Speech and Language Processing

An Introduction to Natural Language Processing,
Computational Linguistics, and Speech Recognition
with Language Models

Third Edition draft

Daniel Jurafsky
Stanford University

James H. Martin
University of Colorado at Boulder

Draft of January 6, 2026. Comments and typos welcome!

Summary of Contents

Contents

Volume I

LARGE LANGUAGE MODELS

In the first part of the book we introduce the fundamental suite of algorithmic and linguistic tools that make up the modern neural large language model. We begin with tokenization and preprocessing, including Unicode, and then proceed to introduce many basic language modeling ideas using simple n-gram language models, we then introduce the algorithms which are the components of large language models: logistic regression, embeddings, and feedforward networks. Next we are ready to introduce the principles of large language modeling, encoder, decoders and pretraining, then the fundamental transformer architecture, then masked language model and other architectures like RNNs and LSTMs, information retrieval and retrieval-based algorithms like RAG, machine translation and the encoder-decoder model, and finally spoken language modeling including both ASR and TTS.

CHAPTER

1 Introduction

La dernière chose qu'on trouve en faisant un ouvrage est de savoir celle qu'il faut mettre la première.
[The last thing you figure out in writing a book is what to put first.]

Pascal

CHAPTER

2 Words and Tokens

```
User:  I need some help, that much seems certain.
ELIZA: WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP
User:  Perhaps I could learn to get along with my mother.
ELIZA: TELL ME MORE ABOUT YOUR FAMILY
User:  My mother takes care of me.
ELIZA: WHO ELSE IN YOU FAMILY TAKES CARE OF YOU
User:  My father.
ELIZA: YOUR FATHER
User:  You are like my father in some ways.
```

Weizenbaum (1966)

ELIZA

The dialogue above is from **ELIZA**, an early natural language processing system that could carry on a limited conversation with a user by imitating the responses of a Rogerian psychotherapist (Weizenbaum, 1966). ELIZA is a surprisingly simple program that uses pattern matching on words to recognize phrases like "I need X" and change the words into suitable outputs like "What would it mean to you if you got X?". ELIZA's mimicry of human conversation, while very crude by modern standards, was remarkably successful: many people who interacted with ELIZA came to believe that it really *understood* them. As a result, this work led researchers to first think about the impacts of chatbots on their users (Weizenbaum, 1976).

tokenization

Of course modern chatbots don't use the simple pattern-based mimicry that ELIZA pioneered. Yet the pattern-based approach to words instantiated in ELIZA is still relevant today in the context of **tokenization**, the task of separating out or **tokenizing** words and word parts from running text. Tokenization, the first step in modern NLP, includes pattern-based approaches that date back to ELIZA.

To understand tokenization we first need to ask: What is a word? Is *um* a word? What about *New York*? Is the nature of words similar across languages? Some languages, like Vietnamese or Cantonese, have very short words while others, like Turkish, have very long words. We also need to think about how to represent words in terms of **characters**. We'll introduce **Unicode**, the modern system for representing characters, and the **UTF-8** text encoding. And we'll introduce the **morpheme**, the meaningful subpart of words (like the morpheme `-er` in the word `longer`)

BPE

The standard way to tokenize text is to use the input characters to guide us. So once we understand the possible subparts of words, we'll introduce the standard **Byte-Pair Encoding** (**BPE**) algorithm that automatically breaks up input text into tokens. This algorithm uses simple statistics of letter sequences to induce a vocabulary of subword tokens. All tokenization systems also depend on **regular expressions** as a processing step. The regular expression is a language for formally specifying and manipulating text strings, an important tool in all modern NLP systems. We'll introduce regular expressions and show examples of their use

regular expressions

Finally, we'll introduce a metric called **edit distance** that measures how similar two words or strings are based on the number of edits (insertions, deletions, substitutions) it takes to change one string into the other. Edit distance plays a role in NLP whenever we need compare two words or strings, for example in the crucial **word error rate** metric for automatic speech recognition.

2.1 Words

How many words are in the following sentence?

```
They picnicked by the pool, then lay back on the grass and
looked at the stars.
```

This sentence has 16 words if we don't count punctuation as words, 18 if we count punctuation. Whether we treat period (“.”), comma (“,”), and so on as words depends on the task. Punctuation is critical for finding boundaries of things (commas, periods, colons) and for identifying some aspects of meaning (question marks, exclamation marks, quotation marks). Large language models generally count punctuation as separate words.

utterance

Spoken language introduces other complications with regard to defining words. What about this utterance from a spoken conversation? (**Utterance** is the technical linguistic term for the spoken correlate of a sentence).

```
I do uh main- mainly business data processing
```

disfluency **fragment** **filled pause**

This utterance has two kinds of **disfluencies**. The broken-off word *main-* is called a **fragment**. Words like *uh* and *um* are called **fillers** or **filled pauses**. Should we consider these to be words? Again, it depends on the application. If we are building a speech transcription system, we might want to eventually strip out the disfluencies. But we also sometimes keep disfluencies around. Disfluencies like *uh* or *um* are actually helpful in speech recognition in predicting the upcoming word, because they may signal that the speaker is restarting the clause or idea, and so for speech recognition they are treated as regular words. Because different people use different disfluencies they can also be a cue to speaker identification. In fact Clark and Fox Tree (2002) showed that *uh* and *um* have different meanings in English. What do you think they are?

word type **word instance**

Perhaps most important, in thinking about what is a word, we need to distinguish two ways of talking about words that will be useful throughout the book. Word **types** are the number of distinct words in a corpus; if the set of words in the vocabulary is V, the number of types is the **vocabulary size** $|V|$. Word **instances** are the total number N of running words.[1] If we ignore punctuation, the picnic sentence has 14 types and 16 instances:

```
They picnicked by the pool, then lay back on the grass and
looked at the stars.
```

We still have decisions to make! For example, should we consider a capitalized string (like *They*) and one that is uncapitalized (like *they*) to be the same word **type**? The answer is that it depends on the task! *They* and *they* might be lumped together as the same type in some tasks where we care less about the formatting, while for other tasks, capitalization is a useful feature and is retained. Sometimes we keep around two versions of a particular NLP model, one with capitalization and one without capitalization.

So far we have been talking about **orthographic words**: words based on our English writing system. But there are many other possible ways to define words. For example, while orthographically `I'm` is one word, grammatically it functions as two words: the subject pronoun `I` and the verb `'m`, short for `am`.

[1] In earlier tradition, and occasionally still, you might see word instances referred to as word *tokens*, but we now try to reserve the word *token* instead to mean the output of subword tokenization algorithms.

Corpus	Types = $\|V\|$	Instances = N
Shakespeare	31 thousand	884 thousand
Brown corpus	38 thousand	1 million
Switchboard telephone conversations	20 thousand	2.4 million
COCA	2 million	440 million
Google n-grams	13 million	1 trillion

Figure 2.1 Rough numbers of wordform types and instances for some English language corpora. The largest, the Google n-grams corpus, contains 13 million types, but this count only includes types appearing 40 or more times, so the true number would be much larger.

The distinctions get even harder to make once we start to think about other languages. For example the writing systems of languages like Chinese, Japanese, and Thai simply don't have orthographic words at all! That is, they don't use spaces to mark potential word-boundaries. In Chinese, for example, words are composed of characters (called **hanzi** in Chinese). Each character generally represents a single unit of meaning (called a **morpheme**, introduced below) and is pronounceable as a single syllable. Words are about 2.4 characters long on average. But since Chinese has no orthographic words, deciding what counts as a word in Chinese is complex. For example, consider the following sentence:

hanzi

(2.1) 姚明进入总决赛 yáo míng jìn rù zǒng jué sài
"Yao Ming reaches the finals"

As Chen et al. (2017b) point out, this could be treated as 3 words (a definition of words called the 'Chinese Treebank' definition, in which Chinese names (family name followed by personal names) are treated as a single word):

(2.2) 姚明 进入 总决赛
YaoMing reaches finals

But the same sentence could be treated as 5 words ('Peking University' standard), in which names are separated into their own units and some adjectives appear as distinct words:

(2.3) 姚 明 进入 总 决赛
Yao Ming reaches overall finals

Finally, it is possible in Chinese simply to ignore words altogether and use characters as the basic elements, treating the sentence as a series of 7 characters, which works pretty well for Chinese since characters are at a reasonable semantic level for most applications (Li et al., 2019):

(2.4) 姚 明 进 入 总 决 赛
Yao Ming enter enter overall decision game

But that method doesn't work for Japanese and Thai, where the individual character is too small a unit.

These issues with defining words makes it hard to use words as the basis for tokenizing text in NLP across languages.

But there's another problem with words. There are too many of them!!! How many words are there in English? When we speak about the number of words in the language, we are generally referring to word types. Fig. 2.1 shows the rough numbers of types and instances computed from some English corpora.

You will notice that the larger the corpora we look at, the more word types we find! That suggests that there is not a clear answer to how many words there are; the answer keeps growing as we see more data! We can see this fact mathematically

Herdan's Law
Heaps' Law

because the relationship between the number of types $|V|$ and number of instances N is called **Herdan's Law** (Herdan, 1960) or **Heaps' Law** (Heaps, 1978) after its discoverers (in linguistics and information retrieval respectively). It is shown in Eq. 2.5, where k and β are positive constants, and $0 < \beta < 1$.

$$|V| = kN^{\beta} \tag{2.5}$$

The value of β depends on the corpus size and the genre; numbers from 0.44 to 0.56 or even higher have often been reported. Roughly we can say that the vocabulary size for a text goes up a little faster than the square root of its length in words.

function words
content words

There are also variants of the law, which capture the fact that we can distinguish roughly two classes of words. One is **function words**, the grammatical words like English *a* and *of*, that tend not to grow indefinitely (a language tends to have a fixed number of these). The other is **content words**: nouns, adjectives and verbs that tend to have meanings about people and places and events. Nouns, and especially particular nouns like names and technical terms do tend to grow indefinitely. So models that are sensitive to this difference between function words and content words have one value of β for the initial part of the corpus where all words are still appearing, and then a second β afterwords for when only the content words are still appearing. Fig. 2.2 shows an example from Tria et al. (2018) showing two values of β (called γ in their figure) for Heaps law computed on the Gutenberg corpus of books. Note that

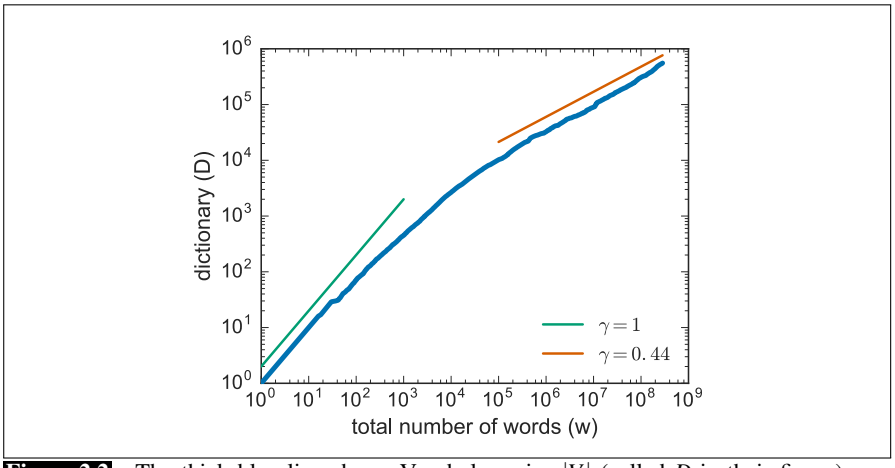


Figure 2.2 The thick blue line shows Vocabulary size $|V|$ (called D in their figure) as a function of text length (their w), computed on the Gutenberg corpus of publicly available books. Note that at the beginning of the corpus, we see both common and function words and the relationship between corpus size and vocabulary is roughly linear (green line, $\gamma = 1$). Later, after the function words have mainly appeared, the number of new words slows down and is closer to the square root of the corpus size. Figure from Tria et al. (2018).

The fact that words grow without end leads to a problem for any computational model. No matter how big our vocabulary, we will never have a vocabulary that captures all the possible words that might occur! That means that our computational model will constantly see **unknown words**: words that it has never seen before. This is a huge problem for machine learning models.

Because of these two problems (first, that many languages don't have orthographic words, and defining them post-hoc is challenging and second, that the number of words grows without bound), language models and other NLP models don't

tend to use words as their unit of processing. Instead, they use smaller units called **subwords** that can be recombined to model new words that our model has never seen before. To think about defining subwords, we first need to talk about units that are smaller than words; **morphemes** and **characters**.

2.2 Morphemes: Parts of Words

Words have parts. At the level of characters, this is obvious. The word *cats* is composed of four characters, 'c', 'a', 't', 's'. But this is also true at a more subtle level: words have components that themselves have coherent meanings. These components are called **morphemes**, and the study of morphemes is called **morphology**. A **morpheme** is a minimal meaning-bearing unit in a language. So, for example, the word *fox* consists of one morpheme (the morpheme *fox*) while the word *cats* consists of two: the morpheme *cat* and the morpheme *-s* that indicates plural.

morphology
morpheme

Here's a sentence in English segmented into morphemes with hyphens:

(2.6) Doc work-ed care-ful-ly wash-ing the glass-es

As we mentioned above, in Chinese, conveniently, the writing system is set up so that each character mainly describes a morpheme. Here's a sentence in Mandarin Chinese with each morpheme character glossed, followed by the translation:

(2.7) 梅 干 菜 用 清 水 泡 软 ， 捞 出 后 ， 沥 干
plum dry vegetable use clear water soak soft , remove out after , drip dry
切 碎
chop fragment
Soak the preserved vegetable in water until soft, remove, drain, and chop

We generally distinguish two broad classes of morphemes: **roots**—the central morpheme of the word, supplying the main meaning—and **affixes**—adding "additional" meanings of various kinds. In the English example above, for the word *worked*, *work* is a root and *-ed* is an affix; similarly for *glasses*, *glass* is a root and *-es* an affix.

root
affix

Affixes themselves fall into two classes, or more correctly a continuum between two poles. At one end, **inflectional morphemes** are grammatical morphemes that tend to play a syntactic role, such as marking agreement. For example, English has the inflectional morpheme *-s* (or *-es*) for marking the **plural** on nouns and the inflectional morpheme *-ed* for marking the past tense on verbs. Inflectional morphemes tend to be productive and often obligatory and their meanings tend to be predictable. **Derivational morphemes** are more idiosyncratic in their application and meaning. Usually they apply only to a specific subclass of words and result in a word of a *different* grammatical class than the root, often with a meaning hard to predict exactly. In the example above, the word *care* (a noun) can be combined with the derivational affix *-full* to produce an adjective (*careful*), and another derivational affix *-ly* to result in an adverb (*carefully*).

inflectional morphemes
derivational morphemes

There is another class of morphemes: **clitics**. A clitic is a morpheme that acts syntactically like a word but is reduced in form and attached (phonologically and sometimes orthographically) to another word. For example the English morpheme *'ve* in the word *I've* is a clitic; it has the grammatical meaning of the word *have*, but in form in cannot appear alone (you can't just say the sentence "'ve"). The English possessive morpheme *'s* in the phrase *the teacher's book* is a clitic. French definite

clitic

article *l'* in the word *l'opera* is a clitic, as are prepositions in Arabic like *b* 'by/with' and conjunctions like *w* 'and'.

morphological typology

The study of how languages vary in their morphology, i.e., how words break up into their parts, is called **morphological typology**. While morphologies of languages can differ along many dimensions, two dimensions are particularly relevant for computational word tokenization.

The first dimension is the **number of morphemes per word**. In some languages, like Vietnamese and Cantonese, each word on average has just over one morpheme. We call languages at this end of the scale **isolating** languages. For example each word in the following Cantonese sentence has one morpheme (and one syllable):

isolating

(2.8) *keoi5 waa6 cyun4 gwok3 zeoi3 daai6 gaan1 uk1 hai6 ni1 gaan1*
he say entire country most big building house is this building
"He said the biggest house in the country was this one"

Alternatively, in languages like Koryak, a Chukotko-Kamchatkan language spoken in the northern part of the Kamchatka peninsula in Russia, a single word may have very many morphemes, corresponding to a whole sentence in English (Arkadiev, 2020; Kurebito, 2017). We call languages toward this end of the scale **synthetic** languages, and the very end of the scale **polysynthetic** languages.

synthetic
polysynthetic

(2.9) *t-ə-nk'e-mejŋ-ə-jetemə-nni-k*
1SG.S-E-midnight-big-E-yurt.cover-E-sew-1SG.S[PFV]
"I sewed a lot of yurt covers in the middle of a night."
(Koryak, Chukotko-Kamchatkan, Russia; Kurebito (2017, 844))

Fig. 2.3 shows an early computation of morphemes per words on a few languages by the linguistic typologist Joseph Greenberg (1960).

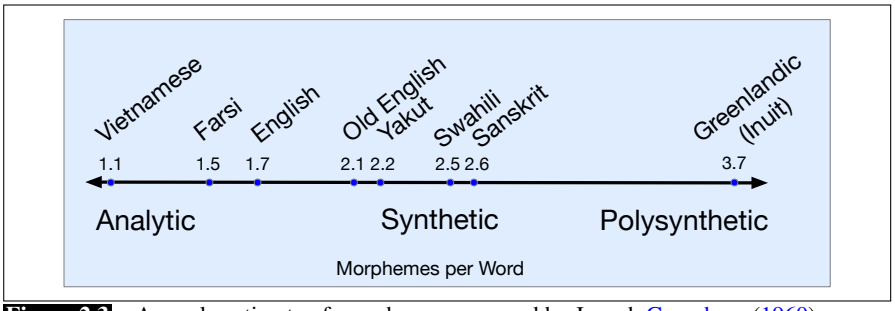


Figure 2.3 An early estimate of morphemes per word by Joseph Greenberg (1960).

agglutinative
fusion

The second dimension is the degree to which morphemes are easily segmentable, ranging from **agglutinative** languages like Turkish, in which morphemes have relatively clean boundaries, to **fusion** languages like Russian, in which a single affix may conflate multiple morphemes, like *-om* in the word *stolom* (table-SG-INSTR-DECL1), which fuses the distinct morphological categories instrumental, singular, and first declension.

The English *-s* suffix in *She read***s** *the article* is an example of fusion, since the suffix means both third person singular but also means present tense, and there's no way to divide up the meaning to different parts of the *-s*.

Although we have loosely talked about these properties (analytic, polysynthetic, fusional, agglutinative) as if they are properties of languages, in fact languages can make use of different morphological systems so it would be more accurate to talk about these as general tendencies.

Nonetheless, the fact morphemes can be hard to define, and that many languages can have complex morphemes that aren't easy to break up into pieces makes it very difficult to use morphemes as a standard for tokenization cross-lingually.

2.3 Unicode

Another option we could consider for tokenization is the level of the individual character. How do we even represent characters across languages and writing system?

Unicode The **Unicode** standard is a method for representing text written using any character in any script of the languages of the world (including dead languages like Sumerian cuneiform, and invented languages like Klingon).

Let's start with a brief historical note about an English-specific subset of Unicode (technically called 'Basic Latin' in Unicode, and commonly referred to as ASCII).

ASCII Starting in the 1960s, the Latin characters used to write English (like the ones used in this sentence), were represented with a code called **ASCII** (American Standard Code for Information Interchange). ASCII represented each character with a single byte. A byte can represent 256 different characters, but ASCII only used 127 of them; the high-order bit of ASCII bytes is always set to 0. (Actually it only used 95 of them and the rest were control codes for an obsolete machine called a teletype). Here's a few ASCII characters with their representation in hex and decimal:

Ch	Hex	Dec	Ch	Hex	Dec		Ch	Hex	Dec	Ch	Hex	Dec
<	3C	60	@	40	64	...	\	5C	92	‘	60	96
=	3D	61	A	41	65	...	[	5D	93	a	61	97
>	3E	62	B	42	66	...	^	5E	94	b	62	98
?	3F	63	C	43	67	...	_	5F	95	c	63	99

Figure 2.4 Some selected ASCII codes for some English letters, with the codes shown both in hexadecimal and decimal.

But ASCII is of course insufficient since there are lots of other characters in the world's writing systems! Even for scripts that use Latin characters, there are many more than the 95 in ASCII. For example, this Spanish phrase (meaning "Sir, replied Sancho") has two non-ASCII characters, ñ and ó:

(2.10) Señor- respondió Sancho-

Devanagari And lots of languages aren't based on Latin characters at all! The **Devanagari** script is used for 120 languages (including Hindi, Marathi, Nepali, Sindhi, and Sanskrit). Here's a Devanagari example from the Hindi text of the Universal Declaration of Human Rights:

अनुच्छेद १(एक): सभी मनुष्य जन्म से स्वतन्त्र तथा मर्यादा और अधिकारों में समान होते हैं। वे तर्क और विवेक से सम्पन्न हैं तथा उन्हें भ्रातृत्व की भावना से परस्पर के प्रति कार्य करना चाहिए।

Chinese has about 100,000 Chinese characters in Unicode (including overlapping and non-overlapping variants used in Chinese, Japanese, Korean, and Vietnamese, collectively referred to as CJKV).

All in all there are more than 150,000 characters and 168 different scripts supported in Unicode 16.0. Even though many scripts from around the world have yet to be added to Unicode, there are so many there, from scripts used by modern languages (Chinese, Arabic, Hindi, Cherokee, Ethiopic, Khmer, N'Ko, Turkish,

Spanish) to scripts of ancient languages (Cuneiform, Ugaritic, Egyptian Hieroglyph, Pahlavi), as well as mathematical symbols, emojis, currency symbols, and more.

2.3.1 Code Points

code point

How does it work? Unicode assigns a unique id, called a **code point**, for each one of these 150,000 characters.

The code point is an abstract representation of the character, and each code point is represented by a number, traditionally written in hexadecimal, from number 0 through 0x10FFFF (which is 1,114,111 decimal). Having over a million code points means there is a lot of room for new characters. It is traditional to represent these code points with the prefix "U+" (which just means "the following is a Unicode hex representation of a code point"). So the code point for the character `a` is `U+0061` which is the same as `0x0061`. (Note that Unicode was designed to be backwards compatible with ASCII, which means that the first 127 code points, including the code for `a`, are identical with ASCII.) Here are some sample code points; some (but not all) come with descriptions:

```
U+0061  a   LATIN SMALL LETTER A
U+0062  b   LATIN SMALL LETTER B
U+0063  c   LATIN SMALL LETTER C
U+00F9  ù   LATIN SMALL LETTER U WITH GRAVE
U+00FA  ú   LATIN SMALL LETTER U WITH ACUTE
U+00FB  û   LATIN SMALL LETTER U WITH CIRCUMFLEX
U+00FC  ü   LATIN SMALL LETTER U WITH DIAERESIS
U+8FDB  进
U+8FDC  远
U+8FDD  违
U+8FDE  连
U+1F600 😀  GRINNING FACE
U+1F00E 🀎  MAHJONG TILE EIGHT OF CHARACTERS
```

glyph

Note that a code point does not specifiy the **glyph**, the visual representation of a character. Glyphs are stored in **fonts**. The code point `U+0061` is an abstract representation of `a`. There can be an indefinite number of visual representations, for example in different fonts like Times Roman (a) or Courier (a), or different font styles like boldface (**a**) or italic (*a*). But all of them are represented by the same code point `U+0061`.

2.3.2 UTF-8 Encoding

While the code point (the unique id) is the abstract Unicode representation of the character, we don't just stick that id in a text file.

encoding

Instead, whenever we need to represent a character in a text string, we write an **encoding** of the character. There are many different possible encoding methods, but the encoding method called UTF-8 is by far the most frequent (for example almost the entire web is encoded in UTF-8).

Let's talk about encodings. The Unicode representation of the word `hello` consists of the following sequence of 5 code points:

```
U+0068  U+0065 U+006C U+006C U+006F
```

We can imagine a very simple encoding method: just write the code point id in a file. Since there are more than 1 million characters, 16 bits (2 bytes) isn't enough, so we'll need to use 4 bytes (32 bit) to capture the 21 bits we need to represent 1.1 million characters. (We could fit it in 3 bytes but it's inconvenient to use multiples of 3 for bytes.)

With this 4-byte representation the word `hello` would be encoded as the following set of bytes:

```
00 00 00 68 00 00 00 65 00 00 00 6C 00 00 00 6C 00 00 00 6F
```

But we don't use this encoding (which is technically called UTF-32) because it makes every file 4 times longer than it would have been in ASCII, making files really big and full of zeros. Also those zeros cause another problem: it turns out that having any byte that is completely zero messes things up for backwards compatibility for ASCII-based systems that historically used a 0 byte as an end-of-string marker.

UTF-8

Instead, the most common encoding standard is **UTF-8** (Unicode Transformation Format 8), which represents characters efficiently (using fewer bytes on average) by writing some characters using fewer bytes and some using more bytes. UTF-8 is thus a **variable-length encoding**.

variable-length encoding

For some characters (the first 127 code points, i.e. the set of ASCII characters), UTF-8 encodes them as a single byte, so the UTF-8 encoding of `hello` is :

```
68 65 6C 6C 6F
```

This conveniently means that files encoded in ASCII are also valid UTF-8 encodings!

But UTF-8 is a variable length encoding, meaning that code points $\geq$128 are encoded as a sequence of two, three, or four bytes. Each of these bytes are between 128 and 255, so they won't be confused with ASCII, and each byte indicates in the first few bits whether it's a 2-byte, 3-byte, or 4-byte encoding.

Code Points		UTF-8 Encoding			
From - To	**Bit Value**	**Byte 1**	**Byte 2**	**Byte 3**	**Byte 4**
U+0000-U+007F	0xxxxxxx	xxxxxxxx			
U+0080-U+07FF	00000yyy yyxxxxxx	110yyyyy	10xxxxxx		
U+0800-U+FFFF	zzzzyyyy yyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	
U+010000-U+10FFFF	000uuuuu zzzzyyyy yyxxxxxx	11110uuu	10uuzzzz	10yyyyyy	10xxxxxx

Figure 2.5 Mapping from Unicode code point to the variable length UTF-8 encoding. For a given code point in the From-To range, the bit value in column 2 is packed into 1, 2, 3, or 4 bytes. Figure adapted from Unicode 16.0 Core Spec Chapter 3 Table 3-6.

Fig. 2.5 shows how this mapping occurs. For example these rules explain how the character ñ, which has code point U+00F1, or bit sequence `00000000 11110001`, (where blue indicates the sequence yyyyy and red the sequence xxxxxx) is encoded into to the two-byte bit sequence `11000011 10110001` or `0xC3B1`. As a result of these rules, the first 127 characters (ASCII) are mapped to one byte, most remaining characters in European, Middle Eastern, and African scripts map to two bytes, most Chinese, Japanese, and Korean characters map to three bytes, and rarer CJKV characters and emojis and some symbols map to 4 bytes.

UTF-8 has a number of advantages. It's relatively efficient, using fewer bytes for commonly-encountered characters, it doesn't use zero bytes (except when literally representing the NULL character which is U+0000), it's backwards compatible with ASCII, and it's self-synchronizing, meaning that if a file is corrupted, it's always possible to find the start of the next or prior character just by moving up to 3 bytes left or right.

Unicode and Python: Starting with Python 3, all Python strings are stored internally as Unicode, each string a sequence of Unicode code points. Thus string functions and regular expressions all apply natively to code points. For example, functions like `len()` of a string return its length in characters, i.e., code points, not its length in bytes.

When reading or writing from a file, however, the code points need to be encoded and decoding using a method like UTF-8. That is, every file is encoded in some encoding. If it's not UTF-8, it's an older encoding method like ASCII or Latin-1 (iso_8859_1). There is no such thing as a text file without an encoding. The encoding method is specified in Python when opening a file for reading and writing.

2.4 Subword Tokenization: Byte-Pair Encoding

tokenization
tokens

Tokenization, the first stage of natural language processing, is the process of segmenting the running input text into **tokens**.

We've seen three candidates for tokens: words, morphemes and characters. But each has problems as a unit. Words and morphemes seem approximately at the right level for NLP processing, since they tend to have consistent meanings, but they are challenging to define formally. Characters are clearer to define, but seem too small a unit to choose for tokens.

In this section we introduce what we do in practice for NLP: use a data-driven approach to define tokens that will generally result in units about the size of morphemes or words, but occasionally use units as small as characters.

Why tokenize the input? One reason is that converting an input to a deterministic fixed set of units means that different algorithms and systems can agree on simple questions. For example, How long is this text? (How many units are in it?). Or: Is *don't* or *New York* one token or two? Standardizing is thus essential for replicability in NLP experiments, and many algorithms that we introduce in this book (like the **perplexity** metric for language models) assume that all texts have a fixed tokenization.

Tokenization algorithms that include smaller tokens for morphemes and letters also eliminate the problem of **unknown words**. What are these? As we will see in the next chapter, NLP algorithms often learn some facts about language from one corpus (a **training** corpus) and then use these facts to make decisions about a separate **test** corpus and its language. Thus if our training corpus contains, say the words *low*, *new*, and *newer*, but not *lower*, then if the word *lower* appears in our test corpus, our system will not know what to do with it.

subwords

To deal with this unknown word problem, modern tokenizers automatically induce sets of tokens that include tokens smaller than words, called **subwords**. Subwords can be arbitrary substrings, or they can be meaning-bearing units like the morphemes *-est* or *-er*. In modern tokenization schemes, many tokens are words, but other tokens are frequently occurring morphemes or other subwords like *-er*. Every unseen word can thus be represented by some sequence of known subword units. For example, if we had happened not to ever see the word *lower*, when it appears we could segment it successfully into *low* and *er* which we had already seen. In the worst case, a really unusual word (perhaps an acronym like *GRPO*) could be tokenized as a sequence of individual letters if necessary.

Two tokenization algorithms are widely used in modern language models: **byte-pair encoding** (BPE) (Sennrich et al., 2016), and **unigram language modeling**

BPE (ULM) (Kudo, 2018).[2] In this section we introduce the **byte-pair encoding** or **BPE** algorithm (Sennrich et al., 2016; Gage, 1994); see Fig. 2.6.

Like most tokenization schemes, the BPE algorithm has two parts: a **trainer**, and an **encoder**. In general in the token training phase we take a raw training corpus (usually roughly pre-separated into words, for example by whitespace) and induce a vocabulary, a set of tokens. Then a token encoder takes a raw test sentence and encodes it into the tokens in the vocabulary that were learned in training.

2.4.1 BPE training

The **BPE** training algorithm iteratively merges frequent neighboring tokens to create longer and longer tokens. The algorithm begins with a vocabulary that is just the set of all individual characters. It then examines the training corpus, and finds the two characters that are most frequently adjacent. Imagine our original corpus is 10 characters long, using a vocabulary of 5 characters, {A, B, C, D, E}:

```
A B D C A B E C A B
```

The most frequent neighboring pair of characters is "A B" so we merge those, add a new merged token 'AB' to the vocabulary, and replace every adjacent 'A' 'B' in the corpus with the new 'AB':

```
AB D C AB E C AB
```

Now we have a vocabulary of 6 possible tokens {A, B, C, D, E, AB}, and the corpus has length 7. And now the most frequent pair of tokens is "C AB", so we merge those, leading to a vocabulary with 7 tokens {A, B, C, D, E, AB, CAB}, and the corpus has length 5.

```
AB D CAB E CAB
```

The algorithm continues to count and merge, creating new longer and longer character strings, until k merges have been done creating k novel tokens; k is thus a parameter of the algorithm. The resulting vocabulary consists of the original set of characters plus k new symbols. That's the core of the algorithm.

The only additional complication is that in practice, instead of running on the raw sequence of characters, the algorithm is usually run only *inside* words. That is, the algorithm does not merge across word boundaries. To do this, the input corpus is often first separated at white space and punctuation (using the regular expressions that we define later in the chapter). This gives a starting set of strings, each corresponding to the characters of a word, (with the white space usually attached to the start of the word), together with the counts of the words. Then while counts come from a corpus, merges are only allowed within the strings.

Let's see how the full algorithm thus works on this tiny synthetic corpus, where we've explicitly marked the spaces between words:[3]

(2.11) `set␣new␣new␣renew␣reset␣renew`

First, we'll break up the corpus into words, with leading whitespace, together with their counts; no merges will be allowed to go beyond these word boundaries. The result looks like the following list of 4 words and a starting vocabulary of 7 characters:

[2] The **SentencePiece** library includes implementations of both of these (Kudo and Richardson, 2018a), and people sometimes use the name **SentencePiece** to simply mean ULM tokenization.

[3] Yes, we realize this isn't a particularly likely or exciting sentence.

corpus	vocabulary
2 _ n e w	_, e, n, r, s, t, w
2 _ r e n e w	
1 s e t	
1 _ r e s e t	

The BPE training algorithm first counts all pairs of adjacent symbols: the most frequent is the pair `n e` because it occurs in `new` (frequency of 2) and `renew` (frequency of 2) for a total of 4 occurrences. We then merge these symbols, treating `ne` as one symbol, and count again:

corpus	vocabulary
2 _ ne w	_, e, n, r, s, t, w, ne
2 _ r e ne w	
1 s e t	
1 _ r e s e t	

Now the most frequent pair is `ne w` (total count=4), which we merge.

corpus	vocabulary
2 _ new	_, e, n, r, s, t, w, ne, new
2 _ r e new	
1 s e t	
1 _ r e s e t	

Next _ r (total count of 3) get merged to _r, and then _r e (total count 3) gets merged to _re. The system has essentially induced that there is a word-initial prefix `re-`:

corpus	vocabulary
2 _ new	_, e, n, r, s, t, w, ne, new, _r, _re
2 _re new	
1 s e t	
1 _re s e t	

If we continue, the next merges are:

merge	current vocabulary
(_, new)	_, e, n, r, s, t, w, ne, new, _r, _re, _new
(_re, new)	_, e, n, r, s, t, w, ne, new, _r, _re, _new, _renew
(s, e)	_, e, n, r, s, t, w, ne, new, _r, _re, _new, _renew, se
(se, t)	_, e, n, r, s, t, w, ne, new, _r, _re, _new, _renew, se, set

function BYTE-PAIR ENCODING(strings C, number of merges k) **returns** vocab V

$V \leftarrow$ all unique characters in C # initial set of tokens is characters
for $i = 1$ **to** k **do** # merge tokens k times
  $t_L, t_R \leftarrow$ Most frequent pair of adjacent tokens in C
  $t_{NEW} \leftarrow t_L + t_R$ # make new token by concatenating
  $V \leftarrow V + t_{NEW}$ # update the vocabulary
  Replace each occurrence of t_L, t_R in C with t_{NEW} # and update the corpus
return V

Figure 2.6 The training part of the BPE algorithm for taking a corpus broken up into individual characters or bytes, and learning a vocabulary by iteratively merging tokens. Figure adapted from Bostrom and Durrett (2020).

2.4.2 BPE encoder

Once we've learned our vocabulary, the BPE **encoder** is used to tokenize a test sentence. The encoder just runs on the test data the merges we have learned from the training data. It runs them greedily, in the order we learned them. (Thus the frequencies in the test data don't play a role, just the frequencies in the training data). So first we segment each test sentence word into characters. Then we apply the first rule: replace every instance of `n e` in the test corpus with `ne`, and then the second rule: replace every instance of `ne w` in the test corpus with `new`, and so on. By the end of course many of the merges simple recreated words in the training set. But the merges also created knowledge of morphemes like the `re-` prefix (that might appear in perhaps unseen combinations like `revisit` or `rearrange`), or the morpheme `new` without an initial space (hence word-internal) that might appear at the start of sentences or in words unseen in training like `anew`.

Of course in real settings BPE is run with tens of thousands of merges on a very large input corpus, to produce vocabulary sizes of 50,000, 100,000, or even 200,000 tokens. The result is that most words can be represented as single tokens, and only the rarer words (and unknown words) will have to be represented by multiple tokens. At least for English. For multilingual systems, the tokens can be dominated by English, leaving fewer tokens for other languages, as we'll discuss below.

2.4.3 BPE in practice

The example above just showed simple BPE learning from sequences of ASCII bytes. How does BPE work with Unicode input? We normally run BPE on the individual bytes of UTF-8-encoded text. That is, we take a Unicode representations of text as a series of code points, encode it in bytes using UTF-8, and we treat each of these individual bytes as the input to BPE. Thus BPE likely begins by rediscovering the 2-byte and common 3-byte sequences that UTF-8 uses to encode various code points. Again, running BPE only inside presegmented words helps avoid problems. Because there are only 256 possible values of a byte, there will be no unknown tokens, although it's possible that BPE will learn some illegal UTF-8 sequences across character boundaries. These will be very rare, and can be eliminated with a filter.

Let's see some examples of the industrial application of the BPE tokenizer used in large systems like OpenAI GPT4o. This tokenizer has 200K tokens, which is a comparatively large number. We can use Tat Dat Duong's Tiktokenizer visualizer (https://tiktokenizer.vercel.app/) to see the number of tokens in a given sentence. For example here's the tokenization of a nonsense sentence we made up; the visualizer uses a center dot to indicate a space:

```
Anyhow,·she's·seen·Jane's·224123·flowers·anyhow!
```

The visualization shows colors to separate out words, but of course the true output of the tokenizer is simply a sequence of unique token ids. (In case you're interested, they were the following 13 tokens: 11865, 8923, 11, 31211, 6177, 23919, 885, 220, 19427, 7633, 18887, 147065, 0)

Notice that most words are their own token, usually including the leading space. Clitics like `'s` are segmented off when they appear on proper nouns like `Jane`, but are counted as part of a word for frequent words like `she's`. Numbers tend to be segmented into chunks of 3 digits. And some words (like *anyhow*) are segmented differently if they appear capitalized sentence-initially (two tokens, `Any` and `how`), then if they appear after a space, lower case (one token `anyhow`).

pretokenization

Some of these are related to preprocessing steps. As we mentioned briefly above, language models usually create their tokens in a **pretokenization** stage that first segments the input using regular expressions, for example breaking the input at spaces and punctuation, stripping off clitics, and breaking numbers into sets of digits. We'll see how to use regular expressions in Section 2.6.

SuperBPE
BoundlessBPE

It's possible to change this pretokenization to allow BPE tokens to span multiple words. For example the **SuperBPE** (Liu et al., 2025) and **BoundlessBPE** (Schmidt et al., 2025) algorithms first induce regular BPE subword tokens by enforcing pretokenization. They then run a second stage of BPE allowing merges across spaces and punctuation. The result is a large set of tokens that can be more efficient (Fig. 2.7).

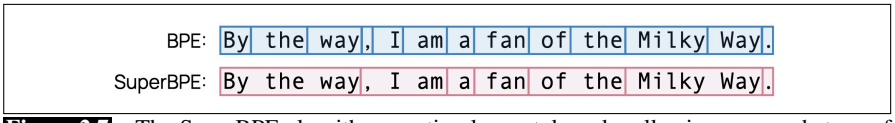


Figure 2.7 The SuperBPE algorithm creating larger tokens by allowing a second stage of merging across spaces. Figure from Liu et al. (2025).

Many of the tokenizers used in practice for large language models are multilingual, trained on many languages. But because the training data for large language models is vastly dominated by English text, these multilingual BPE tokenizers tend to use most of the tokens for English, leaving fewer of them for other languages. The result is that they do a better job of tokenizing English, and the other languages tend to get their words split up into shorter tokens. For example let's look at a Spanish sentence from a recipe for plantains, together with an English translation.

The English has 18 tokens; each of the 14 words is a token (none of the words are split into multiple tokens):

```
In·a·deep·bowl,·mix·the·orange·juice·with·the·sugar,·g
inger,·and·nutmeg.
```

By contrast, the original 16 words in Spanish have been encoded into 33 tokens, a much larger number. Notice that many basic words have been broken into pieces. For example `hondo`, 'deep', has been segmented into `h` and `ondo`. Similarly for `jugo`, 'juice', `nuez`, 'nut' and `jenjibre` 'ginger'):

```
En·un·recipiente·hondo,·mezclar·el·jugo·de·naranja·con
·el·azúcar,·jengibre,·y·nuez·moscada.
```

Spanish is not a particularly low-resource language; this oversegmenting can be even more serious in lower resource languages, often down to individual characters. Oversegmenting into these tiny tokens can cause various problems for the downstream processing of the language. As will become more clear once we introduce transformer models in Chapter 8, such fragmentation can lead to poor representations of meaning, the need for longer contexts, and higher costs to train models (Rust et al., 2021; Ahia et al., 2023).

2.5 Corpora

Words don't appear out of nowhere. Any particular piece of text that we study is produced by one or more specific speakers or writers, in a specific dialect of a

specific language, at a specific time, in a specific place, for a specific function.

Perhaps the most important dimension of variation is the language. NLP algorithms are most useful when they apply across many languages. The world has 7097 languages at the time of this writing, according to the online Ethnologue catalog (Simons and Fennig, 2018). It is important to test algorithms on more than one language, and particularly on languages with different properties; by contrast there is an unfortunate current tendency for NLP algorithms to be developed or tested just on English (Bender, 2019). Even when algorithms are developed beyond English, they tend to be developed for the official languages of large industrialized nations (Chinese, Spanish, Japanese, German etc.), but we don't want to limit tools to just these few languages. Furthermore, most languages also have multiple varieties, often spoken in different regions or by different social groups. Thus, for example, if we're

AAE processing text that uses features of African American English (**AAE**) or African American Vernacular English (AAVE)—the variations of English that can be used by millions of people in African American communities (King 2020)—we must use NLP tools that function with features of those varieties. Twitter posts might use features often used by speakers of African American English, such as constructions like

MAE *iont* (*I don't* in Mainstream American English (**MAE**)), or *talmbout* corresponding to MAE *talking about*, both examples that influence word segmentation (Blodgett et al. 2016, Jones 2015).

It's also quite common for speakers or writers to use multiple languages in a sin-

code switching gle utterance, a phenomenon called **code switching**. Code switching is enormously common across the world; here are examples showing Spanish and (transliterated) Hindi code switching with English (Solorio et al. 2014, Jurgens et al. 2017):

(2.12) Por primera vez veo a @username actually being hateful! it was beautiful:)
[For the first time I get to see @username actually being hateful! it was beautiful:)]

(2.13) dost tha or ra- hega ... dont wory ... but dherya rakhe
["he was and will remain a friend ... don't worry ... but have faith"]

Another dimension of variation is the genre. The text that our algorithms must process might come from newswire, fiction or non-fiction books, scientific articles, Wikipedia, or religious texts. It might come from spoken genres like telephone conversations, business meetings, police body-worn cameras, medical interviews, or transcripts of television shows or movies. It might come from work situations like doctors' notes, legal text, or parliamentary or congressional proceedings.

Text also reflects the demographic characteristics of the writer (or speaker): their age, gender, race, socioeconomic class can all influence the linguistic properties of the text we are processing.

And finally, time matters too. Language changes over time, and for some languages we have good corpora of texts from different historical periods.

Because language is so situated, when developing computational models for language processing from a corpus, it's important to consider who produced the language, in what context, for what purpose. How can a user of a dataset know all these

datasheet details? The best way is for the corpus creator to build a **datasheet** (Gebru et al., 2020) or **data statement** (Bender et al., 2021) for each corpus. A datasheet specifies properties of a dataset like:

Motivation: Why was the corpus collected, by whom, and who funded it?

Situation: When and in what situation was the text written/spoken? For example, was there a task? Was the language originally spoken conversation, edited text, social media communication, monologue vs. dialogue?

Language variety: What language (including dialect/region) was the corpus in?

Speaker demographics: What was, e.g., the age or gender of the text's authors?

Collection process: How big is the data? If it is a subsample how was it sampled? Was the data collected with consent? How was the data pre-processed, and what metadata is available?

Annotation process: What are the annotations, what are the demographics of the annotators, how were they trained, how was the data annotated?

Distribution: Are there copyright or other intellectual property restrictions?

2.6 Regular Expressions

regular expression

One of the most useful tools for text processing in computer science is the **regular expression** (or **regex**), a language for specifying text strings. Regexes are used in every computer language, in text processing tools like Unix `grep`, and in editors like vim or Emacs. And they play an important role in the pre-tokenization step for tokenization algorithms like BPE. Formally, a regular expression is an algebraic notation for characterizing a set of strings. Practically, we can use a regex to search for a string in a text and to specify how to change the string, both of which are key to tokenization.

string

We use regular expressions to search for a **pattern** in a **string** which can be a single line or a longer text. For example, the Python function

```
re.search(pattern,string)
```

scans through the `string` and returns the first match inside it for the `pattern`. In the following examples we generally highlight the exact string that matches the regular expression and show only the first match. We'll use Python syntax, expressing the regex as a raw string delimited by double quotes: `r"regex"`. Raw strings treat backslashes as literal characters, which will be important since many regex patterns we'll introduce use backslashes.

Regular expressions come in different variants, so using an online regex tester can help make sure your regex does what you think it's doing.

2.6.1 Character Disjunction: The Square Bracket

The simplest kind of regular expression is a sequence of simple characters. The pattern `r"Buttercup"` matches the substring `Buttercup` in any string (like the string `I'm called little Buttercup`). But often we need to use special characters. For example, we might want to match *either* some character or another. For example, regular expressions are generally **case sensitive**: `r"s"` matches a lower case `s` but not an upper case `S`. To match both `s` and `S` we can use the **character disjunction** operator, the square braces `[` and `]`. The string of characters inside the braces specifies a disjunction of characters to match. For example, Fig. 2.8 shows that the pattern `r"[mM]"` matches patterns containing either `m` or `M`.

character disjunction

Pattern	Match	String
`r"[mM]ary"`	Mary or mary	"Mary Ann stopped by Mona's"
`r"[abc]"`	'a', 'b', *or* 'c'	"In uomini, in soldati"
`r"[1234567890]"`	any one digit	"plenty of 7 to 5"

Figure 2.8 The use of the brackets `[]` to specify a disjunction of characters.

The regular expression `r"[1234567890]"` specifies any single digit. This can get awkward (imagine typing `r"[ABCDEFGHIJKLMNOPQRSTUVWXYZ]"` to mean an uppercase letter) so the brackets can also be used with a dash (`-`) to specify any one character in a **range**. The pattern `r"[2-5]"` specifies any one of the characters *2*, *3*, *4*, or *5*. The pattern `r"[b-g]"` specifies one of the characters *b*, *c*, *d*, *e*, *f*, or *g*. Some other examples are shown in Fig. 2.9.

range

Regex	Match	Example Patterns Matched
`r"[A-Z]"`	an upper case letter	"we should call it 'Drenched Blossoms' "
`r"[a-z]"`	a lower case letter	"my beans were impatient to be hoed!"
`r"[0-9]"`	a single digit	"Chapter 1: Down the Rabbit Hole"

Figure 2.9 The use of the brackets `[]` plus the dash `-` to specify a range.

The square braces can also be used to specify what a single character *cannot* be, by use of the caret `^`. If the caret `^` is the first symbol after the open square brace `[`, the resulting pattern is negated. For example, the pattern `r"[^a]"` matches any single character (including special characters) except *a*. This is only true when the caret is the first symbol after the open square brace. If it occurs anywhere else, it usually stands for a caret; Fig. 2.10 shows some examples.

Regex	Match (single characters)	Example Patterns Matched
`r"[^A-Z]"`	not an upper case letter	"Oyfn pripetchik"
`r"[^Ss]"`	neither 'S' nor 's'	"I have no exquisite reason for't"
`r"[^.]"`	not a period	"our resident Djinn"
`r"[e^]"`	either 'e' or '^'	"look up ^ now"
`r"a^b"`	the pattern 'a^b'	"look up a^ b now"

Figure 2.10 The caret `^` for negation or just to mean `^`. See below re: the backslash for escaping the period.

2.6.2 Counting, Optionality, and Wildcards

How can we talk about optional elements, like an optional *s* if we want to match both *koala* and *koalas*? We can't use the square brackets, because while they allow us to say "s or S", they don't allow us to say "s or nothing". For this we use the question mark `r"?"`, which means "the preceding character or nothing",. So `r"colou?r"` matches both `color` and `colour`, and `r"koalas?"` matches `koala` or `koalas`.

There's another way to talk about elements that may or may not occur. Consider the language of certain sheep, which consists of strings that look like the following:

```
baa!
baaa!
baaaa!
...
```

This sheep language consists of strings with a `b`, followed by at least two (and arbitrarily more) *a*'s, followed by an exclamation point. To represent this language, we'll use a useful operator that is represented by the asterisk or `*`, called the **Kleene** `*` (generally pronounced "cleany star"). The Kleene star means "zero or more occurrences of the immediately previous character or regular expression". So `r"a*"` means "any string of zero or more `a`s".

Kleene *

Could `r"ba*"` represent the sheep language? It will correctly match `ba` or `baaaaaa`, but there's a problem! It will also match `b`, with no `a`, or `ba` with only one

a. That's because Kleene star means "zero or more occurrences". Instead, for the sheep language we'll want `r"baaa*"`, meaning `b` followed by `aa` followed by zero or more additional `a`s. More complex patterns can also be repeated. So `r"[ab]*"` means "zero or more *a*'s or *b*'s" (not "zero or more right square braces"). This will match strings like *aaaa* or *ababab* or *bbbb*, as well as the empty string. For specifying an integer (a string of digits) we can use `r"[0-9][0-9]*"`. (Why isn't it just `r"[0-9]*"`?)

Kleene +

There is a slightly shorter way to specify "at least one" of some character: the **Kleene +**, which means "one or more occurrences of the immediately preceding character or regular expression". So `r"[0-9]+"` is the normal way to specify "a sequence of digits", and we could also specify the sheep language as `r"baa+!"`.

Besides the Kleene * and Kleene + we can also use explicit numbers as counters, by enclosing them in curly brackets. The operator `r"{3}"` means "exactly 3 occurrences of the previous character or expression". So `r"ax{10}z"` will match `a` followed by exactly 10 `x`'s followed by `z`.

period

An important special character is the **period** (`r"."`), a **wildcard** expression that matches any single character (*except* a newline).

The wildcard is often used together with the Kleene star to mean "any string of characters". For example, suppose we want to find any line in which a particular word, for example, `rose`, appears twice. We can specify this with the regular expression `r"rose.*rose"`, meaning two roses, with a sequence of zero or more characters (of any kind) between them. Fig. 2.11 summarizes.

Regex	Match
*	zero or more occurrences of the previous char or expression
+	one or more occurrences of the previous char or expression
?	zero or one occurrence of the previous char or expression
{n}	exactly *n* occurrences of the previous char or expression
.	any single char
.*	any string of zero or more chars

Figure 2.11 Counting and wildcards.

2.6.3 Anchors and Boundaries

anchors

Anchors are special characters that anchor regular expressions to particular places in a string. The most common anchors are the caret ^ and the dollar sign $. The caret ^ matches the start of a line. The pattern `r"^The"` matches the word `The` only at the start of a line. Thus, the caret ^ has three uses: to match the start of a line, to indicate a negation inside of square brackets, and just to mean a caret. (What are the contexts that allow the system to know which function a given caret is supposed to have?) The dollar sign $ matches the end of a line. So the pattern ␣$ is a useful pattern for matching a space at the end of a line, and `r"^The dog\.$"` matches a line that contains only the phrase *The dog.* with a final period.

Note that we have to use the backslash in the prior example since we want the . to mean "period" and not the wildcard. By contrast, the regular expression `r"^The dog.$"` would match `The dog.` but also `The dog!` and `The dogo`. As we'll discuss below, all the special characters we've defined so far (* + ? . [ ]) need to be backslashed when we mean to use them literally.

There are other anchors: \b matches a word boundary, and \B matches a non word-boundary. Thus, `r"\bthe\b"` matches the word `the` but not the word `other`.

Regex	Match
^	start of line
$	end of line
\b	word boundary
\B	non-word boundary

Figure 2.12 Anchors in regular expressions.

A "word" for the purposes of a regex is defined (based on words in programming languages) as a sequence of digits, underscores, or letters. Thus `r"\b99\b"` will match the string `99` in `There are 99 bottles of beer on the wall` (because 99 follows a space) but not `99` in `There are 299 bottles of beer on the wall` (since 99 follows a number). But it will match `99` in `$99` (since 99 follows a dollar sign ($), which is not a digit, underscore, or letter).

Note that all these anchors and boundary operators technically match the empty string, meaning that they don't eat up any characters of the string. The carat in the pattern `r"^The"` matches the start of `"The"` but doesn't actually advance over the first character `T`. And the pattern `r"the\b the"` matches `the the`; the `\b` is aware of the fact that the space is a boundary, but it matches the empty string right before the space, not the space, so that the space character is available to be matched.

2.6.4 Disjunction, Grouping, and Precedence

Suppose we need to search for texts about pets; perhaps we are particularly interested in cats and dogs. In such a case, we might want to search for either the string *cat* or the string *dog*. Since we can't use the square brackets to search for "cat or dog" (why wouldn't `r"[catdog]"` do the right thing?), we need a new operator, the **disjunction** operator, also called the **pipe** symbol `|`. The pattern `r"cat|dog"` matches either the string `cat` or the string `dog`.

disjunction

Sometimes we need to use this disjunction operator in the midst of a larger sequence. For example, suppose I want to search for mentions of pet fish. How can I specify both *guppy* and *guppies*? We cannot simply say `r"guppy|ies"`, because that would match only the strings `guppy` and `ies`. This is because sequences like `guppy` take **precedence** over the disjunction operator `|`. To make the disjunction operator apply only to a specific pattern, we need to use the parenthesis operators `(` and `)`. Enclosing a pattern in parentheses makes it act like a single character for the purposes of neighboring operators like the pipe `|` and the Kleene*. So the pattern `r"gupp(y|ies)"` would specify that we meant the disjunction only to apply to the suffixes `y` and `ies`.

precedence

The parenthesis operator `(` is also useful when we are using counters like the Kleene*. Unlike the `|` operator, the Kleene* operator applies by default only to a single character, not to a whole sequence. Suppose we want to match repeated instances of a string. Perhaps we have a line that has column labels of the form *Column 1 Column 2 Column 3*. The expression `r"Column␣[0-9]+␣*"` will not match any number of columns; instead, it will match a single column followed by any number of spaces! The star here applies only to the space ␣ that precedes it, not to the whole sequence. With the parentheses, we could write the expression `r"(Column␣[0-9]+␣+)*"` to match the word *Column*, followed by a number and spaces, the whole pattern repeated zero or more times.

This idea that one operator may take precedence over another, requiring us to sometimes use parentheses to specify what we mean, is formalized by the **operator precedence hierarchy** for regular expressions. The following table gives the order

operator precedence

of operator precedence, from highest precedence to lowest precedence.

Parenthesis	`()`
Counters	`* + ? {}`
Sequences and anchors	`the ^my end$`
Disjunction	`|`

Thus, because counters have a higher precedence than sequences, `r"the*"` matches *theeeee* but not *thethe*. Because sequences have a higher precedence than disjunction, `r"the|any"` matches *the* or *any* but not *thany* or *theny*.

Patterns can be ambiguous in another way. Consider the expression `r"[a-z]*"` when matching against the text *once upon a time*. Since `r"[a-z]*"` matches zero or more letters, this expression could match nothing, or just the first letter *o*, *on*, *onc*, or *once*. In these cases regular expressions always match the *largest* string they can; we say that patterns are **greedy**, expanding to cover as much of a string as they can.

greedy

non-greedy

*?

+?

There are, however, ways to enforce **non-greedy** matching, using another meaning of the ? qualifier. The operator ***?** is a Kleene star that matches as little text as possible. The operator **+?** is a Kleene plus that matches as little text as possible.

2.6.5 A Simple Example

Suppose we wanted to write a regex to find cases of the English article *the*. A simple (but incorrect) pattern might be:

```
r"the"
```
(2.14)

One problem is that this pattern will miss the word when it begins a sentence and hence is capitalized (i.e., *The*). This might lead us to the following pattern:

```
r"[tT]he"
```
(2.15)

But we will still overgeneralize, incorrectly return texts with `the` embedded in other words (e.g., *other* or *there*). So we need to specify that we want instances with a word boundary on both sides:

```
r"\b[tT]he\b"
```
(2.16)

The simple process we just went through was based on fixing two kinds of errors: **false positives**, strings that we incorrectly matched like *other* or *there*, and **false negatives**, strings that we incorrectly missed, like *The*. Addressing these two kinds of errors comes up again and again in language processing. Reducing the overall error rate for an application thus involves two antagonistic efforts:

false positives

false negatives

- Increasing **precision** (minimizing false positives)
- Increasing **recall** (minimizing false negatives)

We'll come back to precision and recall with more precise definitions in Chapter 4.

2.6.6 More Operators

Figure 2.13 shows some useful aliases for common ranges:

Finally, certain special characters are referred to by special notation based on the backslash (\) (see Fig. 2.14). The most common of these are the **newline** character `\n` and the **tab** character `\t`.

newline

Regex	Expansion	Match	First Matches
\d	[0-9]	any digit	Party␣of␣5
\D	[^0-9]	any non-digit	Blue␣moon
\w	[a-zA-Z0-9_]	any alphanumeric/underscore	Daiyu
\W	[^\w]	a non-alphanumeric	!!!!
\s	[␣\r\t\n\f]	whitespace (space, tab)	in Concord
\S	[^\s]	Non-whitespace	in␣Concord

Figure 2.13 Aliases for common sets of characters.

How do we refer to characters that are special themselves (like `.`, `*`, `-`, `[`, and `\`) when we mean them literally, not in their special usage? That is, if we are trying to match a period, or a star, or a bracket or paren? To get the literal meaning of a special character, we need to precede them with a backslash, (i.e., `r"\."`, `r"\*"`, `r"\["`, and `r"\\"`).

Regex	Match	First Patterns Matched
*	an asterisk "*"	"K*A*P*L*A*N"
\.	a period "."	"Dr. Livingston, I presume"
\?	a question mark	"Why don't they come and lend a hand?"
\n	a newline	
\t	a tab	

Figure 2.14 Some characters that need to be escaped (via backslash).

2.6.7 Substitutions and Capture Groups

substitution

An important use of regular expressions is in **substitutions**, where we want to replace one string with another. Regular expression can help us specify the string to be replaced as well as the replacement. In Python we use the function `re.sub()` (similar functions exist in other languages and environments).

`re.sub`*(pattern, repl, string)* takes three arguments: a *pattern* to search for, a *replacement* to replace it with, and a *string* in which to do the search and replacing We could for example change every instance of `cherry` to `apricot` in `string`:

```
re.sub(r"cherry", r"apricot", string)
```

Or we could convert to upper case all the instances of a particular name:

```
re.sub(r"janet", r"Janet", string)
```

More often, however, the substitution depends in a more complex way on the string that matched the *pattern*. For example, suppose we have a document in which all the dates are in US format (mm/dd/yyyy) and we want to change them into the format used in the EU and many other regions: (dd-mm-yyyy). The *pattern* `r"\d{2}/\d{2}/\d{4}"` will match a date. But how do we specify in the *replacement* that we want to swap the date and month values?

capture group

The tool in regular expression for this is the **capture group**. A capture group uses parentheses to capture (*store*) the values that we matched in the search, so we can reuse them in the replacement. We put a set of parentheses around the part of the *pattern* we want to capture, and it will get stored in a numbered group (groups are numbered from left to right). Then in the *repl*, we refer back to that group with a number command.

Consider the following expression:

```
re.sub(r"(\d{2})/(\d{2})/(\d{4})", r"\2-\1-\3", string)}
```

We've put parentheses (and) around the two month digits, the two day digits, and the four year digits, thus storing the first 2 digits in group 1, the second 2 digits in group 2, and the final digits in group 3. Then in the *repl* string, we use **number** operators \1, \2, and \3, to refer back to the first, second, and third registers. The result would take a string like

```
The date is 10/15/2011
```

and convert it to

```
The date is 15-10-2011
```

Capture groups can be useful even if we are not doing substitutions. For example we can use them to find repetitions, something we often need in text processing. For example, to find a repeated word in a string, we can use this pattern which searches for a word, captures it in a group, and then refers back to it after whitespace:

```
r"\b([A-Za-z]+)\s+\1\b"
```

Parentheses thus have a double function in regular expressions; they are used to group terms for specifying the order in which operators should apply, and they are used to capture the match. Occasionally we need parentheses for grouping, but don't want to capture the resulting pattern. In that case we use a **non-capturing group**, which is specified by putting the special commands ?: after the open parenthesis, in the form (?: pattern). Non-capture groups are usually used when we are trying to capture only part of a long or complex pattern. Perhaps we are matching a sequence of dates (\d\d/\d\d/\d\d\d\d) separated by spaces and we want to extract only the 15th one. We need to use parenthesis in order to use the counting operator on the first 14, but we don't want to store all the useless information. The following pattern only stores the 15th date in group 1:

non-capturing group

```
r"(?:\d\d/\d\d/\d\d\d\d\s+){14}(\d\d/\d\d/\d\d\d\d)"    (2.17)
```

Substitutions and capture groups are also useful for implementing historically important chatbots like ELIZA (Weizenbaum, 1966). Recall that ELIZA simulates a Rogerian psychologist by carrying on conversations like the following:

$User_2$:	They're always bugging us about something or other.
$ELIZA_2$:	CAN YOU THINK OF A SPECIFIC EXAMPLE
$User_3$:	Well, my boyfriend made me come here.
$ELIZA_3$:	YOUR BOYFRIEND MADE YOU COME HERE
$User_4$:	He says I'm depressed much of the time.
$ELIZA_4$:	I AM SORRY TO HEAR YOU ARE DEPRESSED

ELIZA works by having a series or cascade of regex substitutions each of which matches and changes some part of the input lines. After the input is uppercased, substitutions change all instances of MY to YOUR, and I'M to YOU ARE, and so on. That way when ELIZA repeats back part of the user utterance, it will seem to be referring correctly to the user. The next set of substitutions matches and replaces other patterns in the input, turning the input into a complete response. Here are some examples:

```
re.sub(r".* YOU ARE (DEPRESSED|SAD) .*",r"I AM SORRY TO HEAR YOU ARE \1",input)
re.sub(r".* YOU ARE (DEPRESSED|SAD) .*",r"WHY DO YOU THINK YOU ARE \1",input)
re.sub(r".* ALWAYS .*",r"CAN YOU THINK OF A SPECIFIC EXAMPLE",input)
```

2.6.8 Lookahead Assertions

Finally, there will be times when we need to predict the future: look ahead in the text to see if some pattern matches, but not yet advance the pointer we always keep to where we are in the text, so that we can then deal with the pattern if it occurs, but if it doesn't we can check for something else instead.

lookahead

zero-width

These **lookahead** assertions make use of the `(?` syntax that we saw in the previous section for non-capture groups. The operator `(?= pattern)` is true if `pattern` occurs, but is **zero-width**, i.e. the match pointer doesn't advance, just as we saw with anchors and boundary markers like `\b`. The operator `(?! pattern)` only returns true if a pattern does not match, but again is zero-width and doesn't advance the pointer. Negative lookahead is commonly used when we are parsing some complex pattern but want to rule out a special case. For example suppose we want to capture the first word on the line, but only if it doesn't start with the letter `T`. We can use negative lookahead to do this:

```
r"^(?![tT])(\w+)\b"
```
(2.18)

The first negative lookahead says that the line must not start with a `t` or `T`, but matches the empty string, not moving the match pointer. Then the capture group captures the first word.

2.6.9 Regular Expressions for BPE pre-tokenization

```
>>> import regex as re
>>> pat = re.compile(
... # Contractions: 't and 'm are tokens
...     r"'s|'t|'re|'ve|'m|'ll|'d|"
... # Words:  sequence of Unicode letters (after optional space)
...     r" ?\p{L}+|"
... #Number: sequence of digits (after optional space)
...     r" ?\p{N}+|"
... # Punctuation: sequence of non-alphanumeric/non-space
...                         #(after optional space)
...     r" ?[^\s\p{L}\p{N}]+|"
... # whitespace
...     r"\s+(?!\S)|\s+"
... )
>>> text = "We're 350 dogs! Um, lunch?"
>>> print(pat.findall(text))
['We', "'re", ' 350', ' dogs', '!', ' Um', ',', ' lunch', '?']
>>>
```

Figure 2.15 The GPT-2 pre-tokenizer regular expression, used to split (roughly) on whitespace before running the BPE algorithm.

We described in Section 2.4 how before we run a BPE tokenization algorithm on a corpus we first pre-tokenize, splitting the corpus roughly by whitespace. Then the BPE tokenization algorithm builds up tokens from sequences of characters inside words and doesn't tokenize across word boundaries.

Here's the regular expression used to do this pretokenization that is used for one influential language model, the GPT-2 language model (Radford et al., 2019):

```
r"'s|'t|'re|'ve|'m|'ll|'d| ?\p{L}+| ?\p{N}+| ?[^\s\p{L}\p{N}]+|\s+(?!\S)|\s+"
```

This is quite a complex regular expression, and also makes use of some advanced Unicode-related features we haven't described yet. These features are part of a popular external Python 3 library called `regex` (which is more powerful than the internal Python 3 library called `re`).

For example the `regex` library has special `\p` and `\P` operators that can match if the current character has particular Unicode codepoint properties. For example `\p{L}` matches any Unicode letter, `\P{L}` matches any non-letter, `\p{N}` matches any number, and `\P{N}` matches any non-number.

Fig. 2.15 annotates the regular expression more clearly, and also shows the output of running the GPT-2 tokenizer on the sentence `We're 350 dogs! Um, lunch?`.

Note that the tokenizer splits `We're` into `We` and `'re`, that punctuation is split off from `␣dogs` and `␣lunch`, and that some tokens like `␣dogs` start with a space, and others like `We` and `!` do not.

2.7 Simple Unix Tools for Word Tokenization

For English it is possible to do simple naive word tokenization and frequency computation in a single Unix command-line. As Church (1994) points out, this can be useful when we need quick information about a text corpus. We'll make use of some Unix commands: `tr`, used to systematically change particular characters in the input; `sort`, which sorts input lines in alphabetical order; and `uniq`, which collapses and counts adjacent identical lines.

For example let's begin with the 'complete words' of Shakespeare in one file, `sh.txt`. We can use `tr` to tokenize the words by changing every sequence of non-alphabetic characters to a newline ('A-Za-z' means alphabetic and the -c option complements to non-alphabet, so together they mean to change every non-alphabetic character into a newline. The -s ('squeeze') option is used to replace the result of multiple consecutive changes into a single output, so a series of non-alphabetic characters in a row would all be 'squeezed' into a single newline):

```
tr -sc 'A-Za-z' '\n' < sh.txt
```

The output of this command will be:

```
THE
SONNETS
by
William
Shakespeare
From
fairest
creatures
...
```

Now that there is one word per line, we can sort the lines, and pass them to `uniq -c` which will collapse and count them:

```
tr -sc 'A-Za-z' '\n' < sh.txt | sort | uniq -c
```

with the following output:

```
1945 A
72 AARON
19 ABBESS
25 Aaron
6 Abate
1 Abates
...
```

Alternatively, we can collapse all the upper case to lower case:

```
tr -sc 'A-Za-z' '\n' < sh.txt | tr A-Z a-z | sort | uniq -c
```

whose output is

```
14725 a
  97 aaron
   1 abaissiez
  10 abandon
   2 abandoned
   2 abase
   1 abash
  14 abate
   ...
```

Now we can sort again to find the frequent words. The `-n` option to `sort` means to sort numerically rather than alphabetically, and the `-r` option means to sort in reverse order (highest-to-lowest):

```
tr -sc 'A-Za-z' '\n' < sh.txt | tr A-Z a-z | sort | uniq -c | sort -n -r
```

The results show that the most frequent words in Shakespeare, as in any other corpus, are the short **function words** like articles, pronouns, prepositions:

```
27378 the
26084 and
22538 i
19771 to
17481 of
14725 a
13826 you
   ...
```

Unix tools of this sort can be very handy in building quick word count statistics for any corpus in English. For anything more complex, we generally turn to the more sophisticated tokenization algorithms we've discussed above.

2.8 Rule-based tokenization

While data-based tokenization like BPE is the most common way of doing tokenization, there are also situations where we want to constrain our tokens to be words and not subwords. This might be useful if we are running parsing algorithms for English where the parser might need grammatical words as input. Or it can be useful for any linguistic application where we have some a prior definition of the token that we

are interested in studying. Or it can be useful for social science applications where orthographic words are useful domains of study.

In rule-based tokenization, we pre-define a standard and implement rules to implement that kind of tokenization. Let's explore this for English word tokenization.

We have some desiderata for English. We often want to break off punctuation as a separate token; commas are a useful piece of information for parsers, and periods help indicate sentence boundaries. But we'll often want to keep the punctuation that occurs word internally, in examples like *m.p.h.*, *Ph.D.*, *AT&T*, and *cap'n*. Special characters and numbers will need to be kept in prices ($45.55) and dates (`01/02/06`); we don't want to segment that price into separate tokens of "45" and "55". And there are URLs (`https://www.stanford.edu`), Twitter hashtags (`#nlproc`), or email addresses (`someone@cs.colorado.edu`).

Number expressions introduce complications; in addition to appearing at word boundaries, commas appear inside numbers in English, every three digits: `555,500.50`. Tokenization differs by language; languages like Spanish, French, and German, for example, use a comma to mark the decimal point, and spaces (or sometimes periods) where English puts commas, for example, `555 500,50`.

clitic

A rule-based tokenizer can also be used to expand **clitic** contractions that are marked by apostrophes, converting `what're` to the two tokens `what are`, and `we're` to `we are`. A clitic is a part of a word that can't stand on its own, and can only occur when it is attached to another word. Such contractions occur in other alphabetic languages, including French pronouns (`j'ai` and articles `l'homme`).

Depending on the application, tokenization algorithms may also tokenize multiword expressions like `New York` or `rock 'n' roll` as a single token, which requires a multiword expression dictionary of some sort. Rule-based tokenization is thus intimately tied up with **named entity recognition**, the task of detecting names, dates, and organizations (Chapter 17).

Penn Treebank tokenization

One commonly used tokenization standard is known as the **Penn Treebank tokenization** standard, used for the parsed corpora (treebanks) released by the Linguistic Data Consortium (LDC), the source of many useful datasets. This standard separates out clitics (*doesn't* becomes *does* plus *n't*), keeps hyphenated words together, and separates out all punctuation (to save space we're showing visible spaces '␣' between tokens, although newlines is a more common output):

Input: `"The San Francisco-based restaurant," they said, "doesn't charge $10".`

Output: `"␣The␣San␣Francisco-based␣restaurant␣,␣"␣they␣said␣,␣"␣does␣n't␣charge␣$␣10␣"␣.`

In practice, since tokenization is run before any other language processing, it needs to be very fast. For rule-based word tokenization we generally use deterministic algorithms based on regular expressions compiled into efficient finite state automata. For example, Fig. 2.16 shows a basic regular expression that can be used to tokenize English with the `nltk.regexp_tokenize` function of the Python-based Natural Language Toolkit (NLTK) (Bird et al. 2009; `https://www.nltk.org`).

Carefully designed deterministic algorithms can deal with the ambiguities that arise, such as the fact that the apostrophe needs to be tokenized differently when used as a genitive marker (as in *the book's cover*), a quotative as in *'The other class', she said*, or in clitics like *they're*.

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     (?:[A-Z]\.)+        # abbreviations, e.g. U.S.A.
...   | \w+(?:-\w+)*        # words with optional internal hyphens
...   | \$?\d+(?:\.\d+)?%?  # currency, percentages, e.g. $12.40, 82%
...   | \.\.\.              # ellipsis
...   | [][.,;"'?():_`-]  # these are separate tokens; includes ], [
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

Figure 2.16 A Python trace of regular expression tokenization in the NLTK Python-based natural language processing toolkit (Bird et al., 2009), commented for readability; the `(?x)` verbose flag tells Python to strip comments and whitespace. Figure from Chapter 3 of Bird et al. (2009).

2.8.1 Sentence Segmentation

sentence segmentation

Rule-based segmentation is commonly used for another kind of tokenization process: the sentence. **Sentence segmentation** is a step that is can be optionally applied in text processing. It is especially important when applying NLP algorithms to tasks of detecting structure, like parse structure.

Sentence segmentation depends on the language and the genre. The most useful cues for segmenting a text into sentences in English written text tend to be punctuation, like periods, question marks, and exclamation points. Question marks and exclamation points are relatively unambiguous markers of sentence boundaries, and simple rules can segment sentences when they appear.

The period character ".", on the other hand, is ambiguous between a sentence boundary marker and a marker of abbreviations like *Dr.* or *Inc.* The previous sentence that you just read showed an even more complex case of this ambiguity, in which the final period of *Inc.* marked both an abbreviation and the sentence boundary marker. For this reason, sentence tokenization and word tokenization can be addressed jointly.

Many English sentence tokenization methods work by first deciding (often based on deterministic rules, but sometimes via machine learning) whether a period is part of the word or is a sentence-boundary marker. An abbreviation dictionary can help determine whether the period is part of a commonly used abbreviation; the dictionaries can be hand-built or machine-learned (Kiss and Strunk, 2006), as can the final sentence splitter. In the Stanford CoreNLP toolkit (Manning et al., 2014), for example sentence splitting is rule-based, a deterministic consequence of tokenization; a sentence ends when a sentence-ending punctuation (., !, or ?) is not already grouped with other characters into a token (such as for an abbreviation or number), optionally followed by additional final quotes or brackets.

2.9 Minimum Edit Distance

We often need a way to compare how similar two words or strings are. As we'll see in later chapters, this comes up most commonly in tasks like automatic speech recognition or machine translation, where we want to know how similar the sequence

of words is to some reference sequence of words.

minimum edit distance

Edit distance gives us a way to quantify these intuitions about string similarity. More formally, the **minimum edit distance** between two strings is defined as the minimum number of editing operations (operations like insertion, deletion, substitution) needed to transform one string into another. In this section we'll introduce edit distance for single words, but the algorithm applies equally to entire strings.

The gap between *intention* and *execution*, for example, is 5 (delete an `i`, substitute `e` for `n`, substitute `x` for `t`, insert `c`, substitute `u` for `n`). It's much easier to see this by looking at the most important visualization for string distances, an **alignment** between the two strings, shown in Fig. 2.17. Given two sequences, an **alignment** is a correspondence between substrings of the two sequences. Thus, we say `I` **aligns** with the empty string, `N` with `E`, and so on. Beneath the aligned strings is another representation; a series of symbols expressing an **operation list** for converting the top string into the bottom string: **d** for deletion, **s** for substitution, **i** for insertion.

alignment

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

Figure 2.17 Representing the minimum edit distance between two strings as an **alignment**. The final row gives the operation list for converting the top string into the bottom string: d for deletion, s for substitution, i for insertion.

We can also assign a particular cost or weight to each of these operations. The **Levenshtein** distance between two sequences is the simplest weighting factor in which each of the three operations has a cost of 1 (Levenshtein, 1966)—we assume that the substitution of a letter for itself, for example, `t` for `t`, has zero cost. The Levenshtein distance between *intention* and *execution* is 5. Levenshtein also proposed an alternative version of his metric in which each insertion or deletion has a cost of 1 and substitutions are not allowed. (This is equivalent to allowing substitution, but giving each substitution a cost of 2 since any substitution can be represented by one insertion and one deletion). Using this version, the Levenshtein distance between *intention* and *execution* is 8.

2.9.1 The Minimum Edit Distance Algorithm

How do we find the minimum edit distance? We can think of this as a search task, in which we are searching for the shortest path—a sequence of edits—from one string to another.

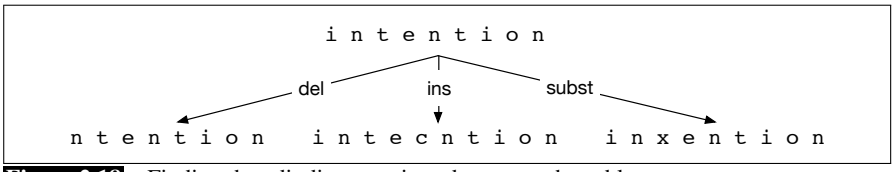


Figure 2.18 Finding the edit distance viewed as a search problem

The space of all possible edits is enormous, so we can't search naively. However, lots of distinct edit paths will end up in the same state (string), so rather than recomputing all those paths, we could just remember the shortest path to a state each time

dynamic programming

we saw it. We can do this by using **dynamic programming**. Dynamic programming is the name for a class of algorithms, first introduced by Bellman (1957), that apply a table-driven method to solve problems by combining solutions to subproblems. Some of the most commonly used algorithms in natural language processing make use of dynamic programming, such as the **Viterbi** algorithm (Chapter 17) and the **CKY** algorithm for parsing (Chapter 18).

The intuition of a dynamic programming problem is that a large problem can be solved by properly combining the solutions to various subproblems. Consider the shortest path of transformed words that represents the minimum edit distance between the strings *intention* and *execution* shown in Fig. 2.19.

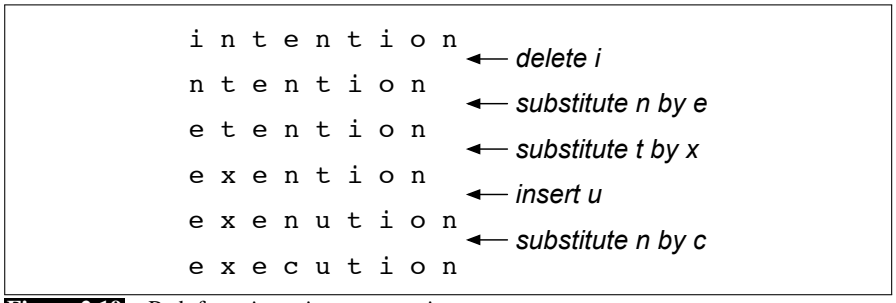


Figure 2.19 Path from *intention* to *execution*.

Imagine some string (perhaps it is *exention*) that is in this optimal path (whatever it is). The intuition of dynamic programming is that if *exention* is in the optimal operation list, then the optimal sequence must also include the optimal path from *intention* to *exention*. Why? If there were a shorter path from *intention* to *exention*, then we could use it instead, resulting in a shorter overall path, and the optimal sequence wouldn't be optimal, thus leading to a contradiction.

minimum edit distance algorithm

The **minimum edit distance algorithm** was named by Wagner and Fischer (1974) but independently discovered by many people (see the Historical Notes section of Chapter 17).

Let's first define the minimum edit distance between two strings. Given two strings, the source string X of length n, and target string Y of length m, we'll define $D[i,j]$ as the edit distance between $X[1..i]$ and $Y[1..j]$, i.e., the first i characters of X and the first j characters of Y. The edit distance between X and Y is thus $D[n,m]$.

We'll use dynamic programming to compute $D[n,m]$ bottom up, combining solutions to subproblems. In the base case, with a source substring of length i but an empty target string, going from i characters to 0 requires i deletes. With a target substring of length j but an empty source going from 0 characters to j characters requires j inserts. Having computed $D[i,j]$ for small i,j we then compute larger $D[i,j]$ based on previously computed smaller values. The value of $D[i,j]$ is computed by taking the minimum of the three possible paths through the matrix which arrive there:

$$D[i,j] = \min \begin{cases} D[i-1,j] + \text{del-cost}(source[i]) \\ D[i,j-1] + \text{ins-cost}(target[j]) \\ D[i-1,j-1] + \text{sub-cost}(source[i], target[j]) \end{cases} \tag{2.19}$$

We mentioned above two versions of Levenshtein distance, one in which substitutions cost 1 and one in which substitutions cost 2 (i.e., are equivalent to an insertion plus a deletion). Let's here use that second version of Levenshtein distance in which the insertions and deletions each have a cost of 1 (ins-cost(·) = del-cost(·) = 1), and

substitutions have a cost of 2 (except substitution of identical letters has zero cost). Under this version of Levenshtein, the computation for $D[i,j]$ becomes:

$$D[i,j] = \min \begin{cases} D[i-1,j]+1 \\ D[i,j-1]+1 \\ D[i-1,j-1]+\begin{cases} 2; & \text{if } source[i] \neq target[j] \\ 0; & \text{if } source[i] = target[j] \end{cases} \end{cases} \tag{2.20}$$

The algorithm is summarized in Fig. 2.20; Fig. 2.21 shows the results of applying the algorithm to the distance between *intention* and *execution* with the version of Levenshtein in Eq. 2.20.

```
function MIN-EDIT-DISTANCE(source, target) returns min-distance

  n ← LENGTH(source)
  m ← LENGTH(target)
  Create a distance matrix D[n+1,m+1]

  # Initialization: the zeroth row and column is the distance from the empty string
  D[0,0] = 0
  for each row i from 1 to n do
    D[i,0] ← D[i-1,0] + del-cost(source[i])
  for each column j from 1 to m do
    D[0,j] ← D[0,j-1] + ins-cost(target[j])

  # Recurrence relation:
  for each row i from 1 to n do
      for each column j from 1 to m do
        D[i,j] ← MIN( D[i−1,j] + del-cost(source[i]),
                      D[i−1,j−1] + sub-cost(source[i], target[j]),
                      D[i,j−1] + ins-cost(target[j]))
  # Termination
  return D[n,m]
```

Figure 2.20 The minimum edit distance algorithm, an example of the class of dynamic programming algorithms. The various costs can either be fixed (e.g., $\forall x, \text{ins-cost}(x) = 1$) or can be specific to the letter (to model the fact that some letters are more likely to be inserted than others). We assume that there is no cost for substituting a letter for itself (i.e., $\text{sub-cost}(x,x) = 0$).

Alignment Knowing the minimum edit distance is useful for algorithms like finding potential spelling error corrections. But the edit distance algorithm is important in another way; with a small change, it can also provide the minimum cost **alignment** between two strings. Aligning two strings is useful throughout speech and language processing. In speech recognition, minimum edit distance alignment is used to compute the word error rate (Chapter 15). Alignment plays a role in machine translation, in which sentences in a parallel corpus (a corpus with a text in two languages) need to be matched to each other.

To extend the edit distance algorithm to produce an alignment, we can start by visualizing an alignment as a path through the edit distance matrix. Figure 2.22 shows this path with boldfaced cells. Each boldfaced cell represents an alignment of a pair of letters in the two strings. If two boldfaced cells occur in the same row,

Src\Tar	#	e	x	e	c	u	t	i	o	n
#	0	1	2	3	4	5	6	7	8	9
i	1	2	3	4	5	6	7	6	7	8
n	2	3	4	5	6	7	8	7	8	7
t	3	4	5	6	7	8	7	8	9	8
e	4	3	4	5	6	7	8	9	10	9
n	5	4	5	6	7	8	9	10	11	10
t	6	5	6	7	8	9	8	9	10	11
i	7	6	7	8	9	10	9	8	9	10
o	8	7	8	9	10	11	10	9	8	9
n	9	8	9	10	11	12	11	10	9	8

Figure 2.21 Computation of minimum edit distance between *intention* and *execution* with the algorithm of Fig. 2.20, using Levenshtein distance with cost of 1 for insertions or deletions, 2 for substitutions.

there will be an insertion in going from the source to the target; two boldfaced cells in the same column indicate a deletion.

Figure 2.22 also shows the intuition of how to compute this alignment path. The computation proceeds in two steps. In the first step, we augment the minimum edit distance algorithm to store backpointers in each cell. The backpointer from a cell points to the previous cell (or cells) that we came from in entering the current cell. We've shown a schematic of these backpointers in Fig. 2.22. Some cells have multiple backpointers because the minimum extension could have come from multiple previous cells. In the second step, we perform a **backtrace**. In a backtrace, we start from the last cell (at the final row and column), and follow the pointers back through the dynamic programming matrix. Each complete path between the final cell and the initial cell is a minimum distance alignment. Exercise 2.7 asks you to modify the minimum edit distance algorithm to store the pointers and compute the backtrace to output an alignment.

backtrace

	#	e	x	e	c	u	t	i	o	n
#	0	← 1	← 2	← 3	← 4	← 5	← 6	← 7	← 8	← 9
i	↑ **1**	↖←↑ 2	↖←↑ 3	↖←↑ 4	↖←↑ 5	↖←↑ 6	↖←↑ 7	↖ 6	← 7	← 8
n	↑ 2	↖←↑ **3**	↖←↑ 4	↖←↑ 5	↖←↑ 6	↖←↑ 7	↖←↑ 8	↑ 7	↖←↑ 8	↖ 7
t	↑ 3	↖←↑ 4	↖←↑ **5**	↖←↑ 6	↖←↑ 7	↖←↑ 8	↖ 7	←↑ 8	↖←↑ 9	↑ 8
e	↑ 4	↖ 3	← 4	↖← **5**	← **6**	← 7	←↑ 8	↖←↑ 9	↖←↑ 10	↑ 9
n	↑ 5	↑ 4	↖←↑ 5	↖←↑ 6	↖←↑ 7	↖←↑ **8**	↖←↑ 9	↖←↑ 10	↖←↑ 11	↖↑ 10
t	↑ 6	↑ 5	↖←↑ 6	↖←↑ 7	↖←↑ 8	↖←↑ 9	↖ **8**	← 9	← 10	←↑ 11
i	↑ 7	↑ 6	↖←↑ 7	↖←↑ 8	↖←↑ 9	↖←↑ 10	↑ 9	↖ **8**	← 9	← 10
o	↑ 8	↑ 7	↖←↑ 8	↖←↑ 9	↖←↑ 10	↖←↑ 11	↑ 10	↑ 9	↖ **8**	← 9
n	↑ 9	↑ 8	↖←↑ 9	↖←↑ 10	↖←↑ 11	↖←↑ 12	↑ 11	↑ 10	↑ 9	↖ **8**

Figure 2.22 When entering a value in each cell, we mark which of the three neighboring cells we came from with up to three arrows. After the table is full we compute an **alignment** (minimum edit path) by using a **backtrace**, starting at the **8** in the lower-right corner and following the arrows back. The sequence of bold cells represents one possible minimum cost alignment between the two strings, again using Levenshtein distance with cost of 1 for insertions or deletions, 2 for substitutions. Diagram design after Gusfield (1997).

While we worked our example with simple Levenshtein distance, the algorithm in Fig. 2.20 allows arbitrary weights on the operations. For spelling correction, for example, substitutions are more likely to happen between letters that are next to

each other on the keyboard. The **Viterbi** algorithm is a probabilistic extension of minimum edit distance. Instead of computing the "minimum edit distance" between two strings, Viterbi computes the "maximum probability alignment" of one string with another. We'll discuss this more in Chapter 17.

2.10 Summary

This chapter introduced the fundamental concepts of tokens and tokenization in language processing. We discussed the linguistic levels of words, morphemes, and characters, introduced Unicode code points and the UTF-8 encoding, introduced the **BPE** algorithm for tokenization, and introduced the **regular expression** and the **minimum edit distance** algorithm for comparing strings. Here's a summary of the main points we covered about these ideas:

- Words and morphemes are useful units of representation, but difficult to define formally.
- **Unicode** is a system for representing characters in the many scripts used to write the languages of the world.
- Each character is represented internally with a unique id called a **code point**, and can be encoded in a file via encoding methods like **UTF-8**, which is a variable-length encoding.
- **Byte-Pair Encoding** or **BPE** is the standard way to induce tokens in a data-driven way. It is the first step in most large language models.
- **BPE** tokens are often roughly word or morpheme-sized, although they can be as small as single characters.
- The **regular expression** language is a powerful tool for pattern-matching.
- Basic operations in regular expressions include **disjunction** of symbols (`[]`, `|`), **counters** (`*`, `+`, and `{n,m}`), **anchors** (`^`, `$`), capture groups (`(,)`), and substitutions.
- The **minimum edit distance** between two strings is the minimum number of operations it takes to edit one into the other. Minimum edit distance can be computed by **dynamic programming**, which also results in an **alignment** of the two strings.

Historical Notes

For more on Herdan's law and Heaps' Law, see Herdan (1960, p. 28), Heaps (1978), Egghe (2007) and Baayen (2001);

Unicode drew on ASCII and ISO character encoding standards. Early drafts were worked out in discussions between engineers from Xerox and Apple. An early draft standard was published in 1988, with a more formal release of the Unicode Stanford in 1991. What became UTF-8 began with ISO drafts in 1989, with various extensions. The self-synchronizing aspects were famously outlined on a placemat in a New Jersey dinner in 1992 by Ken Thompson.

Word tokenization and other text normalization algorithms have been applied since the beginning of the field. This include stemming, like the widely used stemmer of Lovins (1968), and applications to the digital humanities like those of by Packard (1973), who built an affix-stripping morphological parser for Ancient Greek.

BPE, originally a text compression method proposed by Gage (1994), was applied to subword tokenization in the context of early neural machine translation by Sennrich et al. (2016). It was then taken up in OpenAI's GPT-2 (Radford et al., 2019) as the default tokenization method, and also included in the open-source SentencePiece library (Kudo and Richardson, 2018b). There is a nice a public implementation, `minbpe`, https://github.com/karpathy/minbpe, by Andrej Karpathy, who also has a popular lecture introducing BPE (https://www.youtube.com/watch?v=zduSFxRajkE).

Kleene 1951; 1956 first defined regular expressions and the finite automaton, based on the McCulloch-Pitts neuron. Ken Thompson was one of the first to build regular expressions compilers into editors for text searching (Thompson, 1968). His editor *ed* included a command "g/regular expression/p", or Global Regular Expression Print, which later became the Unix `grep` utility.

NLTK is an essential tool that offers both useful Python libraries (https://www.nltk.org) and textbook descriptions (Bird et al., 2009) of many algorithms including text normalization and corpus interfaces.

For more on edit distance, see Gusfield (1997). Our example measuring the edit distance from 'intention' to 'execution' was adapted from Kruskal (1983). There are various publicly available packages to compute edit distance, including Unix `diff` and the NIST `sclite` program (NIST, 2005).

In his autobiography Bellman (1984) explains how he originally came up with the term *dynamic programming*:

> "...The 1950s were not good years for mathematical research. [the] Secretary of Defense ...had a pathological fear and hatred of the word, research... I decided therefore to use the word, "programming". I wanted to get across the idea that this was dynamic, this was multistage... I thought, let's ... take a word that has an absolutely precise meaning, namely dynamic... it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to."

Exercises

2.1 Write regular expressions for the following languages.

1. the set of all alphabetic strings;
2. the set of all lower case alphabetic strings ending in a *b*;
3. the set of all strings from the alphabet a, b such that each a is immediately preceded by and immediately followed by a b;

2.2 Write regular expressions for the following languages. By "word", we mean an alphabetic string separated from other words by whitespace, any relevant punctuation, line breaks, and so forth.

1. the set of all strings with two consecutive repeated words (e.g., "Humbert Humbert" and "the the" but not "the bug" or "the big bug");
2. all strings that start at the beginning of the line with an integer and that end at the end of the line with a word;

3. all strings that have both the word *grotto* and the word *raven* in them (but not, e.g., words like *grottos* that merely *contain* the word *grotto*);
4. write a pattern that places the first word of an English sentence in a register. Deal with punctuation.

2.3 Implement an ELIZA-like program, using substitutions such as those described on page 25. You might want to choose a different domain than a Rogerian psychologist, although keep in mind that you would need a domain in which your program can legitimately engage in a lot of simple repetition.

2.4 Compute the edit distance (using insertion cost 1, deletion cost 1, substitution cost 1) of "leda" to "deal". Show your work (using the edit distance grid).

2.5 Figure out whether *drive* is closer to *brief* or to *divers* and what the edit distance is to each. You may use any version of *distance* that you like.

2.6 Now implement a minimum edit distance algorithm and use your hand-computed results to check your code.

2.7 Augment the minimum edit distance algorithm to output an alignment; you will need to store pointers and add a stage to compute the backtrace.

CHAPTER

3 N-gram Language Models

"You are uniformly charming!" cried he, with a smile of associating and now and then I bowed and they perceived a chaise and four to wish for.
Random sentence generated from a Jane Austen trigram model

Predicting is difficult—especially about the future, as the old quip goes. But how about predicting something that seems much easier, like the next word someone is going to say? What word, for example, is likely to follow

```
The water of Walden Pond is so beautifully ...
```

You might conclude that a likely word is `blue`, or `green`, or `clear`, but probably not `refrigerator` nor `this`. In this chapter we formalize this intuition by introducing n-gram **language models** or **LMs**. A language model is a machine learning model that predicts upcoming words. More formally, a language model assigns a **probability** to each possible next word, or equivalently gives a probability distribution over possible next words. Language models can also assign a probability to an entire sentence. Thus an LM could tell us that the following sequence has a much higher probability of appearing in a text:

language model
LM

```
all of a sudden I notice three guys standing on the sidewalk
```

than does this same set of words in a different order:

```
on guys all I of notice sidewalk three a sudden standing the
```

Why would we want to predict upcoming words? The main reason is that **large language models** are built just by training them to predict words!! As we'll see in chapters 5-9, large language models learn an enormous amount about language solely from being trained to predict upcoming words from neighboring words.

This probabilistic knowledge can be very practical. Consider correcting grammar or spelling errors like `Their are two midterms`, in which `There` was mistyped as `Their`, or `Everything has improve`, in which `improve` should have been `improved`. The phrase `There are` is more probable than `Their are`, and `has improved` than `has improve`, so a language model can help users select the more grammatical variant.

Or for a speech system to recognize that you said `I will be back soonish` and not `I will be bassoon dish`, it helps to know that `back soonish` is a more probable sequence. Language models can also help in **augmentative and alternative communication** (Trnka et al. 2007, Kane et al. 2017). People can use **AAC** systems if they are physically unable to speak or sign but can instead use eye gaze or other movements to select words from a menu. Word prediction can be used to suggest likely words for the menu.

AAC

In this chapter we introduce the simplest kind of language model: the **n-gram**

n-gram

language model. An n-gram is a sequence of n words: a 2-gram (which we'll call **bigram**) is a two-word sequence of words like `The water`, or `water of`, and a 3-gram (a **trigram**) is a three-word sequence of words like `The water of`, or `water of Walden`. But we also (in a bit of terminological ambiguity) use the word 'n-gram' to mean a probabilistic model that can estimate the probability of a word given the n-1 previous words, and thereby also to assign probabilities to entire sequences.

In later chapters we will introduce the much more powerful neural **large language models**, based on the **transformer** architecture of Chapter 8. But because n-grams have a remarkably simple and clear formalization, we use them to introduce some major concepts of large language modeling, including **training and test sets**, **perplexity**, **sampling**, and **interpolation**.

3.1 N-Grams

Let's begin with the task of computing $P(w|h)$, the probability of a word w given some history h. Suppose the history h is "`The water of Walden Pond is so beautifully` " and we want to know the probability that the next word is `blue`:

$$P(\texttt{blue}|\texttt{The water of Walden Pond is so beautifully}) \tag{3.1}$$

One way to estimate this probability is directly from relative frequency counts: take a very large corpus, count the number of times we see `The water of Walden Pond is so beautifully`, and count the number of times this is followed by `blue`. This would be answering the question "Out of the times we saw the history h, how many times was it followed by the word w", as follows:

$$P(\texttt{blue}|\texttt{The water of Walden Pond is so beautifully}) = \frac{C(\texttt{The water of Walden Pond is so beautifully blue})}{C(\texttt{The water of Walden Pond is so beautifully})} \tag{3.2}$$

If we had a large enough corpus, we could compute these two counts and estimate the probability from Eq. 3.2. But even the entire web isn't big enough to give us good estimates for counts of entire sentences. This is because language is **creative**; new sentences are invented all the time, and we can't expect to get accurate counts for such large objects as entire sentences. For this reason, we'll need more clever ways to estimate the probability of a word w given a history h, or the probability of an entire word sequence W.

Let's start with some notation. First, throughout this chapter we'll continue to refer to **words**, although in practice we usually compute language models over **tokens** like the BPE tokens of page 13. To represent the probability of a particular random variable X_i taking on the value "the", or $P(X_i = \text{"the"})$, we will use the simplification $P(the)$. We'll represent a sequence of n words either as $w_1 \dots w_n$ or $w_{1:n}$. Thus the expression $w_{1:n-1}$ means the string $w_1, w_2, ..., w_{n-1}$, but we'll also be using the equivalent notation $w_{<n}$, which can be read as "all the elements of w from w_1 up to and including w_{n-1}". For the joint probability of each word in a sequence having a particular value $P(X_1 = w_1, X_2 = w_2, X_3 = w_3, ..., X_n = w_n)$ we'll use $P(w_1, w_2, ..., w_n)$.

Now, how can we compute probabilities of entire sequences like $P(w_1, w_2, ..., w_n)$? One thing we can do is decompose this probability using the **chain rule of proba-**

bility:

$$P(X_1 ... X_n) = P(X_1)P(X_2|X_1)P(X_3|X_{1:2}) \dots P(X_n|X_{1:n-1}) = \prod_{k=1}^{n} P(X_k|X_{1:k-1}) \tag{3.3}$$

Applying the chain rule to words, we get

$$P(w_{1:n}) = P(w_1)P(w_2|w_1)P(w_3|w_{1:2}) \dots P(w_n|w_{1:n-1}) = \prod_{k=1}^{n} P(w_k|w_{1:k-1}) \tag{3.4}$$

The chain rule shows the link between computing the joint probability of a sequence and computing the conditional probability of a word given previous words. Equation 3.4 suggests that we could estimate the joint probability of an entire sequence of words by multiplying together a number of conditional probabilities. But using the chain rule doesn't really seem to help us! We don't know any way to compute the exact probability of a word given a long sequence of preceding words, $P(w_n|w_{1:n-1})$. As we said above, we can't just estimate by counting the number of times every word occurs following every long string in some corpus, because language is creative and any particular context might have never occurred before!

3.1.1 The Markov assumption

The intuition of the n-gram model is that instead of computing the probability of a word given its entire history, we can **approximate** the history by just the last few words.

bigram

The **bigram** model, for example, approximates the probability of a word given all the previous words $P(w_n|w_{1:n-1})$ by using only the conditional probability given the preceding word $P(w_n|w_{n-1})$. In other words, instead of computing the probability

$$P(\texttt{blue}|\texttt{The water of Walden Pond is so beautifully}) \tag{3.5}$$

we approximate it with the probability

$$P(\texttt{blue}|\texttt{beautifully}) \tag{3.6}$$

When we use a bigram model to predict the conditional probability of the next word, we are thus making the following approximation:

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-1}) \tag{3.7}$$

Markov

The assumption that the probability of a word depends only on the previous word is called a **Markov** assumption. Markov models are the class of probabilistic models that assume we can predict the probability of some future unit without looking too far into the past. We can generalize the bigram (which looks one word into the past)

n-gram

to the trigram (which looks two words into the past) and thus to the **n-gram** (which looks $n-1$ words into the past).

Let's see a general equation for this n-gram approximation to the conditional probability of the next word in a sequence. We'll use N here to mean the n-gram

size, so $N = 2$ means bigrams and $N = 3$ means trigrams. Then we approximate the probability of a word given its entire context as follows:

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-N+1:n-1}) \tag{3.8}$$

Given the bigram assumption for the probability of an individual word, we can compute the probability of a complete word sequence by substituting Eq. 3.7 into Eq. 3.4:

$$P(w_{1:n}) \approx \prod_{k=1}^{n} P(w_k|w_{k-1}) \tag{3.9}$$

3.1.2 How to estimate probabilities

maximum likelihood estimation

normalize

How do we estimate these bigram or n-gram probabilities? An intuitive way to estimate probabilities is called **maximum likelihood estimation** or **MLE**. We get the MLE estimate for the parameters of an n-gram model by getting counts from a corpus, and **normalizing** the counts so that they lie between 0 and 1. For probabilistic models, normalizing means dividing by some total count so that the resulting probabilities fall between 0 and 1 and sum to 1.

For example, to compute a particular bigram probability of a word w_n given a previous word w_{n-1}, we'll compute the count of the bigram $C(w_{n-1}w_n)$ and normalize by the sum of all the bigrams that share the same first word w_{n-1}:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1}w)} \tag{3.10}$$

We can simplify this equation, since the sum of all bigram counts that start with a given word w_{n-1} must be equal to the unigram count for that word w_{n-1} (the reader should take a moment to be convinced of this):

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})} \tag{3.11}$$

Let's work through an example using a mini-corpus of three sentences. We'll first need to augment each sentence with a special symbol <s> at the beginning of the sentence, to give us the bigram context of the first word. We'll also need a special end-symbol </s>.[1]

```
<s> I am Sam </s>
<s> Sam I am </s>
<s> I do not like green eggs and ham </s>
```

Here are the calculations for some of the bigram probabilities from this corpus

$P(\texttt{I}|\texttt{<s>}) = \frac{2}{3} = 0.67$ $P(\texttt{Sam}|\texttt{<s>}) = \frac{1}{3} = 0.33$ $P(\texttt{am}|\texttt{I}) = \frac{2}{3} = 0.67$

$P(\texttt{</s>}|\texttt{Sam}) = \frac{1}{2} = 0.5$ $P(\texttt{Sam}|\texttt{am}) = \frac{1}{2} = 0.5$ $P(\texttt{do}|\texttt{I}) = \frac{1}{3} = 0.33$

For the general case of MLE n-gram parameter estimation:

$$P(w_n|w_{n-N+1:n-1}) = \frac{C(w_{n-N+1:n-1}\ w_n)}{C(w_{n-N+1:n-1})} \tag{3.12}$$

[1] We need the end-symbol to make the bigram grammar a true probability distribution. Without an end-symbol, instead of the sentence probabilities of all sentences summing to one, the sentence probabilities for all sentences *of a given length* would sum to one. This model would define an infinite set of probability distributions, with one distribution per sentence length. See Exercise 3.5.

relative frequency

Equation 3.12 (like Eq. 3.11) estimates the n-gram probability by dividing the observed frequency of a particular sequence by the observed frequency of a prefix. This ratio is called a **relative frequency**. We said above that this use of relative frequencies as a way to estimate probabilities is an example of maximum likelihood estimation or MLE. In MLE, the resulting parameter set maximizes the likelihood of the training set T given the model M (i.e., $P(T|M)$). For example, suppose the word *Chinese* occurs 400 times in a corpus of a million words. What is the probability that a random word selected from some other text of, say, a million words will be the word *Chinese*? The MLE of its probability is $\frac{400}{1000000}$ or 0.0004. Now 0.0004 is not the best possible estimate of the probability of *Chinese* occurring in all situations; it might turn out that in some other corpus or context *Chinese* is a very unlikely word. But it is the probability that makes it *most likely* that Chinese will occur 400 times in a million-word corpus. We present ways to modify the MLE estimates slightly to get better probability estimates in Section 3.6.

Let's move on to some examples from a real but tiny corpus, drawn from the now-defunct Berkeley Restaurant Project, a dialogue system from the last century that answered questions about a database of restaurants in Berkeley, California (Jurafsky et al., 1994). Here are some sample user queries (text-normalized, by lower casing and with punctuation striped) (a sample of 9332 sentences is on the website):

can you tell me about any good cantonese restaurants close by
tell me about chez panisse
i'm looking for a good place to eat breakfast
when is caffe venezia open during the day

Figure 3.1 shows the bigram counts from part of a bigram grammar from text-normalized Berkeley Restaurant Project sentences. Note that the majority of the values are zero. In fact, we have chosen the sample words to cohere with each other; a matrix selected from a random set of eight words would be even more sparse.

	i	**want**	**to**	**eat**	**chinese**	**food**	**lunch**	**spend**
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

Figure 3.1 Bigram counts for eight of the words (out of $V = 1446$) in the Berkeley Restaurant Project corpus of 9332 sentences. Zero counts are in gray. Each cell shows the count of the column label word following the row label word. Thus the cell in row **i** and column **want** means that **want** followed **i** 827 times in the corpus.

Figure 3.2 shows the bigram probabilities after normalization (dividing each cell in Fig. 3.1 by the appropriate unigram for its row, taken from the following set of unigram counts):

i	**want**	**to**	**eat**	**chinese**	**food**	**lunch**	**spend**
2533	927	2417	746	158	1093	341	278

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

Figure 3.2 Bigram probabilities for eight words in the Berkeley Restaurant Project corpus of 9332 sentences. Zero probabilities are in gray.

Here are a few other useful probabilities:

$$P(\texttt{i}|\texttt{<s>}) = 0.25 \qquad P(\texttt{english}|\texttt{want}) = 0.0011$$
$$P(\texttt{food}|\texttt{english}) = 0.5 \qquad P(\texttt{</s>}|\texttt{food}) = 0.68$$

Now we can compute the probability of sentences like *I want English food* or *I want Chinese food* by simply multiplying the appropriate bigram probabilities together, as follows:

$$\begin{aligned} P(\texttt{<s> i want english food </s>}) &= P(\texttt{i}|\texttt{<s>})P(\texttt{want}|\texttt{i})P(\texttt{english}|\texttt{want}) \\ &\quad P(\texttt{food}|\texttt{english})P(\texttt{</s>}|\texttt{food}) \\ &= 0.25 \times 0.33 \times 0.0011 \times 0.5 \times 0.68 \\ &= 0.000031 \end{aligned}$$

We leave it as Exercise 3.2 to compute the probability of *i want chinese food*.

What kinds of linguistic phenomena are captured in these bigram statistics? Some of the bigram probabilities above encode some facts that we think of as strictly **syntactic** in nature, like the fact that what comes after *eat* is usually a noun or an adjective, or that what comes after *to* is usually a verb. Others might be a fact about the personal assistant task, like the high probability of sentences beginning with the words *I*. And some might even be cultural rather than linguistic, like the higher probability that people are looking for Chinese versus English food.

3.1.3 Dealing with scale in large n-gram models

In practice, language models can be very large, leading to practical issues.

log probabilities

Log probabilities Language model probabilities are always stored and computed in log space as **log probabilities**. This is because probabilities are (by definition) less than or equal to 1, and so the more probabilities we multiply together, the smaller the product becomes. Multiplying enough n-grams together would result in numerical underflow. Adding in log space is equivalent to multiplying in linear space, so we combine log probabilities by adding them. By adding log probabilities instead of multiplying probabilities, we get results that are not as small. We do all computation and storage in log space, and just convert back into probabilities if we need to report probabilities at the end by taking the exp of the logprob:

$$p_1 \times p_2 \times p_3 \times p_4 = \exp(\log p_1 + \log p_2 + \log p_3 + \log p_4) \tag{3.13}$$

In practice throughout this book, we'll use log to mean natural log (ln) when the base is not specified.

trigram
4-gram
5-gram

Longer context Although for pedagogical purposes we have only described bigram models, when there is sufficient training data we use **trigram** models, which condition on the previous two words, or **4-gram** or **5-gram** models. For these larger n-grams, we'll need to assume extra contexts to the left and right of the sentence end. For example, to compute trigram probabilities at the very beginning of the sentence, we use two pseudo-words for the first trigram (i.e., $P(\texttt{I}|\texttt{<s><s>})$).

Some large n-gram datasets have been created, like the million most frequent n-grams drawn from the Corpus of Contemporary American English (COCA), a curated 1 billion word corpus of American English (Davies, 2020), Google's Web 5-gram corpus from 1 trillion words of English web text (Franz and Brants, 2006), or the Google Books Ngrams corpora (800 billion tokens from Chinese, English, French, German, Hebrew, Italian, Russian, and Spanish) (Lin et al., 2012a)).

It's even possible to use extremely long-range n-gram context. The infini-gram (∞-gram) project (Liu et al., 2024) allows n-grams of any length. Their idea is to avoid the expensive (in space and time) pre-computation of huge n-gram count tables. Instead, n-gram probabilities with arbitrary n are computed quickly at inference time by using an efficient representation called suffix arrays. This allows computing of n-grams of every length for enormous corpora of 5 trillion tokens.

Efficiency considerations are important when building large n-gram language models. It is standard to quantize the probabilities using only 4-8 bits (instead of 8-byte floats), store the word strings on disk and represent them in memory only as a 64-bit hash, and represent n-grams in special data structures like 'reverse tries'. It is also common to prune n-gram language models, for example by only keeping n-grams with counts greater than some threshold or using entropy to prune less-important n-grams (Stolcke, 1998). Efficient language model toolkits like KenLM (Heafield 2011, Heafield et al. 2013) use sorted arrays and use merge sorts to efficiently build the probability tables in a minimal number of passes through a large corpus.

3.2 Evaluating Language Models: Training and Test Sets

extrinsic evaluation

The best way to evaluate the performance of a language model is to embed it in an application and measure how much the application improves. Such end-to-end evaluation is called **extrinsic evaluation**. Extrinsic evaluation is the only way to know if a particular improvement in the language model (or any component) is really going to help the task at hand. Thus for evaluating n-gram language models that are a component of some task like speech recognition or machine translation, we can compare the performance of two candidate language models by running the speech recognizer or machine translator twice, once with each language model, and seeing which gives the more accurate transcription.

intrinsic evaluation

Unfortunately, running big NLP systems end-to-end is often very expensive. Instead, it's helpful to have a metric that can be used to quickly evaluate potential improvements in a language model. An **intrinsic evaluation** metric is one that measures the quality of a model independent of any application. In the next section we'll introduce **perplexity**, which is the standard intrinsic metric for measuring language model performance, both for simple n-gram language models and for the more sophisticated neural large language models of Chapter 8.

training set
development set
test set

In order to evaluate any machine learning model, we need to have at least three distinct datasets: the **training set**, the **development set**, and the **test set**.

The **training set** is the data we use to learn the parameters of our model; for simple n-gram language models it's the corpus from which we get the counts that we normalize into the probabilities of the n-gram language model.

The **test set** is a different, held-out set of data, not overlapping with the training set, that we use to evaluate the model. We need a separate test set to give us an unbiased estimate of how well the model we trained can generalize when we apply it to some new unknown dataset. A machine learning model that perfectly captured the training data, but performed terribly on any other data, wouldn't be much use when it comes time to apply it to any new data or problem! We thus measure the quality of an n-gram model by its performance on this unseen test set or test corpus.

How should we choose a training and test set? The test set should reflect the language we want to use the model for. If we're going to use our language model for speech recognition of chemistry lectures, the test set should be text of chemistry lectures. If we're going to use it as part of a system for translating hotel booking requests from Chinese to English, the test set should be text of hotel booking requests. If we want our language model to be general purpose, then the test set should be drawn from a wide variety of texts. In such cases we might collect a lot of texts from different sources, and then divide it up into a training set and a test set. It's important to do the dividing carefully; if we're building a general purpose model, we don't want the test set to consist of only text from one document, or one author, since that wouldn't be a good measure of general performance.

Thus if we are given a corpus of text and want to compare the performance of two different n-gram models, we divide the data into training and test sets, and train the parameters of both models on the training set. We can then compare how well the two trained models fit the test set.

But what does it mean to "fit the test set"? The standard answer is simple: whichever language model assigns a **higher probability** to the test set—which means it more accurately predicts the test set—is a better model. Given two probabilistic models, the better model is the one that better predicts the details of the test data, and hence will assign a higher probability to the test data.

Since our evaluation metric is based on test set probability, it's important not to let the test sentences into the training set. Suppose we are trying to compute the probability of a particular "test" sentence. If our test sentence is part of the training corpus, we will mistakenly assign it an artificially high probability when it occurs in the test set. We call this situation **training on the test set** or also **data contamination**. Training on the test set introduces a bias that makes the probabilities all look too high, and causes huge inaccuracies in **perplexity**, the probability-based metric we introduce below.

data contamination

Even if we don't train on the test set, if we test our language model on the test set many times after making different changes, we might implicitly tune to its characteristics, by noticing which changes seem to make the model better. For this reason, we only want to run our model on the test set once, or a very few number of times, once we are sure our model is ready.

development test

For this reason we normally instead have a third dataset called a **development** test set or, **devset**. We do all our testing on this dataset until the very end, and then we test on the test set once to see how good our model is.

How do we divide our data into training, development, and test sets? We want our test set to be as large as possible, since a small test set may be accidentally unrepresentative, but we also want as much training data as possible. At the minimum, we would want to pick the smallest test set that gives us enough statistical power

to measure a statistically significant difference between two potential models. It's important that the devset be drawn from the same kind of text as the test set, since its goal is to measure how we would do on the test set.

3.3 Evaluating Language Models: Perplexity

We said above that we evaluate language models based on which one assigns a higher probability to the test set. A better model is better at predicting upcoming words, and so it will be less surprised by (i.e., assign a higher probability to) each word when it occurs in the test set. Indeed, a perfect language model would correctly guess each next word in a corpus, assigning it a probability of 1, and all the other words a probability of zero. So given a test corpus, a better language model will assign a higher probability to it than a worse language model.

But in fact, we often do not use raw probability as our metric for evaluating language models. The reason is that the probability of a test set (or any sequence) depends on the number of words or tokens in it; the probability of a test set gets smaller the longer the text. It's useful to have a metric that is per-word, normalized by length, so we could compare across texts of different lengths. There is a such a metric! It's a function of probability called **perplexity**, and it is used for evaluating large language models as well as n-gram models.

perplexity

The **perplexity** (sometimes abbreviated as PP or PPL) of a language model on a test set is the inverse probability of the test set (one over the probability of the test set), normalized by the number of words (or tokens). For this reason it's sometimes called the per-word or per-token perplexity. We normalize by the number of words N by taking the Nth root. For a test set $W = w_1 w_2 \dots w_N$,:

$$\begin{aligned} \text{perplexity}(W) &= P(w_1 w_2 \dots w_N)^{-\frac{1}{N}} \\ &= \sqrt[N]{\frac{1}{P(w_1 w_2 \dots w_N)}} \end{aligned} \tag{3.14}$$

Or we can use the chain rule to expand the probability of W:

$$\text{perplexity}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1 \dots w_{i-1})}} \tag{3.15}$$

Note that because of the inverse in Eq. 3.15, the higher the probability of the word sequence, the lower the perplexity. Thus the **the lower the perplexity of a model on the data, the better the model**. Minimizing perplexity is equivalent to maximizing the test set probability according to the language model. Why does perplexity use the inverse probability? It turns out the inverse arises from the original definition of perplexity from cross-entropy rate in information theory; for those interested, the explanation is in the advanced Section 3.7. Meanwhile, we just have to remember that perplexity has an inverse relationship with probability.

The details of computing the perplexity of a test set W depends on which language model we use. Here's the perplexity of W with a unigram language model (just the geometric mean of the inverse of the unigram probabilities):

$$\text{perplexity}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i)}} \tag{3.16}$$

The perplexity of W computed with a bigram language model is still a geometric mean, but now of the inverse of the bigram probabilities:

$$\text{perplexity}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})}} \tag{3.17}$$

What we generally use for word sequence in Eq. 3.15 or Eq. 3.17 is the entire sequence of words in some test set. Since this sequence will cross many sentence boundaries, if our vocabulary includes a between-sentence token <EOS> or separate begin- and end-sentence markers <s> and </s> then we can include them in the probability computation. If we do, then we also include one token per sentence in the total count of word tokens N.[2]

We mentioned above that perplexity is a function of both the text and the language model: given a text W, different language models will have different perplexities. Because of this, perplexity can be used to compare different language models. For example, here we trained unigram, bigram, and trigram models on 38 million words from the *Wall Street Journal* newspaper. We then computed the perplexity of each of these models on a WSJ test set using Eq. 3.16 for unigrams, Eq. 3.17 for bigrams, and the corresponding equation for trigrams. The table below shows the perplexity of the 1.5 million word test set according to each of the language models.

	Unigram	**Bigram**	**Trigram**
Perplexity	962	170	109

As we see above, the more information the n-gram gives us about the word sequence, the higher the probability the n-gram will assign to the string. A trigram model is less surprised than a unigram model because it has a better idea of what words might come next, and so it assigns them a higher probability. And the higher the probability, the lower the perplexity (since as Eq. 3.15 showed, perplexity is related inversely to the probability of the test sequence according to the model). So a lower perplexity tells us that a language model is a better predictor of the test set.

Note that in computing perplexities, the language model must be constructed without any knowledge of the test set, or else the perplexity will be artificially low. And the perplexity of two language models is only comparable if they use identical vocabularies.

An (intrinsic) improvement in perplexity does not guarantee an (extrinsic) improvement in the performance of a language processing task like speech recognition or machine translation. Nonetheless, because perplexity usually correlates with task improvements, it is commonly used as a convenient evaluation metric. Still, when possible a model's improvement in perplexity should be confirmed by an end-to-end evaluation on a real task.

3.3.1 Perplexity as Weighted Average Branching Factor

It turns out that perplexity can also be thought of as the **weighted average branching factor** of a language. The branching factor of a language is the number of possible next words that can follow any word. For example consider a mini artificial

[2] For example if we use both begin and end tokens, we would include the end-of-sentence marker </s> but not the beginning-of-sentence marker <s> in our count of N; This is because the end-sentence token is followed directly by the begin-sentence token with probability almost 1, so we don't want the probability of that fake transition to influence our perplexity.

language that is deterministic (no probabilities), any word can follow any word, and whose vocabulary consists of only three colors:

$$L = \{\texttt{red}, \texttt{blue}, \texttt{green}\} \tag{3.18}$$

The branching factor of this language is 3.

Now let's make a probabilistic version of the same LM, let's call it A, where each word follows each other with equal probability $\frac{1}{3}$ (it was trained on a training set with equal counts for the 3 colors), and a test set T = "`red red red red blue`".

Let's first convince ourselves that if we compute the perplexity of this artificial color language on this test set (or any such test set) we indeed get 3. By Eq. 3.15, the perplexity of A on T is:

$$\begin{aligned} \text{perplexity}_A(T) &= P_A(\texttt{red red red red blue})^{-\frac{1}{5}} \\ &= \left(\left(\frac{1}{3}\right)^5\right)^{-\frac{1}{5}} \\ &= \left(\frac{1}{3}\right)^{-1} = 3 \end{aligned} \tag{3.19}$$

But now suppose `red` was very likely in the training set of a different LM B, and so B has the following probabilities:

$$P(\texttt{red}) = 0.8 \quad P(\texttt{green}) = 0.1 \quad P(\texttt{blue}) = 0.1 \tag{3.20}$$

We should expect the perplexity of the same test set `red red red red blue` for language model B to be lower since most of the time the next color will be red, which is very predictable, i.e. has a high probability. So the probability of the test set will be higher, and since perplexity is inversely related to probability, the perplexity will be lower. Thus, although the branching factor is still 3, the perplexity or *weighted* branching factor is smaller:

$$\begin{aligned} \text{perplexity}_B(T) &= P_B(\texttt{red red red red blue})^{-1/5} \\ &= 0.04096^{-\frac{1}{5}} \\ &= 0.527^{-1} = 1.89 \end{aligned} \tag{3.21}$$

3.4 Sampling sentences from a language model

sampling

One important way to visualize what kind of knowledge a language model embodies is to sample from it. **Sampling** from a distribution means to choose random points according to their likelihood. Thus sampling from a language model—which represents a distribution over sentences—means to generate some sentences, choosing each sentence according to its likelihood as defined by the model. Thus we are more likely to generate sentences that the model thinks have a high probability and less likely to generate sentences that the model thinks have a low probability.

This technique of visualizing a language model by sampling was first suggested very early on by Shannon (1948) and Miller and Selfridge (1950). It's simplest to visualize how this works for the unigram case. Imagine all the words of the English language covering the number line between 0 and 1, each word covering an interval

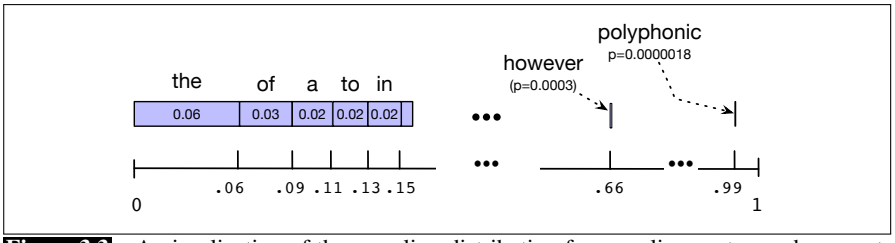


Figure 3.3 A visualization of the sampling distribution for sampling sentences by repeatedly sampling unigrams. The blue bar represents the relative frequency of each word (we've ordered them from most frequent to least frequent, but the choice of order is arbitrary). The number line shows the cumulative probabilities. If we choose a random number between 0 and 1, it will fall in an interval corresponding to some word. The expectation for the random number to fall in the larger intervals of one of the frequent words (*the*, *of*, *a*) is much higher than in the smaller interval of one of the rare words (*polyphonic*).

proportional to its frequency. Fig. 3.3 shows a visualization, using a unigram LM computed from the text of this book. We choose a random value between 0 and 1, find that point on the probability line, and print the word whose interval includes this chosen value. We continue choosing random numbers and generating words until we randomly generate the sentence-final token </s>.

We can use the same technique to generate bigrams by first generating a random bigram that starts with <s> (according to its bigram probability). Let's say the second word of that bigram is w. We next choose a random bigram starting with w (again, drawn according to its bigram probability), and so on.

3.5 Generalizing vs. overfitting the training set

The n-gram model, like many statistical models, is dependent on the training corpus. One implication of this is that the probabilities often encode specific facts about a given training corpus. Another implication is that n-grams do a better and better job of modeling the training corpus as we increase the value of n.

We can use the sampling method from the prior section to visualize both of these facts! To give an intuition for the increasing power of higher-order n-grams, Fig. 3.4 shows random sentences generated from unigram, bigram, trigram, and 4-gram models trained on Shakespeare's works.

The longer the context, the more coherent the sentences. The unigram sentences show no coherent relation between words nor any sentence-final punctuation. The bigram sentences have some local word-to-word coherence (especially considering punctuation as words). The trigram sentences are beginning to look a lot like Shakespeare. Indeed, the 4-gram sentences look a little too much like Shakespeare. The words *It cannot be but so* are directly from *King John*. This is because, not to put the knock on Shakespeare, his oeuvre is not very large as corpora go ($N = 884,647, V = 29,066$), and our n-gram probability matrices are ridiculously sparse. There are $V^2 = 844,000,000$ possible bigrams alone, and the number of possible 4-grams is $V^4 = 7 \times 10^{17}$. Thus, once the generator has chosen the first 3-gram (*It cannot be*), there are only seven possible next words for the 4th element (*but*, *I*, *that*, *thus*, *this*, and the period).

To get an idea of the dependence on the training set, let's look at LMs trained on a completely different corpus: the *Wall Street Journal* (WSJ) newspaper. Shakespeare

1 gram	–To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have
	–Hill he late speaks; or! a more to leg less first you enter
2 gram	–Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow.
	–What means, sir. I confess she? then all sorts, he is trim, captain.
3 gram	–Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done.
	–This shall forbid it should be branded, if renown made it empty.
4 gram	–King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in;
	–It cannot be but so.

Figure 3.4 Eight sentences randomly generated from four n-gram models computed from Shakespeare's works. All characters were mapped to lower-case and punctuation marks were treated as words. Output is hand-corrected for capitalization to improve readability.

and the WSJ are both English, so we might have expected some overlap between our n-grams for the two genres. Fig. 3.5 shows sentences generated by unigram, bigram, and trigram models trained on 40 million words from WSJ.

1 gram	Months the my and issue of year foreign new exchange's september were recession exchange new endorsed a acquire to six executives
2 gram	Last December through the way to preserve the Hudson corporation N. B. E. C. Taylor would seem to complete the major central planners one point five percent of U. S. E. has already old M. X. corporation of living on information such as more frequently fishing to keep her
3 gram	They also point to ninety nine point six billion dollars from two hundred four oh six three percent of the rates of interest stores as Mexico and Brazil on market conditions

Figure 3.5 Three sentences randomly generated from three n-gram models computed from 40 million words of the *Wall Street Journal*, lower-casing all characters and treating punctuation as words. Output was then hand-corrected for capitalization to improve readability.

Compare these examples to the pseudo-Shakespeare in Fig. 3.4. While they both model "English-like sentences", there is no overlap in the generated sentences, and little overlap even in small phrases. Statistical models are pretty useless as predictors if the training sets and the test sets are as different as Shakespeare and the WSJ.

How should we deal with this problem when we build n-gram models? One step is to be sure to use a training corpus that has a similar **genre** to whatever task we are trying to accomplish. To build a language model for translating legal documents, we need a training corpus of legal documents. To build a language model for a question-answering system, we need a training corpus of questions.

It is equally important to get training data in the appropriate **dialect** or **variety**, especially when processing social media posts or spoken transcripts. For example some tweets will use features of African American English (AAE)— the name for the many variations of language used in African American communities (King, 2020). Such features can include words like *finna*—an auxiliary verb that marks immediate future tense —that don't occur in other varieties, or spellings like *den* for *then*, in tweets like this one (Blodgett and O'Connor, 2017):

(3.22) Bored af den my phone finna die!!!

while tweets from English-based languages like Nigerian Pidgin have markedly different vocabulary and n-gram patterns from American English (Jurgens et al., 2017):

(3.23) @username R u a wizard or wat gan sef: in d mornin - u tweet, afternoon - u tweet, nyt gan u dey tweet. beta get ur IT placement wiv twitter

Is it possible for the testset nonetheless to have a word we have never seen before? What happens if the word *Jurafsky* never occurs in our training set, but pops up in the test set? The answer is that although words might be unseen, we normally run our NLP algorithms not on words but on **subword tokens**. With subword tokenization (like the BPE algorithm of Chapter 2) any word can be modeled as a sequence of known smaller subwords, if necessary by a sequence of tokens corresponding to individual letters. So although for convenience we've been referring to words in this chapter, the language model vocabulary is normally the set of tokens rather than words, and in this way the test set can never contain unseen tokens.

3.6 Smoothing, Interpolation, and Backoff

There is a problem with using maximum likelihood estimates for probabilities: any finite training corpus will be missing some perfectly acceptable English word sequences. That is, cases where a particular n-gram never occurs in the training data but appears in the test set. Perhaps our training corpus has the words `ruby` and `slippers` in it but just happens not to have the phrase `ruby slippers`.

zeros

These unseen sequences or **zeros**—sequences that don't occur in the training set but do occur in the test set—are a problem for two reasons. First, their presence means we are underestimating the probability of word sequences that might occur, which hurts the performance of any application we want to run on this data. Second, if the probability of any word in the test set is 0, the probability of the whole test set is 0. Perplexity is defined based on the inverse probability of the test set. Thus if some words in context have zero probability, we can't compute perplexity at all, since we can't divide by zero!

smoothing **discounting**

The standard way to deal with putative "zero probability n-grams" that should really have some non-zero probability is called **smoothing** or **discounting**. Smoothing algorithms shave off a bit of probability mass from some more frequent events and give it to unseen events. Here we'll introduce some simple smoothing algorithms: **Laplace (add-one) smoothing**, **stupid backoff**, and n-gram **interpolation**.

3.6.1 Laplace Smoothing

Laplace smoothing

The simplest way to do smoothing is to add one to all the n-gram counts, before we normalize them into probabilities. All the counts that used to be zero will now have a count of 1, the counts of 1 will be 2, and so on. This algorithm is called **Laplace smoothing**. Laplace smoothing does not perform well enough to be used in modern n-gram models, but it usefully introduces many of the concepts that we see in other smoothing algorithms, gives a useful baseline, and is also a practical smoothing algorithm for other tasks like **text classification** (Appendix K).

Let's start with the application of Laplace smoothing to unigram probabilities. Recall that the unsmoothed maximum likelihood estimate of the unigram probability

of the word w_i is its count c_i normalized by the total number of word tokens N:

$$P(w_i) = \frac{c_i}{N}$$

add-one

Laplace smoothing merely adds one to each count (hence its alternate name **add-one** smoothing). Since there are V words in the vocabulary and each one was incremented, we also need to adjust the denominator to take into account the extra V observations. (What happens to our P values if we don't increase the denominator?)

$$P_{\text{Laplace}}(w_i) = \frac{c_i + 1}{N + V} \tag{3.24}$$

Now that we have the intuition for the unigram case, let's smooth our Berkeley Restaurant Project bigrams. Figure 3.6 shows the add-one smoothed counts for the bigrams in Fig. 3.1.

	i	want	to	eat	chinese	food	lunch	spend
i	6	828	1	10	1	1	1	3
want	3	1	609	2	7	7	6	2
to	3	1	5	687	3	1	7	212
eat	1	1	3	1	17	3	43	1
chinese	2	1	1	1	1	83	2	1
food	16	1	16	1	2	5	1	1
lunch	3	1	1	1	1	2	1	1
spend	2	1	2	1	1	1	1	1

Figure 3.6 Add-one smoothed bigram counts for eight of the words (out of $V = 1446$) in the Berkeley Restaurant Project corpus of 9332 sentences. Previously-zero counts are in gray.

Figure 3.7 shows the add-one smoothed probabilities for the bigrams in Fig. 3.2, computed by Eq. 3.26 below. Recall that normal bigram probabilities are computed by normalizing each row of counts by the unigram count:

$$P_{\text{MLE}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})} \tag{3.25}$$

For add-one smoothed bigram counts, we need to augment the unigram count in the denominator by the number of total word types in the vocabulary V. We can see why this is in the following equation, which makes it explicit that the unigram count in the denominator is really the sum over all the bigrams that start with w_{n-1}. Since we add one to each of these, and there are V of them, we add a total of V to the denominator:

$$P_{\text{Laplace}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n) + 1}{\sum_w (C(w_{n-1}w) + 1)} = \frac{C(w_{n-1}w_n) + 1}{C(w_{n-1}) + V} \tag{3.26}$$

Thus, each of the unigram counts given on page 42 will need to be augmented by $V = 1446$. The result, using Eq. 3.26, is the smoothed bigram probabilities in Fig. 3.7.

One useful visualization technique is to reconstruct an **adjusted count matrix** so we can see how much a smoothing algorithm has changed the original counts. This adjusted count C^* is the count that, if divided by $C(w_{n-1})$, would result in the smoothed probability. This adjusted count is easier to compare directly with the MLE counts. That is, the Laplace probability can equally be expressed as the adjusted count divided by the (non-smoothed) denominator from Eq. 3.25:

$$P_{\text{Laplace}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n) + 1}{C(w_{n-1}) + V} = \frac{C^*(w_{n-1}w_n)}{C(w_{n-1})}$$

	i	want	to	eat	chinese	food	lunch	spend
i	0.0015	0.21	0.00025	0.0025	0.00025	0.00025	0.00025	0.00075
want	0.0013	0.00042	0.26	0.00084	0.0029	0.0029	0.0025	0.00084
to	0.00078	0.00026	0.0013	0.18	0.00078	0.00026	0.0018	0.055
eat	0.00046	0.00046	0.0014	0.00046	0.0078	0.0014	0.02	0.00046
chinese	0.0012	0.00062	0.00062	0.00062	0.00062	0.052	0.0012	0.00062
food	0.0063	0.00039	0.0063	0.00039	0.00079	0.002	0.00039	0.00039
lunch	0.0017	0.00056	0.00056	0.00056	0.00056	0.0011	0.00056	0.00056
spend	0.0012	0.00058	0.0012	0.00058	0.00058	0.00058	0.00058	0.00058

Figure 3.7 Add-one smoothed bigram probabilities for eight of the words (out of $V = 1446$) in the BeRP corpus of 9332 sentences computed by Eq. 3.26. Previously-zero probabilities are in gray.

Rearranging terms, we can solve for $C^*(w_{n-1}w_n)$:

$$C^*(w_{n-1}w_n) = \frac{[C(w_{n-1}w_n)+1] \times C(w_{n-1})}{C(w_{n-1})+V} \tag{3.27}$$

Figure 3.8 shows the reconstructed counts, computed by Eq. 3.27.

	i	want	to	eat	chinese	food	lunch	spend
i	3.8	527	0.64	6.4	0.64	0.64	0.64	1.9
want	1.2	0.39	238	0.78	2.7	2.7	2.3	0.78
to	1.9	0.63	3.1	430	1.9	0.63	4.4	133
eat	0.34	0.34	1	0.34	5.8	1	15	0.34
chinese	0.2	0.098	0.098	0.098	0.098	8.2	0.2	0.098
food	6.9	0.43	6.9	0.43	0.86	2.2	0.43	0.43
lunch	0.57	0.19	0.19	0.19	0.19	0.38	0.19	0.19
spend	0.32	0.16	0.32	0.16	0.16	0.16	0.16	0.16

Figure 3.8 Add-one reconstituted counts for eight words (of $V = 1446$) in the BeRP corpus of 9332 sentences, computed by Eq. 3.27. Previously-zero counts are in gray.

Note that add-one smoothing has made a very big change to the counts. Comparing Fig. 3.8 to the original counts in Fig. 3.1, we can see that $C(want\ to)$ changed from 608 to 238. We can see this in probability space as well: $P(to|want)$ decreases from 0.66 in the unsmoothed case to 0.26 in the smoothed case. Looking at the **discount** d, defined as the ratio between new and old counts, shows us how strikingly the counts for each prefix word have been reduced; the discount for the bigram *want to* is 0.39, while the discount for *Chinese food* is 0.10, a factor of 10. The sharp change occurs because too much probability mass is moved to all the zeros.

3.6.2 Add-k smoothing

One alternative to add-one smoothing is to move a bit less of the probability mass from the seen to the unseen events. Instead of adding 1 to each count, we add a fractional count k (0.5? 0.01?). This algorithm is therefore called **add-k smoothing**.

add-k

$$P^*_{\text{Add-k}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)+k}{C(w_{n-1})+kV} \tag{3.28}$$

Add-k smoothing requires that we have a method for choosing k; this can be done, for example, by optimizing on a **devset**. Although add-k is useful for some tasks (including text classification), it turns out that it still doesn't work well for

language modeling, generating counts with poor variances and often inappropriate discounts (Gale and Church, 1994).

3.6.3 Language Model Interpolation

There is an alternative source of knowledge we can draw on to solve the problem of zero frequency n-grams. If we are trying to compute $P(w_n|w_{n-2}w_{n-1})$ but we have no examples of a particular trigram $w_{n-2}w_{n-1}w_n$, we can instead estimate its probability by using the bigram probability $P(w_n|w_{n-1})$. Similarly, if we don't have counts to compute $P(w_n|w_{n-1})$, we can look to the unigram $P(w_n)$. In other words, sometimes using **less context** can help us generalize more for contexts that the model hasn't learned much about.

interpolation

The most common way to use this n-gram hierarchy is called **interpolation**: computing a new probability by interpolating (weighting and combining) the trigram, bigram, and unigram probabilities. In simple linear interpolation, we combine different order n-grams by linearly interpolating them. Thus, we estimate the trigram probability $P(w_n|w_{n-2}w_{n-1})$ by mixing together the unigram, bigram, and trigram probabilities, each weighted by a λ:

$$\begin{aligned}\hat{P}(w_n|w_{n-2}w_{n-1}) &= \lambda_1 P(w_n) \\ &\quad +\lambda_2 P(w_n|w_{n-1}) \\ &\quad +\lambda_3 P(w_n|w_{n-2}w_{n-1})\end{aligned} \tag{3.29}$$

The λs must sum to 1, making Eq. 3.29 equivalent to a weighted average. In a slightly more sophisticated version of linear interpolation, each λ weight is computed by conditioning on the context. This way, if we have particularly accurate counts for a particular bigram, we assume that the counts of the trigrams based on this bigram will be more trustworthy, so we can make the λs for those trigrams higher and thus give that trigram more weight in the interpolation. Equation 3.30 shows the equation for interpolation with context-conditioned weights, where each *lambda* takes an argument that is the two prior word context:

$$\begin{aligned}\hat{P}(w_n|w_{n-2}w_{n-1}) &= \lambda_1(w_{n-2:n-1})P(w_n) \\ &\quad +\lambda_2(w_{n-2:n-1})P(w_n|w_{n-1}) \\ &\quad +\lambda_3(w_{n-2:n-1})P(w_n|w_{n-2}w_{n-1})\end{aligned} \tag{3.30}$$

held-out

How are these λ values set? Both the simple interpolation and conditional interpolation λs are learned from a **held-out** corpus. A held-out corpus is an additional training corpus, so-called because we hold it out from the training data, that we use to set these λ values.[3] We do so by choosing the λ values that maximize the likelihood of the held-out corpus. That is, we fix the n-gram probabilities and then search for the λ values that—when plugged into Eq. 3.29—give us the highest probability of the held-out set. There are various ways to find this optimal set of λs. One way is to use the **EM** algorithm, an iterative learning algorithm that converges on locally optimal λs (Jelinek and Mercer, 1980).

3.6.4 Stupid Backoff

backoff

An alternative to interpolation is **backoff**. In a backoff model, if the n-gram we need

[3] Held-out corpora are generally used to set **hyperparameters**, which are special parameters, unlike regular counts that are learned from the training data; we'll discuss hyperparameters in Chapter 6.

has zero counts, we approximate it by backing off to the (n-1)-gram. We continue backing off until we reach a history that has some counts. For a backoff model to give a correct probability distribution, we have to **discount** the higher-order n-grams to save some probability mass for the lower order n-grams. In practice, instead of discounting, it's common to use a much simpler non-discounted backoff algorithm called **stupid backoff** (Brants et al., 2007).

discount

stupid backoff

Stupid backoff gives up the idea of trying to make the language model a true probability distribution. There is no discounting of the higher-order probabilities. If a higher-order n-gram has a zero count, we simply backoff to a lower order n-gram, weighed by a fixed (context-independent) weight. This algorithm does not produce a probability distribution, so we'll follow Brants et al. (2007) in referring to it as S:

$$S(w_i|w_{i-N+1:i-1}) = \begin{cases} \dfrac{\text{count}(w_{i-N+1:i})}{\text{count}(w_{i-N+1:i-1})} & \text{if count}(w_{i-N+1:i}) > 0 \\ \lambda S(w_i|w_{i-N+2:i-1}) & \text{otherwise} \end{cases} \tag{3.31}$$

The backoff terminates in the unigram, which has score $S(w) = \frac{count(w)}{N}$. Brants et al. (2007) find that a value of 0.4 worked well for λ.

3.7 Advanced: Perplexity's Relation to Entropy

We introduced perplexity in Section 3.3 as a way to evaluate n-gram models on a test set. A better n-gram model is one that assigns a higher probability to the test data, and perplexity is a normalized version of the probability of the test set. The perplexity measure actually arises from the information-theoretic concept of cross-entropy, which explains otherwise mysterious properties of perplexity (why the inverse probability, for example?) and its relationship to entropy. **Entropy** is a measure of information. Given a random variable X ranging over whatever we are predicting (words, letters, parts of speech), the set of which we'll call χ, and with a particular probability function, call it $p(x)$, the entropy of the random variable X is:

Entropy

$$H(X) = -\sum_{x \in \chi} p(x) \log_2 p(x) \tag{3.32}$$

The log can, in principle, be computed in any base. If we use log base 2, the resulting value of entropy will be measured in **bits**.

One intuitive way to think about entropy is as a lower bound on the number of bits it would take to encode a certain decision or piece of information in the optimal coding scheme. Consider an example from the standard information theory textbook Cover and Thomas (1991). Imagine that we want to place a bet on a horse race but it is too far to go all the way to Yonkers Racetrack, so we'd like to send a short message to the bookie to tell him which of the eight horses to bet on. One way to encode this message is just to use the binary representation of the horse's number as the code; thus, horse 1 would be `001`, horse 2 `010`, horse 3 `011`, and so on, with horse 8 coded as `000`. If we spend the whole day betting and each horse is coded with 3 bits, on average we would be sending 3 bits per race.

Can we do better? Suppose that the spread is the actual distribution of the bets placed and that we represent it as the prior probability of each horse as follows:

Horse 1	$\frac{1}{2}$	Horse 5	$\frac{1}{64}$
Horse 2	$\frac{1}{4}$	Horse 6	$\frac{1}{64}$
Horse 3	$\frac{1}{8}$	Horse 7	$\frac{1}{64}$
Horse 4	$\frac{1}{16}$	Horse 8	$\frac{1}{64}$

The entropy of the random variable X that ranges over horses gives us a lower bound on the number of bits and is

$$\begin{aligned} H(X) &= -\sum_{i=1}^{i=8} p(i)\log_2 p(i) \\ &= -\tfrac{1}{2}\log_2\tfrac{1}{2} - \tfrac{1}{4}\log_2\tfrac{1}{4} - \tfrac{1}{8}\log_2\tfrac{1}{8} - \tfrac{1}{16}\log_2\tfrac{1}{16} - 4(\tfrac{1}{64}\log_2\tfrac{1}{64}) \\ &= 2 \text{ bits} \end{aligned} \tag{3.33}$$

A code that averages 2 bits per race can be built with short encodings for more probable horses, and longer encodings for less probable horses. For example, we could encode the most likely horse with the code `0`, and the remaining horses as `10`, then `110`, `1110`, `111100`, `111101`, `111110`, and `111111`.

What if the horses are equally likely? We saw above that if we used an equal-length binary code for the horse numbers, each horse took 3 bits to code, so the average was 3. Is the entropy the same? In this case each horse would have a probability of $\frac{1}{8}$. The entropy of the choice of horses is then

$$H(X) = -\sum_{i=1}^{i=8} \frac{1}{8}\log_2\frac{1}{8} = -\log_2\frac{1}{8} = 3 \text{ bits} \tag{3.34}$$

Until now we have been computing the entropy of a single variable. But most of what we will use entropy for involves *sequences*. For a grammar, for example, we will be computing the entropy of some sequence of words $W = \{w_1, w_2, \ldots, w_n\}$. One way to do this is to have a variable that ranges over sequences of words. For example we can compute the entropy of a random variable that ranges over all sequences of words of length n in some language L as follows:

$$H(w_1, w_2, \ldots, w_n) = -\sum_{w_{1:n}\in L} p(w_{1:n})\log p(w_{1:n}) \tag{3.35}$$

entropy rate

We could define the **entropy rate** (we could also think of this as the **per-word entropy**) as the entropy of this sequence divided by the number of words:

$$\frac{1}{n}H(w_{1:n}) = -\frac{1}{n}\sum_{w_{1:n}\in L} p(w_{1:n})\log p(w_{1:n}) \tag{3.36}$$

But to measure the true entropy of a language, we need to consider sequences of infinite length. If we think of a language as a stochastic process L that produces a sequence of words, and allow W to represent the sequence of words $w_1, \ldots, w_n$, then L's entropy rate $H(L)$ is defined as

$$\begin{aligned} H(L) &= \lim_{n\to\infty}\frac{1}{n}H(w_{1:n}) \\ &= -\lim_{n\to\infty}\frac{1}{n}\sum_{W\in L} p(w_{1:n})\log p(w_{1:n}) \end{aligned} \tag{3.37}$$

The Shannon-McMillan-Breiman theorem (Algoet and Cover 1988, Cover and Thomas 1991) states that if the language is regular in certain ways (to be exact, if it is both stationary and ergodic),

$$H(L) = \lim_{n\to\infty} -\frac{1}{n}\log p(w_{1:n}) \tag{3.38}$$

That is, we can take a single sequence that is long enough instead of summing over all possible sequences. The intuition of the Shannon-McMillan-Breiman theorem is that a long-enough sequence of words will contain in it many other shorter sequences and that each of these shorter sequences will reoccur in the longer sequence according to their probabilities.

Stationary

A stochastic process is said to be **stationary** if the probabilities it assigns to a sequence are invariant with respect to shifts in the time index. In other words, the probability distribution for words at time t is the same as the probability distribution at time $t+1$. Markov models, and hence n-grams, are stationary. For example, in a bigram, P_i is dependent only on P_{i-1}. So if we shift our time index by x, P_{i+x} is still dependent on P_{i+x-1}. But natural language is not stationary, since as we show in Appendix D, the probability of upcoming words can be dependent on events that were arbitrarily distant and time dependent. Thus, our statistical models only give an approximation to the correct distributions and entropies of natural language.

To summarize, by making some incorrect but convenient simplifying assumptions, we can compute the entropy of some stochastic process by taking a very long sample of the output and computing its average log probability.

cross-entropy

Now we are ready to introduce **cross-entropy**. The cross-entropy is useful when we don't know the actual probability distribution p that generated some data. It allows us to use some m, which is a model of p (i.e., an approximation to p). The cross-entropy of m on p is defined by

$$H(p,m) = \lim_{n\to\infty} -\frac{1}{n}\sum_{W\in L} p(w_1,\ldots,w_n)\log m(w_1,\ldots,w_n) \tag{3.39}$$

That is, we draw sequences according to the probability distribution p, but sum the log of their probabilities according to m.

Again, following the Shannon-McMillan-Breiman theorem, for a stationary ergodic process:

$$H(p,m) = \lim_{n\to\infty} -\frac{1}{n}\log m(w_1 w_2 \ldots w_n) \tag{3.40}$$

This means that, as for entropy, we can estimate the cross-entropy of a model m on some distribution p by taking a single sequence that is long enough instead of summing over all possible sequences.

What makes the cross-entropy useful is that the cross-entropy $H(p,m)$ is an upper bound on the entropy $H(p)$. For any model m:

$$H(p) \leq H(p,m) \tag{3.41}$$

This means that we can use some simplified model m to help estimate the true entropy of a sequence of symbols drawn according to probability p. The more accurate m is, the closer the cross-entropy $H(p,m)$ will be to the true entropy $H(p)$. Thus, the difference between $H(p,m)$ and $H(p)$ is a measure of how accurate a model is. Between two models m_1 and m_2, the more accurate model will be the one with the

lower cross-entropy. (The cross-entropy can never be lower than the true entropy, so a model cannot err by underestimating the true entropy.)

We are finally ready to see the relation between perplexity and cross-entropy as we saw it in Eq. 3.40. Cross-entropy is defined in the limit as the length of the observed word sequence goes to infinity. We approximate this cross-entropy by relying on a (sufficiently long) sequence of fixed length. This approximation to the cross-entropy of a model $M = P(w_i|w_{i-N+1:i-1})$ on a sequence of words W is

$$H(W) = -\frac{1}{N}\log P(w_1 w_2 \ldots w_N) \tag{3.42}$$

perplexity

The **perplexity** of a model P on a sequence of words W is now formally defined as 2 raised to the power of this cross-entropy:

$$\begin{aligned}\text{Perplexity}(W) &= 2^{H(W)} \\ &= P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} \\ &= \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}}\end{aligned}$$

3.8 Summary

This chapter introduced language modeling via the n-gram model, a classic model that allows us to introduce many of the basic concepts in language modeling.

- Language models offer a way to assign a probability to a sentence or other sequence of words or tokens, and to predict a word or token from preceding words or tokens.
- **N-grams** are perhaps the simplest kind of language model. They are Markov models that estimate words from a fixed window of previous words. N-gram models can be trained by counting in a **training corpus** and normalizing the counts (the **maximum likelihood estimate**).
- N-gram **language models** can be evaluated on a **test set** using **perplexity.**
- The **perplexity** of a test set according to a language model is a function of the probability of the test set: the inverse test set probability according to the model, normalized by the length.
- **Sampling** from a language model means to generate some sentences, choosing each sentence according to its likelihood as defined by the model.
- **Smoothing** algorithms provide a way to estimate probabilities for events that were unseen in training. Commonly used smoothing algorithms for n-grams include add-1 smoothing, or rely on lower-order n-gram counts through **interpolation**.

Historical Notes

The underlying mathematics of the n-gram was first proposed by Markov (1913), who used what are now called **Markov chains** (bigrams and trigrams) to predict whether an upcoming letter in Pushkin's *Eugene Onegin* would be a vowel or a consonant. Markov classified 20,000 letters as V or C and computed the bigram and

trigram probability that a given letter would be a vowel given the previous one or two letters. Shannon (1948) applied n-grams to compute approximations to English word sequences. Based on Shannon's work, Markov models were commonly used in engineering, linguistic, and psychological work on modeling word sequences by the 1950s. In a series of extremely influential papers starting with Chomsky (1956) and including Chomsky (1957) and Miller and Chomsky (1963), Noam Chomsky argued that "finite-state Markov processes", while a possibly useful engineering heuristic, were incapable of being a complete cognitive model of human grammatical knowledge. These arguments led many linguists and computational linguists to ignore work in statistical modeling for decades.

The resurgence of n-gram language models came from Fred Jelinek and colleagues at the IBM Thomas J. Watson Research Center, who were influenced by Shannon, and James Baker at CMU, who was influenced by the prior, classified work of Leonard Baum and colleagues on these topics at labs like the US Institute for Defense Analyses (IDA) after they were declassified. Independently these two labs successfully used n-grams in their speech recognition systems at the same time (Baker 1975b, Jelinek et al. 1975, Baker 1975a, Bahl et al. 1983, Jelinek 1990). The terms "language model" and "perplexity" were first used for this technology by the IBM group. Jelinek and his colleagues used the term *language model* in a pretty modern way, to mean the entire set of linguistic influences on word sequence probabilities, including grammar, semantics, discourse, and even speaker characteristics, rather than just the particular n-gram model itself.

Add-one smoothing derives from Laplace's 1812 law of succession and was first applied as an engineering solution to the zero frequency problem by Jeffreys (1948) based on an earlier Add-K suggestion by Johnson (1932). Problems with the add-one algorithm are summarized in Gale and Church (1994).

A wide variety of different language modeling and smoothing techniques were proposed in the 80s and 90s, including Good-Turing discounting—first applied to the n-gram smoothing at IBM by Katz (Nádas 1984, Church and Gale 1991)— Witten-Bell discounting (Witten and Bell, 1991), and varieties of **class-based n-gram** models that used information about word classes. Starting in the late 1990s, Chen and Goodman performed a number of carefully controlled experiments comparing different algorithms and parameters (Chen and Goodman 1999, Goodman 2006, inter alia). They showed the advantages of **Modified Interpolated Kneser-Ney**, which became the standard baseline for n-gram language modeling around the turn of the century, especially because they showed that caches and class-based models provided only minor additional improvement. SRILM (Stolcke, 2002) and KenLM (Heafield 2011, Heafield et al. 2013) are publicly available toolkits for building n-gram language models.

class-based n-gram

Large language models are based on **neural networks** rather than n-grams, enabling them to solve the two major problems with n-grams: (1) the number of parameters increases exponentially as the n-gram order increases, and (2) n-grams have no way to generalize from training examples to test set examples unless they use identical words. Neural language models instead project words into a **continuous** space in which words with similar contexts have similar representations. We'll introduce transformer-based **large language models** in Chapter 8, along the way introducing **feedforward** language models (Bengio et al. 2006, Schwenk 2007) in Chapter 6 and **recurrent** language models (Mikolov, 2012) in Chapter 13.

Exercises

3.1 Write out the equation for trigram probability estimation (modifying Eq. 3.11). Now write out all the non-zero trigram probabilities for the `I am Sam` corpus on page 41.

3.2 Calculate the probability of the sentence `i want chinese food`. Give two probabilities, one using Fig. 3.2 and the 'useful probabilities' just below it on page 43, and another using the add-1 smoothed table in Fig. 3.7. Assume the additional add-1 smoothed probabilities $P(\texttt{i} \mid \texttt{<s>}) = 0.19$ and $P(\texttt{</s>} \mid \texttt{food}) = 0.40$.

3.3 Which of the two probabilities you computed in the previous exercise is higher, unsmoothed or smoothed? Explain why.

3.4 We are given the following corpus, modified from the one in the chapter:

```
<s> I am Sam </s>
<s> Sam I am </s>
<s> I am Sam </s>
<s> I do not like green eggs and Sam </s>
```

Using a bigram language model with add-one smoothing, what is P(Sam | am)? Include <s> and </s> in your counts just like any other token.

3.5 Suppose we didn't use the end-symbol </s>. Train an unsmoothed bigram grammar on the following training corpus without using the end-symbol </s>:

```
<s> a b
<s> b b
<s> b a
<s> a a
```

Demonstrate that your bigram model does not assign a single probability distribution across all sentence lengths by showing that the sum of the probability of the four possible 2 word sentences over the alphabet {a,b} is 1.0, and the sum of the probability of all possible 3 word sentences over the alphabet {a,b} is also 1.0.

3.6 Suppose we train a trigram language model with add-one smoothing on a given corpus. The corpus contains V word types. Express a formula for estimating P(w3|w1,w2), where w3 is a word which follows the bigram (w1,w2), in terms of various n-gram counts and V. Use the notation c(w1,w2,w3) to denote the number of times that trigram (w1,w2,w3) occurs in the corpus, and so on for bigrams and unigrams.

3.7 We are given the following corpus, modified from the one in the chapter:

```
<s> I am Sam </s>
<s> Sam I am </s>
<s> I am Sam </s>
<s> I do not like green eggs and Sam </s>
```

If we use linear interpolation smoothing between a maximum-likelihood bigram model and a maximum-likelihood unigram model with $\lambda_1 = \frac{1}{2}$ and $\lambda_2 = \frac{1}{2}$, what is P(Sam|am)? Include <s> and </s> in your counts just like any other token.

3.8 Write a program to compute unsmoothed unigrams and bigrams.

3.9 Run your n-gram program on two different small corpora of your choice (you might use email text or newsgroups). Now compare the statistics of the two corpora. What are the differences in the most common unigrams between the two? How about interesting differences in bigrams?

3.10 Add an option to your program to generate random sentences.

3.11 Add an option to your program to compute the perplexity of a test set.

3.12 You are given a training set of 100 numbers that consists of 91 zeros and 1 each of the other digits 1-9. Now we see the following test set: 0 0 0 0 0 3 0 0 0 0. What is the unigram perplexity?

CHAPTER

4 Logistic Regression and Text Classification

En sus remotas páginas está escrito que los animales se dividen en:
a. pertenecientes al Emperador
b. embalsamados
c. amaestrados
d. lechones
e. sirenas
f. fabulosos
g. perros sueltos
h. incluidos en esta clasificación
i. que se agitan como locos
j. innumerables
k. dibujados con un pincel finísimo de pelo de camello
l. etcétera
m. que acaban de romper el jarrón
n. que de lejos parecen moscas
Borges (1964)

Classification lies at the heart of language processing and intelligence. Recognizing a letter, a word, or a face, sorting mail, assigning grades to homeworks; these are all examples of assigning a category to an input. The challenges of classification were famously highlighted by the fabulist Jorge Luis Borges (1964), who imagined an ancient mythical encyclopedia that classified animals into:

> *(a) those that belong to the Emperor, (b) embalmed ones, (c) those that are trained, (d) suckling pigs, (e) mermaids, (f) fabulous ones, (g) stray dogs, (h) those that are included in this classification, (i) those that tremble as if they were mad, (j) innumerable ones, (k) those drawn with a very fine camel's hair brush, (l) others, (m) those that have just broken a flower vase, (n) those that resemble flies from a distance.*

Luckily, the classes we use for language processing are easier to define than those of Borges. In this chapter we introduce the **logistic regression** algorithm for classification, and apply it to **text categorization**, the task of assigning a label or category to a text or document. We'll focus on one text categorization task, **sentiment analysis**, the categorization of **sentiment**, the positive or negative orientation that a writer expresses toward some object. A review of a movie, book, or product expresses the author's sentiment toward the product, while an editorial or political text expresses sentiment toward an action or candidate. Extracting sentiment is thus relevant for fields from marketing to politics.

text categorization

sentiment analysis

For the binary task of labeling a text as indicating positive or negative stance, words (like *awesome* and *love*, or *awful* and *ridiculously* are very informative, as we can see from these sample extracts from movie/restaurant reviews:

\+ *...awesome caramel sauce and sweet toasty almonds. I love this place!*
− *...awful pizza and ridiculously overpriced...*

There are many text classification tasks. In **spam detection** we assign an email to one of the two classes *spam* or *not-spam*. **Language id** is the task of determining what language a text is written in, while **authorship attribution** is the task of determining a text's author, relevant to both humanistic and forensic analysis.

spam detection

language id

authorship attribution

But what makes classification so important is that **language modeling** can also be viewed as classification: each word can be thought of as a class, and so predicting the next word is classifying the context-so-far into a class for each next word. As we'll see, this intuition underlies large language models.

The algorithm for classification we introduce in this chapter, logistic regression, is equally important, in a number of ways. First, logistic regression has a close relationship with neural networks. As we will see in Chapter 6, a neural network can be viewed as a series of logistic regression classifiers stacked on top of each other. Second, logistic regression introduces ideas that are fundamental to neural networks and language models, like the **sigmoid** and **softmax** functions, the **logit**, and the key **gradient descent** algorithm for learning. Finally, logistic regression is also one of the most important analytic tools in the social and natural sciences.

sigmoid
softmax
logit

4.1 Machine learning and classification

The goal of classification is to take a single input (we call each input an **observation**), extract some useful **features** or properties of the input, and thereby **classify** the observation into one of a set of discrete classes. We'll call the input x, and say that the output comes from a fixed set of output classes $Y = \{y_1, y_2, ..., y_M\}$. Our goal is return a predicted class from Y. Sometimes you'll see the output classes referred to as the set C instead of Y.

observation

For sentiment analysis, the input x might be a review, or some other text. And the output set Y might be the set:

$$\{\texttt{positive}, \texttt{negative}\}$$

or the set

$$\{0, 1\}$$

For language id, the input might be a text that we need to know what language it was written in, and the output set Y is the set of languages, i.e.,

$$Y = \{\texttt{Abkhaz}, \texttt{Ainu}, \texttt{Albanian}, \texttt{Amharic}, ...\texttt{Zulu}, \texttt{Zuñi}\}$$

There are many ways to do classification. One method is to use rules handwritten by humans. For example, we might have a rule like:

```
If the word ''love'' appears in x, and it's not preceded by the
word ''don't", classify as positive
```

Handwritten rules can be components of modern NLP systems, such as the handwritten lists of positive and negative words that can be used in sentiment analysis, as we'll see below. But rules can be fragile, as situations or data change over time, and for many tasks there are complex interactions between different features (like the example of negation with "don't" in the rule above), so it can be quite hard for humans to come up with rules that are successful over many situations.

Another method that we will introduce later is to ask a large language model (of the type we will introduce in Chapter 7) by prompting the model to give a label to some text. Prompting can be powerful, but again has weaknesses: language models often hallucinate, and may not be able to explain why they chose the class they did.

For these reasons the most common way to do classification is to use **supervised machine learning**. Supervised machine learning is a paradigm in which, in

supervised machine learning

addition to the input and the set of output classes, we have a **labeled training set** and a **learning algorithm**. We talked about training sets in Chapter 3 as a locus for computing n-gram statistics. But in supervised machine learning the training set is **labeled**, meaning that it contains a set of input observations, each observation associated with the correct output (a 'supervision signal'). We can generally refer to a training set of m input/output pairs, where each input x is a text, in the case of text classification, and each is hand-labeled with an associated class (the correct label):.

$$\text{training set:} \quad \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\} \tag{4.1}$$

We'll use superscripts in parentheses to refer to individual observations or instances in the training set. So for sentiment classification, a training set might be a set of sentences or other texts, each with their correct sentiment label.

Our goal is to learn from this training set a classifier that is capable of mapping from a **new** input x to its correct class $y \in Y$. It does this by learning to find features in these training sentences (perhaps words like "awesome" or "awful"). **Probabilistic classifiers** like logistic regression are classifiers that in addition to giving an answer (the class this observation is in), also give the *probability* of the observation being in the class. This distribution over the classes can be useful information for downstream decisions; avoiding making discrete decisions early on can be useful when combining systems. Probabilistic classifiers like logistic regression have four components:

1. A **feature representation** of the input. For each input observation $x^{(i)}$, this will be a vector of features $[x_1, x_2, ..., x_n]$. We will generally refer to feature i for input $x^{(j)}$ as $x_i^{(j)}$, sometimes simplified as x_i, but we will also see the notation f_i, $f_i(x)$, or, for multiclass classification, $f_i(c, x)$.
2. A classification function that computes the estimated class, by computing the probability $P(y = y_i|x)$ for each output class y_i. We will introduce the **sigmoid** and **softmax** tools for classification.
3. An **objective function** that we want to optimize for learning, usually involving minimizing a loss function corresponding to error on training examples. We will introduce the **cross-entropy loss function**.
4. An algorithm for optimizing the objective function. We introduce the **stochastic gradient descent** algorithm.

At the highest level, logistic regression, and really any probabilistic machine learning classifier, has two phases

training: We train the system (in the case of logistic regression that means training the weights w and b, introduced below) using stochastic gradient descent and the cross-entropy loss.

test: Given a test example x we compute the probability $P(y = y_i|x)$ for each output class y_i. Then, given this vector of probabilities, we return the higher probability label $y = 1$ or $y = 0$.

Logistic regression can be used to classify an observation into one of two classes (like 'positive sentiment' and 'negative sentiment'), or into one of many classes. Because the mathematics for the two-class case is simpler, we'll first describe this special case of logistic regression in the next few sections, beginning with the **sigmoid** function, and then turn to **multinomial logistic regression** for more than two classes and the use of the **softmax** function in Section 4.7.

4.2 The sigmoid function

The goal of binary logistic regression is to train a classifier that can make a binary decision about the class of a new input observation. Here we introduce the **sigmoid** classifier that will help us make this decision.

Consider a single input observation x, which we will represent by a vector of features $[x_1, x_2, ..., x_n]$. (We'll show sample features in the next subsection.) The classifier output y can be 1 (meaning the observation is a member of the class) or 0 (the observation is not a member of the class). We want to know the probability $P(y = 1|x)$ that this observation is a member of the class. So perhaps the decision is "positive sentiment" versus "negative sentiment", the features represent counts of words in a document, $P(y = 1|x)$ is the probability that the document has positive sentiment, and $P(y = 0|x)$ is the probability that the document has negative sentiment.

Logistic regression solves this task by learning, from a training set, a vector of **weights** and a **bias term**. Each weight w_i is a real number, and is associated with one of the input features x_i. The weight w_i represents how important that input feature is to the classification decision, and can be positive (providing evidence that the instance being classified belongs in the positive class) or negative (providing evidence that the instance being classified belongs in the negative class). Thus we might expect in a sentiment task the word *awesome* to have a high positive weight, and *abysmal* to have a very negative weight. The **bias term**, also called the **intercept**, is another real number that's added to the weighted inputs.

bias term
intercept

To make a decision on a test instance—after we've learned the weights in training—the classifier first multiplies each x_i by its weight w_i, sums up the weighted features, and adds the bias term b. The resulting single number z expresses the weighted sum of the evidence for the class.

$$z = \left(\sum_{i=1}^{n} w_i x_i\right) + b \tag{4.2}$$

dot product

In the rest of the book we'll represent such sums using the **dot product** notation from linear algebra. The dot product of two vectors $\mathbf{a}$ and $\mathbf{b}$, written as $\mathbf{a} \cdot \mathbf{b}$, is the sum of the products of the corresponding elements of each vector. (Notice that we represent vectors using the boldface notation $\mathbf{b}$). Thus the following is an equivalent formation to Eq. 4.2:

$$z = \mathbf{w} \cdot \mathbf{x} + b \tag{4.3}$$

But note that nothing in Eq. 4.3 forces z to be a legal probability, that is, to lie between 0 and 1. In fact, since weights are real-valued, the output might even be negative; z ranges from $-\infty$ to ∞.

sigmoid

logistic function

To create a probability, we'll pass z through the **sigmoid** function, $\sigma(z)$. The sigmoid function (named because it looks like an s) is also called the **logistic function**, and gives logistic regression its name. The sigmoid has the following equation, shown graphically in Fig. 4.1:

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+\exp(-z)} \tag{4.4}$$

(For the rest of the book, we'll use the notation $\exp(x)$ to mean e^x.) The sigmoid has a number of advantages; it takes a real-valued number and maps it into the range

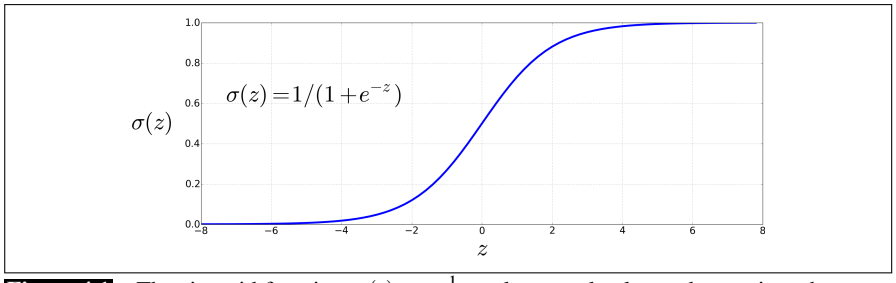


Figure 4.1 The sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ takes a real value and maps it to the range $(0,1)$. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

$(0,1)$, which is just what we want for a probability. Because it is nearly linear around 0 but flattens toward the ends, it tends to squash outlier values toward 0 or 1. And it's differentiable, which as we'll see in Section 4.15 will be handy for learning.

We're almost there. If we apply the sigmoid to the sum of the weighted features, we get a number between 0 and 1. To make it a probability, we just need to make sure that the two cases, $P(y=1)$ and $P(y=0)$, sum to 1. We can do this as follows:

$$\begin{aligned} P(y=1) &= \sigma(\mathbf{w}\cdot\mathbf{x}+b) \\ &= \frac{1}{1+\exp(-(\mathbf{w}\cdot\mathbf{x}+b))} \\ P(y=0) &= 1-\sigma(\mathbf{w}\cdot\mathbf{x}+b) \\ &= 1-\frac{1}{1+\exp(-(\mathbf{w}\cdot\mathbf{x}+b))} \\ &= \frac{\exp(-(\mathbf{w}\cdot\mathbf{x}+b))}{1+\exp(-(\mathbf{w}\cdot\mathbf{x}+b))} \end{aligned} \tag{4.5}$$

The sigmoid function has the property

$$1-\sigma(x) = \sigma(-x) \tag{4.6}$$

so we could also have expressed $P(y=0)$ as $\sigma(-(\mathbf{w}\cdot\mathbf{x}+b))$.

logit

Finally, two terminological points. First, the input to the sigmoid function, the score $z = \mathbf{w}\cdot\mathbf{x}+b$ from Eq. 4.3, is often called the **logit**. This is because the logit function is the inverse of the sigmoid. The logit function is the log of the odds ratio $\frac{p}{1-p}$:

$$\text{logit}(p) = \sigma^{-1}(p) = \ln\frac{p}{1-p} \tag{4.7}$$

Using the term **logit** for z is a way of reminding us that by using the sigmoid to turn z (which ranges from $-\infty$ to ∞) into a probability, we are implicitly interpreting z as not just any real-valued number, but as specifically a log odds.

y hat $\hat{y}$

Second, for binary classification it will be convenient to refer to the output of the sigmoid function $\sigma(\mathbf{w}\cdot\mathbf{x}+b)$ which is computing $P(y=1)$, as $\hat{y}$. We pronounce this symbol as **y hat**. We thus use $\hat{y}$ to mean "the probability of the input observation having the positive class". When we introduce multinomial or softmax logistic regression in Section 4.7, we will introduce an extension of $\hat{y}$ that will be a vector of probabities over all the output classes, but for binary classification we just keep one scalar probability, knowing that we can always compute the missing probability $P(y=0)$ as $1-P(y=1)$.

4.3 Classification with Logistic Regression

The sigmoid function from the prior section thus gives us a way to take an instance x and compute the probability $P(y=1|x)$.

How do we make a decision about which class to apply to a test instance x? For a given x, we say yes if the probability $P(y=1|x)$ is more than .5, and no otherwise. We call .5 the **decision boundary**:

decision boundary

$$\text{decision}(x) = \begin{cases} 1 & \text{if } P(y=1|x) > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

Let's have some examples of applying logistic regression as a classifier for language tasks.

4.3.1 Sentiment Classification

Suppose we are doing binary sentiment classification on movie review text, and we would like to know whether to assign the sentiment class + or − to a review document *doc*. We'll represent each input observation by the 6 features $x_1 \ldots x_6$ of the input shown in the following table; Fig. 4.2 shows features in a sample mini test document.

Var	Definition	Value in Fig. 4.2
x_1	count(positive lexicon words ∈ doc)	3
x_2	count(negative lexicon words ∈ doc)	2
x_3	$\begin{cases} 1 & \text{if "no"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	1
x_4	count(1st and 2nd pronouns ∈ doc)	3
x_5	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	0
x_6	ln(word+punctuation count of doc)	ln(66) = 4.19

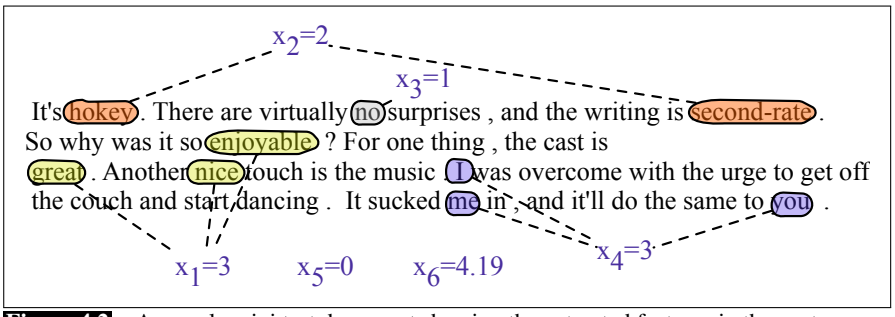


Figure 4.2 A sample mini test document showing the extracted features in the vector x.

Let's assume for the moment that we've already learned a real-valued weight for each of these features, and that the 6 weights corresponding to the 6 features are $[2.5, -5.0, -1.2, 0.5, 2.0, 0.7]$, while $b = 0.1$. (We'll discuss in the next section how the weights are learned.) The weight w_1, for example indicates how important a feature the number of positive lexicon words (*great*, *nice*, *enjoyable*, etc.) is to

a positive sentiment decision, while w_2 tells us the importance of negative lexicon words. Note that $w_1 = 2.5$ is positive, while $w_2 = -5.0$, meaning that negative words are negatively associated with a positive sentiment decision, and are about twice as important as positive words.

Given these 6 features and the input review x, $P(+|x)$ and $P(-|x)$ can be computed using Eq. 4.5:

$$\begin{aligned} P(+|x) = P(y=1|x) &= \sigma(\mathbf{w}\cdot\mathbf{x}+b) \\ &= \sigma([2.5,-5.0,-1.2,0.5,2.0,0.7]\cdot[3,2,1,3,0,4.19]+0.1) \\ &= \sigma(.833) \\ &= 0.70 \qquad (4.8) \\ P(-|x) = P(y=0|x) &= 1-\sigma(\mathbf{w}\cdot\mathbf{x}+b) \\ &= 0.30 \end{aligned}$$

4.3.2 Other classification tasks and features

period disambiguation

Logistic regression is applied to all sorts of NLP tasks, and any property of the input can be a feature. Consider the task of **period disambiguation**: deciding if a period is the end of a sentence or part of a word, by classifying each period into one of two classes, EOS (end-of-sentence) and not-EOS. We might use features like x_1 below expressing that the current word is lower case, perhaps with a positive weight. Or a feature expressing that the current word is in our abbreviations dictionary ("Prof."), perhaps with a negative weight. A feature can also express a combination of properties. For example a period following an upper case word is likely to be an EOS, but if the word itself is *St.* and the previous word is capitalized then the period is likely part of a shortening of the word *street* following a street name.

$$\begin{aligned} x_1 &= \begin{cases} 1 & \text{if } \text{“}Case(w_i) = \text{Lower”} \\ 0 & \text{otherwise} \end{cases} \\ x_2 &= \begin{cases} 1 & \text{if } \text{“}w_i \in \text{AcronymDict”} \\ 0 & \text{otherwise} \end{cases} \\ x_3 &= \begin{cases} 1 & \text{if } \text{“}w_i = \text{St. \& } Case(w_{i-1}) = \text{Upper”} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Designing versus learning features: In classic models, features are designed by hand by examining the training set with an eye to linguistic intuitions and literature, supplemented by insights from error analysis on the training set of an early version of a system. We can also consider **feature interactions**, complex features that are combinations of more primitive features. We saw such a feature for period disambiguation above, where a period on the word *St.* was less likely to be the end of the sentence if the previous word was capitalized. Features can be created automatically via **feature templates**, abstract specifications of features. For example a bigram template for period disambiguation might create a feature for every pair of words that occurs before a period in the training set. Thus the feature space is sparse, since we only have to create a feature if that n-gram exists in that position in the training set. The feature is generally created as a hash from the string descriptions. A user description of a feature as, "bigram(American breakfast)" is hashed into a unique integer i that becomes the feature number f_i.

feature interactions

feature templates

It should be clear from the prior paragraph that designing features by hand requires extensive human effort. For this reason, recent NLP systems avoid hand-

designed features and instead focus on **representation learning**: ways to learn features automatically in an unsupervised way from the input. We'll introduce methods for representation learning in Chapter 5 and Chapter 6.

Scaling input features: When different input features have extremely different ranges of values, it's common to rescale them so they have comparable ranges. We **standardize** input values by centering them to result in a zero mean and a standard deviation of one (this transformation is sometimes called the **z-score**). That is, if μ_i is the mean of the values of feature x_i across the m observations in the input dataset, and σ_i is the standard deviation of the values of features x_i across the input dataset, we can replace each feature x_i by a new feature x'_i computed as follows:

standardize
z-score

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)} \qquad \sigma_i = \sqrt{\frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2}$$
$$x'_i = \frac{x_i - \mu_i}{\sigma_i} \tag{4.9}$$

normalize

Alternatively, we can **normalize** the input features values to lie between 0 and 1:

$$x'_i = \frac{x_i - \min(x_i)}{\max(x_i) - \min(x_i)} \tag{4.10}$$

Having input data with comparable range is useful when comparing values across features. Data scaling is especially important in large neural networks, since it helps speed up gradient descent.

Another very common type of scaling for natural language data is to take logs, for example when inputs are word counts, or bigram counts, or anything else that follows a Zipfian distribution.

4.3.3 Processing many examples at once

We've shown the equations for logistic regression for a single example. But in practice we'll of course want to process an entire test set with many examples. Let's suppose we have a test set consisting of m test examples each of which we'd like to classify. We'll continue to use the notation from page 64, in which a superscript value in parentheses refers to the example index in some set of data (either for training or for test). So in this case each test example $x^{(i)}$ has a feature vector $\mathbf{x}^{(i)}$, $1 \leq i \leq m$. (As usual, we'll represent vectors and matrices in bold.)

One way to compute $\hat{y}^{(i)}$ (which is how we refer to $P(y^{(1)} = 1)$, i.e., the probability that the output for input $x^{(i)}i$ has the value 1), is just to have a for-loop and compute each test example one at a time:

$$\begin{aligned} &\textbf{foreach} && x^{(i)} && \text{in input } [x^{(1)}, x^{(2)}, ..., x^{(m)}] \\ &&& \hat{y}^{(i)} = P(y^{(i)} = 1) && = \sigma(\mathbf{w} \cdot \mathbf{x^{(i)}} + b) \end{aligned} \tag{4.11}$$

For the first 3 test examples, then, we would be separately computing the predicted output probability as follows:

$$\begin{aligned} \hat{y}^{(1)} = P(y^{(1)} = 1 | x^{(1)}) &= \sigma(\mathbf{w} \cdot \mathbf{x^{(1)}} + b) \\ \hat{y}^{(2)} = P(y^{(2)} = 1 | x^{(2)}) &= \sigma(\mathbf{w} \cdot \mathbf{x^{(2)}} + b) \\ \hat{y}^{(3)} = P(y^{(3)} = 1 | x^{(3)}) &= \sigma(\mathbf{w} \cdot \mathbf{x^{(3)}} + b) \end{aligned}$$

But it turns out that we can slightly modify our original equation Eq. 4.5 to do this much more efficiently. We'll use matrix arithmetic to assign a class to all the examples with one matrix operation!

First, we'll pack all the input feature vectors for each input x into a single input matrix $\mathbf{X}$, where each row i is a row vector consisting of the feature vector for input example $x^{(i)}$ (i.e., the vector $\mathbf{x}^{(i)}$). Assuming each example has f features and weights, $\mathbf{X}$ will therefore be a matrix of shape $[m \times f]$, as follows:

$$\mathbf{X} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_f^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_f^{(2)} \\ x_1^{(3)} & x_2^{(3)} & \dots & x_f^{(3)} \\ \dots & & & \end{bmatrix} \tag{4.12}$$

Now if we introduce $\mathbf{b}$ as a vector of length m which consists of the scalar bias term b repeated m times, $\mathbf{b} = [b, b, ..., b]$, and $\hat{\mathbf{y}} = [\hat{y}^{(1)}, \hat{y}^{(2)} ... \hat{y}^{(m)}]$ as the vector of outputs (one scalar $\hat{y}^{(i)} = P(y^{(i)} = 1)$ for each input $x^{(i)}$ and its feature vector $\mathbf{x}^{(i)}$), and represent the weight vector $\mathbf{w}$ as a column vector, we can compute all the outputs with a single matrix multiplication and one addition:

$$\hat{\mathbf{y}} = \sigma(\mathbf{X}\mathbf{w} + \mathbf{b}) \tag{4.13}$$

You should convince yourself that Eq. 4.13 computes the same thing as our for-loop in Eq. 4.11. For example $\hat{y}^{(1)}$, the first entry of the output vector $\hat{\mathbf{y}}$, will correctly be:

$$\hat{y}^{(1)} = \sigma\left([x_1^{(1)}, x_2^{(1)}, ..., x_f^{(1)}] \cdot [w_1, w_2, ..., w_f] + b\right) \tag{4.14}$$

Note that we had to reorder $\mathbf{X}$ and $\mathbf{w}$ from the order they appeared in Eq. 4.5 to make the multiplications come out properly. Here is Eq. 4.13 again with the shapes shown:

$$\begin{array}{cccccc} \hat{\mathbf{y}} & = & \sigma(\mathbf{X} & \mathbf{w} & + & \mathbf{b}) \\ (m \times 1) & & (m \times f) & (f \times 1) & & (m \times 1) \end{array} \tag{4.15}$$

Modern compilers and compute hardware can compute this matrix operation very efficiently, making the computation much faster, which becomes important when training or testing on very large datasets.

Note by the way that we could have kept $\mathbf{X}$ and $\mathbf{w}$ in the original order (as $\hat{\mathbf{y}} = \sigma(\mathbf{w}\mathbf{X} + \mathbf{b})$) if we had chosen to define $\mathbf{X}$ differently as a matrix of column vectors, one vector for each input example, instead of row vectors, and then it would have shape $[f \times m]$. But we conventionally represent inputs as rows.

4.4 Learning in Logistic Regression

How are the parameters of the model, the weights $\mathbf{w}$ and bias b, learned? Binary logistic regression is an instance of supervised classification in which we know the correct label y (either 0 or 1) for each observation x. What the system produces via Eq. 4.5 is $\hat{y}$, the system's probability of y being 1, which we can think of an estimate of the true y (which is either 1 or 0). We want to learn parameters (meaning $\mathbf{w}$ and b) that make the probability value $\hat{y}$ for each training observation as close as possible

to the true y. That is, if y is 1, then we want the system's estimate $\hat{y} = P(y = 1)$ to be high, i.e. as close to 1 as possible. If y is in fact 0, then we want the system's estimate $\hat{y} = P(y = 1)$ to be low, i.e. as close to 0 as possible.

This requires two components that we foreshadowed in the introduction to the chapter. The first is a metric for how close $\hat{y}$ (the systems's estimate of the probability that the observation is in the positive class) is to the true gold label y. Rather than measure similarity, we usually talk about the opposite of this: the *distance* between the system output and the gold output, and we call this distance the **loss** function or the **cost function**. In the next section we'll introduce the loss function that is commonly used for logistic regression and also for neural networks, the **cross-entropy loss**.

loss

The second thing we need is an optimization algorithm for iteratively updating the weights so as to minimize this loss function. The standard algorithm for this is **gradient descent**; we'll introduce the **stochastic gradient descent** algorithm in the following section.

We'll describe these algorithms for the simpler case of binary logistic regression in the next two sections, and then turn to multinomial logistic regression in Section 4.8.

4.5 The cross-entropy loss function

We need a loss function that expresses, for an observation x, how close the classifier output ($\hat{y} = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$) is to the correct output (y, which is 0 or 1). Recall that $\hat{y}$ expresses the probability that the classifier assigns to observation x being in the positive class 1. So both $\hat{y}$ and y express probabilities, but y is always 0 or 1 but $\hat{y}$ can also lie in between. We'll call this measure of loss or distance:

$$L(\hat{y}, y) = \text{How much } \hat{y} \text{ differs from the true } y \tag{4.16}$$

We do this via a loss function that prefers the correct class labels of the training examples to be *more likely*. This is called **conditional maximum likelihood estimation**: we choose the parameters w, b that **maximize the log probability of the true y labels in the training data** given the observations x. The resulting loss function is the **negative log likelihood loss**, generally called the **cross-entropy loss**.

cross-entropy loss

Let's derive this loss function, applied to a single observation x. We'd like to learn weights that maximize the probability of the correct label $p(y|x)$. Since there are only two discrete outcomes (1 or 0), this is a Bernoulli distribution, and we can express the probability $p(y|x)$ that our classifier produces for one observation as the following (keeping in mind that if $y = 1$, Eq. 4.17 simplifies to $\hat{y}$; if $y = 0$, Eq. 4.17 simplifies to $1 - \hat{y}$):

$$p(y|x) = \hat{y}^y (1 - \hat{y})^{1-y} \tag{4.17}$$

Now we take the log of both sides. This will turn out to be handy mathematically, and doesn't hurt us; whatever values maximize a probability will also maximize the log of the probability:

$$\begin{aligned} \log p(y|x) &= \log \left[\hat{y}^y (1 - \hat{y})^{1-y} \right] \\ &= y \log \hat{y} + (1 - y) \log(1 - \hat{y}) \end{aligned} \tag{4.18}$$

Eq. 4.18 describes a log likelihood that should be maximized. In order to turn this into a loss function (something that we need to minimize), we'll just flip the sign on Eq. 4.18. The result is the cross-entropy loss L_{CE}:

$$L_{\text{CE}}(\hat{y},y) = -\log p(y|x) = -[y\log\hat{y}+(1-y)\log(1-\hat{y})] \tag{4.19}$$

Finally, we can plug in the definition of $\hat{y} = \sigma(\mathbf{w}\cdot\mathbf{x}+b)$:

$$L_{\text{CE}}(\hat{y},y) = -[y\log\sigma(\mathbf{w}\cdot\mathbf{x}+b)+(1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))] \tag{4.20}$$

Let's see if this loss function does the right thing for our example from Fig. 4.2. We want the loss to be smaller if the model's estimate is close to correct, and bigger if the model is confused. So first let's suppose the correct gold label for the sentiment example in Fig. 4.2 is positive, i.e., $y = 1$. In this case our model is doing well, since from Eq. 4.8 it indeed gave the example a higher probability of being positive (.70) than negative (.30). If we plug $\sigma(\mathbf{w}\cdot\mathbf{x}+b) = .70$ and $y = 1$ into Eq. 4.20, the right side of the equation drops out, leading to the following loss (we'll use log to mean natural log when the base is not specified):

$$\begin{aligned} L_{\text{CE}}(\hat{y},y) &= -[y\log\sigma(\mathbf{w}\cdot\mathbf{x}+b)+(1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))] \\ &= -[\log\sigma(\mathbf{w}\cdot\mathbf{x}+b)] \\ &= -\log(.70) \\ &= .36 \end{aligned}$$

By contrast, let's pretend instead that the example in Fig. 4.2 was actually negative, i.e., $y = 0$ (perhaps the reviewer went on to say "But bottom line, the movie is terrible! I beg you not to see it!"). In this case our model is confused and we'd want the loss to be higher. Now if we plug $y = 0$ and $1-\sigma(\mathbf{w}\cdot\mathbf{x}+b) = .30$ from Eq. 4.8 into Eq. 4.20, the left side of the equation drops out:

$$\begin{aligned} L_{\text{CE}}(\hat{y},y) &= -[y\log\sigma(\mathbf{w}\cdot\mathbf{x}+b)+(1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))] \\ &= -[\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))] \\ &= -\log(.30) \\ &= 1.2 \end{aligned}$$

Sure enough, the loss for predicting the correct label (.36) is less than the loss for predicting the incorrect label (1.2).

Why does minimizing this negative log probability do what we want? A perfect classifier would assign probability 1 to the correct outcome ($y = 1$ or $y = 0$) and probability 0 to the incorrect outcome. That means if y equals 1, the higher $\hat{y}$ is (the closer it is to 1), the better the classifier; the lower $\hat{y}$ is (the closer it is to 0), the worse the classifier. If y equals 0, instead, the higher $1-\hat{y}$ is (closer to 1), the better the classifier. The negative log of $\hat{y}$ (if the true y equals 1) or $1-\hat{y}$ (if the true y equals 0) is a convenient loss metric since it goes from 0 (negative log of 1, no loss) to infinity (negative log of 0, infinite loss). This loss function also ensures that as the probability of the correct answer is maximized, the probability of the incorrect answer is minimized; since the two sum to one, any increase in the probability of the correct answer is coming at the expense of the incorrect answer. It's called the cross-entropy loss, because Eq. 4.18 is also the formula for the **cross-entropy** between the true probability distribution y and our estimated distribution $\hat{y}$.

Now we know what we want to minimize; in the next section, we'll see how to find the minimum.

4.6 Gradient Descent

Our goal with gradient descent is to find the optimal weights: minimize the loss function we've defined for the model. In Eq. 4.21 below, we'll explicitly represent the fact that the cross-entropy loss function L_{CE} is parameterized by the weights. In machine learning in general we refer to the parameters being learned as θ; in the case of logistic regression $\theta = \{\mathbf{w}, b\}$. So we'll represent $\hat{y}^{(i)}$ as $f(x^{(i)}; \theta)$ to make the dependence on θ more obvious. The goal is to find the set of weights which minimizes the loss function, averaged over all examples:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} L_{\text{CE}}(f(x^{(i)}; \theta), y^{(i)}) \tag{4.21}$$

How shall we find the minimum of this (or any) loss function? Gradient descent is a method that finds a minimum of a function by figuring out in which direction (in the space of the parameters θ) the function's slope is rising the most steeply, and moving in the opposite direction. The intuition is that if you are hiking in a canyon and trying to descend most quickly down to the river at the bottom, you might look around yourself in all directions, find the direction where the ground is sloping the steepest, and walk downhill in that direction.

convex

For logistic regression, this loss function is conveniently **convex**. A convex function has at most one minimum; there are no local minima to get stuck in, so gradient descent starting from any point is guaranteed to find the minimum. (By contrast, the loss for multi-layer neural networks is non-convex, and gradient descent may get stuck in local minima for neural network training and never find the global optimum.)

Although the algorithm (and the concept of gradient) are designed for direction *vectors*, let's first consider a visualization of the case where the parameter of our system is just a single scalar w, shown in Fig. 4.3.

Given a random initialization of w at some value w^1, and assuming the loss function L happened to have the shape in Fig. 4.3, we need the algorithm to tell us whether at the next iteration we should move left (making w^2 smaller than w^1) or right (making w^2 bigger than w^1) to reach the minimum.

gradient

The gradient descent algorithm answers this question by finding the **gradient** of the loss function at the current point and moving in the opposite direction. The gradient of a function of many variables is a vector pointing in the direction of the greatest increase in a function. The gradient is a multi-variable generalization of the slope, so for a function of one variable like the one in Fig. 4.3, we can informally think of the gradient as the slope. The dotted line in Fig. 4.3 shows the slope of this hypothetical loss function at point $w = w^1$. You can see that the slope of this dotted line is negative. Thus to find the minimum, gradient descent tells us to go in the opposite direction: moving w in a positive direction.

learning rate

The magnitude of the amount to move in gradient descent is the value of the slope $\frac{d}{dw} L(f(x;w),y)$ weighted by a **learning rate** η. A higher (faster) learning rate means that we should move w more on each step. The change we make in our parameter is the learning rate times the gradient (or the slope, in our single-variable example):

$$w^{t+1} = w^t - \eta \frac{d}{dw} L(f(x;w),y) \tag{4.22}$$

Now let's extend the intuition from a function of one scalar variable w to many

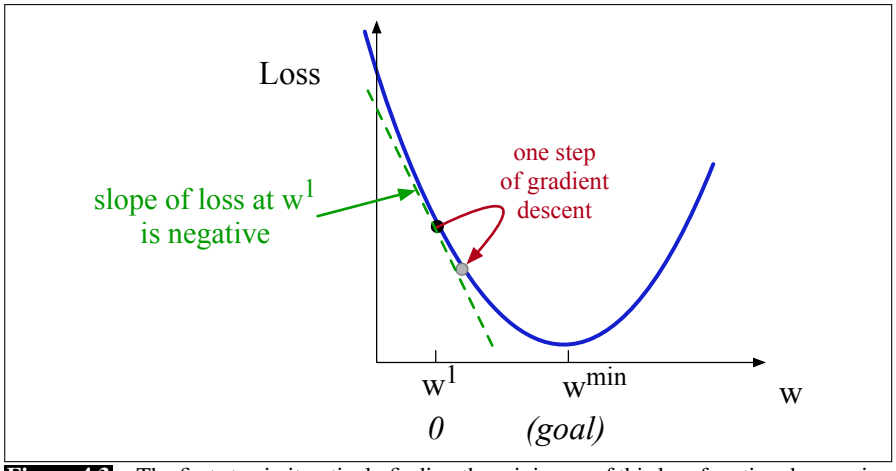


Figure 4.3 The first step in iteratively finding the minimum of this loss function, by moving w in the reverse direction from the slope of the function. Since the slope is negative, we need to move w in a positive direction, to the right. Here superscripts are used for learning steps, so w^1 means the initial value of w (which is 0), w^2 the value at the second step, and so on.

variables, because we don't just want to move left or right, we want to know where in the N-dimensional space (of the N parameters that make up θ) we should move. The **gradient** is just such a vector; it expresses the directional components of the sharpest slope along each of those N dimensions. If we're just imagining two weight dimensions (say for one weight w and one bias b), the gradient might be a vector with two orthogonal components, each of which tells us how much the ground slopes in the w dimension and in the b dimension. Fig. 4.4 shows a visualization of the value of a 2-dimensional gradient vector taken at the red point.

In an actual logistic regression, the parameter vector $\mathbf{w}$ is much longer than 1 or 2, since the input feature vector $\mathbf{x}$ can be quite long, and we need a weight w_i for each x_i. For each dimension/variable w_i in $\mathbf{w}$ (plus the bias b), the gradient will have a component that tells us the slope with respect to that variable. In each dimension w_i, we express the slope as a partial derivative $\frac{\partial}{\partial w_i}$ of the loss function. Essentially we're asking: "How much would a small change in that variable w_i influence the total loss function L?"

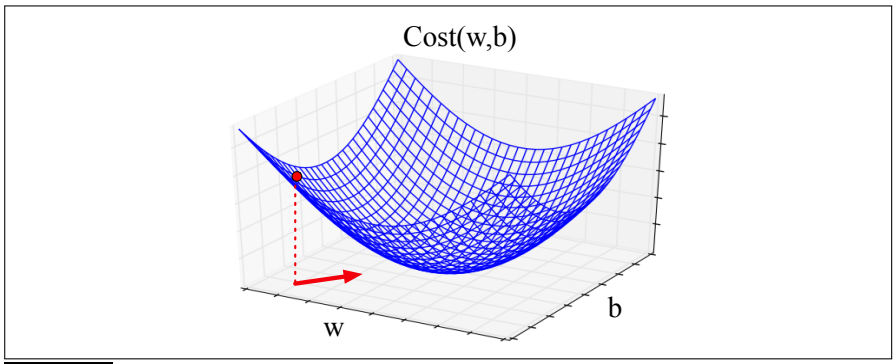


Figure 4.4 Visualization of the gradient vector at the red point in two dimensions w and b, showing a red arrow in the x-y plane pointing in the direction we will go to look for the minimum: the opposite direction of the gradient (recall that the gradient points in the direction of increase not decrease).

Formally, then, the gradient of a multi-variable function f is a vector in which each component expresses the partial derivative of f with respect to one of the variables. We'll use the inverted Greek delta symbol ∇ to refer to the gradient, and represent $\hat{y}$ as $f(x;\theta)$ to make the dependence on θ more obvious:

$$\nabla L(f(x;\theta),y) = \begin{bmatrix} \frac{\partial}{\partial w_1} L(f(x;\theta),y) \\ \frac{\partial}{\partial w_2} L(f(x;\theta),y) \\ \vdots \\ \frac{\partial}{\partial w_n} L(f(x;\theta),y) \\ \frac{\partial}{\partial b} L(f(x;\theta),y) \end{bmatrix} \tag{4.23}$$

The final equation for updating θ based on the gradient is thus

$$\theta^{t+1} = \theta^t - \eta \nabla L(f(x;\theta),y) \tag{4.24}$$

4.6.1 The Gradient for Logistic Regression

In order to update θ, we need a definition for the gradient $\nabla L(f(x;\theta),y)$. Recall that for logistic regression, the cross-entropy loss function is:

$$L_{\text{CE}}(\hat{y},y) = -[y \log \sigma(\mathbf{w}\cdot\mathbf{x}+b) + (1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))] \tag{4.25}$$

It turns out that the derivative of this function for one observation vector x is Eq. 4.26 (the interested reader can see Section 4.15 for the derivation of this equation):

$$\begin{aligned} \frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial w_j} &= [\sigma(\mathbf{w}\cdot\mathbf{x}+b)-y]x_j \\ &= (\hat{y}-y)x_j \end{aligned} \tag{4.26}$$

You'll also sometimes see this equation in the equivalent form:

$$\frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial w_j} = -(y-\hat{y})x_j \tag{4.27}$$

Note in these equations that the gradient with respect to a single weight w_j represents a very intuitive value: the difference between the true y and our estimated $\hat{y} = \sigma(\mathbf{w}\cdot\mathbf{x}+b)$ for that observation, multiplied by the corresponding input value x_j.

By the way, we'll also need a term for the partial derivative with respect to b. That turns out to be:

$$\begin{aligned} \frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial b} &= \sigma(\mathbf{w}\cdot\mathbf{x}+b)-y \\ &= \hat{y}-y \end{aligned} \tag{4.28}$$

4.6.2 The Stochastic Gradient Descent Algorithm

Stochastic gradient descent is an online algorithm that minimizes the loss function by computing its gradient after each training example, and nudging θ in the right direction (the opposite direction of the gradient). (An "online algorithm" is one that processes its input example by example, rather than waiting until it sees the entire input.) Stochastic gradient descent is called **stochastic** because it chooses a single

function STOCHASTIC GRADIENT DESCENT($L()$, $f()$, x, y) **returns** θ
  # where: L is the loss function
  # f is a function parameterized by θ
  # x is the set of training inputs $x^{(1)}$, $x^{(2)}$,..., $x^{(m)}$
  # y is the set of training outputs (labels) $y^{(1)}$, $y^{(2)}$,..., $y^{(m)}$

$\theta \leftarrow 0$ # (or small random values)
repeat til done # see caption
  For each training tuple $(x^{(i)}, y^{(i)})$ (in random order)
    1. Optional (for reporting): # How are we doing on this tuple?
      Compute $\hat{y}^{(i)} = f(x^{(i)}; \theta)$ # What is our estimated output $\hat{y}$?
      Compute the loss $L(\hat{y}^{(i)}, y^{(i)})$ # How far off is $\hat{y}^{(i)}$ from the true output $y^{(i)}$?
    2. $g \leftarrow \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$ # How should we move θ to maximize loss?
    3. $\theta \leftarrow \theta - \eta\, g$ # Go the other way instead
return θ

Figure 4.5 The stochastic gradient descent algorithm. Step 1 (computing the loss) is used mainly to report how well we are doing on the current tuple; we don't need to compute the loss in order to compute the gradient. The algorithm can terminate when it converges (when the gradient norm $< \epsilon$), or when progress halts (for example when the loss starts going up on a held-out set). Weights are initialized to 0 for logistic regression, but to small random values for neural networks, as we'll see in Chapter 6.

random example at a time; in Section 4.6.4 we'll discuss other versions of gradient descent that batch many examples at once. Fig. 4.5 shows the algorithm.

hyperparameter

The learning rate η is a **hyperparameter** that must be adjusted. If it's too high, the learner will take steps that are too large, overshooting the minimum of the loss function. If it's too low, the learner will take steps that are too small, and take too long to get to the minimum. It is common to start with a higher learning rate and then slowly decrease it, so that it is a function of the iteration k of training; the notation η_k can be used to mean the value of the learning rate at iteration k.

We'll discuss hyperparameters in more detail in Chapter 6, but in short, they are a special kind of parameter for any machine learning model. Unlike regular parameters of a model (weights like w and b), which are learned by the algorithm from the training set, hyperparameters are special parameters chosen by the algorithm designer that affect how the algorithm works.

4.6.3 Working through an example

Let's walk through a single step of the gradient descent algorithm. We'll use a simplified version of the example in Fig. 4.2 as it sees a single observation x, whose correct value is $y = 1$ (this is a positive review), and with a feature vector $\mathbf{x} = [x_1, x_2]$ consisting of these two features:

$$x_1 = 3 \quad \text{(count of positive lexicon words)}$$
$$x_2 = 2 \quad \text{(count of negative lexicon words)}$$

Let's assume the initial weights and bias in θ^0 are all set to 0, and the initial learning rate η is 0.1:

$$w_1 = w_2 = b = 0$$
$$\eta = 0.1$$

The single update step requires that we compute the gradient, multiplied by the learning rate

$$\theta^{t+1} = \theta^t - \eta \nabla_\theta L(f(x^{(i)};\theta), y^{(i)})$$

In our mini example there are three parameters, so the gradient vector has 3 dimensions, for w_1, w_2, and b. We can compute the first gradient as follows:

$$\nabla_{w,b} L = \begin{bmatrix} \frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial w_1} \\ \frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial w_2} \\ \frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial b} \end{bmatrix} = \begin{bmatrix} (\sigma(\mathbf{w}\cdot\mathbf{x}+b)-y)x_1 \\ (\sigma(\mathbf{w}\cdot\mathbf{x}+b)-y)x_2 \\ \sigma(\mathbf{w}\cdot\mathbf{x}+b)-y \end{bmatrix} = \begin{bmatrix} (\sigma(0)-1)x_1 \\ (\sigma(0)-1)x_2 \\ \sigma(0)-1 \end{bmatrix} = \begin{bmatrix} -0.5x_1 \\ -0.5x_2 \\ -0.5 \end{bmatrix} = \begin{bmatrix} -1.5 \\ -1.0 \\ -0.5 \end{bmatrix}$$

Now that we have a gradient, we compute the new parameter vector θ^1 by moving θ^0 in the opposite direction from the gradient:

$$\theta^1 = \begin{bmatrix} w_1 \\ w_2 \\ b \end{bmatrix} - \eta \begin{bmatrix} -1.5 \\ -1.0 \\ -0.5 \end{bmatrix} = \begin{bmatrix} .15 \\ .1 \\ .05 \end{bmatrix}$$

So after one step of gradient descent, the weights have shifted to be: $w_1 = .15$, $w_2 = .1$, and $b = .05$.

Note that this observation x happened to be a positive example. We would expect that after seeing more negative examples with high counts of negative words, that the weight $\mathbf{w}_2$ would shift to have a negative value.

4.6.4 Mini-batch training

Stochastic gradient descent is called stochastic because it chooses a single random example at a time, moving the weights so as to improve performance on that single example. That can result in very choppy movements, so it's common to compute the gradient over batches of training instances rather than a single instance.

batch training

For example in **batch training** we compute the gradient over the entire dataset. By seeing so many examples, batch training offers a superb estimate of which direction to move the weights, at the cost of spending a lot of time processing every single example in the training set to compute this perfect direction.

mini-batch

A compromise is **mini-batch** training: we train on a group of m examples (perhaps 512, or 1024) that is less than the whole dataset. (If m is the size of the dataset, then we are doing **batch** gradient descent; if $m = 1$, we are back to doing stochastic gradient descent.) Mini-batch training also has the advantage of computational efficiency. The mini-batches can easily be vectorized, choosing the size of the mini-batch based on the computational resources. This allows us to process all the examples in one mini-batch in parallel and then accumulate the loss, something that's not possible with individual or batch training.

We just need to define mini-batch versions of the cross-entropy loss function we defined in Section 4.5 and the gradient in Section 4.6.1. Let's extend the cross-entropy loss for one example from Eq. 4.19 to mini-batches of size m. We'll continue to use the notation that $x^{(i)}$ and $y^{(i)}$ mean the ith training features and training label,

respectively. We make the assumption that the training examples are independent:

$$\begin{aligned}\log p(\text{training labels}) &= \log \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}) \\ &= \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}) \\ &= -\sum_{i=1}^{m} L_{\text{CE}}(\hat{y}^{(i)}, y^{(i)}) \end{aligned} \tag{4.29}$$

Now the cost function for the mini-batch of m examples is the average loss for each example:

$$\begin{aligned} Cost(\hat{y}, y) &= \frac{1}{m}\sum_{i=1}^{m} L_{\text{CE}}(\hat{y}^{(i)}, y^{(i)}) \\ &= -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log \sigma(\mathbf{w}\cdot\mathbf{x}^{(i)}+b) + (1-y^{(i)}) \log\left(1-\sigma(\mathbf{w}\cdot\mathbf{x}^{(i)}+b)\right) \end{aligned} \tag{4.30}$$

The mini-batch gradient is the average of the individual gradients from Eq. 4.26:

$$\frac{\partial Cost(\hat{y}, y)}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}\left[\sigma(\mathbf{w}\cdot\mathbf{x}^{(i)}+b) - y^{(i)}\right] x_j^{(i)} \tag{4.31}$$

Instead of using the sum notation, we can more efficiently compute the gradient in its matrix form, following the vectorization we saw on page 70, where we have a matrix $\mathbf{X}$ of size $[m \times f]$ representing the m inputs in the batch, and a vector $\mathbf{y}$ of size $[m \times 1]$ representing the correct outputs:

$$\begin{aligned} \frac{\partial Cost(\hat{y}, y)}{\partial \mathbf{w}} &= \frac{1}{m}(\hat{\mathbf{y}} - \mathbf{y})^{\intercal}\mathbf{X} \\ &= \frac{1}{m}(\sigma(\mathbf{X}\mathbf{w} + \mathbf{b}) - \mathbf{y})^{\intercal}\mathbf{X} \end{aligned} \tag{4.32}$$

4.7 Multinomial logistic regression

Sometimes we need more than two classes. Perhaps we might want to do 3-way sentiment classification (positive, negative, or neutral). Or we could be assigning some of the labels we will introduce in Chapter 17, like the part of speech of a word (choosing from 10, 30, or even 50 different parts of speech), or the named entity type of a phrase (choosing from tags like person, location, organization). Or, for large language models, we'll be predicting the next word out of the $|V|$ possible words in the vocabulary, so it's $|V|$-way classification.

multinomial logistic regression

In such cases we use **multinomial logistic regression**, also called **softmax regression** (in older NLP literature you will sometimes see the name **maxent classifier**). In multinomial logistic regression we want to label each observation with a class k from a set of K classes, under the stipulation that only one of these classes is the correct one (sometimes called **hard classification**; an observation can not be in

multiple classes). Let's use the following representation: the output $\mathbf{y}$ for each input $\mathbf{x}$ will be a vector of length K. If class c is the correct class, we'll set $y_c = 1$, and set all the other elements of $\mathbf{y}$ to be 0, i.e., $y_c = 1$ and $y_j = 0 \;\; \forall j \neq c$. A vector like this $\mathbf{y}$, with one value=1 and the rest 0, is called a **one-hot vector**. The job of the classifier is to produce an estimate vector $\hat{\mathbf{y}}$. For each class k, the value $\hat{y}_k$ will be the classifier's estimate of the probability $P(y_k = 1|\mathbf{x})$.

4.7.1 Softmax

softmax

The multinomial logistic classifier uses a generalization of the sigmoid, called the **softmax** function, to compute $p(y_k = 1|\mathbf{x})$. The softmax function takes a vector $\mathbf{z} = [z_1, z_2, ..., z_K]$ of K arbitrary values and maps them to a probability distribution, with each value in the range [0,1], and all the values summing to 1. Like the sigmoid, it is an exponential function.

For a vector $\mathbf{z}$ of dimensionality K, the softmax is defined as:

$$\text{softmax}(z_i) \;=\; \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)} \quad 1 \leq i \leq K \tag{4.33}$$

The softmax of an input vector $\mathbf{z} = [z_1, z_2, ..., z_K]$ is thus a vector itself:

$$\text{softmax}(\mathbf{z}) \;=\; \left[\frac{\exp(z_1)}{\sum_{i=1}^{K} \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{K} \exp(z_i)}, ..., \frac{\exp(z_K)}{\sum_{i=1}^{K} \exp(z_i)} \right] \tag{4.34}$$

The denominator $\sum_{i=1}^{K} \exp(\mathbf{z}_i)$ is used to normalize all the values into probabilities. Thus for example given a vector:

$$\mathbf{z} = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$$

the resulting (rounded) softmax($\mathbf{z}$) is

$$[0.05, 0.09, 0.01, 0.1, 0.74, 0.01]$$

Like the sigmoid, the softmax has the property of squashing values toward 0 or 1. Thus if one of the inputs is larger than the others, it will tend to push its probability toward 1, and suppress the probabilities of the smaller inputs.

Finally, note that, just as for the sigmoid, we refer to $\mathbf{z}$, the vector of scores that is the input to the softmax, as **logits** (see Eq. 4.7).

4.7.2 Applying softmax in logistic regression

When we apply softmax for logistic regression, the input will (just as for the sigmoid) be the dot product between a weight vector $\mathbf{w}$ and an input vector $\mathbf{x}$ (plus a bias). But now we'll need separate weight vectors $\mathbf{w}_k$ and bias b_k for each of the K classes. The probability of each of our output classes $\hat{y}_k$ can thus be computed as:

$$P(y_k = 1|\mathbf{x}) \;=\; \frac{\exp(\mathbf{w}_k \cdot \mathbf{x} + b_k)}{\sum_{j=1}^{K} \exp(\mathbf{w}_j \cdot \mathbf{x} + b_j)} \tag{4.35}$$

The form of Eq. 4.35 makes it seem that we would compute each output separately. Instead, it's more common to set up the equation for more efficient computation by modern vector processing hardware. We'll do this by representing the

set of K weight vectors as a weight matrix W and a bias vector $\mathbf{b}$. Each row k of $\mathbf{W}$ corresponds to the vector of weights w_k. $\mathbf{W}$ thus has shape $[K \times f]$, for K the number of output classes and f the number of input features. The bias vector $\mathbf{b}$ has one value for each of the K output classes. If we represent the weights in this way, we can compute $\hat{\mathbf{y}}$, the vector of output probabilities for each of the K classes, by a single elegant equation:

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{4.36}$$

If you work out the matrix arithmetic, you can see that the estimated score of the first output class $\hat{y}_1$ (before we take the softmax) will correctly turn out to be $\mathbf{w}_1 \cdot \mathbf{x} + b_1$.

prototype

One helpful interpretation of the weight matrix $\mathbf{W}$ is to see each row $\mathbf{w}_k$ as a **prototype** of class k. The weight vector $\mathbf{w}_k$ that is learned represents the class as a kind of template. Since two vectors that are more similar to each other have a higher dot product with each other, the dot product acts as a similarity function. Logistic regression is thus learning a protoype representation for each class, such that incoming vectors are assigned the class k they are most similar to from the K classes (Doumbouya et al., 2025).

Fig. 4.6 shows the difference between binary and multinomial logistic regression by illustrating the weight vector versus weight matrix in the computation of the output class probabilities.

4.7.3 Features in Multinomial Logistic Regression

Features in multinomial logistic regression act like features in binary logistic regression, with the difference mentioned above that we'll need separate weight vectors and biases for each of the K classes. Recall our binary exclamation point feature x_5 from page 67:

$$x_5 = \begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$$

In binary classification a positive weight w_5 on a feature influences the classifier toward $y = 1$ (positive sentiment) and a negative weight influences it toward $y = 0$ (negative sentiment) with the absolute value indicating how important the feature is. For multinomial logistic regression, by contrast, with separate weights for each class, a feature can be evidence for or against each individual class.

In 3-way multiclass sentiment classification, for example, we must assign each document one of the 3 classes $+$, $-$, or 0 (neutral). Now a feature related to exclamation marks might have a negative weight for 0 documents, and a positive weight for $+$ or $-$ documents:

Feature	Definition	$w_{5,+}$	$w_{5,-}$	$w_{5,0}$
$f_5(x)$	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	3.5	3.1	-5.3

Because these feature weights are dependent both on the input text and the output class, we sometimes make this dependence explicit and represent the features themselves as $f(x, y)$: a function of both the input and the class. Using such a notation $f_5(x)$ above could be represented as three features $f_5(x, +)$, $f_5(x, -)$, and $f_5(x, 0)$, each of which has a single weight. We'll use this kind of notation in our description of the CRF in Chapter 17.

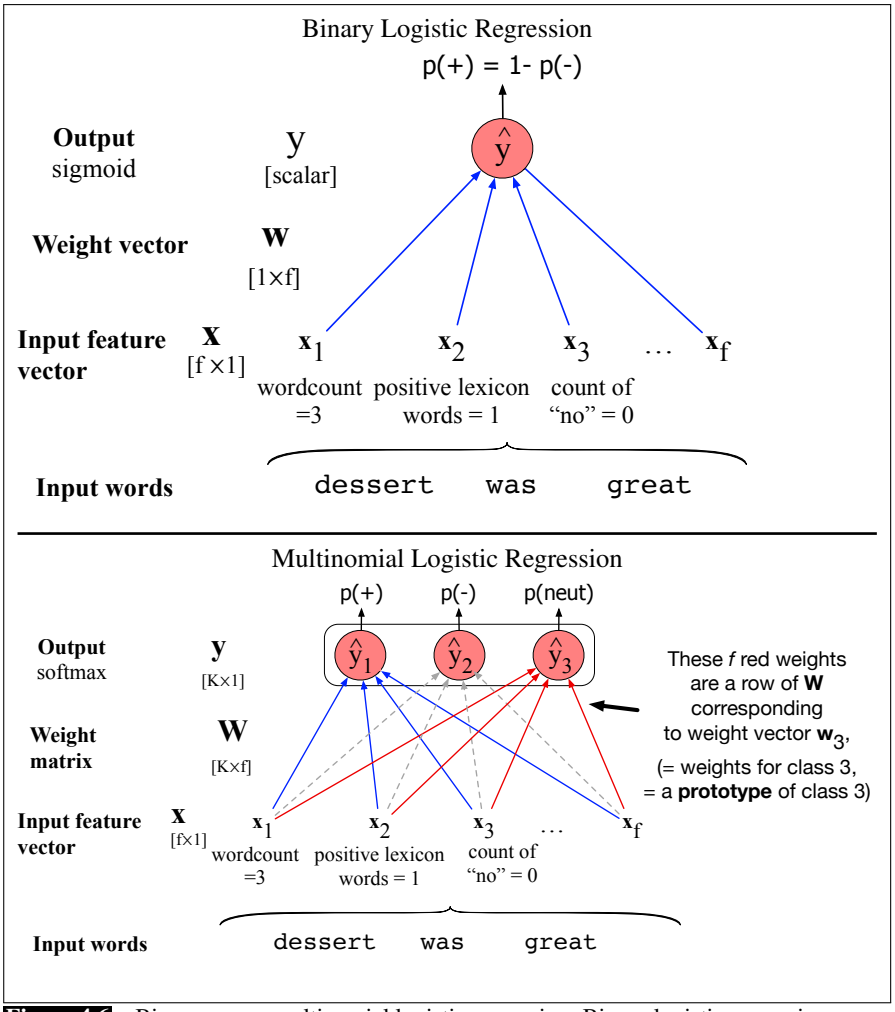


Figure 4.6 Binary versus multinomial logistic regression. Binary logistic regression uses a single weight vector $\mathbf{w}$, and has a scalar output $\hat{y}$. In multinomial logistic regression we have K separate weight vectors corresponding to the K classes, all packed into a single weight matrix $\mathbf{W}$, and a vector output $\hat{\mathbf{y}}$. We omit the biases from both figures for clarity.

4.8 Learning in Multinomial Logistic Regression

The loss function for multinomial logistic regression generalizes the loss function for binary logistic regression from 2 to K classes. Recall that the cross-entropy loss for binary logistic regression (repeated from Eq. 4.19) is:

$$L_{\text{CE}}(\hat{y},y) = -\log p(y|x) \;=\; -[y\log\hat{y} + (1-y)\log(1-\hat{y})] \tag{4.37}$$

The loss function for multinomial logistic regression generalizes the two terms in Eq. 4.37 (one that is non-zero when $y = 1$ and one that is non-zero when $y = 0$) to K terms. As we mentioned above, for multinomial regression we'll represent both $\mathbf{y}$ and $\hat{\mathbf{y}}$ as vectors. The true label $\mathbf{y}$ is a vector with K elements, each corresponding to a class, with $y_c = 1$ if the correct class is c, with all other elements of $\mathbf{y}$ being 0. And our classifier will produce an estimate vector with K elements $\hat{\mathbf{y}}$, each element $\hat{y}_k$ of which represents the estimated probability $p(y_k = 1|\mathbf{x})$.

The loss function for a single example $\mathbf{x}$, generalizing from binary logistic regression, is the sum of the logs of the K output classes, each weighted by the indicator function $\mathbf{y}_k$ (Eq. 4.38). This turns out to be just the negative log probability of the correct class c (Eq. 4.39):

$$\begin{aligned} L_{\text{CE}}(\hat{\mathbf{y}},\mathbf{y}) &= -\sum_{k=1}^{K} y_k \log \hat{y}_k && (4.38)\\ &= -\log \hat{y}_c, \quad \text{(where } c \text{ is the correct class)} && (4.39)\\ &= -\log \hat{p}(y_c = 1|\mathbf{x}) \quad \text{(where } c \text{ is the correct class)} \\ &= -\log \frac{\exp(\mathbf{w_c}\cdot\mathbf{x}+b_c)}{\sum_{j=1}^{K}\exp(\mathbf{w_j}\cdot\mathbf{x}+b_j)} \quad (c \text{ is the correct class}) && (4.40) \end{aligned}$$

How did we get from Eq. 4.38 to Eq. 4.39? Because only one class (let's call it c) is the correct one, the vector $\mathbf{y}$ takes the value 1 only for this value of k, i.e., has $y_c = 1$ and $y_j = 0 \ \ \forall j \neq c$. That means the terms in the sum in Eq. 4.38 will all be 0 except for the term corresponding to the true class c. Hence the cross-entropy loss is simply the log of the output probability corresponding to the correct class, and we therefore also call Eq. 4.39 the **negative log likelihood loss**.

negative log likelihood loss

Of course for gradient descent we don't need the loss, we need its gradient. The gradient for a single example turns out to be very similar to the gradient for binary logistic regression, $(\hat{y}-y)x$, that we saw in Eq. 4.26. Let's consider one piece of the gradient, the derivative for a single weight. For each class k, the weight of the ith element of input $\mathbf{x}$ is $w_{k,i}$. What is the partial derivative of the loss with respect to $w_{k,i}$? This derivative turns out to be just the difference between the true value for the class k (which is either 1 or 0) and the probability the classifier outputs for class k, weighted by the value of the input x_i corresponding to the ith element of the weight vector for class k:

$$\begin{aligned} \frac{\partial L_{\text{CE}}}{\partial w_{k,i}} &= -(y_k - \hat{y}_k)x_i \\ &= -(y_k - p(y_k = 1|x))x_i \\ &= -\left(y_k - \frac{\exp(\mathbf{w_k}\cdot\mathbf{x}+b_k)}{\sum_{j=1}^{K}\exp(\mathbf{w_j}\cdot\mathbf{x}+b_j)}\right)x_i && (4.41) \end{aligned}$$

We'll return to this case of the gradient for softmax regression when we introduce neural networks in Chapter 6, and at that time we'll also discuss the derivation of this gradient in equations Eq. 6.35–Eq. 6.43.

4.9 Evaluation: Precision, Recall, F-measure

To introduce the methods for evaluating text classification, let's first consider some simple binary *detection* tasks. For example, in spam detection, our goal is to label every text as being in the spam category ("positive") or not in the spam category ("negative"). For each item (email document) we therefore need to know whether our system called it spam or not. We also need to know whether the email is actually spam or not, i.e. the human-defined labels for each document that we are trying to match. We will refer to these human labels as the **gold labels**.

gold labels

Or imagine you're the CEO of the *Delicious Pie Company* and you need to know what people are saying about your pies on social media, so you build a system that detects tweets concerning Delicious Pie. Here the positive class is tweets about Delicious Pie and the negative class is all other tweets.

In both cases, we need a metric for knowing how well our spam detector (or pie-tweet-detector) is doing. To evaluate any system for detecting things, we start by building a **confusion matrix** like the one shown in Fig. 4.7. A confusion matrix is a table for visualizing how an algorithm performs with respect to the human gold labels, using two dimensions (system output and gold labels), and each cell labeling a set of possible outcomes. In the spam detection case, for example, true positives are documents that are indeed spam (indicated by human-created gold labels) that our system correctly said were spam. False negatives are documents that are indeed spam but our system incorrectly labeled as non-spam.

confusion matrix

To the bottom right of the table is the equation for *accuracy*, which asks what percentage of all the observations (for the spam or pie examples that means all emails or tweets) our system labeled correctly. Although accuracy might seem a natural metric, we generally don't use it for text classification tasks. That's because accuracy doesn't work well when the classes are unbalanced (as indeed they are with spam, which is a large majority of email, or with tweets, which are mainly not about pie).

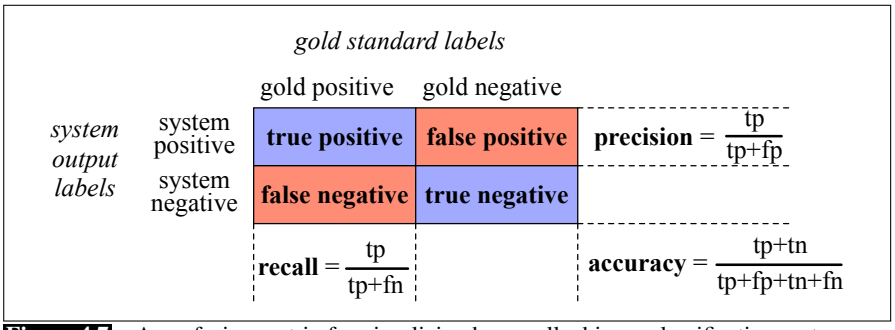


Figure 4.7 A confusion matrix for visualizing how well a binary classification system performs against gold standard labels.

To make this more explicit, imagine that we looked at a million tweets, and let's say that only 100 of them are discussing their love (or hatred) for our pie, while the other 999,900 are tweets about something completely unrelated. Imagine a simple classifier that stupidly classified every tweet as "not about pie". This classifier would have 999,900 true negatives and only 100 false negatives for an accuracy of 999,900/1,000,000 or 99.99%! What an amazing accuracy level! Surely we should be happy with this classifier? But of course this fabulous 'no pie' classifier would be completely useless, since it wouldn't find a single one of the customer comments we are looking for. In other words, accuracy is not a good metric when the goal is to discover something that is rare, or at least not completely balanced in frequency, which is a very common situation in the world.

precision

That's why instead of accuracy we generally turn to two other metrics shown in Fig. 4.7: **precision** and **recall**. **Precision** measures the percentage of the items that the system detected (i.e., the system labeled as positive) that are in fact positive (i.e., are positive according to the human gold labels). Precision is defined as

$$\textbf{Precision} = \frac{\text{true positives}}{\text{true positives + false positives}}$$

recall

Recall measures the percentage of items actually present in the input that were correctly identified by the system. Recall is defined as

$$\textbf{Recall} = \frac{\text{true positives}}{\text{true positives + false negatives}}$$

Precision and recall will help solve the problem with the useless "nothing is pie" classifier. This classifier, despite having a fabulous accuracy of 99.99%, has a terrible recall of 0 (since there are no true positives, and 100 false negatives, the recall is 0/100). You should convince yourself that the precision at finding relevant tweets is equally problematic. Thus precision and recall, unlike accuracy, emphasize true positives: finding the things that we are supposed to be looking for.

F-measure

There are many ways to define a single metric that incorporates aspects of both precision and recall. The simplest of these combinations is the **F-measure** (van Rijsbergen, 1975) , defined as:

$$F_\beta = \frac{(\beta^2+1)PR}{\beta^2 P + R}$$

The β parameter differentially weights the importance of recall and precision, based perhaps on the needs of an application. Values of $\beta > 1$ favor recall, while values of $\beta < 1$ favor precision. When $\beta = 1$, precision and recall are equally balanced; this is the most frequently used metric, and is called $F_{\beta=1}$ or just F_1:

F1

$$F_1 = \frac{2PR}{P+R} \tag{4.42}$$

F-measure comes from a weighted harmonic mean of precision and recall. The harmonic mean of a set of numbers is the reciprocal of the arithmetic mean of reciprocals:

$$\text{HarmonicMean}(a_1, a_2, a_3, a_4, ..., a_n) = \frac{n}{\frac{1}{a_1}+\frac{1}{a_2}+\frac{1}{a_3}+...+\frac{1}{a_n}} \tag{4.43}$$

and hence F-measure is

$$F = \frac{1}{\alpha\frac{1}{P}+(1-\alpha)\frac{1}{R}} \quad \text{or} \left(\text{with } \beta^2 = \frac{1-\alpha}{\alpha}\right) \quad F = \frac{(\beta^2+1)PR}{\beta^2 P + R} \tag{4.44}$$

Harmonic mean is used because the harmonic mean of two values is closer to the minimum of the two values than the arithmetic mean is. Thus it weighs the lower of the two numbers more heavily, which is more conservative in this situation.

4.9.1 Evaluating with more than two classes

The simple example we gave above in defining precision and recall involved text classification with only two classes. But as we saw in Section 4.7, lots of NLP classification tasks have more than two classes. Even for sentiment analysis we often have 3 classes (positive, negative, neutral) and many more classes are common in tasks like text classification or emotion detection, and so on.

To deal with more than two classes we'll need to slightly modify our definitions of precision and recall. Consider the sample confusion matrix for a hypothetical 3-way *one-of* email categorization decision (urgent, normal, spam) shown in Fig. 4.8.

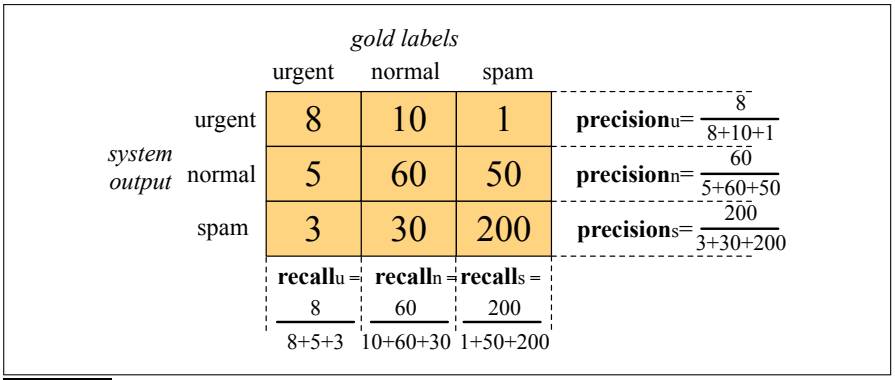


Figure 4.8 Confusion matrix for a three-class categorization task, showing for each pair of classes (c_1, c_2), how many documents from c_1 were (in)correctly assigned to c_2.

The matrix shows, for example, that the system mistakenly labeled one spam document as urgent, and we have shown how to compute a distinct precision and recall value for each class. In order to derive a single metric that tells us how well the system is doing, we can combine these values in two ways. In **macroaveraging**, we compute the performance for each class, and then average over classes. In **microaveraging**, we collect the decisions for all classes into a single confusion matrix, and then compute precision and recall from that table. Fig. 4.9 shows the confusion matrix for each class separately, and shows the computation of microaveraged and macroaveraged precision.

macroaveraging

microaveraging

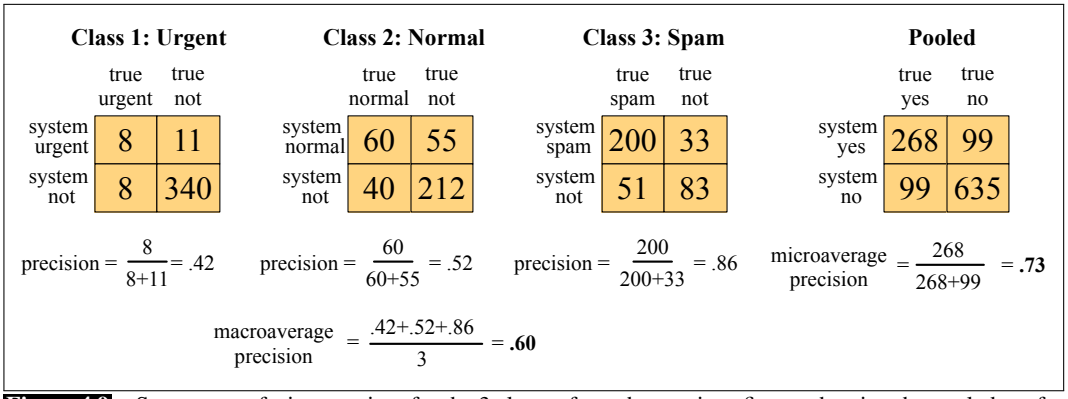


Figure 4.9 Separate confusion matrices for the 3 classes from the previous figure, showing the pooled confusion matrix and the microaveraged and macroaveraged precision.

As the figure shows, a microaverage is dominated by the more frequent class (in this case spam), since the counts are pooled. The macroaverage better reflects the statistics of the smaller classes, and so is more appropriate when performance on all the classes is equally important.

4.10 Test sets and Cross-validation

The training and testing procedure for text classification follows what we saw with language modeling (Section 3.2): we use the training set to train the model, then use the **development test set** (also called a **devset**) to perhaps tune some parameters,

development test set

devset

and in general decide what the best model is. Once we come up with what we think is the best model, we run it on the (hitherto unseen) test set to report its performance.

While the use of a devset avoids overfitting the test set, having a fixed training set, devset, and test set creates another problem: in order to save lots of data for training, the test set (or devset) might not be large enough to be representative. Wouldn't it be better if we could somehow use all our data for training and still use all our data for test? We can do this by **cross-validation**.

cross-validation

In cross-validation, we choose a number k, and partition our data into k disjoint subsets called **folds**. Now we choose one of those k folds as a test set, train our classifier on the remaining $k-1$ folds, and then compute the error rate on the test set. Then we repeat with another fold as the test set, again training on the other $k-1$ folds. We do this sampling process k times and average the test set error rate from these k runs to get an average error rate. If we choose $k = 10$, we would train 10 different models (each on 90% of our data), test the model 10 times, and average these 10 values. This is called **10-fold cross-validation**.

folds

10-fold cross-validation

The only problem with cross-validation is that because all the data is used for testing, we need the whole corpus to be blind; we can't examine any of the data to suggest possible features and in general see what's going on, because we'd be peeking at the test set, and such cheating would cause us to overestimate the performance of our system. However, looking at the corpus to understand what's going on is important in designing NLP systems! What to do? For this reason, it is common to create a fixed training set and test set, then do 10-fold cross-validation inside the training set, but compute error rate the normal way in the test set, as shown in Fig. 4.10.

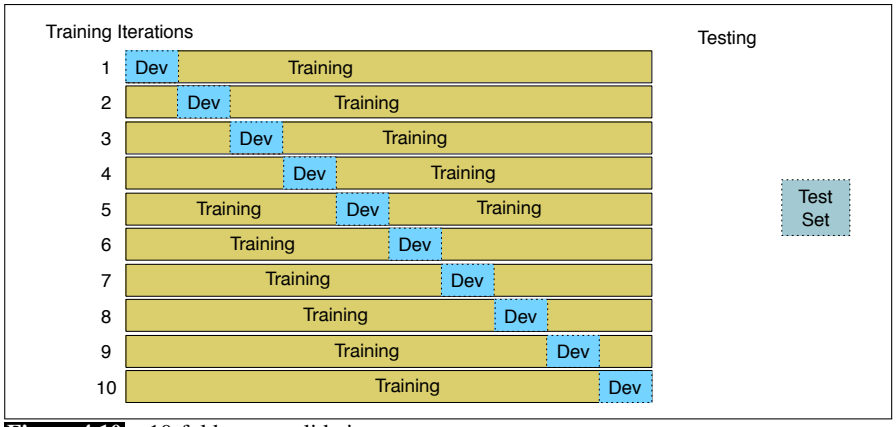


Figure 4.10 10-fold cross-validation

4.11 Statistical Significance Testing

In building systems we often need to compare the performance of two systems. How can we know if the new system we just built is better than our old one? Or better than some other system described in the literature? This is the domain of statistical hypothesis testing, and in this section we introduce tests for statistical significance for NLP classifiers, drawing especially on the work of Dror et al. (2020) and Berg-Kirkpatrick et al. (2012).

Suppose we're comparing the performance of classifiers A and B on a metric M such as F_1, or accuracy. Perhaps we want to know if our new sentiment classifier A gets a higher F_1 score than our previous sentiment classifier B on a particular test set x. Let's call $M(A,x)$ the score that system A gets on test set x, and $\delta(x)$ the performance difference between A and B on x:

$$\delta(x) = M(A,x) - M(B,x) \tag{4.45}$$

effect size

We would like to know if $\delta(x) > 0$, meaning that our new classifier A has a higher F_1 than our old classifier B on x. $\delta(x)$ is called the **effect size**; a bigger δ means that A seems to be way better than B; a small δ means A seems to be only a little better.

Why don't we just check if $\delta(x)$ is positive? Suppose we do, and we find that the F_1 score of A is higher than B's by .04. Can we be certain that A is better? We cannot! That's because A might just be accidentally better than B on this particular x. We need something more: we want to know if A's superiority over B is likely to hold again if we checked another test set x', or under some other set of circumstances.

In the paradigm of statistical hypothesis testing, we test this by formalizing two hypotheses.

$$\begin{aligned} H_0 &: \delta(x) \leq 0 \\ H_1 &: \delta(x) > 0 \end{aligned} \tag{4.46}$$

null hypothesis

The hypothesis H_0, called the **null hypothesis**, supposes that $\delta(x)$ is actually negative or zero, meaning that A is not better than B. We would like to know if we can confidently rule out this hypothesis, and instead support H_1, that A is better.

We do this by creating a random variable X ranging over all test sets. Now we ask how likely is it, if the null hypothesis H_0 was correct, that among these test sets we would encounter the value of $\delta(x)$ that we found, if we repeated the experiment a great many times. We formalize this likelihood as the **p-value**: the probability, assuming the null hypothesis H_0 is true, of seeing the $\delta(x)$ that we saw or one even greater

p-value

$$P(\delta(X) \geq \delta(x) | H_0 \text{ is true}) \tag{4.47}$$

So in our example, this p-value is the probability that we would see $\delta(x)$ assuming A is **not** better than B. If $\delta(x)$ is huge (let's say A has a very respectable F_1 of .9 and B has a terrible F_1 of only .2 on x), we might be surprised, since that would be extremely unlikely to occur if H_0 were in fact true, and so the p-value would be low (unlikely to have such a large δ if A is in fact not better than B). But if $\delta(x)$ is very small, it might be less surprising to us even if H_0 were true and A is not really better than B, and so the p-value would be higher.

A very small p-value means that the difference we observed is very unlikely under the null hypothesis, and we can reject the null hypothesis. What counts as very small? It is common to use values like .05 or .01 as the thresholds. A value of .01 means that if the p-value (the probability of observing the δ we saw assuming H_0 is true) is less than .01, we reject the null hypothesis and assume that A is indeed better than B. We say that a result (e.g., "A is better than B") is **statistically significant** if the δ we saw has a probability that is below the threshold and we therefore reject this null hypothesis.

statistically significant

How do we compute this probability we need for the p-value? In NLP we generally don't use simple parametric tests like t-tests or ANOVAs that you might be familiar with. Parametric tests make assumptions about the distributions of the test

statistic (such as normality) that don't generally hold in our cases. So in NLP we usually use non-parametric tests based on sampling: we artificially create many versions of the experimental setup. For example, if we had lots of different test sets x' we could just measure all the $\delta(x')$ for all the x'. That gives us a distribution. Now we set a threshold (like .01) and if we see in this distribution that 99% or more of those deltas are smaller than the delta we observed, i.e., that p-value(x)—the probability of seeing a $\delta(x)$ as big as the one we saw—is less than .01, then we can reject the null hypothesis and agree that $\delta(x)$ was a sufficiently surprising difference and A is really a better algorithm than B.

approximate randomization

There are two common non-parametric tests used in NLP: **approximate randomization** (Noreen, 1989) and the **bootstrap test**. We will describe bootstrap below, showing the paired version of the test, which again is most common in NLP.

paired

Paired tests are those in which we compare two sets of observations that are aligned: each observation in one set can be paired with an observation in another. This happens naturally when we are comparing the performance of two systems on the same test set; we can pair the performance of system A on an individual observation x_i with the performance of system B on the same x_i.

4.11.1 The Paired Bootstrap Test

bootstrap test

The **bootstrap test** (Efron and Tibshirani, 1993) can apply to any metric; from precision, recall, or F1 to the BLEU metric used in machine translation. The word

bootstrapping

bootstrapping refers to repeatedly drawing large numbers of samples with replacement (called **bootstrap samples**) from an original set. The intuition of the bootstrap test is that we can create many virtual test sets from an observed test set by repeatedly sampling from it. The method only makes the assumption that the sample is representative of the population.

Consider a tiny text classification example with a test set x of 10 documents. The first row of Fig. 4.11 shows the results of two classifiers (A and B) on this test set. Each document is labeled by one of the four possibilities (A and B both right, both wrong, A right and B wrong, A wrong and B right). A slash through a letter ($\not{B}$) means that that classifier got the answer wrong. On the first document both A and B get the correct class (AB), while on the second document A got it right but B got it wrong (A$\not{B}$). If we assume for simplicity that our metric is accuracy, A has an accuracy of .70 and B of .50, so $\delta(x)$ is .20.

Now we create a large number b (perhaps 10^5) of virtual test sets $x^{(i)}$, each of size $n = 10$. Fig. 4.11 shows a couple of examples. To create each virtual test set $x^{(i)}$, we repeatedly ($n = 10$ times) select a cell from row x with replacement. For example, to create the first cell of the first virtual test set $x^{(1)}$, if we happened to randomly select the second cell of the x row, we would copy the value A$\not{B}$ into our new cell, and move on to create the second cell of $x^{(1)}$, each time sampling (randomly choosing) from the original x with replacement.

Now that we have the b test sets, providing a sampling distribution, we can do statistics on how often A has an accidental advantage. There are various ways to compute this advantage; here we follow the version laid out in Berg-Kirkpatrick et al. (2012). Assuming H_0 (A isn't better than B), we would expect that $\delta(X)$, estimated over many test sets, would be zero or negative; a much higher value would be surprising, since H_0 specifically assumes A isn't better than B. To measure exactly how surprising our observed $\delta(x)$ is, we would in other circumstances compute the p-value by counting over many test sets how often $\delta(x^{(i)})$ exceeds the expected zero

	1	2	3	4	5	6	7	8	9	10	A%	B%	$\delta()$
x	AB	A~~B~~	AB	~~A~~B	A~~B~~	~~A~~B	A~~B~~	AB	~~A~~B	A~~B~~	.70	.50	.20
$x^{(1)}$	A~~B~~	AB	A~~B~~	~~A~~B	~~A~~B	A~~B~~	~~A~~B	AB	~~A~~B	AB	.60	.60	.00
$x^{(2)}$	A~~B~~	AB	~~A~~B	~~A~~B	~~A~~B	AB	~~A~~B	A~~B~~	AB	AB	.60	.70	-.10
...													
$x^{(b)}$													

Figure 4.11 The paired bootstrap test: Examples of b pseudo test sets $x^{(i)}$ being created from an initial true test set x. Each pseudo test set is created by sampling $n = 10$ times with replacement; thus an individual sample is a single cell, a document with its gold label and the correct or incorrect performance of classifiers A and B. Of course real test sets don't have only 10 examples, and b needs to be large as well.

value by $\delta(x)$ or more:

$$\text{p-value}(x) = \frac{1}{b}\sum_{i=1}^{b} \mathbb{1}\left(\delta(x^{(i)}) - \delta(x) \geq 0\right)$$

(We use the notation $\mathbb{1}(x)$ to mean "1 if x is true, and 0 otherwise".) However, although it's generally true that the expected value of $\delta(X)$ over many test sets, (again assuming A isn't better than B) is 0, this **isn't** true for the bootstrapped test sets we created. That's because we didn't draw these samples from a distribution with 0 mean; we happened to create them from the original test set x, which happens to be biased (by .20) in favor of A. So to measure how surprising is our observed $\delta(x)$, we actually compute the p-value by counting over many test sets how often $\delta(x^{(i)})$ exceeds the expected value of $\delta(x)$ by $\delta(x)$ or more:

$$\begin{aligned}\text{p-value}(x) &= \frac{1}{b}\sum_{i=1}^{b} \mathbb{1}\left(\delta(x^{(i)}) - \delta(x) \geq \delta(x)\right) \\ &= \frac{1}{b}\sum_{i=1}^{b} \mathbb{1}\left(\delta(x^{(i)}) \geq 2\delta(x)\right) \end{aligned} \tag{4.48}$$

So if for example we have 10,000 test sets $x^{(i)}$ and a threshold of .01, and in only 47 of the test sets do we find that A is accidentally better $\delta(x^{(i)}) \geq 2\delta(x)$, the resulting p-value of .0047 is smaller than .01, indicating that the delta we found, $\delta(x)$ is indeed sufficiently surprising and unlikely to have happened by accident, and we can reject the null hypothesis and conclude A is better than B.

The full algorithm for the bootstrap is shown in Fig. 4.12. It is given a test set x, a number of samples b, and counts the percentage of the b bootstrap test sets in which $\delta(x^{(i)}) > 2\delta(x)$. This percentage then acts as a one-sided empirical p-value.

4.12 Avoiding Harms in Classification

representational harms

It is important to avoid harms that may result from classifiers. One class of harms is **representational harms** (Crawford 2017, Blodgett et al. 2020), harms caused by a system that demeans a social group, for example by perpetuating negative stereotypes about them. For example Kiritchenko and Mohammad (2018) examined the performance of 200 sentiment analysis systems on pairs of sentences that

function BOOTSTRAP(test set x, num of samples b) **returns** p-*value*(x)

Calculate $\delta(x)$ # how much better does algorithm A do than B on x
$s = 0$
for $i = 1$ **to** b **do**
  for $j = 1$ **to** n **do** # Draw a bootstrap sample $x^{(i)}$ of size n
    Select a member of x at random and add it to $x^{(i)}$
  Calculate $\delta(x^{(i)})$ # how much better does algorithm A do than B on $x^{(i)}$
  $s \leftarrow s + 1$ **if** $\delta(x^{(i)}) \geq 2\delta(x)$
p-value(x) $\approx \frac{s}{b}$ # on what % of the b samples did algorithm A beat expectations?
return p-value(x) # if very few did, our observed δ is probably not accidental

Figure 4.12 A version of the paired bootstrap algorithm after Berg-Kirkpatrick et al. (2012).

were identical except for containing either a common African American first name (like *Shaniqua*) or a common European American first name (like *Stephanie*), chosen from the Caliskan et al. (2017) study discussed in Chapter 5. They found that most systems assigned lower sentiment and more negative emotion to sentences with African American names, reflecting and perpetuating stereotypes that associate African Americans with negative emotions (Popp et al., 2003).

toxicity detection

In other tasks classifiers may lead to both representational harms and other harms, such as silencing. For example the important text classification task of **toxicity detection** is the task of detecting hate speech, abuse, harassment, or other kinds of toxic language. While the goal of such classifiers is to help reduce societal harm, toxicity classifiers can themselves cause harms. For example, researchers have shown that some widely used toxicity classifiers incorrectly flag as being toxic sentences that are non-toxic but simply mention identities like women (Park et al., 2018), blind people (Hutchinson et al., 2020) or gay people (Dixon et al., 2018; Dias Oliva et al., 2021), or simply use linguistic features characteristic of varieties like African-American Vernacular English (Sap et al. 2019, Davidson et al. 2019). Such false positive errors could lead to the silencing of discourse by or about these groups.

These model problems can be caused by biases or other problems in the training data; in general, machine learning systems replicate and even amplify the biases in their training data. But these problems can also be caused by the labels (for example due to biases in the human labelers), by the resources used (like lexicons, or model components like pretrained embeddings), or even by model architecture (like what the model is trained to optimize). While the mitigation of these biases (for example by carefully considering the training data sources) is an important area of research, we currently don't have general solutions. For this reason it's important, when introducing any NLP model, to study these kinds of factors and make them clear. One way to do this is by releasing a **model card** (Mitchell et al., 2019) for each version of a model. A model card documents a machine learning model with information like:

model card

- training algorithms and parameters
- training data sources, motivation, and preprocessing
- evaluation data sources, motivation, and preprocessing
- intended use and users
- model performance across different demographic or other groups and envi-

ronmental situations

4.13 Interpreting models

interpretable

Often we want to know more than just the correct classification of an observation. We want to know why the classifier made the decision it did. That is, we want our decision to be **interpretable**. Interpretability can be hard to define strictly, but the core idea is that as humans we should know why our algorithms reach the conclusions they do. Because the features to logistic regression are often human-designed, one way to understand a classifier's decision is to understand the role each feature plays in the decision. Logistic regression can be combined with statistical tests (the likelihood ratio test, or the Wald test); investigating whether a particular feature is significant by one of these tests, or inspecting its magnitude (how large is the weight w associated with the feature?) can help us interpret why the classifier made the decision it makes. This is enormously important for building transparent models.

Furthermore, in addition to its use as a classifier, logistic regression in NLP and many other fields is widely used as an analytic tool for testing hypotheses about the effect of various explanatory variables (features). In text classification, perhaps we want to know if logically negative words (*no, not, never*) are more likely to be associated with negative sentiment, or if negative reviews of movies are more likely to discuss the cinematography. However, in doing so it's necessary to control for potential confounds: other factors that might influence sentiment (the movie genre, the year it was made, perhaps the length of the review in words). Or we might be studying the relationship between NLP-extracted linguistic features and non-linguistic outcomes (hospital readmissions, political outcomes, or product sales), but need to control for confounds (the age of the patient, the county of voting, the brand of the product). In such cases, logistic regression allows us to test whether some feature is associated with some outcome above and beyond the effect of other features.

4.14 Advanced: Regularization

Numquam ponenda est pluralitas sine necessitate
'Plurality should never be proposed unless needed'
William of Occam

overfitting
generalize
regularization

There is a problem with learning weights that make the model perfectly match the training data. If a feature is perfectly predictive of the outcome because it happens to only occur in one class, it will be assigned a very high weight. The weights for features will attempt to perfectly fit details of the training set, in fact too perfectly, modeling noisy factors that just accidentally correlate with the class. This problem is called **overfitting**. A good model should be able to **generalize** well from the training data to the unseen test set, but a model that overfits will have poor generalization.

To avoid overfitting, a new **regularization** term $R(\theta)$ is added to the loss function in Eq. 4.21, resulting in the following loss for a batch of m examples (slightly

rewritten from Eq. 4.21 to be maximizing log probability rather than minimizing loss, and removing the $\frac{1}{m}$ term which doesn't affect the argmax):

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) - \alpha R(\theta) \tag{4.49}$$

The new regularization term $R(\theta)$ is used to penalize large weights. Thus a setting of the weights that matches the training data perfectly— but uses many weights with high values to do so—will be penalized more than a setting that matches the data a little less well, but does so using smaller weights. The higher the regularization strength parameter α, the lower the model's weights will be, reducing its reliance on the training data. Note that regularization is not normally applied to the bias term, which acts as a kind of threshold that helps deal with uncentered data and class priors.

L2 regularization

There are two common ways to compute this regularization term $R(\theta)$. **L2 regularization** is a quadratic function of the weight values named because it uses the (square of the) L2 norm of the weight values. The L2 norm, $||\theta||_2$, is the same as the **Euclidean distance** of the vector θ from the origin. If θ consists of n weights, then:

$$R(\theta) = ||\theta||_2^2 = \sum_{j=1}^{n} \theta_j^2 \tag{4.50}$$

The L2 regularized loss function becomes:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \left[\sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)};\theta) \right] - \alpha \sum_{j=1}^{n} \theta_j^2 \tag{4.51}$$

L1 regularization

L1 regularization is a linear function of the weight values, named after the L1 norm $||W||_1$, the sum of the absolute values of the weights, or **Manhattan distance** (the Manhattan distance is the distance you'd have to walk between two points in a city with a street grid like New York):

$$R(\theta) = ||\theta||_1 = \sum_{i=1}^{n} |\theta_i| \tag{4.52}$$

The L1 regularized loss function becomes:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \left[\sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)};\theta) \right] - \alpha \sum_{j=1}^{n} |\theta_j| \tag{4.53}$$

lasso
ridge

These kinds of regularization come from statistics, where L1 regularization is called **lasso regression** (Tibshirani, 1996) and L2 regularization is called **ridge regression**, and both are commonly used in language processing. L2 regularization is easier to optimize because of its simple derivative (the derivative of θ^2 is just 2θ), while L1 regularization is more complex (the derivative of $|\theta|$ is non-continuous at zero). But while L2 prefers weight vectors with many small weights, L1 prefers sparse solutions with some larger weights but many more weights set to zero. Thus L1 regularization leads to much sparser weight vectors, that is, far fewer features. Again, for both these types of regularization we general ignore the bias term and only consider the other weights.

Both L1 and L2 regularization have Bayesian interpretations as constraints on the prior of how weights should look. L1 regularization can be viewed as a Laplace prior on the weights. L2 regularization corresponds to assuming that weights are distributed according to a Gaussian distribution with mean $\mu = 0$. In a Gaussian or normal distribution, the further away a value is from the mean, the lower its probability (scaled by the variance σ). By using a Gaussian prior on the weights, we are saying that weights prefer to have the value 0. A Gaussian for a weight θ_j is

$$\frac{1}{\sqrt{2\pi\sigma_j^2}} \exp\left(-\frac{(\theta_j - \mu_j)^2}{2\sigma_j^2}\right) \tag{4.54}$$

If we multiply each weight by a Gaussian prior on the weight, we are thus maximizing the following constraint:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \prod_{i=1}^{m} P(y^{(i)}|x^{(i)}) \times \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi\sigma_j^2}} \exp\left(-\frac{(\theta_j - \mu_j)^2}{2\sigma_j^2}\right) \tag{4.55}$$

which in log space, with $\mu = 0$, and assuming $2\sigma^2 = 1$, corresponds to

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) - \alpha \sum_{j=1}^{n} \theta_j^2 \tag{4.56}$$

which is in the same form as Eq. 4.51.

4.15 Advanced: Deriving the Gradient Equation

In this section we give the derivation of the gradient of the cross-entropy loss function L_{CE} for logistic regression. Let's start with some quick calculus refreshers. First, the derivative of $\ln(x)$:

$$\frac{d}{dx}\ln(x) = \frac{1}{x} \tag{4.57}$$

Second, the (very elegant) derivative of the sigmoid:

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z)) \tag{4.58}$$

chain rule

Finally, the **chain rule** of derivatives. Suppose we are computing the derivative of a composite function $f(x) = u(v(x))$. The derivative of $f(x)$ is the derivative of $u(x)$ with respect to $v(x)$ times the derivative of $v(x)$ with respect to x:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx} \tag{4.59}$$

First, we want to know the derivative of the loss function with respect to a single weight w_j (we'll need to compute it for each weight, and for the bias):

$$\begin{aligned}\frac{\partial L_{\text{CE}}}{\partial w_j} &= \frac{\partial}{\partial w_j}-[y\log\sigma(\mathbf{w}\cdot\mathbf{x}+b)+(1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))]\\ &= -\left[\frac{\partial}{\partial w_j}y\log\sigma(\mathbf{w}\cdot\mathbf{x}+b)+\frac{\partial}{\partial w_j}(1-y)\log[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]\right]\end{aligned} \tag{4.60}$$

Next, using the chain rule, and relying on the derivative of log:

$$\frac{\partial L_{\text{CE}}}{\partial w_j} = -\frac{y}{\sigma(\mathbf{w}\cdot\mathbf{x}+b)}\frac{\partial}{\partial w_j}\sigma(\mathbf{w}\cdot\mathbf{x}+b)-\frac{1-y}{1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)}\frac{\partial}{\partial w_j}[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)] \tag{4.61}$$

Rearranging terms:

$$\frac{\partial L_{\text{CE}}}{\partial w_j} = -\left[\frac{y}{\sigma(\mathbf{w}\cdot\mathbf{x}+b)}-\frac{1-y}{1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)}\right]\frac{\partial}{\partial w_j}\sigma(\mathbf{w}\cdot\mathbf{x}+b)$$

And now plugging in the derivative of the sigmoid, and using the chain rule one more time, we end up with Eq. 4.62:

$$\begin{aligned}\frac{\partial L_{\text{CE}}}{\partial w_j} &= -\left[\frac{y-\sigma(\mathbf{w}\cdot\mathbf{x}+b)}{\sigma(\mathbf{w}\cdot\mathbf{x}+b)[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]}\right]\sigma(\mathbf{w}\cdot\mathbf{x}+b)[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]\frac{\partial(\mathbf{w}\cdot\mathbf{x}+b)}{\partial w_j}\\ &= -\left[\frac{y-\sigma(\mathbf{w}\cdot\mathbf{x}+b)}{\sigma(\mathbf{w}\cdot\mathbf{x}+b)[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]}\right]\sigma(\mathbf{w}\cdot\mathbf{x}+b)[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]x_j\\ &= -[y-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]x_j\\ &= [\sigma(\mathbf{w}\cdot\mathbf{x}+b)-y]x_j\end{aligned} \tag{4.62}$$

4.16 Summary

This chapter introduced the **logistic regression** model of **classification**.

- Logistic regression is a supervised machine learning classifier that extracts real-valued features from the input, multiplies each by a weight, sums them, and passes the sum through a **sigmoid** function to generate a probability. A threshold is used to make a decision.
- Logistic regression can be used with two classes (e.g., positive and negative sentiment) or with multiple classes (**multinomial logistic regression**, for example for n-ary text classification, part-of-speech labeling, etc.).
- Multinomial logistic regression uses the **softmax** function to compute probabilities.
- The weights (vector w and bias b) are learned from a labeled training set via a loss function, such as the **cross-entropy loss**, that must be minimized.
- Minimizing this loss function is a **convex optimization** problem, and iterative algorithms like **gradient descent** are used to find the optimal weights.
- **Regularization** is used to avoid overfitting.
- Logistic regression is also one of the most useful analytic tools, because of its ability to transparently study the importance of individual features.

Historical Notes

Logistic regression was developed in the field of statistics, where it was used for the analysis of binary data by the 1960s, and was particularly common in medicine (Cox, 1969). Starting in the late 1970s it became widely used in linguistics as one of the formal foundations of the study of linguistic variation (Sankoff and Labov, 1979).

Nonetheless, logistic regression didn't become common in natural language processing until the 1990s, when it seems to have appeared simultaneously from two directions. The first source was the neighboring fields of information retrieval and speech processing, both of which had made use of regression, and both of which lent many other statistical techniques to NLP. Indeed a very early use of logistic regression for document routing was one of the first NLP applications to use (LSI) embeddings as word representations (Schütze et al., 1995).

maximum entropy

At the same time in the early 1990s logistic regression was developed and applied to NLP at IBM Research under the name **maximum entropy** modeling or **maxent** (Berger et al., 1996), seemingly independent of the statistical literature. Under that name it was applied to language modeling (Rosenfeld, 1996), part-of-speech tagging (Ratnaparkhi, 1996), parsing (Ratnaparkhi, 1997), coreference resolution (Kehler, 1997b), and text classification (Nigam et al., 1999).

There are a variety of sources covering the many kinds of text classification tasks. For sentiment analysis see Pang and Lee (2008), and Liu and Zhang (2012). Stamatatos (2009) surveys authorship attribute algorithms. On language identification see Jauhiainen et al. (2019); Jaech et al. (2016) is an important early neural system. The task of newswire indexing was often used as a test case for text classification algorithms, based on the Reuters-21578 collection of newswire articles.

See Manning et al. (2008) and Aggarwal and Zhai (2012) on text classification; classification in general is covered in machine learning textbooks (Hastie et al. 2001, Witten and Frank 2005, Bishop 2006, Murphy 2012).

Non-parametric methods for computing statistical significance were used first in NLP in the MUC competition (Chinchor et al., 1993), and even earlier in speech recognition (Gillick and Cox 1989, Bisani and Ney 2004). Our description of the bootstrap draws on the description in Berg-Kirkpatrick et al. (2012). Recent work has focused on issues including multiple test sets and multiple metrics (Søgaard et al. 2014, Dror et al. 2017).

information gain

Feature selection is a method of removing features that are unlikely to generalize well. Features are generally ranked by how informative they are about the classification decision. A very common metric, **information gain**, tells us how many bits of information the presence of the word gives us for guessing the class. Other feature selection metrics include χ^2, pointwise mutual information, and GINI index; see Yang and Pedersen (1997) for a comparison and Guyon and Elisseeff (2003) for an introduction to feature selection.

Exercises

CHAPTER

5 Embeddings

荃者所以在鱼，得鱼而忘荃 Nets are for fish;
Once you get the fish, you can forget the net.
言者所以在意，得意而忘言 Words are for meaning;
Once you get the meaning, you can forget the words
庄子(Zhuangzi), Chapter 26

The asphalt that Los Angeles is famous for occurs mainly on its freeways. But in the middle of the city is another patch of asphalt, the La Brea tar pits, and this asphalt preserves millions of fossil bones from the last of the Ice Ages of the Pleistocene Epoch. One of these fossils is the *Smilodon*, or saber-toothed tiger, instantly recognizable by its long canines. Five million years ago or so, a completely different saber-tooth tiger called *Thylacosmilus* lived in Argentina and other parts of South America. Thylacosmilus was a marsupial whereas Smilodon was a placental mammal, but Thylacosmilus had the same long upper canines and, like Smilodon, had a protective bone flange on the lower jaw. The similarity of these two mammals is one of many examples of parallel or convergent evolution, in which particular contexts or environments lead to the evolution of very similar structures in different species (Gould, 1980).

The role of context is also important in the similarity of a less biological kind of organism: the word. Words that occur in *similar contexts* tend to have *similar meanings*. This link between similarity in how words are distributed and similarity in what they mean is called the **distributional hypothesis**. The hypothesis was first formulated in the 1950s by linguists like Joos (1950), Harris (1954), and Firth (1957), who noticed that words which are synonyms (like *oculist* and *eye-doctor*) tended to occur in the same environment (e.g., near words like *eye* or *examined*) with the amount of meaning difference between two words "corresponding roughly to the amount of difference in their environments" (Harris, 1954, p. 157).

distributional hypothesis

In this chapter we introduce **embeddings**, vector representations of the meaning of words that are learned directly from word distributions in texts. Embeddings lie at the heart of large language models and other modern applications. The **static embeddings** we introduce here underlie the more powerful dynamic or **contextualized embeddings** like **BERT** that we will see in Chapter 9 and Chapter 8.

embeddings

The linguistic field that studies embeddings and their meanings is called **vector semantics**. Embeddings are also the first example in this book of **representation learning**, automatically learning useful representations of the input text. Finding such **self-supervised** ways to learn representations of language, instead of creating representations by hand via **feature engineering**, is an important principle of modern NLP (Bengio et al., 2013).

vector semantics
representation learning

5.1 Lexical Semantics

Let's begin by introducing some basic principles of word meaning. How should we represent the meaning of a word? In the n-gram models of Chapter 3, and in classical NLP applications, our only representation of a word is as a string of letters, or an index in a vocabulary list. This representation is not that different from a tradition in philosophy, perhaps you've seen it in introductory logic classes, in which the meaning of words is represented by just spelling the word with small capital letters; representing the meaning of "dog" as DOG, and "cat" as CAT, or by using an apostrophe (DOG').

Representing the meaning of a word by capitalizing it is a pretty unsatisfactory model. You might have seen a version of a joke due originally to semanticist Barbara Partee (Carlson, 1977):

> Q: What's the meaning of life?
> A: LIFE'

Surely we can do better than this! After all, we'll want a model of word meaning to do all sorts of things for us. It should tell us that some words have similar meanings (*cat* is similar to *dog*), others are antonyms (*cold* is the opposite of *hot*), some have positive connotations (*happy*) while others have negative connotations (*sad*). It should represent the fact that the meanings of *buy*, *sell*, and *pay* offer differing perspectives on the same underlying purchasing event. (If I buy something from you, you've probably sold it to me, and I likely paid you.) More generally, a model of word meaning should allow us to draw inferences to address meaning-related tasks like question-answering or dialogue.

lexical semantics

In this section we summarize some of these desiderata, drawing on results in the linguistic study of word meaning, which is called **lexical semantics**; we'll return to and expand on this list in Appendix G and Chapter 21.

Lemmas and Senses Let's start by looking at how one word (we'll choose *mouse*) might be defined in a dictionary (simplified from the online dictionary WordNet):

```
mouse (N)
1.  any of numerous small rodents...
2.  a hand-operated device that controls a cursor...
```

lemma
citation form

Here the form *mouse* is the **lemma**, also called the **citation form**. The form *mouse* would also be the lemma for the word *mice*; dictionaries don't have separate definitions for inflected forms like *mice*. Similarly *sing* is the lemma for *sing*, *sang*, *sung*. In many languages the infinitive form is used as the lemma for the verb, so Spanish *dormir* "to sleep" is the lemma for *duermes* "you sleep". The specific forms *sung* or *carpets* or *sing* or *duermes* are called **wordforms**.

wordform

As the example above shows, each lemma can have multiple meanings; the lemma *mouse* can refer to the rodent or the cursor control device. We call each of these aspects of the meaning of *mouse* a **word sense**. The fact that lemmas can be **polysemous** (have multiple senses) can make interpretation difficult (is someone who searches for "mouse info" looking for a pet or a widget?). Chapter 9 and Appendix G will discuss the problem of polysemy, and introduce **word sense disambiguation**, the task of determining which sense of a word is being used in a particular context.

Synonymy One important component of word meaning is the relationship between word senses. For example when one word has a sense whose meaning is

identical to a sense of another word, or nearly identical, we say the two senses of those two words are **synonyms**. Synonyms include such pairs as

synonym

couch/sofa vomit/throw up filbert/hazelnut car/automobile

A more formal definition of synonymy (between words rather than senses) is that two words are synonymous if they are substitutable for one another in any sentence without changing the *truth conditions* of the sentence, the situations in which the sentence would be true.

While substitutions between some pairs of words like *car* / *automobile* or *water* / H_2O are truth preserving, the words are still not identical in meaning. Indeed, probably no two words are absolutely identical in meaning. One of the fundamental tenets of semantics, called the **principle of contrast** (Girard 1718, Bréal 1897, Clark 1987), states that a difference in linguistic form is always associated with some difference in meaning. For example, the word H_2O is used in scientific contexts and would be inappropriate in a hiking guide—*water* would be more appropriate— and this genre difference is part of the meaning of the word. In practice, the word *synonym* is therefore used to describe a relationship of approximate or rough synonymy.

principle of contrast

Word Similarity While words don't have many synonyms, most words do have lots of *similar* words. *Cat* is not a synonym of *dog*, but *cats* and *dogs* are certainly similar words. In moving from synonymy to similarity, it will be useful to shift from talking about relations between word senses (like synonymy) to relations between words (like similarity). Dealing with words avoids having to commit to a particular representation of word senses, which will turn out to simplify our task.

The notion of word **similarity** is very useful in larger semantic tasks. Knowing how similar two words are can help in computing how similar the meaning of two phrases or sentences are, a very important component of tasks like question answering, paraphrasing, and summarization. One way of getting values for word similarity is to ask humans to judge how similar one word is to another. A number of datasets have resulted from such experiments. For example the SimLex-999 dataset (Hill et al., 2015) gives values on a scale from 0 to 10, like the examples below, which range from near-synonyms (*vanish*, *disappear*) to pairs that scarcely seem to have anything in common (*hole*, *agreement*):

similarity

vanish	disappear	9.8
belief	impression	5.95
muscle	bone	3.65
modest	flexible	0.98
hole	agreement	0.3

Word Relatedness The meaning of two words can be related in ways other than similarity. One such class of connections is called word **relatedness** (Budanitsky and Hirst, 2006), also traditionally called word **association** in psychology.

relatedness

association

Consider the meanings of the words *coffee* and *cup*. Coffee is not similar to cup; they share practically no features (coffee is a plant or a beverage, while a cup is a manufactured object with a particular shape). But coffee and cup are clearly related; they are associated by co-participating in an everyday event (the event of drinking coffee out of a cup). Similarly *scalpel* and *surgeon* are not similar but are related eventively (a surgeon tends to make use of a scalpel).

One common kind of relatedness between words is if they belong to the same **semantic field**. A semantic field is a set of words which cover a particular semantic domain and bear structured relations with each other. For example, words might be

semantic field

related by being in the semantic field of hospitals (*surgeon*, *scalpel*, *nurse*, *anesthetic*, *hospital*), restaurants (*waiter*, *menu*, *plate*, *food*, *chef*), or houses (*door*, *roof*, *kitchen*, *family*, *bed*). Semantic fields are also related to **topic models**, like **Latent Dirichlet Allocation**, **LDA**, which apply unsupervised learning on large sets of texts to induce sets of associated words from text. Semantic fields and topic models are very useful tools for discovering topical structure in documents.

topic models

In Appendix G we'll introduce more relations between senses like **hypernymy** or **IS-A**, **antonymy** (opposites) and **meronymy** (part-whole relations).

connotations

Connotation Finally, words have *affective meanings* or **connotations**. The word *connotation* has different meanings in different fields, but here we use it to mean the aspects of a word's meaning that are related to a writer or reader's emotions, sentiment, opinions, or evaluations. For example some words have positive connotations (*wonderful*) while others have negative connotations (*dreary*). Even words whose meanings are similar in other ways can vary in connotation; consider the difference in connotations between *fake*, *knockoff*, *forgery*, on the one hand, and *copy*, *replica*, *reproduction* on the other, or *innocent* (positive connotation) and *naive* (negative connotation). Some words describe positive evaluation (*great*, *love*) and others negative evaluation (*terrible*, *hate*). Positive or negative evaluation language is called **sentiment**, as we saw in Appendix K, and word sentiment plays a role in important tasks like sentiment analysis, stance detection, and applications of NLP to the language of politics and consumer reviews.

sentiment

Early work on affective meaning (Osgood et al., 1957) found that words varied along three important dimensions of affective meaning:

valence: the pleasantness of the stimulus

arousal: the intensity of emotion provoked by the stimulus

dominance: the degree of control exerted by the stimulus

Thus words like *happy* or *satisfied* are high on valence, while *unhappy* or *annoyed* are low on valence. *Excited* is high on arousal, while *calm* is low on arousal. *Controlling* is high on dominance, while *awed* or *influenced* are low on dominance. Each word is thus represented by three numbers, corresponding to its value on each of the three dimensions:

	Valence	Arousal	Dominance
courageous	8.0	5.5	7.4
music	7.7	5.6	6.5
heartbreak	2.5	5.7	3.6
cub	6.7	4.0	4.2

Osgood et al. (1957) noticed that in using these 3 numbers to represent the meaning of a word, the model was representing each word as a point in a three-dimensional space, a vector whose three dimensions corresponded to the word's rating on the three scales. This revolutionary idea that word meaning could be represented as a point in space (e.g., that part of the meaning of *heartbreak* can be represented as the point $[2.5, 5.7, 3.6]$) was the first expression of the vector semantics models that we introduce next.

5.2 Vector Semantics: The Intuition

vector semantics

Vector semantics is the standard way to represent word meaning in NLP, helping

us model many of the aspects of word meaning we saw in the previous section. The roots of the model lie in the 1950s when two big ideas converged: Osgood's 1957 idea mentioned above to use a point in three-dimensional space to represent the connotation of a word, and the proposal by linguists like Joos (1950), Harris (1954), and Firth (1957) to define the meaning of a word by its **distribution** in language use, meaning its neighboring words or grammatical environments. Their idea was that two words that occur in very similar distributions (whose neighboring words are similar) have similar meanings.

For example, suppose you didn't know the meaning of the word *ongchoi* (a recent borrowing from Cantonese) but you see it in the following contexts:

(5.1) Ongchoi is delicious sauteed with garlic.

(5.2) Ongchoi is superb over rice.

(5.3) ...ongchoi leaves with salty sauces...

And suppose that you had seen many of these context words in other contexts:

(5.4) ...spinach sauteed with garlic over rice...

(5.5) ...chard stems and leaves are delicious...

(5.6) ...collard greens and other salty leafy greens

The fact that *ongchoi* occurs with words like *rice* and *garlic* and *delicious* and *salty*, as do words like *spinach*, *chard*, and *collard greens* might suggest that ongchoi is a leafy green similar to these other leafy greens.[1] We can implement the same intuition computationally by just counting words in the context of *ongchoi*.

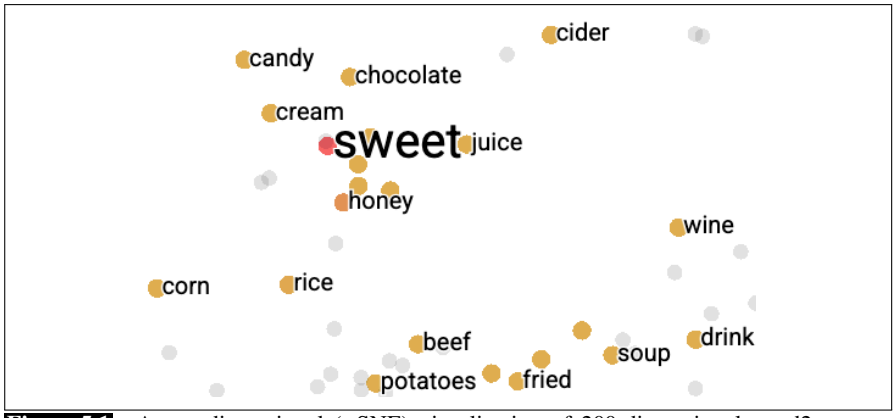


Figure 5.1 A two-dimensional (t-SNE) visualization of 200-dimensional word2vec embeddings for some words close to the word *sweet*, showing that words with similar meanings are nearby in space. Visualization created using the TensorBoard Embedding Projector https://projector.tensorflow.org/.

embeddings

The idea of vector semantics is to represent a word as a point in a multidimensional semantic space that is derived (in different ways we'll see) from the distributions of word neighbors. Vectors for representing words are called **embeddings**. The word "embedding" derives historically from its mathematical sense as a mapping from one space or structure to another, although the meaning has shifted; see the end of the chapter.

Fig. 5.1 shows a visualization of embeddings learned by the word2vec algorithm, showing the location of selected words (neighbors of "sweet") projected down from

[1] It's in fact *Ipomoea aquatica*, a relative of morning glory sometimes called *water spinach* in English.

200-dimensional space into a 2-dimensional space. Note that the nearest neighbors of sweet are semantically related words like honey, candy, juice, chocolate. This idea that similar words are neighbors in high-dimensional space offers enormous power to language models and other NLP applications. For example the sentiment classifiers of Chapter 4 depend on the same words appearing in the training and test sets. But by representing words as embeddings, a classifier can assign sentiment as long as it sees some words with *similar meanings*. And as we'll see, vector semantic models like the ones showed in Fig. 5.1 can be learned automatically from text without supervision.

In this chapter we'll begin with a simple pedagogical model of embeddings in which the meaning of a word is defined by a vector with the counts of nearby words. We introduce this model as a helpful way to understand the concept of vectors and what it means for a vector to be a representation of word meaning, but more sophisticated variants like the **tf-idf model** we will introduce in Chapter 11 are important methods you should understand. We will see that this method results in very long vectors that are **sparse**, i.e. mostly zeros (since most words simply never occur in the context of others). We'll then introduce the **word2vec** model family for constructing short, **dense** vectors that have even more useful semantic properties.

We'll also introduce the **cosine**, the standard way to use embeddings to compute *semantic similarity*, between two words, two sentences, or two documents, an important tool in practical applications.

5.3 Simple count-based embeddings

"The most important attributes of a vector in 3-space are {Location, Location, Location}"
Randall Munroe, the hover from https://xkcd.com/2358/

Let's now introduce the first way to compute word vector embeddings. This simplest vector model of meaning is based on the **co-occurrence matrix**, a way of representing how often words co-occur. We'll define a particular kind of co-occurrence matrix, the **word-context matrix**, in which each row in the matrix represents a word in the vocabulary and each column represents how often each other word in the vocabulary appears nearby. This matrix is thus of dimensionality $|V| \times |V|$ and each cell records the number of times the row (target) word and the column (context) word co-occur nearby in some training corpus.

word-context matrix

What do we mean by 'nearby'? We could implement various methods, but let's start with a very simple one: a context window around the word, let's say of 4 words to the left and 4 words to the right. If we do that, each cell will represents the number of times (in some training corpus) the column word occurs in such a ± 4 word window around the row word.

Let's see how this works for 4 words: *cherry*, *strawberry*, *digital*, and *information*. For each word we took a single instance from a corpus, and we show the ± 4 word window from that instance:

is traditionally followed by	**cherry**	pie, a traditional dessert
often mixed, such as	**strawberry**	rhubarb pie. Apple pie
computer peripherals and personal	**digital**	assistants. These devices usually
a computer. This includes	**information**	available on the internet

If we then take every occurrence of each word in a large corpus and count the context words around it, we get a word-context co-occurrence matrix. The full word-

context co-occurrence matrix is very large, because for each word in the vocabulary (since $|V|$) we have to count how often it occurs with every other word in the vocabulary, hence dimensionality $|V| \times |V|$. Let's therefore instead sketch the process on a smaller scale. Imagine that we are going to look at only the 4 words, and only consider the following 3 context words: *a*, *computer*, and *pie*. Furthermore let's assume we only count occurrences in the mini-corpus above.

So before looking at Fig. 5.2, compute by hand the counts for these 3 context words for the four words *cherry*, *strawberry*, *digital*, and *information*.

	a	**computer**	**pie**
cherry	1	0	1
strawberry	0	0	2
digital	0	1	0
information	1	1	0

Figure 5.2 Co-occurrence vectors for four words with counts from the 4 windows above, showing just 3 of the potential context word dimensions. The vector for *cherry* is outlined in red. Note that a real vector would have vastly more dimensions and thus be even sparser.

Hopefully your count matches what is shown in Fig. 5.2, so that each cell represents the number of times a particular word (defined by the row) occurs in a particular context (defined by the word column).

vector

Each row, then, is a vector representing a word. To review some basic linear algebra, a **vector** is, at heart, just a list or array of numbers. So *cherry* is represented as the list [1,0,1] (the first **row vector** in Fig. 5.2) and *information* is represented as the list [1,1,0] (the fourth row vector).

vector space
dimension

A **vector space** is a collection of vectors, and is characterized by its **dimension**. Vectors in a 3-dimensional vector space have an element for each dimension of the space. We will loosely refer to a vector in a 3-dimensional space as a 3-dimensional vector, with one element along each dimension. In the example in Fig. 5.2, we've chosen to make the document vectors of dimension 3, just so they fit on the page; in real term-document matrices, the document vectors would have dimensionality $|V|$, the vocabulary size.

The ordering of the numbers in a vector space indicates the different dimensions on which documents vary. The third dimension for all these vectors corresponds to the number of times *pie* occurs in the context. The second dimension for all of them corresponds to the number of times the word *computer* occurs. Notice that the vectors for *information* and *digital* have the same value (1) for this "computer" dimension.

In reality, we don't compute word vectors on a single context window. Instead, we compute them over an entire corpus. Let's see what some real counts look like. Let's look at some vectors computed in this way. Fig. 5.3 shows a subset of the word-word co-occurrence matrix for these four words, where, again because it's impossible to visualize all $|V|$ possible context words on the page of this textbook, we show a subset of 6 of the dimensions, with counts computed from the Wikipedia corpus (Davies, 2015).

Note in Fig. 5.3 that the two words *cherry* and *strawberry* are more similar to each other (both *pie* and *sugar* tend to occur in their window) than they are to other words like *digital*; conversely, *digital* and *information* are more similar to each other than, say, to *strawberry*.

We can think of the vector for a document as a point in $|V|$-dimensional space; thus the documents in Fig. 5.3 are points in 3-dimensional space. Fig. 5.4 shows a spatial visualization.

	aardvark	...	computer	data	result	pie	sugar	...
cherry	0	...	2	8	9	442	25	...
strawberry	0	...	0	0	1	60	19	...
digital	0	...	1670	1683	85	5	4	...
information	0	...	3325	3982	378	5	13	...

Figure 5.3 Co-occurrence vectors for four words in the Wikipedia corpus, showing six of the dimensions (hand-picked for pedagogical purposes). The vector for *digital* is outlined in red. Note that a real vector would have vastly more dimensions and thus be much sparser, i.e. would have zero values in most dimensions.

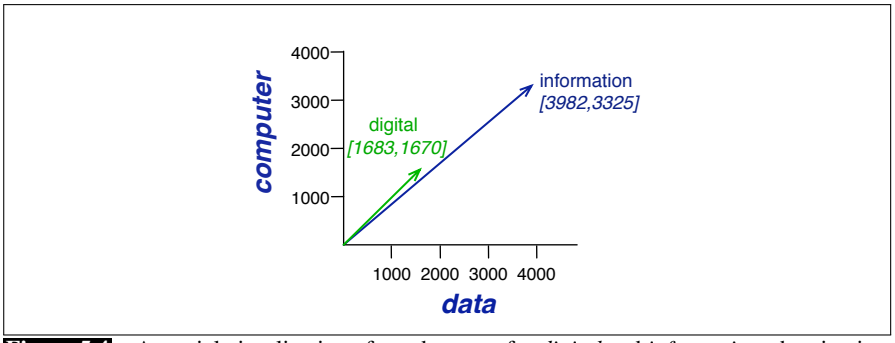


Figure 5.4 A spatial visualization of word vectors for *digital* and *information*, showing just two of the dimensions, corresponding to the words *data* and *computer*.

Note that $|V|$, the dimensionality of the vector, is generally the size of the vocabulary, often between 10,000 and 50,000 words (using the most frequent words in the training corpus; keeping words after about the most frequent 50,000 or so is generally not helpful). Since most of these numbers are zero these are **sparse** vector representations; there are efficient algorithms for storing and computing with sparse matrices.

It's also possible to apply various kinds of weighting functions to the counts in these cells. The most popular such weighting is tf-idf, which we'll introduce in Chapter 11, but there have historically been a wide variety of other weightings.

Now that we have some intuitions, let's move on to examine the details of computing word similarity.

5.4 Cosine for measuring similarity

To measure similarity between two target words v and w, we need a metric that takes two vectors (of the same dimensionality, either both with words as dimensions, hence of length $|V|$, or both with documents as dimensions, of length $|D|$) and gives a measure of their similarity. By far the most common similarity metric is the **cosine** of the angle between the vectors.

dot product
inner product

The cosine—like most measures for vector similarity used in NLP—is based on the **dot product** operator from linear algebra, also called the **inner product**:

$$\text{dot product}(\mathbf{v},\mathbf{w}) = \mathbf{v}\cdot\mathbf{w} = \sum_{i=1}^{N} v_i w_i = v_1 w_1 + v_2 w_2 + \ldots + v_N w_N \tag{5.7}$$

The dot product acts as a similarity metric because it will tend to be high just when the two vectors have large values in the same dimensions. Alternatively, vectors that

have zeros in different dimensions—orthogonal vectors—will have a dot product of 0, representing their strong dissimilarity.

This raw dot product, however, has a problem as a similarity metric: it favors long vectors. The **vector length** is defined as

vector length

$$|\mathbf{v}| = \sqrt{\sum_{i=1}^{N} v_i^2} \tag{5.8}$$

The dot product is higher if a vector is longer, with higher values in each dimension. More frequent words have longer vectors, since they tend to co-occur with more words and have higher co-occurrence values with each of them. The raw dot product thus will be higher for frequent words. But this is a problem; we'd like a similarity metric that tells us how similar two words are regardless of their frequency.

We modify the dot product to normalize for the vector length by dividing the dot product by the lengths of each of the two vectors. This **normalized dot product** turns out to be the same as the cosine of the angle between the two vectors, following from the definition of the dot product between two vectors $\mathbf{a}$ and $\mathbf{b}$:

$$\begin{aligned} \mathbf{a} \cdot \mathbf{b} &= |\mathbf{a}||\mathbf{b}| \cos\theta \\ \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}||\mathbf{b}|} &= \cos\theta \end{aligned} \tag{5.9}$$

cosine

The **cosine** similarity metric between two vectors $\mathbf{v}$ and $\mathbf{w}$ thus can be computed as:

$$\text{cosine}(\mathbf{v},\mathbf{w}) = \frac{\mathbf{v} \cdot \mathbf{w}}{|\mathbf{v}||\mathbf{w}|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2}\sqrt{\sum_{i=1}^{N} w_i^2}} \tag{5.10}$$

unit vector

For some applications we pre-normalize each vector, by dividing it by its length, creating a **unit vector** of length 1. Thus we could compute a unit vector from $\mathbf{a}$ by dividing it by $|\mathbf{a}|$. For unit vectors, the dot product is the same as the cosine.

The cosine value ranges from 1 for vectors pointing in the same direction, through 0 for orthogonal vectors, to -1 for vectors pointing in opposite directions. But since raw frequency values are non-negative, the cosine for these vectors ranges from 0–1.

Let's see how the cosine computes which of the words *cherry* or *digital* is closer in meaning to *information*, just using raw counts from the following shortened table:

	pie	data	computer
cherry	442	8	2
digital	5	1683	1670
information	5	3982	3325

$$\begin{aligned} \cos(\text{cherry},\text{information}) &= \frac{442*5+8*3982+2*3325}{\sqrt{442^2+8^2+2^2}\sqrt{5^2+3982^2+3325^2}} = .018 \\ \cos(\text{digital},\text{information}) &= \frac{5*5+1683*3982+1670*3325}{\sqrt{5^2+1683^2+1670^2}\sqrt{5^2+3982^2+3325^2}} = .996 \end{aligned}$$

The model decides that *information* is way closer to *digital* than it is to *cherry*, a result that seems sensible. Fig. 5.5 shows a visualization.

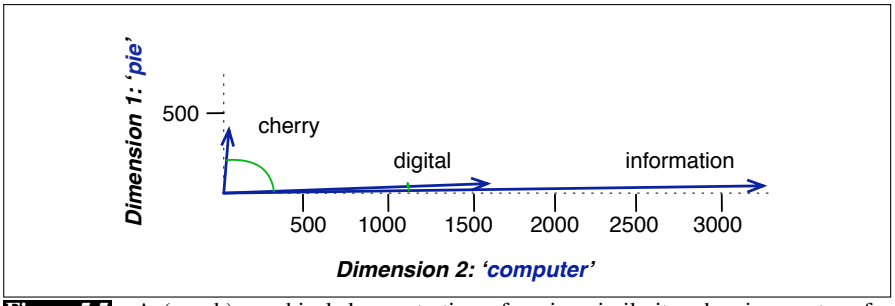


Figure 5.5 A (rough) graphical demonstration of cosine similarity, showing vectors for three words (*cherry*, *digital*, and *information*) in the two dimensional space defined by counts of the words *computer* and *pie* nearby. The figure doesn't show the cosine, but it highlights the angles; note that the angle between *digital* and *information* is smaller than the angle between *cherry* and *information*. When two vectors are more similar, the cosine is larger but the angle is smaller; the cosine has its maximum (1) when the angle between two vectors is smallest ($0°$); the cosine of all other angles is less than 1.

Cosine similarity can be used to estimate word similarity, for tasks like finding word paraphrases, tracking changes in word meaning, or automatically discovering meanings of words in different corpora. For example, we can fine the 10 most similar words to any target word w by computing the cosines between w and each of the $|V|-1$ other words, sorting, and looking at the top 10.

5.5 Word2vec

In the previous sections we saw how to represent a word as a sparse, long vector with dimensions corresponding to words in the vocabulary. We now introduce a more powerful word representation: **embeddings**, short dense vectors. Unlike the vectors we've seen so far, embeddings are **short**, with number of dimensions d ranging from 50-1000, rather than the much larger vocabulary size $|V|$.These d dimensions don't have a clear interpretation. And the vectors are **dense**: instead of vector entries being sparse, mostly-zero counts or functions of counts, the values will be real-valued numbers that can be negative.

It turns out that dense vectors work better in every NLP task than sparse vectors. While we don't completely understand all the reasons for this, we have some intuitions. Representing words as 300-dimensional dense vectors requires our classifiers to learn far fewer weights than if we represented words as 50,000-dimensional vectors, and the smaller parameter space possibly helps with generalization and avoiding overfitting. Dense vectors may also do a better job of capturing synonymy. For example, in a sparse vector representation, dimensions for synonyms like *car* and *automobile* dimension are distinct and unrelated; sparse vectors may thus fail to capture the similarity between a word with *car* as a neighbor and a word with *automobile* as a neighbor.

skip-gram
SGNS
word2vec

In this section we introduce one method for computing embeddings: **skip-gram with negative sampling**, sometimes called **SGNS**. The skip-gram algorithm is one of two algorithms in a software package called **word2vec**, and so sometimes the algorithm is loosely referred to as word2vec (Mikolov et al. 2013a, Mikolov et al. 2013b). The word2vec methods are fast, efficient to train, and easily available online with code and pretrained embeddings. Word2vec embeddings are **static em-**

static embeddings

beddings, meaning that the method learns one fixed embedding for each word in the vocabulary. In Chapter 9 we'll introduce methods for learning dynamic **contextual embeddings** like the popular family of **BERT** representations, in which the vector for each word is different in different contexts.

The intuition of word2vec is that instead of counting how often each context word c occurs near, say, *apricot*, we'll instead train a classifier on a binary prediction task: "Is word c likely to show up near *apricot*?" We don't actually care about this prediction task; instead we'll take the learned classifier *weights* as the word embeddings.

The revolutionary intuition here is that we can just use running text as implicitly supervised training data for such a classifier; a word c that occurs near the target word *apricot* acts as gold 'correct answer' to the question "Is word c likely to show up near *apricot*?" This method, often called **self-supervision**, avoids the need for any sort of hand-labeled supervision signal. This idea was first proposed in the task of neural language modeling, when Bengio et al. (2003) and Collobert et al. (2011) showed that a neural language model (a neural network that learned to predict the next word from prior words) could just use the next word in running text as its supervision signal, and could be used to learn an embedding representation for each word as part of doing this prediction task.

self-supervision

We'll see how to do neural networks in the next chapter, but word2vec is a much simpler model than the neural network language model, in two ways. First, word2vec simplifies the task (making it binary classification instead of word prediction). Second, word2vec simplifies the architecture (training a logistic regression classifier instead of a multi-layer neural network with hidden layers that demand more sophisticated training algorithms). The intuition of skip-gram is:

1. Treat the target word and a neighboring context word as positive examples.
2. Randomly sample other words in the lexicon to get negative samples.
3. Use logistic regression to train a classifier to distinguish those two cases.
4. Use the learned weights as the embeddings.

5.5.1 The classifier

Let's start by thinking about the classification task, and then turn to how to train. Imagine a sentence like the following, with a target word *apricot*, and assume we're using a window of ±2 context words:

```
... lemon,  a [tablespoon of apricot jam,      a] pinch ...
               c1         c2    w     c3       c4
```

Our goal is to train a classifier such that, given a tuple (w,c) of a target word w paired with a candidate context word c (for example (*apricot*, *jam*), or perhaps (*apricot*, *aardvark*)) it will return the probability that c is a real context word (true for *jam*, false for *aardvark*):

$$P(+|w,c) \tag{5.11}$$

The probability that word c is not a real context word for w is just 1 minus Eq. 5.11:

$$P(-|w,c) = 1 - P(+|w,c) \tag{5.12}$$

How does the classifier compute the probability P? The intuition of the skip-gram model is to base this probability on embedding similarity: a word is likely to occur near the target if its embedding vector is similar to the target embedding. To compute similarity between these dense embeddings, we rely on the intuition that two vectors are similar if they have a high **dot product** (after all, cosine is just a normalized dot product). In other words:

$$Similarity(w,c) \approx \mathbf{c} \cdot \mathbf{w} \tag{5.13}$$

The dot product $\mathbf{c} \cdot \mathbf{w}$ is not a probability, it's just a number ranging from $-\infty$ to ∞ (since the elements in word2vec embeddings can be negative, the dot product can be negative). To turn the dot product into a probability, we'll use the **logistic** or **sigmoid** function $\sigma(x)$, the fundamental core of logistic regression:

$$\sigma(x) = \frac{1}{1+\exp(-x)} \tag{5.14}$$

We model the probability that word c is a real context word for target word w as:

$$P(+|w,c) = \sigma(\mathbf{c} \cdot \mathbf{w}) = \frac{1}{1+\exp(-\mathbf{c} \cdot \mathbf{w})} \tag{5.15}$$

The sigmoid function returns a number between 0 and 1, but to make it a probability we'll also need the total probability of the two possible events (c is a context word, and c isn't a context word) to sum to 1. We thus estimate the probability that word c is not a real context word for w as:

$$\begin{aligned} P(-|w,c) &= 1 - P(+|w,c) \\ &= \sigma(-\mathbf{c} \cdot \mathbf{w}) = \frac{1}{1+\exp(\mathbf{c} \cdot \mathbf{w})} \end{aligned} \tag{5.16}$$

Equation 5.15 gives us the probability for one word, but there are many context words in the window. Skip-gram makes the simplifying assumption that all context words are independent, allowing us to just multiply their probabilities:

$$P(+|w,c_{1:L}) = \prod_{i=1}^{L} \sigma(\mathbf{c_i} \cdot \mathbf{w}) \tag{5.17}$$

$$\log P(+|w,c_{1:L}) = \sum_{i=1}^{L} \log \sigma(\mathbf{c_i} \cdot \mathbf{w}) \tag{5.18}$$

In summary, skip-gram trains a probabilistic classifier that, given a test target word w and its context window of L words $c_{1:L}$, assigns a probability based on how similar this context window is to the target word. The probability is based on applying the logistic (sigmoid) function to the dot product of the embeddings of the target word with each context word. To compute this probability, we just need embeddings for each target word and context word in the vocabulary.

Fig. 5.6 shows the intuition of the parameters we'll need. Skip-gram actually stores two embeddings for each word, one for the word as a target, and one for the word considered as context. Thus the parameters we need to learn are two matrices $\mathbf{W}$ and $\mathbf{C}$, each containing an embedding for every one of the $|V|$ words in the vocabulary V.[2] Let's now turn to learning these embeddings (which is the real goal of training this classifier in the first place).

[2] In principle the target matrix and the context matrix could use different vocabularies, but we'll simplify by assuming one shared vocabulary V.

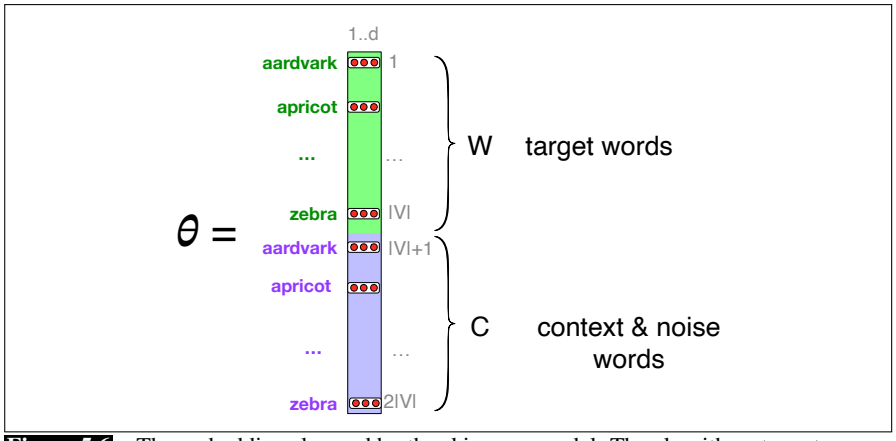


Figure 5.6 The embeddings learned by the skipgram model. The algorithm stores two embeddings for each word, the target embedding (sometimes called the input embedding) and the context embedding (sometimes called the output embedding). The parameter θ that the algorithm learns is thus a matrix of $2|V|$ vectors, each of dimension d, formed by concatenating two matrices, the target embeddings **W** and the context+noise embeddings **C**.

5.5.2 Learning skip-gram embeddings

The learning algorithm for skip-gram embeddings takes as input a corpus of text, and a chosen vocabulary size N. It begins by assigning a random embedding vector for each of the N vocabulary words, and then proceeds to iteratively shift the embedding of each word w to be more like the embeddings of words that occur nearby in texts, and less like the embeddings of words that don't occur nearby. Let's start by considering a single piece of training data:

```
... lemon,  a [tablespoon of apricot jam,      a] pinch ...
              c1         c2    w     c3       c4
```

This example has a target word w (apricot), and 4 context words in the $L = \pm 2$ window, resulting in 4 positive training instances (on the left below):

positive examples +

w	c_{pos}
apricot	tablespoon
apricot	of
apricot	jam
apricot	a

negative examples -

w	c_{neg}	w	c_{neg}
apricot	aardvark	apricot	seven
apricot	my	apricot	forever
apricot	where	apricot	dear
apricot	coaxial	apricot	if

For training a binary classifier we also need negative examples. In fact skip-gram with negative sampling (SGNS) uses more negative examples than positive examples (with the ratio between them set by a parameter k). So for each of these (w, c_{pos}) training instances we'll create k negative samples, each consisting of the target w plus a 'noise word' c_{neg}. A noise word is a random word from the lexicon, constrained not to be the target word w. The table right above shows the setting where $k = 2$, so we'll have 2 negative examples in the negative training set $-$ for each positive example w, c_{pos}.

The noise words are chosen according to their weighted unigram probability $p_\alpha(w)$, where α is a weight. If we were sampling according to unweighted probability $P(w)$, it would mean that with unigram probability $P(\text{"}the\text{"})$ we would choose the word *the* as a noise word, with unigram probability $P(\text{"}aardvark\text{"})$ we would

choose *aardvark*, and so on. But in practice it is common to set $\alpha = 0.75$, i.e. use the weighting $P_{\frac{3}{4}}(w)$:

$$P_\alpha(w) = \frac{count(w)^\alpha}{\sum_{w'} count(w')^\alpha} \tag{5.19}$$

Setting $\alpha = .75$ gives better performance because it gives rare noise words slightly higher probability: for rare words, $P_\alpha(w) > P(w)$. To illustrate this intuition, it might help to work out the probabilities for an example with $\alpha = .75$ and two events, $P(a) = 0.99$ and $P(b) = 0.01$:

$$\begin{aligned} P_\alpha(a) &= \frac{.99^{.75}}{.99^{.75} + .01^{.75}} = 0.97 \\ P_\alpha(b) &= \frac{.01^{.75}}{.99^{.75} + .01^{.75}} = 0.03 \end{aligned} \tag{5.20}$$

Thus using $\alpha = .75$ increases the probability of the rare event b from 0.01 to 0.03.

Given the set of positive and negative training instances, and an initial set of embeddings, the goal of the learning algorithm is to adjust those embeddings to

- Maximize the similarity of the target word, context word pairs (w, c_{pos}) drawn from the positive examples
- Minimize the similarity of the (w, c_{neg}) pairs from the negative examples.

If we consider one word/context pair (w, c_{pos}) with its k noise words $c_{neg_1} ... c_{neg_k}$, we can express these two goals as the following loss function L to be minimized (hence the $-$); here the first term expresses that we want the classifier to assign the real context word c_{pos} a high probability of being a neighbor, and the second term expresses that we want to assign each of the noise words c_{neg_i} a high probability of being a non-neighbor, all multiplied because we assume independence:

$$\begin{aligned} L(w, c_{pos}, c_{neg*}) &= -\log \left[P(+|w, c_{pos}) \prod_{i=1}^{k} P(-|w, c_{neg_i}) \right] \\ &= -\left[\log P(+|w, c_{pos}) + \sum_{i=1}^{k} \log P(-|w, c_{neg_i}) \right] \\ &= -\left[\log P(+|w, c_{pos}) + \sum_{i=1}^{k} \log \left(1 - P(+|w, c_{neg_i})\right) \right] \\ &= -\left[\log \sigma(c_{pos} \cdot w) + \sum_{i=1}^{k} \log \sigma(-c_{neg_i} \cdot w) \right] \end{aligned} \tag{5.21}$$

That is, we want to maximize the dot product of the word with the actual context words, and minimize the dot products of the word with the k negative sampled non-neighbor words.

We minimize this loss function using stochastic gradient descent. Fig. 5.7 shows the intuition of one step of learning.

To get the gradient, we need to take the derivative of Eq. 5.21 with respect to the different embeddings. It turns out the derivatives are the following (we leave the

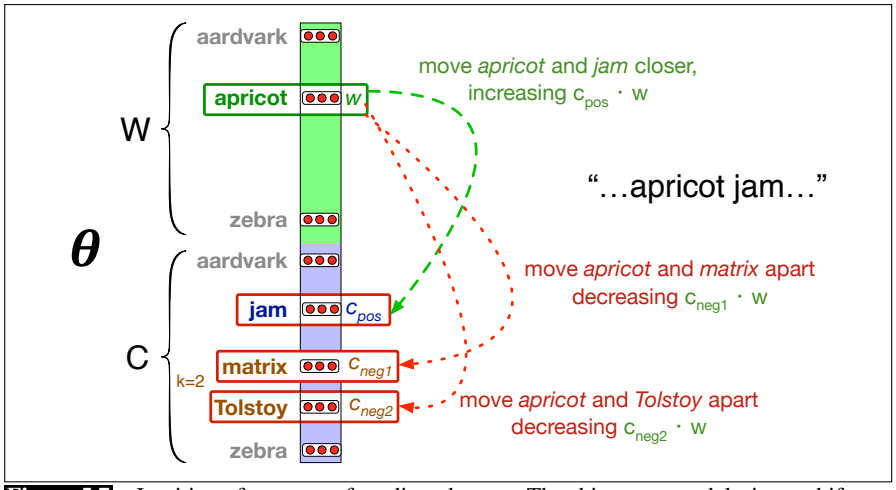


Figure 5.7 Intuition of one step of gradient descent. The skip-gram model tries to shift embeddings so the target embeddings (here for *apricot*) are closer to (have a higher dot product with) context embeddings for nearby words (here *jam*) and further from (lower dot product with) context embeddings for noise words that don't occur nearby (here *Tolstoy* and *matrix*).

proof as an exercise at the end of the chapter):

$$\frac{\partial L}{\partial c_{pos}} = [\sigma(\mathbf{c}_{pos} \cdot \mathbf{w}) - 1]\mathbf{w} \tag{5.22}$$

$$\frac{\partial L}{\partial c_{neg_i}} = [\sigma(\mathbf{c}_{neg_i} \cdot \mathbf{w})]\mathbf{w} \tag{5.23}$$

$$\frac{\partial L}{\partial w} = [\sigma(\mathbf{c}_{pos} \cdot \mathbf{w}) - 1]\mathbf{c}_{pos} + \sum_{i=1}^{k}[\sigma(\mathbf{c}_{neg_i} \cdot \mathbf{w})]\mathbf{c}_{neg_i} \tag{5.24}$$

The update equations going from time step t to $t+1$ in stochastic gradient descent are thus:

$$\mathbf{c}_{pos}^{t+1} = \mathbf{c}_{pos}^{t} - \eta[\sigma(\mathbf{c}_{pos}^{t} \cdot \mathbf{w}^{t}) - 1]\mathbf{w}^{t} \tag{5.25}$$

$$\mathbf{c}_{neg_i}^{t+1} = \mathbf{c}_{neg_i}^{t} - \eta[\sigma(\mathbf{c}_{neg_i}^{t} \cdot \mathbf{w}^{t})]\mathbf{w}^{t} \tag{5.26}$$

$$\mathbf{w}^{t+1} = \mathbf{w}^{t} - \eta\left[[\sigma(\mathbf{c}_{pos}^{t} \cdot \mathbf{w}^{t}) - 1]\mathbf{c}_{pos}^{t} + \sum_{i=1}^{k}[\sigma(\mathbf{c}_{neg_i}^{t} \cdot \mathbf{w}^{t})]\mathbf{c}_{neg_i}^{t}\right] \tag{5.27}$$

Just as in logistic regression, then, the learning algorithm starts with randomly initialized **W** and **C** matrices, and then walks through the training corpus using gradient descent to move **W** and **C** so as to minimize the loss in Eq. 5.21 by making the updates in (Eq. 5.25)-(Eq. 5.27).

target embedding
context embedding

Recall that the skip-gram model learns **two** separate embeddings for each word i: the **target embedding** $\mathbf{w}_i$ and the **context embedding** $\mathbf{c}_i$, stored in two matrices, the **target matrix W** and the **context matrix C**. It's common to just add them together, representing word i with the vector $\mathbf{w}_i + \mathbf{c}_i$. Alternatively we can throw away the **C** matrix and just represent each word i by the vector $\mathbf{w}_i$.

As with the simple count-based methods like tf-idf, the context window size affects the performance of skip-gram embeddings, and experiments often tune the context window size parameter on a devset.

5.5.3 Other kinds of static embeddings

fasttext

There are many kinds of static embeddings. An extension of word2vec, **fasttext** (Bojanowski et al., 2017), addresses a problem with word2vec as we have presented it so far: it has no good way to deal with **unknown words**—words that appear in a test corpus but were unseen in the training corpus. A related problem is word sparsity, such as in languages with rich morphology, where some of the many forms for each noun and verb may only occur rarely. Fasttext deals with these problems by using subword models, representing each word as itself plus a bag of constituent n-grams, with special boundary symbols < and > added to each word. For example, with $n = 3$ the word *where* would be represented by the sequence <where> plus the character n-grams:

<wh, whe, her, ere, re>

Then a skipgram embedding is learned for each constituent n-gram, and the word *where* is represented by the sum of all of the embeddings of its constituent n-grams. Unknown words can then be presented only by the sum of the constituent n-grams. A fasttext open-source library, including pretrained embeddings for 157 languages, is available at https://fasttext.cc.

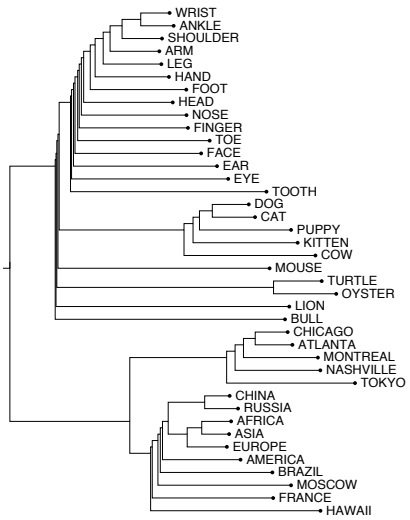
Another very widely used static embedding model is GloVe (Pennington et al., 2014), short for Global Vectors, because the model is based on capturing global corpus statistics. GloVe is based on ratios of probabilities from the word-word co-occurrence matrix.

It turns out that dense embeddings like word2vec actually have an elegant mathematical relationship with count-based embeddings, in which word2vec can be seen as implicitly optimizing a function of a count matrix with a particular (PPMI) weighting (Levy and Goldberg, 2014c).

5.6 Visualizing Embeddings

> "I see well in many dimensions as long as the dimensions are around two."
>
> The late economist Martin Shubik

Visualizing embeddings is an important goal in helping understand, apply, and improve these models of word meaning. But how can we visualize a (for example) 100-dimensional vector?



The simplest way to visualize the meaning of a word w embedded in a space is to list the most similar words to w by sorting the vectors for all words in the vocabulary by their cosine with the vector for w. For example the 7 closest words to *frog* using a particular set of embeddings computed with the GloVe algorithm are: *frogs*, *toad*, *litoria*, *leptodactylidae*, *rana*, *lizard*, and *eleutherodactylus* (Pennington et al., 2014).

Yet another visualization method is to use a clustering algorithm to show a hierarchical representation of which words are similar to others in the embedding space. The uncaptioned figure on the left uses hierarchical clustering of some embedding vectors for nouns as a visualization method (Rohde et al., 2006).

Probably the most common visualization method, however, is to project the 100 dimensions of a word down into 2 dimensions. Fig. 5.1 showed one such visualization, as does Fig. 5.9, using a projection method called t-SNE (van der Maaten and Hinton, 2008).

5.7 Semantic properties of embeddings

In this section we briefly summarize some of the semantic properties of embeddings that have been studied.

Different types of similarity or association: One parameter of vector semantic models that is relevant to both sparse PPMI vectors and dense word2vec vectors is the size of the context window used to collect counts. This is generally between 1 and 10 words on each side of the target word (for a total context of 2-20 words).

The choice depends on the goals of the representation. Shorter context windows tend to lead to representations that are a bit more syntactic, since the information is coming from immediately nearby words. When the vectors are computed from short context windows, the most similar words to a target word w tend to be semantically similar words with the same parts of speech. When vectors are computed from long context windows, the highest cosine words to a target word w tend to be words that are topically related but not similar.

For example Levy and Goldberg (2014a) showed that using skip-gram with a window of ± 2, the most similar words to the word *Hogwarts* (from the *Harry Potter* series) were names of other fictional schools: *Sunnydale* (from *Buffy the Vampire Slayer*) or *Evernight* (from a vampire series). With a window of ± 5, the most similar words to *Hogwarts* were other words topically related to the *Harry Potter* series: *Dumbledore*, *Malfoy*, and *half-blood*.

first-order co-occurrence

second-order co-occurrence

It's also often useful to distinguish two kinds of similarity or association between words (Schütze and Pedersen, 1993). Two words have **first-order co-occurrence** (sometimes called **syntagmatic association**) if they are typically nearby each other. Thus *wrote* is a first-order associate of *book* or *poem*. Two words have **second-order co-occurrence** (sometimes called **paradigmatic association**) if they have similar neighbors. Thus *wrote* is a second-order associate of words like *said* or *remarked*.

Analogy/Relational Similarity: Another semantic property of embeddings is their ability to capture relational meanings. In an important early vector space model of cognition, Rumelhart and Abrahamson (1973) proposed the **parallelogram model** for solving simple analogy problems of the form *a is to b as a* is to what?*. In such problems, a system is given a problem like *apple:tree::grape:?*, i.e., *apple is to tree as grape is to* _____, and must fill in the word *vine*. In the parallelogram model, illustrated in Fig. 5.8, the vector from the word *apple* to the word *tree* ($= \overrightarrow{\text{tree}} - \overrightarrow{\text{apple}}$) is added to the vector for *grape* ($\overrightarrow{\text{grape}}$); the nearest word to that point is returned.

parallelogram model

In early work with sparse embeddings, scholars showed that sparse vector models of meaning could solve such analogy problems (Turney and Littman, 2005), but the parallelogram method received more modern attention because of its success with word2vec or GloVe vectors (Mikolov et al. 2013c, Levy and Goldberg 2014b, Pennington et al. 2014). For example, the result of the expression $\overrightarrow{\text{king}} - \overrightarrow{\text{man}} + \overrightarrow{\text{woman}}$ is a vector close to $\overrightarrow{\text{queen}}$. Similarly, $\overrightarrow{\text{Paris}} - \overrightarrow{\text{France}} + \overrightarrow{\text{Italy}}$ results in a vector that is close to $\overrightarrow{\text{Rome}}$. The embedding model thus seems to be extract-

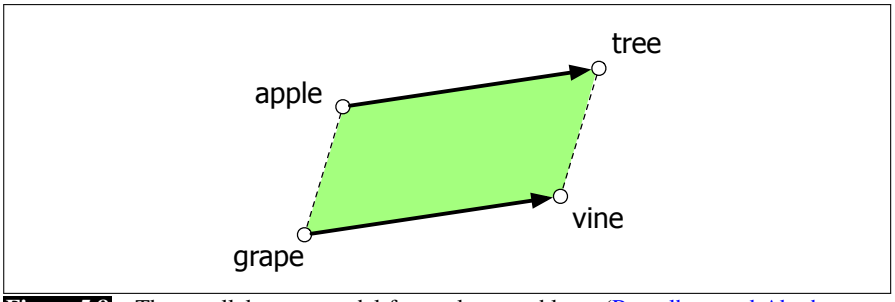


Figure 5.8 The parallelogram model for analogy problems (Rumelhart and Abrahamson, 1973): the location of $\overrightarrow{\text{vine}}$ can be found by subtracting $\overrightarrow{\text{apple}}$ from $\overrightarrow{\text{tree}}$ and adding $\overrightarrow{\text{grape}}$.

ing representations of relations like MALE-FEMALE, or CAPITAL-CITY-OF, or even COMPARATIVE/SUPERLATIVE, as shown in Fig. 5.9 from GloVe.

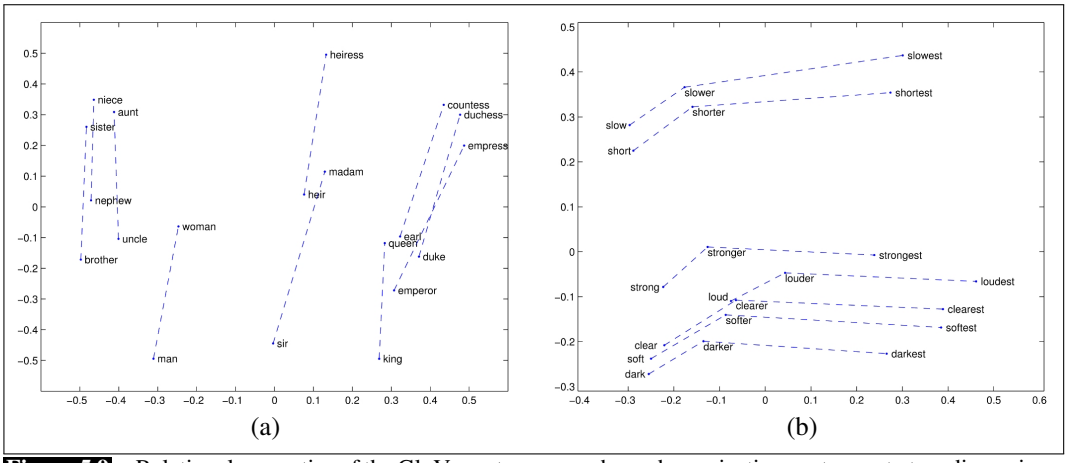


Figure 5.9 Relational properties of the GloVe vector space, shown by projecting vectors onto two dimensions. (a) $\overrightarrow{\text{king}} - \overrightarrow{\text{man}} + \overrightarrow{\text{woman}}$ is close to $\overrightarrow{\text{queen}}$. (b) offsets seem to capture comparative and superlative morphology (Pennington et al., 2014).

For a $\mathbf{a} : \mathbf{b} :: \mathbf{a}^* : \mathbf{b}^*$ problem, meaning the algorithm is given vectors $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{a}^*$ and must find $\mathbf{b}^*$, the parallelogram method is thus:

$$\hat{\mathbf{b}}^* = \underset{\mathbf{x}}{\text{argmin}}\ \text{distance}(\mathbf{x}, \mathbf{b} - \mathbf{a} + \mathbf{a}^*) \tag{5.28}$$

with some distance function, such as Euclidean distance.

There are some caveats. For example, the closest value returned by the parallelogram algorithm in word2vec or GloVe embedding spaces is usually not in fact b* but one of the 3 input words or their morphological variants (i.e., *cherry:red :: potato:x* returns *potato* or *potatoes* instead of *brown*), so these must be explicitly excluded. Furthermore while embedding spaces perform well if the task involves frequent words, small distances, and certain relations (like relating countries with their capitals or verbs/nouns with their inflected forms), the parallelogram method with embeddings doesn't work as well for other relations (Linzen 2016, Gladkova et al. 2016, Schluter 2018, Ethayarajh et al. 2019a), and indeed Peterson et al. (2020) argue that the parallelogram method is in general too simple to model the human cognitive process of forming analogies of this kind.

5.7.1 Embeddings and Historical Semantics

Embeddings can also be a useful tool for studying how meaning changes over time, by computing multiple embedding spaces, each from texts written in a particular time period. For example Fig. 5.10 shows a visualization of changes in meaning in English words over the last two centuries, computed by building separate embedding spaces for each decade from historical corpora like Google n-grams (Lin et al., 2012b) and the Corpus of Historical American English (Davies, 2012).

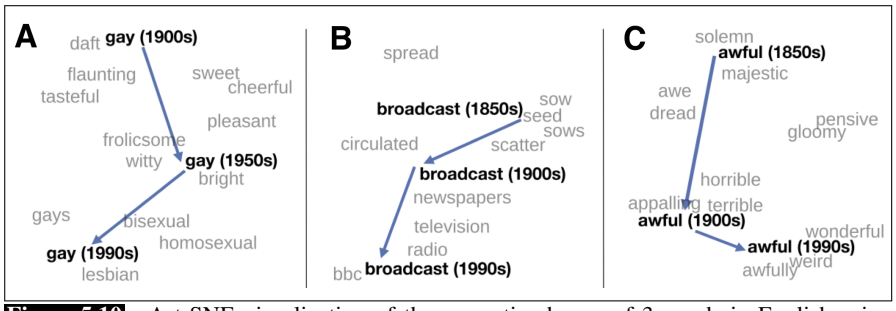


Figure 5.10 A t-SNE visualization of the semantic change of 3 words in English using word2vec vectors. The modern sense of each word, and the grey context words, are computed from the most recent (modern) time-point embedding space. Earlier points are computed from earlier historical embedding spaces. The visualizations show the changes in the word *gay* from meanings related to "cheerful" or "frolicsome" to referring to homosexuality, the development of the modern "transmission" sense of *broadcast* from its original sense of sowing seeds, and the pejoration of the word *awful* as it shifted from meaning "full of awe" to meaning "terrible or appalling" (Hamilton et al., 2016b).

5.8 Bias and Embeddings

In addition to their ability to learn word meaning from text, embeddings, alas, also reproduce the implicit biases and stereotypes that were latent in the text. As the prior section just showed, embeddings can roughly model relational similarity: 'queen' as the closest word to 'king' - 'man' + 'woman' implies the analogy *man:woman::king:queen*. But these same embedding analogies also exhibit gender stereotypes. For example Bolukbasi et al. (2016) find that the closest occupation to 'computer programmer' - 'man' + 'woman' in word2vec embeddings trained on news text is 'homemaker', and that the embeddings similarly suggest the analogy 'father' is to 'doctor' as 'mother' is to 'nurse'. This could result in what Crawford (2017) and Blodgett et al. (2020) call an **allocational harm**, when a system allocates resources (jobs or credit) unfairly to different groups. For example algorithms that use embeddings as part of a search for hiring potential programmers or doctors might thus incorrectly downweight documents with women's names.

allocational harm

It turns out that embeddings don't just reflect the statistics of their input, but also **amplify** bias; gendered terms become **more** gendered in embedding space than they were in the input text statistics (Zhao et al. 2017, Ethayarajh et al. 2019b, Jia et al. 2020), and biases are more exaggerated than in actual labor employment statistics (Garg et al., 2018).

bias amplification

Embeddings also encode the implicit associations that are a property of human reasoning. The Implicit Association Test (Greenwald et al., 1998) measures peo-

ple's associations between concepts (like 'flowers' or 'insects') and attributes (like 'pleasantness' and 'unpleasantness') by measuring differences in the latency with which they label words in the various categories.[3] Using such methods, people in the United States have been shown to associate African-American names with unpleasant words (more than European-American names), male names more with mathematics and female names with the arts, and old people's names with unpleasant words (Greenwald et al. 1998, Nosek et al. 2002a, Nosek et al. 2002b). Caliskan et al. (2017) replicated all these findings of implicit associations using GloVe vectors and cosine similarity instead of human latencies. For example African-American names like 'Leroy' and 'Shaniqua' had a higher GloVe cosine with unpleasant words while European-American names ('Brad', 'Greg', 'Courtney') had a higher cosine with pleasant words. These problems with embeddings are an example of a **representational harm** (Crawford 2017, Blodgett et al. 2020), which is a harm caused by a system demeaning or even ignoring some social groups. Any embedding-aware algorithm that made use of word sentiment could thus exacerbate bias against African Americans.

representational harm

Recent research focuses on ways to try to remove these kinds of biases, for example by developing a transformation of the embedding space that removes gender stereotypes but preserves definitional gender (Bolukbasi et al. 2016, Zhao et al. 2017) or changing the training procedure (Zhao et al., 2018b). However, although these sorts of **debiasing** may reduce bias in embeddings, they do not eliminate it (Gonen and Goldberg, 2019), and this remains an open problem.

debiasing

Historical embeddings are also being used to measure biases in the past. Garg et al. (2018) used embeddings from historical texts to measure the association between embeddings for occupations and embeddings for names of various ethnicities or genders (for example the relative cosine similarity of women's names versus men's to occupation words like 'librarian' or 'carpenter') across the 20th century. They found that the cosines correlate with the empirical historical percentages of women or ethnic groups in those occupations. Historical embeddings also replicated old surveys of ethnic stereotypes; the tendency of experimental participants in 1933 to associate adjectives like 'industrious' or 'superstitious' with, e.g., Chinese ethnicity, correlates with the cosine between Chinese last names and those adjectives using embeddings trained on 1930s text. They also were able to document historical gender biases, such as the fact that embeddings for adjectives related to competence ('smart', 'wise', 'thoughtful', 'resourceful') had a higher cosine with male than female words, and showed that this bias has been slowly decreasing since 1960. We return in later chapters to this question about the role of bias in natural language processing.

5.9 Evaluating Vector Models

The most important evaluation metric for vector models is extrinsic evaluation on tasks, i.e., using vectors in an NLP task and seeing whether this improves performance over some other model.

[3] Roughly speaking, if humans associate 'flowers' with 'pleasantness' and 'insects' with 'unpleasantness', when they are instructed to push a green button for 'flowers' (daisy, iris, lilac) and 'pleasant words' (love, laughter, pleasure) and a red button for 'insects' (flea, spider, mosquito) and 'unpleasant words' (abuse, hatred, ugly) they are faster than in an incongruous condition where they push a red button for 'flowers' and 'unpleasant words' and a green button for 'insects' and 'pleasant words'.

Nonetheless it is useful to have intrinsic evaluations. The most common metric is to test their performance on **similarity**, computing the correlation between an algorithm's word similarity scores and word similarity ratings assigned by humans. **WordSim-353** (Finkelstein et al., 2002) is a commonly used set of ratings from 0 to 10 for 353 noun pairs; for example (*plane*, *car*) had an average score of 5.77. **SimLex-999** (Hill et al., 2015) is a more complex dataset that quantifies similarity (*cup, mug*) rather than relatedness (*cup, coffee*), and includes concrete and abstract adjective, noun and verb pairs. The **TOEFL dataset** is a set of 80 questions, each consisting of a target word with 4 additional word choices; the task is to choose which is the correct synonym, as in the example: *Levied is closest in meaning to: imposed, believed, requested, correlated* (Landauer and Dumais, 1997). All of these datasets present words without context.

Slightly more realistic are intrinsic similarity tasks that include context. The Stanford Contextual Word Similarity (SCWS) dataset (Huang et al., 2012) and the Word-in-Context (WiC) dataset (Pilehvar and Camacho-Collados, 2019) offer richer evaluation scenarios. SCWS gives human judgments on 2,003 pairs of words in their sentential context, while WiC gives target words in two sentential contexts that are either in the same or different senses; see Appendix G. The *semantic textual similarity* task (Agirre et al. 2012, Agirre et al. 2015) evaluates the performance of sentence-level similarity algorithms, consisting of a set of pairs of sentences, each pair with human-labeled similarity scores.

Another task used for evaluation is the analogy task, discussed on page 112, where the system has to solve problems of the form *a is to b as a* is to b**, given *a, b,* and *a** and having to find *b** (Turney and Littman, 2005). A number of sets of tuples have been created for this task (Mikolov et al. 2013a, Mikolov et al. 2013c, Gladkova et al. 2016), covering morphology (*city:cities::child:children*), lexicographic relations (*leg:table::spout:teapot*) and encyclopedia relations (*Beijing:China::Dublin:Ireland*), some drawing from the SemEval-2012 Task 2 dataset of 79 different relations (Jurgens et al., 2012).

All embedding algorithms suffer from inherent variability. For example because of randomness in the initialization and the random negative sampling, algorithms like word2vec may produce different results even from the same dataset, and individual documents in a collection may strongly impact the resulting embeddings (Tian et al. 2016, Hellrich and Hahn 2016, Antoniak and Mimno 2018). When embeddings are used to study word associations in particular corpora, therefore, it is best practice to train multiple embeddings with bootstrap sampling over documents and average the results (Antoniak and Mimno, 2018).

5.10 Summary

- In vector semantics, a word is modeled as a vector—a point in high-dimensional space, also called an **embedding**. In this chapter we focus on **static embeddings**, where each word is mapped to a fixed embedding.
- Vector semantic models fall into two classes: **sparse** and **dense**. In sparse models each dimension corresponds to a word in the vocabulary V and cells are functions of **co-occurrence counts**. The **word-context** or **term-term** matrix has a row for each (target) word in the vocabulary and a column for each context term in the vocabulary.

- Dense vector models typically have dimensionality 50–1000. **Word2vec** algorithms like **skip-gram** are a popular way to compute dense embeddings. Skip-gram trains a logistic regression classifier to compute the probability that two words are 'likely to occur nearby in text'. This probability is computed from the dot product between the embeddings for the two words.
- Skip-gram uses stochastic gradient descent to train the classifier, by learning embeddings that have a high dot product with embeddings of words that occur nearby and a low dot product with noise words.
- Other important embedding algorithms include **GloVe**, a method based on ratios of word co-occurrence probabilities.
- Whether using sparse or dense vectors, word and document similarities are computed by some function of the **dot product** between vectors. The cosine of two vectors—a normalized dot product—is the most popular such metric.

Historical Notes

The idea of vector semantics arose out of research in the 1950s in three distinct fields: linguistics, psychology, and computer science, each of which contributed a fundamental aspect of the model.

The idea that meaning is related to the distribution of words in context was widespread in linguistic theory of the 1950s, among distributionalists like Zellig Harris, Martin Joos, and J. R. Firth, and semioticians like Thomas Sebeok. As Joos (1950) put it,

> the linguist's "meaning" of a morpheme... is by definition the set of conditional probabilities of its occurrence in context with all other morphemes.

The idea that the meaning of a word might be modeled as a point in a multidimensional semantic space came from psychologists like Charles E. Osgood, who had been studying how people responded to the meaning of words by assigning values along scales like *happy/sad* or *hard/soft*. Osgood et al. (1957) proposed that the meaning of a word in general could be modeled as a point in a multidimensional Euclidean space, and that the similarity of meaning between two words could be modeled as the distance between these points in the space.

mechanical indexing

A final intellectual source in the 1950s and early 1960s was the field then called **mechanical indexing**, now known as **information retrieval**. In what became known as the **vector space model** for information retrieval (Salton 1971, Sparck Jones 1986), researchers demonstrated new ways to define the meaning of words in terms of vectors (Switzer, 1965), and refined methods for word similarity based on measures of statistical association between words like mutual information (Giuliano, 1965) and idf (Sparck Jones, 1972), and showed that the meaning of documents could be represented in the same vector spaces used for words. Around the same time, (Cordier, 1965) showed that factor analysis of word association probabilities could be used to form dense vector representations of words.

Some of the philosophical underpinning of the distributional way of thinking came from the late writings of the philosopher Wittgenstein, who was skeptical of the possibility of building a completely formal theory of meaning definitions for each word. Wittgenstein suggested instead that "the meaning of a word is its use in the language" (Wittgenstein, 1953, PI 43). That is, instead of using some logical language to define each word, or drawing on denotations or truth values, Wittgenstein's

idea is that we should define a word by how it is used by people in speaking and understanding in their day-to-day interactions, thus prefiguring the movement toward embodied and experiential models in linguistics and NLP (Glenberg and Robertson 2000, Lake and Murphy 2021, Bisk et al. 2020, Bender and Koller 2020).

More distantly related is the idea of defining words by a vector of discrete features, which has roots at least as far back as Descartes and Leibniz (Wierzbicka 1992, Wierzbicka 1996). By the middle of the 20th century, beginning with the work of Hjelmslev (Hjelmslev, 1969) (originally 1943) and fleshed out in early models of generative grammar (Katz and Fodor, 1963), the idea arose of representing meaning with **semantic features**, symbols that represent some sort of primitive meaning. For example words like *hen*, *rooster*, or *chick*, have something in common (they all describe chickens) and something different (their age and sex), representable as:

semantic feature

hen	`+female, +chicken, +adult`
rooster	`-female, +chicken, +adult`
chick	`+chicken, -adult`

The dimensions used by vector models of meaning to define words, however, are only abstractly related to this idea of a small fixed number of hand-built dimensions. Nonetheless, there has been some attempt to show that certain dimensions of embedding models do contribute some specific compositional aspect of meaning like these early semantic features.

The use of dense vectors to model word meaning, and indeed the term **embedding**, grew out of the **latent semantic indexing** (LSI) model (Deerwester et al., 1988) recast as **LSA** (**latent semantic analysis**) (Deerwester et al., 1990). In LSA **singular value decomposition**—**SVD**— is applied to a term-document matrix (each cell weighted by log frequency and normalized by entropy), and then the first 300 dimensions are used as the LSA embedding. Singular Value Decomposition (SVD) is a method for finding the most important dimensions of a dataset, those dimensions along which the data varies the most. LSA was then quickly widely applied: as a cognitive model (Landauer and Dumais, 1997), and for tasks like spell checking (Jones and Martin, 1997), language modeling (Bellegarda 1997, Coccaro and Jurafsky 1998, Bellegarda 2000), morphology induction (Schone and Jurafsky 2000, Schone and Jurafsky 2001b), multiword expressions (MWEs) (Schone and Jurafsky, 2001a), and essay grading (Rehder et al., 1998). Related models were simultaneously developed and applied to word sense disambiguation by Schütze (1992b). LSA also led to the earliest use of embeddings to represent words in a probabilistic classifier, in the logistic regression document router of Schütze et al. (1995). The idea of SVD on the term-term matrix (rather than the term-document matrix) as a model of meaning for NLP was proposed soon after LSA by Schütze (1992b). Schütze applied the low-rank (97-dimensional) embeddings produced by SVD to the task of word sense disambiguation, analyzed the resulting semantic space, and also suggested possible techniques like dropping high-order dimensions. See Schütze (1997).

SVD

A number of alternative matrix models followed on from the early SVD work, including Probabilistic Latent Semantic Indexing (PLSI) (Hofmann, 1999), Latent Dirichlet Allocation (LDA) (Blei et al., 2003), and Non-negative Matrix Factorization (NMF) (Lee and Seung, 1999).

The LSA community seems to have first used the word "embedding" in Landauer et al. (1997), in a variant of its mathematical meaning as a mapping from one space or mathematical structure to another. In LSA, the word embedding seems to have described the mapping from the space of sparse count vectors to the latent space of

SVD dense vectors. Although the word thus originally meant the mapping from one space to another, it has metonymically shifted to mean the resulting dense vector in the latent space, and it is in this sense that we currently use the word.

By the next decade, Bengio et al. (2003) and Bengio et al. (2006) showed that neural language models could also be used to develop embeddings as part of the task of word prediction. Collobert and Weston (2007), Collobert and Weston (2008), and Collobert et al. (2011) then demonstrated that embeddings could be used to represent word meanings for a number of NLP tasks. Turian et al. (2010) compared the value of different kinds of embeddings for different NLP tasks. Mikolov et al. (2011) showed that recurrent neural nets could be used as language models. The idea of simplifying the hidden layer of these neural net language models to create the skip-gram (and also CBOW) algorithms was proposed by Mikolov et al. (2013a). The negative sampling training algorithm was proposed in Mikolov et al. (2013b). There are numerous surveys of static embeddings and their parameterizations (Bullinaria and Levy 2007, Bullinaria and Levy 2012, Lapesa and Evert 2014, Kiela and Clark 2014, Levy et al. 2015).

See Manning et al. (2008) and Chapter 11 for a deeper understanding of the role of vectors in information retrieval, including how to compare queries with documents, more details on tf-idf, and issues of scaling to very large datasets. See Kim (2019) for a clear and comprehensive tutorial on word2vec. Cruse (2004) is a useful introductory linguistic text on lexical semantics.

Exercises

CHAPTER

6 Neural Networks

> "[M]achines of this character can behave in a very complicated manner when the number of units is large."
>
> Alan Turing (1948) "Intelligent Machines", page 6

Neural networks are a fundamental computational tool for language processing, and a very old one. They are called neural because their origins lie in the **McCulloch-Pitts neuron** (McCulloch and Pitts, 1943), a simplified model of the biological neuron as a kind of computing element that could be described in terms of propositional logic. But the modern use in language processing no longer draws on these early biological inspirations.

Instead, a modern neural network is a network of small computing units, each of which takes a vector of input values and produces a single output value. In this chapter we introduce the neural net applied to classification. The architecture we introduce is called a **feedforward network** because the computation proceeds iteratively from one layer of units to the next. The use of modern neural nets is often called **deep learning**, because modern networks are often **deep** (have many layers).

feedforward

deep learning

Neural networks share much of the same mathematics as logistic regression. But neural networks are a more powerful classifier than logistic regression, and indeed a minimal neural network (technically one with a single 'hidden layer') can be shown to learn any function.

Neural net classifiers are different from logistic regression in another way. With logistic regression, we applied the regression classifier to many different tasks by developing many rich kinds of feature templates based on domain knowledge. When working with neural networks, it is more common to avoid most uses of rich hand-derived features, instead building neural networks that take raw tokens as inputs and learn to induce features as part of the process of learning to classify. We saw examples of this kind of representation learning for embeddings in Chapter 5, and we'll see lots of examples once we start studying deep transformers networks. Nets that are very deep are particularly good at representation learning. For that reason deep neural nets are the right tool for tasks that offer sufficient data to learn features automatically.

In this chapter we'll introduce feedforward networks as classifiers, first with hand-built features, and then using the embeddings that we studied in Chapter 5. In subsequent chapters we'll introduce many other kinds of neural models, most importantly the **transformer** and **attention**, (Chapter 8), but also **recurrent neural networks** (Chapter 13) and **convolutional neural networks** (Chapter 15). And in the next chapter we'll introduce the paradigm of neural large language models.

6.1 Units

The building block of a neural network is a single computational unit. A unit takes a set of real valued numbers as input, performs some computation on them, and produces an output.

At its heart, a neural unit is taking a weighted sum of its inputs, with one additional term in the sum called a **bias term**. Given a set of inputs $x_1 ... x_n$, a unit has a set of corresponding weights $w_1 ... w_n$ and a bias b, so the weighted sum z can be represented as:

bias term

$$z = b + \sum_i w_i x_i \tag{6.1}$$

Often it's more convenient to express this weighted sum using vector notation; recall from linear algebra that a **vector** is, at heart, just a list or array of numbers. Thus we'll talk about z in terms of a weight vector w, a scalar bias b, and an input vector x, and we'll replace the sum with the convenient **dot product**:

vector

$$z = \mathbf{w} \cdot \mathbf{x} + b \tag{6.2}$$

As defined in Eq. 6.2, z is just a real valued number.

Finally, instead of using z, a linear function of x, as the output, neural units apply a non-linear function f to z. We will refer to the output of this function as the **activation** value for the unit, a. Since we are just modeling a single unit, the activation for the node is in fact the final output of the network, which we'll generally call y. So the value y is defined as:

activation

$$y = a = f(z)$$

We'll discuss three popular non-linear functions f below (the sigmoid, the tanh, and the rectified linear unit or ReLU) but it's pedagogically convenient to start with the **sigmoid** function since we saw it in Chapter 4:

sigmoid

$$y = \sigma(z) = \frac{1}{1+e^{-z}} \tag{6.3}$$

The sigmoid (shown in Fig. 6.1) has a number of advantages; it maps the output into the range $(0,1)$, which is useful in squashing outliers toward 0 or 1. And it's differentiable, which as we saw in Section 4.15 will be handy for learning.

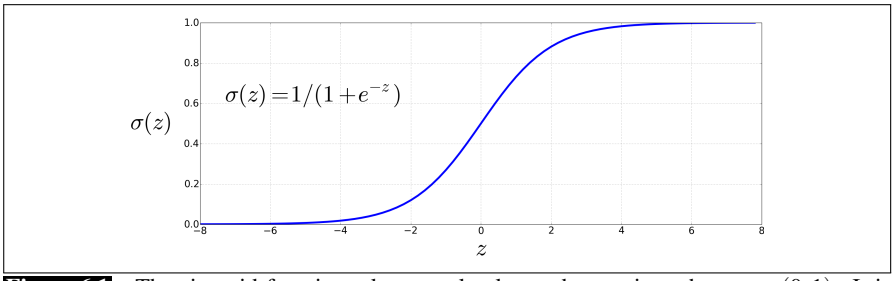


Figure 6.1 The sigmoid function takes a real value and maps it to the range $(0,1)$. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

Substituting Eq. 6.2 into Eq. 6.3 gives us the output of a neural unit:

$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1+\exp(-(\mathbf{w} \cdot \mathbf{x} + b))} \tag{6.4}$$

Fig. 6.2 shows a final schematic of a basic neural unit. In this example the unit takes 3 input values x_1, x_2, and x_3, and computes a weighted sum, multiplying each value by a weight (w_1, w_2, and w_3, respectively), adds them to a bias term b, and then passes the resulting sum through a sigmoid function to result in a number between 0 and 1.

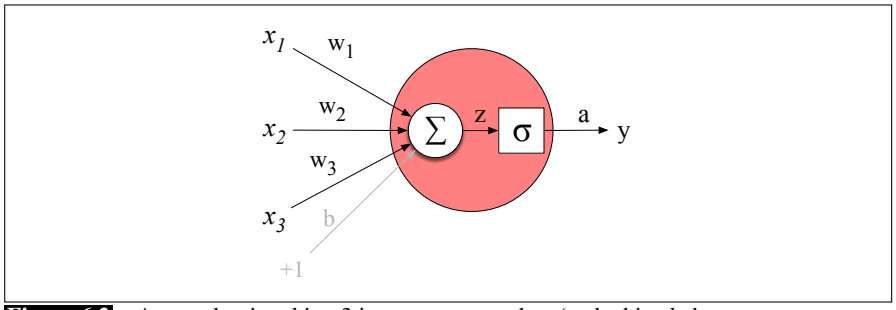


Figure 6.2 A neural unit, taking 3 inputs x_1, x_2, and x_3 (and a bias b that we represent as a weight for an input clamped at +1) and producing an output y. We include some convenient intermediate variables: the output of the summation, z, and the output of the sigmoid, a. In this case the output of the unit y is the same as a, but in deeper networks we'll reserve y to mean the final output of the entire network, leaving a as the activation of an individual node.

Let's walk through an example just to get an intuition. Let's suppose we have a unit with the following weight vector and bias:

$$\begin{aligned} \mathbf{w} &= [0.2, 0.3, 0.9] \\ b &= 0.5 \end{aligned}$$

What would this unit do with the following input vector:

$$\mathbf{x} = [0.5, 0.6, 0.1]$$

The resulting output y would be:

$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1+e^{-(\mathbf{w} \cdot \mathbf{x} + b)}} = \frac{1}{1+e^{-(.5*.2+.6*.3+.1*.9+.5)}} = \frac{1}{1+e^{-0.87}} = .70$$

tanh

In practice, the sigmoid is not commonly used as an activation function. A function that is very similar but almost always better is the **tanh** function shown in Fig. 6.3a; tanh is a variant of the sigmoid that ranges from -1 to +1:

$$y = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{6.5}$$

ReLU

The simplest activation function, and perhaps the most commonly used, is the rectified linear unit, also called the **ReLU**, shown in Fig. 6.3b. It's just the same as z when z is positive, and 0 otherwise:

$$y = \text{ReLU(z)} = max(z, 0) \tag{6.6}$$

These activation functions have different properties that make them useful for different language applications or network architectures. For example, the tanh function has the nice properties of being smoothly differentiable and mapping outlier values toward the mean. The rectifier function, on the other hand, has nice properties that

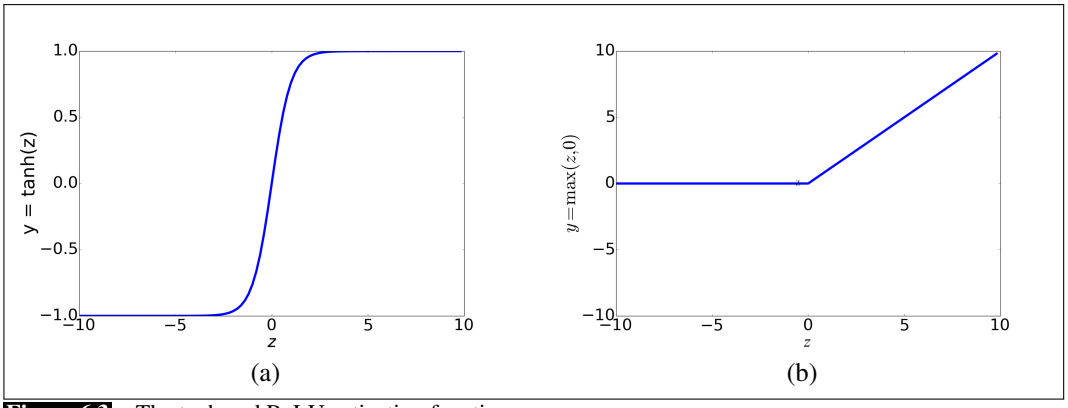


Figure 6.3 The tanh and ReLU activation functions.

saturated

result from it being very close to linear. In the sigmoid or tanh functions, very high values of z result in values of y that are **saturated**, i.e., extremely close to 1, and have derivatives very close to 0. Zero derivatives cause problems for learning, because as we'll see in Section 6.6, we'll train networks by propagating an error signal backwards, multiplying gradients (partial derivatives) from each layer of the network; gradients that are almost 0 cause the error signal to get smaller and smaller until it is too small to be used for training, a problem called the **vanishing gradient** problem. Rectifiers don't have this problem, since the derivative of ReLU for high values of z is 1 rather than very close to 0.

vanishing gradient

6.2 The XOR problem

Early in the history of neural networks it was realized that the power of neural networks, as with the real neurons that inspired them, comes from combining these units into larger networks.

One of the most clever demonstrations of the need for multi-layer networks was the proof by Minsky and Papert (1969) that a single neural unit cannot compute some very simple functions of its input. Consider the task of computing elementary logical functions of two inputs, like AND, OR, and XOR. As a reminder, here are the truth tables for those functions:

AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

perceptron

This example was first shown for the **perceptron**, which is a very simple neural unit that has a binary output and has a very simple step function as its non-linear activation function. The output y of a perceptron is 0 or 1, and is computed as follows (using the same weight $\mathbf{w}$, input $\mathbf{x}$, and bias b as in Eq. 6.2):

$$y = \begin{cases} 0, & \text{if } \mathbf{w}\cdot\mathbf{x}+b \leq 0 \\ 1, & \text{if } \mathbf{w}\cdot\mathbf{x}+b > 0 \end{cases} \tag{6.7}$$

It's very easy to build a perceptron that can compute the logical AND and OR functions of its binary inputs; Fig. 6.4 shows the necessary weights.

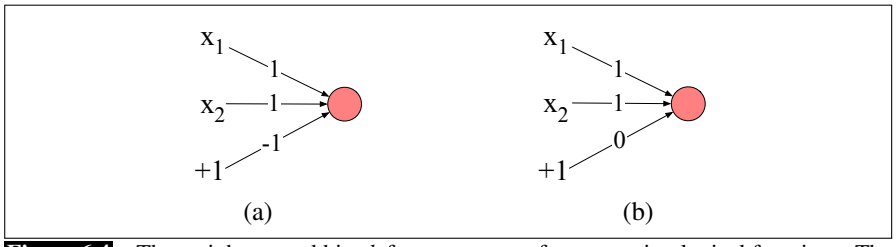


Figure 6.4 The weights w and bias b for perceptrons for computing logical functions. The inputs are shown as x_1 and x_2 and the bias as a special node with value $+1$ which is multiplied with the bias weight b. (a) logical AND, with weights $w_1 = 1$ and $w_2 = 1$ and bias weight $b = -1$. (b) logical OR, with weights $w_1 = 1$ and $w_2 = 1$ and bias weight $b = 0$. These weights/biases are just one from an infinite number of possible sets of weights and biases that would implement the functions.

It turns out, however, that it's not possible to build a perceptron to compute logical XOR! (It's worth spending a moment to give it a try!)

The intuition behind this important result relies on understanding that a perceptron is a linear classifier. For a two-dimensional input x_1 and x_2, the perceptron equation, $w_1x_1 + w_2x_2 + b = 0$ is the equation of a line. (We can see this by putting it in the standard linear format: $x_2 = (-w_1/w_2)x_1 + (-b/w_2)$.) This line acts as a **decision boundary** in two-dimensional space in which the output 0 is assigned to all inputs lying on one side of the line, and the output 1 to all input points lying on the other side of the line. If we had more than 2 inputs, the decision boundary becomes a hyperplane instead of a line, but the idea is the same, separating the space into two categories.

decision boundary

Fig. 6.5 shows the possible logical inputs (00, 01, 10, and 11) and the line drawn by one possible set of parameters for an AND and an OR classifier. Notice that there is simply no way to draw a line that separates the positive cases of XOR (01 and 10) from the negative cases (00 and 11). We say that XOR is not a **linearly separable** function. Of course we could draw a boundary with a curve, or some other function, but not a single line.

linearly separable

6.2.1 The solution: neural networks

While the XOR function cannot be calculated by a single perceptron, it can be calculated by a layered network of perceptron units. Rather than see this with networks of simple perceptrons, however, let's see how to compute XOR using two layers of ReLU-based units following Goodfellow et al. (2016). Fig. 6.6 shows a figure with the input being processed by two layers of neural units. The middle layer (called h) has two units, and the output layer (called y) has one unit. A set of weights and biases are shown that allows the network to correctly compute the XOR function.

Let's walk through what happens with the input $\mathbf{x} = [0, 0]$. If we multiply each input value by the appropriate weight, sum, and then add the bias b, we get the vector [0, -1], and we then apply the rectified linear transformation to give the output of the $\mathbf{h}$ layer as [0, 0]. Now we once again multiply by the weights, sum, and add the bias (0 in this case) resulting in the value 0. The reader should work through the computation of the remaining 3 possible input pairs to see that the resulting y values are 1 for the inputs [0, 1] and [1, 0] and 0 for [0, 0] and [1, 1].

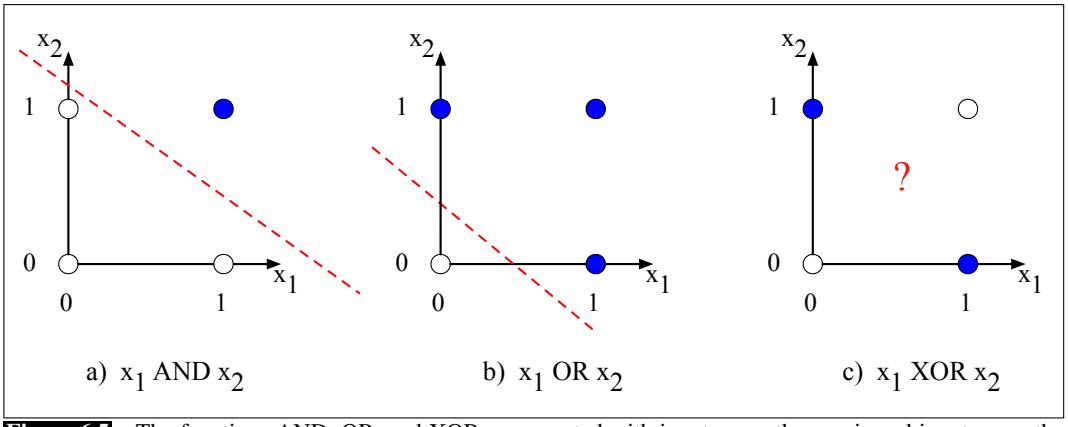


Figure 6.5 The functions AND, OR, and XOR, represented with input x_1 on the x-axis and input x_2 on the y-axis. Filled circles represent perceptron outputs of 1, and white circles perceptron outputs of 0. There is no way to draw a line that correctly separates the two categories for XOR. Figure styled after Russell and Norvig (2002).

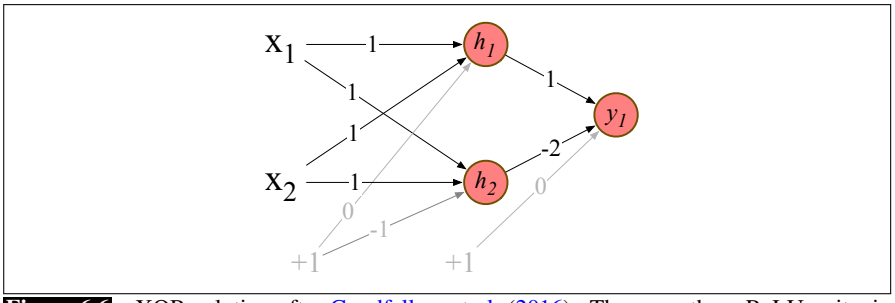


Figure 6.6 XOR solution after Goodfellow et al. (2016). There are three ReLU units, in two layers; we've called them h_1, h_2 (h for "hidden layer") and y_1. As before, the numbers on the arrows represent the weights w for each unit, and we represent the bias b as a weight on a unit clamped to +1, with the bias weights/units in gray.

It's also instructive to look at the intermediate results, the outputs of the two hidden nodes h_1 and h_2. We showed in the previous paragraph that the $\mathbf{h}$ vector for the inputs $\mathbf{x} = [0, 0]$ was $[0, 0]$. Fig. 6.7b shows the values of the $\mathbf{h}$ layer for all 4 inputs. Notice that hidden representations of the two input points $\mathbf{x} = [0, 1]$ and $\mathbf{x} = [1, 0]$ (the two cases with XOR output = 1) are merged to the single point $\mathbf{h} = [1, 0]$. The merger makes it easy to linearly separate the positive and negative cases of XOR. In other words, we can view the hidden layer of the network as forming a representation of the input.

In this example we just stipulated the weights in Fig. 6.6. But for real examples the weights for neural networks are learned automatically using the error backpropagation algorithm to be introduced in Section 6.6. That means the hidden layers will learn to form useful representations. This intuition, that neural networks can automatically learn useful representations of the input, is one of their key advantages, and one that we will return to again and again in later chapters.

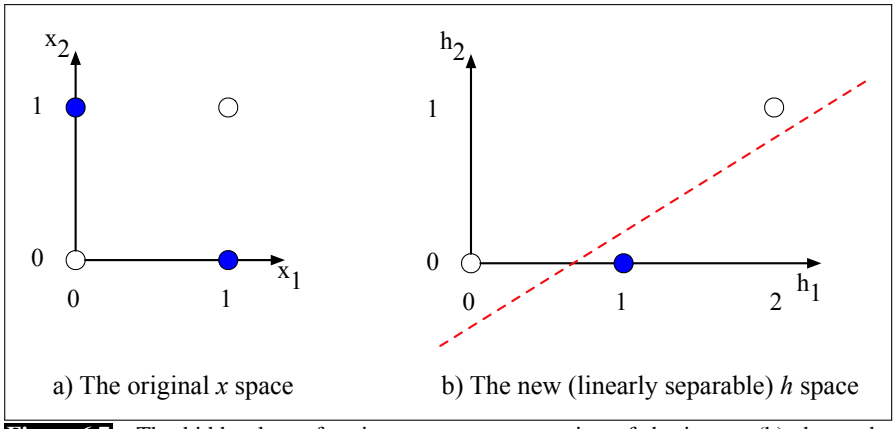


Figure 6.7 The hidden layer forming a new representation of the input. (b) shows the representation of the hidden layer, **h**, compared to the original input representation **x** in (a). Notice that the input point [0, 1] has been collapsed with the input point [1, 0], making it possible to linearly separate the positive and negative cases of XOR. After Goodfellow et al. (2016).

6.3 Feedforward Neural Networks

feedforward network

Let's now walk through a slightly more formal presentation of the simplest kind of neural network, the **feedforward network**. A feedforward network is a multilayer network in which the units are connected with no cycles; the outputs from units in each layer are passed to units in the next higher layer, and no outputs are passed back to lower layers. (In Chapter 13 we'll introduce networks with cycles, called **recurrent neural networks**.)

multi-layer perceptrons **MLP**

For historical reasons multilayer networks, especially feedforward networks, are sometimes called **multi-layer perceptrons** (or **MLP**s); this is a technical misnomer, since the units in modern multilayer networks aren't perceptrons (perceptrons have a simple step-function as their activation function, but modern networks are made up of units with many kinds of non-linearities like ReLUs and sigmoids), but at some point the name stuck.

Simple feedforward networks have three kinds of nodes: input units, hidden units, and output units.

Fig. 6.8 shows a picture. The input layer **x** is a vector of simple scalar values just as we saw in Fig. 6.2.

hidden layer

The core of the neural network is the **hidden layer h** formed of **hidden units** $\mathbf{h}_i$, each of which is a neural unit as described in Section 6.1, taking a weighted sum of its inputs and then applying a non-linearity. In the standard architecture, each layer is **fully-connected**, meaning that each unit in each layer takes as input the outputs from all the units in the previous layer, and there is a link between every pair of units from two adjacent layers. Thus each hidden unit sums over all the input units.

fully-connected

Recall that a single hidden unit has as parameters a weight vector and a bias. We represent the parameters for the entire hidden layer by combining the weight vector and bias for each unit i into a single weight matrix $\mathbf{W}$ and a single bias vector b for the whole layer (see Fig. 6.8). Each element $\mathbf{W}_{ji}$ of the weight matrix $\mathbf{W}$ represents the weight of the connection from the ith input unit x_i to the jth hidden unit h_j.

The advantage of using a single matrix $\mathbf{W}$ for the weights of the entire layer is that now the hidden layer computation for a feedforward network can be done very

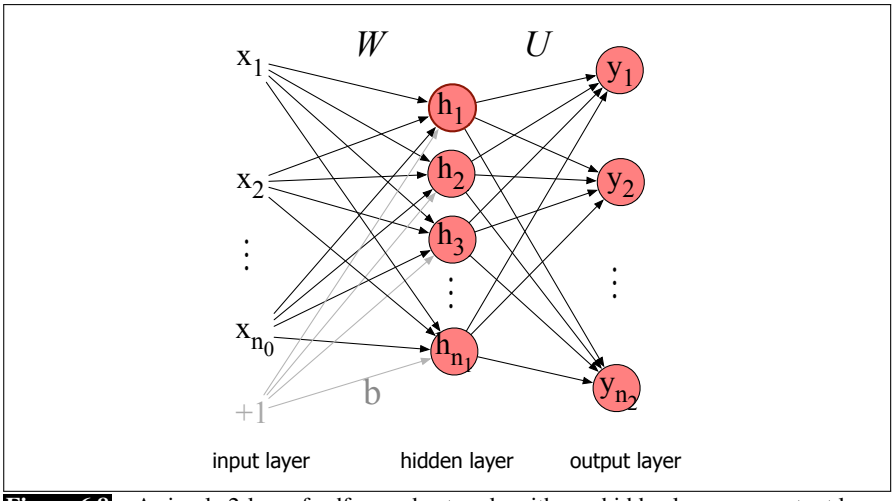


Figure 6.8 A simple 2-layer feedforward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

efficiently with simple matrix operations. In fact, the computation only has three steps: multiplying the weight matrix by the input vector $\mathbf{x}$, adding the bias vector $\mathbf{b}$, and applying the activation function g (such as the sigmoid, tanh, or ReLU activation function defined above).

The output of the hidden layer, the vector $\mathbf{h}$, is thus the following (for this example we'll use the sigmoid function σ as our activation function):

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{6.8}$$

Notice that we're applying the σ function here to a vector, while in Eq. 6.3 it was applied to a scalar. We're thus allowing $\sigma(\cdot)$, and indeed any activation function $g(\cdot)$, to apply to a vector element-wise, so $g[z_1, z_2, z_3] = [g(z_1), g(z_2), g(z_3)]$.

Let's introduce some constants to represent the dimensionalities of these vectors and matrices. We'll refer to the input layer as layer 0 of the network, and have n_0 represent the number of inputs, so $\mathbf{x}$ is a vector of real numbers of dimension n_0, or more formally $\mathbf{x} \in \mathbb{R}^{n_0}$, a column vector of dimensionality $[n_0 \times 1]$. Let's call the hidden layer layer 1 and the output layer layer 2. The hidden layer has dimensionality n_1, so $\mathbf{h} \in \mathbb{R}^{n_1}$ and also $\mathbf{b} \in \mathbb{R}^{n_1}$ (since each hidden unit can take a different bias value). And the weight matrix $\mathbf{W}$ has dimensionality $\mathbf{W} \in \mathbb{R}^{n_1 \times n_0}$, i.e. $[n_1 \times n_0]$.

Take a moment to convince yourself that the matrix multiplication in Eq. 6.8 will compute the value of each $\mathbf{h}_j$ as $\sigma\left(\sum_{i=1}^{n_0} \mathbf{W}_{ji}\mathbf{x}_i + \mathbf{b}_j\right)$.

As we saw in Section 6.2, the resulting value $\mathbf{h}$ (for *hidden* but also for *hypothesis*) forms a *representation* of the input. The role of the output layer is to take this new representation $\mathbf{h}$ and compute a final output. This output could be a real-valued number, but in many cases the goal of the network is to make some sort of classification decision, and so we will focus on the case of classification.

If we are doing a binary task like sentiment classification, we might have a single output node, and its scalar value y is the probability of positive versus negative sentiment. If we are doing multinomial classification, such as assigning a part-of-speech tag, we might have one output node for each potential part-of-speech, whose output value is the probability of that part-of-speech, and the values of all the output nodes must sum to one. The output layer is thus a vector $\mathbf{y}$ that gives a probability distribution across the output nodes.

Let's see how this happens. Like the hidden layer, the output layer has a weight matrix (let's call it $\mathbf{U}$), but some models don't include a bias vector $\mathbf{b}$ in the output layer, so we'll simplify by eliminating the bias vector in this example. The weight matrix is multiplied by its input vector ($\mathbf{h}$) to produce the intermediate output $\mathbf{z}$:

$$\mathbf{z} = \mathbf{U}\mathbf{h}$$

There are n_2 output nodes, so $\mathbf{z} \in \mathbb{R}^{n_2}$, weight matrix $\mathbf{U}$ has dimensionality $\mathbf{U} \in \mathbb{R}^{n_2 \times n_1}$, and element $\mathbf{U}_{ij}$ is the weight from unit j in the hidden layer to unit i in the output layer.

However, $\mathbf{z}$ can't be the output of the classifier, since it's a vector of real-valued numbers, while what we need for classification is a vector of probabilities. There is a convenient function for **normalizing** a vector of real values, by which we mean converting it to a vector that encodes a probability distribution (all the numbers lie between 0 and 1 and sum to 1): the **softmax** function that we saw on page 79 of Chapter 4. More generally for any vector $\mathbf{z}$ of dimensionality d, the softmax is defined as:

normalizing

softmax

$$\text{softmax}(\mathbf{z}_i) = \frac{\exp(\mathbf{z}_i)}{\sum_{j=1}^{d} \exp(\mathbf{z}_j)} \quad 1 \le i \le d \tag{6.9}$$

Thus for example given a vector

$$\mathbf{z} = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1], \tag{6.10}$$

the softmax function will normalize it to a probability distribution (shown rounded):

$$\text{softmax}(\mathbf{z}) = [0.055, 0.090, 0.0067, 0.10, 0.74, 0.010] \tag{6.11}$$

You may recall that we used softmax to create a probability distribution from a vector of real-valued numbers (computed from summing weights times features) in the multinomial version of logistic regression in Chapter 4.

That means we can think of a neural network classifier with one hidden layer as building a vector $\mathbf{h}$ which is a hidden layer representation of the input, and then running standard multinomial logistic regression on the features that the network develops in $\mathbf{h}$. By contrast, in Chapter 4 the features were mainly designed by hand via feature templates. So a neural network is like multinomial logistic regression, but (a) with many layers, since a deep neural network is like layer after layer of logistic regression classifiers; (b) with those intermediate layers having many possible activation functions (tanh, ReLU, sigmoid) instead of just sigmoid (although we'll continue to use σ for convenience to mean any activation function); (c) rather than forming the features by feature templates, the prior layers of the network induce the feature representations themselves.

Here are the final equations for a feedforward network with a single hidden layer, which takes an input vector $\mathbf{x}$, outputs a probability distribution $\mathbf{y}$, and is parameterized by weight matrices $\mathbf{W}$ and $\mathbf{U}$ and a bias vector $\mathbf{b}$:

$$\begin{aligned} \mathbf{h} &= \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \\ \mathbf{z} &= \mathbf{U}\mathbf{h} \\ \mathbf{y} &= \text{softmax}(\mathbf{z}) \end{aligned} \tag{6.12}$$

And just to remember the shapes of all our variables, $\mathbf{x} \in \mathbb{R}^{n_0}$, $\mathbf{h} \in \mathbb{R}^{n_1}$, $\mathbf{b} \in \mathbb{R}^{n_1}$, $\mathbf{W} \in \mathbb{R}^{n_1 \times n_0}$, $\mathbf{U} \in \mathbb{R}^{n_2 \times n_1}$, and the output vector $\mathbf{y} \in \mathbb{R}^{n_2}$. We'll call this network a 2-layer network (we traditionally don't count the input layer when numbering layers, but do count the output layer). So by this terminology logistic regression is a 1-layer network.

6.3.1 More details on feedforward networks

Let's now set up some notation to make it easier to talk about deeper networks of depth more than 2. We'll use superscripts in square brackets to mean layer numbers, starting at 0 for the input layer. So $\mathbf{W}^{[1]}$ will mean the weight matrix for the (first) hidden layer, and $\mathbf{b}^{[1]}$ will mean the bias vector for the (first) hidden layer. n_j will mean the number of units at layer j. We'll use $g(\cdot)$ to stand for the activation function, which will tend to be ReLU or tanh for intermediate layers and softmax for output layers. We'll use $\mathbf{a}^{[i]}$ to mean the output from layer i, and $\mathbf{z}^{[i]}$ to mean the combination of previous layer output, weights and biases $\mathbf{W}^{[i]}\mathbf{a}^{[i-1]} + \mathbf{b}^{[i]}$. The 0th layer is for inputs, so we'll refer to the inputs $\mathbf{x}$ more generally as $\mathbf{a}^{[0]}$.

Thus we can re-represent our 2-layer net from Eq. 6.12 as follows:

$$\begin{aligned}
\mathbf{z}^{[1]} &= \mathbf{W}^{[1]}\mathbf{a}^{[0]} + \mathbf{b}^{[1]} \\
\mathbf{a}^{[1]} &= g^{[1]}(\mathbf{z}^{[1]}) \\
\mathbf{z}^{[2]} &= \mathbf{W}^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]} \\
\mathbf{a}^{[2]} &= g^{[2]}(\mathbf{z}^{[2]}) \\
\hat{\mathbf{y}} &= \mathbf{a}^{[2]}
\end{aligned} \tag{6.13}$$

Note that with this notation, the equations for the computation done at each layer are the same. The algorithm for computing the forward step in an n-layer feedforward network, given the input vector $a^{[0]}$ is thus simply:

for i **in** 1,...,n
  $\mathbf{z}^{[i]} = \mathbf{W}^{[i]}\,\mathbf{a}^{[i-1]} + \mathbf{b}^{[i]}$
  $\mathbf{a}^{[i]} = g^{[i]}(\mathbf{z}^{[i]})$
$\hat{\mathbf{y}} = \mathbf{a}^{[n]}$

It's often useful to have a name for the final set of activations right before the final softmax. So however many layers we have, we'll generally call the unnormalized values in the final vector $\mathbf{z}^{[n]}$, the vector of scores right before the final softmax, the
logits **logits** (see Eq. 4.7).

The need for non-linear activation functions One of the reasons we use non-linear activation functions for each layer in a neural network is that if we did not, the resulting network is exactly equivalent to a single-layer network. Let's see why this is true. Imagine the first two layers of such a network of purely linear layers:

$$\begin{aligned}
\mathbf{z}^{[1]} &= \mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]} \\
\mathbf{z}^{[2]} &= \mathbf{W}^{[2]}\mathbf{z}^{[1]} + \mathbf{b}^{[2]}
\end{aligned}$$

We can rewrite the function that the network is computing as:

$$\begin{aligned}
\mathbf{z}^{[2]} &= \mathbf{W}^{[2]}\mathbf{z}^{[1]} + \mathbf{b}^{[2]} \\
&= \mathbf{W}^{[2]}(\mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}) + \mathbf{b}^{[2]} \\
&= \mathbf{W}^{[2]}\mathbf{W}^{[1]}\mathbf{x} + \mathbf{W}^{[2]}\mathbf{b}^{[1]} + \mathbf{b}^{[2]} \\
&= \mathbf{W}'\mathbf{x} + \mathbf{b}'
\end{aligned} \tag{6.14}$$

This generalizes to any number of layers. So without non-linear activation functions, a multilayer network is just a notational variant of a single layer network with a different set of weights, and we lose all the representational power of multilayer networks.

Replacing the bias unit In describing networks, we will sometimes use a slightly simplified notation that represents exactly the same function without referring to an explicit bias node b. Instead, we add a dummy node $\mathbf{a}_0$ to each layer whose value will always be 1. Thus layer 0, the input layer, will have a dummy node $\mathbf{a}_0^{[0]} = 1$, layer 1 will have $\mathbf{a}_0^{[1]} = 1$, and so on. This dummy node still has an associated weight, and that weight represents the bias value b. For example instead of an equation like

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{6.15}$$

we'll use:

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x}) \tag{6.16}$$

But now instead of our vector $\mathbf{x}$ having n_0 values: $\mathbf{x} = \mathbf{x}_1, \ldots, \mathbf{x}_{n_0}$, it will have $n_0 + 1$ values, with a new 0th dummy value $\mathbf{x}_0 = 1$: $\mathbf{x} = \mathbf{x}_0, \ldots, \mathbf{x}_{n_0}$. And instead of computing each $\mathbf{h}_j$ as follows:

$$\mathbf{h}_j = \sigma\left(\sum_{i=1}^{n_0} \mathbf{W}_{ji}\mathbf{x}_i + \mathbf{b}_j\right), \tag{6.17}$$

we'll instead use:

$$\mathbf{h}_j = \sigma\left(\sum_{i=0}^{n_0} \mathbf{W}_{ji}\mathbf{x}_i\right), \tag{6.18}$$

where the value $\mathbf{W}_{j0}$ replaces what had been $\mathbf{b}_j$. Fig. 6.9 shows a visualization.

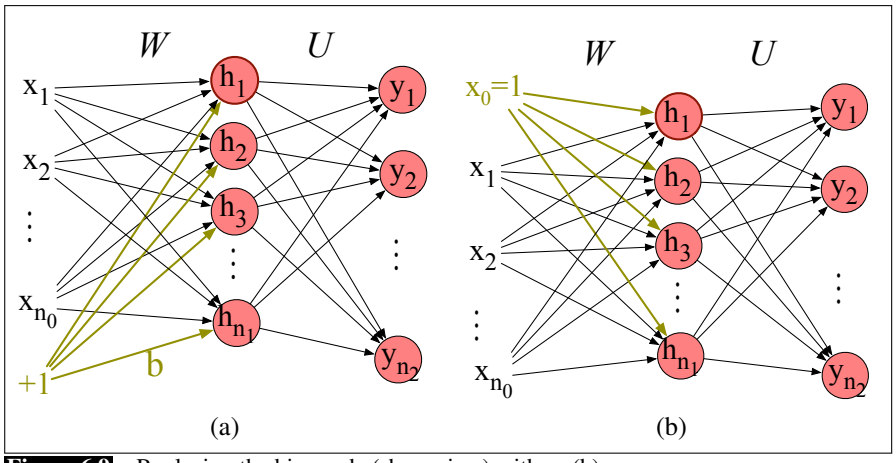


Figure 6.9 Replacing the bias node (shown in a) with x_0 (b).

We'll continue showing the bias as b when we go over the learning algorithm in Section 6.6, but going forward in the book, for most figures and some equations we'll use this simplified notation without explicit bias terms.

6.4 Feedforward networks for NLP: Classification

Let's see how to apply feedforward networks to NLP classification tasks. In practice, simple feedforward networks aren't the way we do text classification; for real applications we would use more sophisticated architectures like the BERT transformers

of Chapter 9. Nonetheless seeing a feedforward network text classifier will let us introduce key ideas that will play a role throughout the rest of the book, including the ideas of the **embedding matrix**, representation **pooling**, and **representation learning**.

But before introducing any of these ideas, let's start with a classifier by making only minimal change from the sentiment classifiers we saw in Chapter 4. Like them, we'll take hand-built features, pass them through a classifier, and produce a class probability. The only difference is that we'll use a neural network instead of logistic regression as the classifier.

6.4.1 Neural net classifiers with hand-built features

Let's begin with a simple 2-layer sentiment classifier by taking our logistic regression classifier from Chapter 4, which corresponds to a 1-layer network, and just adding a hidden layer. The input element $\mathbf{x}_i$ can be scalar features like those in Fig. 4.2, e.g., $\mathbf{x}_1$ = count(words $\in$ doc), $\mathbf{x}_2$ = count(positive lexicon words $\in$ doc), $\mathbf{x}_3$ = 1 if "no" $\in$ doc, and so on, for a total of d features. And the output layer $\hat{\mathbf{y}}$ could have two nodes (one each for positive and negative), or 3 nodes (positive, negative, neutral), in which case $\hat{\mathbf{y}}_1$ would be the estimated probability of positive sentiment, $\hat{\mathbf{y}}_2$ the probability of negative and $\hat{\mathbf{y}}_3$ the probability of neutral. The resulting equations would be just what we saw above for a 2-layer network (as always, we'll continue to use the σ to stand for any non-linearity, whether sigmoid, ReLU or other).

$$\begin{aligned}
\mathbf{x} &= [\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_d] \quad \text{(each } \mathbf{x}_i \text{ is a hand-designed feature)} \\
\mathbf{h} &= \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \\
\mathbf{z} &= \mathbf{U}\mathbf{h} \\
\hat{\mathbf{y}} &= \text{softmax}(\mathbf{z}) \qquad (6.19)
\end{aligned}$$

Fig. 6.10 shows a sketch of this architecture. As we mentioned earlier, adding this hidden layer to our logistic regression classifier allows the network to represent the non-linear interactions between features. This alone might give us a better sentiment classifier.

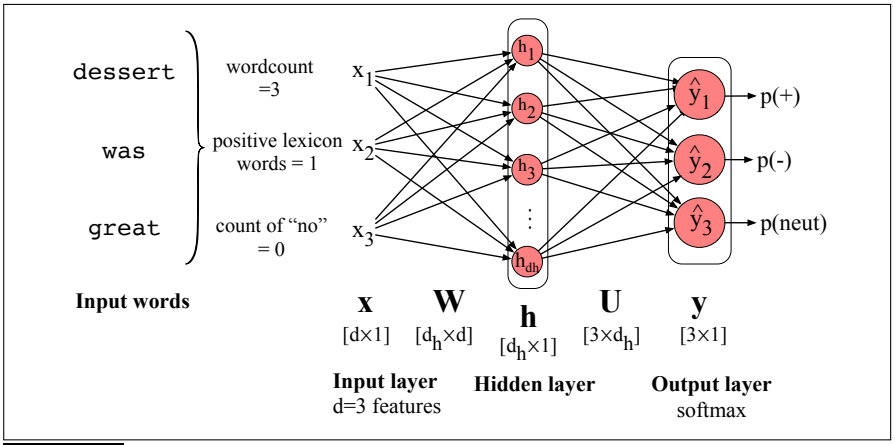


Figure 6.10 Feedforward network sentiment analysis using traditional hand-built features of the input text.

6.4.2 Vectorizing for parallelizing inference

While Eq. 6.19 shows how to classify a single example x, in practice we want to efficiently classify an entire test set of m examples. We do this by vectorizing the process, just as we saw with logistic regression; instead of using for-loops to go through each example, we'll use matrix multiplication to do the entire computation of an entire test set at once. First, we pack all the input feature vectors for each input x into a single input matrix $\mathbf{X}$, with each row i a row vector consisting of the features for input example $x^{(i)}$ (i.e., the vector $\mathbf{x}^{(i)}$). If the dimensionality of our input feature vector is d, $\mathbf{X}$ will be a matrix of shape $[m \times d]$.

Because we are now modeling each input as a row vector rather than a column vector, we also need to slightly modify Eq. 6.19. $\mathbf{X}$ is of shape $[m \times d]$ and $\mathbf{W}$ is of shape $[d_h \times d]$, so we'll reorder how we multiply $\mathbf{X}$ and $\mathbf{W}$ and transpose $\mathbf{W}$ so they correctly multiply to yield a matrix $\mathbf{H}$ of shape $[m \times d_h]$. [1]

The bias vector $\mathbf{b}$ from Eq. 6.19 of shape $[1 \times d_h]$ will now have to be replicated into a matrix of shape $[m \times d_h]$. We'll need to similarly reorder the next step and transpose $\mathbf{U}$. Finally, our output matrix $\hat{\mathbf{Y}}$ will be of shape $[m \times 3]$ (or more generally $[m \times d_o]$, where d_o is the number of output classes), with each row i of our output matrix $\hat{\mathbf{Y}}$ consisting of the output vector $\hat{\mathbf{y}}^{(i)}$. Here are the final equations for computing the output class distribution for an entire test set:

$$\begin{aligned} \mathbf{H} &= \sigma(\mathbf{X}\mathbf{W}^\intercal + \mathbf{b}) \\ \mathbf{Z} &= \mathbf{H}\mathbf{U}^\intercal \\ \hat{\mathbf{Y}} &= \text{softmax}(\mathbf{Z}) \end{aligned} \tag{6.20}$$

In this book, we'll sometimes see orderings like $\mathbf{W}\mathbf{X} + \mathbf{b}$ and sometimes $\mathbf{X}\mathbf{W} + \mathbf{b}$. That's why it's always important to be very aware of the shapes of your weight matrices participating in any given equation.

6.5 Embeddings as the input to neural net classifiers

While hand-built features are a traditional way to design classifiers, most applications of neural networks for NLP don't use hand-built human-engineered features as inputs. Instead, we draw on deep learning's ability to learn features from the data by representing tokens as embeddings. For this section we'll represent each token by its static word2vec or GloVe embeddings that we saw how to compute in Chapter 5. By static embedding, we mean that each token is represented by a fixed vector that we train once, and then just put into a big dictionary. When we want to refer to that token, we grab its embedding out of the dictionary.

However when we apply neural models to the task of language modeling (as we'll see in Chapter 8) the situation is more complex, and we'll use a more powerful kind of embedding called a *contextual embedding*. Contextual embeddings are different for each time a word occurs in a different context. Furthermore, we'll have the network learn these embeddings as part of the task of word prediction.

So let's explore the text classification domain above, but using static embeddings as features instead of the hand-designed features. Let's focus on the inference stage,

[1] Note that we could have kept the original order of our products if we had instead made our input matrix $\mathbf{X}$ represent each input as a column vector instead of a row vector, making it of shape $[d \times m]$. But representing inputs as row vectors is convenient and common in neural network models.

in which we have already learned embeddings for all the input tokens. An embedding is a vector of dimension d that represents the input token. The dictionary of static embeddings in which we store these embeddings is the **embedding matrix** $\mathbf{E}$. Each row of the embedding matrix represents each token of the vocabulary V as a (row) vector of dimensionality d. Since $\mathbf{E}$ has a row for each of the $|V|$ tokens in the vocabulary, $\mathbf{E}$ has shape $[|V| \times d]$. This embedding matrix $\mathbf{E}$ plays a role whenever we are using embeddings as input to neural NLP systems, including in the transformer-based large language models we will introduce over the next chapters.

embedding matrix

Given an input token string like `dessert was great` we first convert the tokens into vocabulary indices (these were created when we first tokenized the input using BPE or SentencePiece). So the representation of `dessert was great` might be $\mathbf{w} = [3, 9824, 226]$. Next we use indexing to select the corresponding rows from $\mathbf{E}$ (row 3, row 9824, row 226).

Another way to think about selecting token embeddings from the embedding matrix is to represent input tokens as one-hot vectors of shape $[1 \times |V|]$, i.e., with one dimension for each word in the vocabulary. Recall that in a **one-hot vector** all the elements are 0 except one, the element whose dimension is the word's index in the vocabulary, which has value 1. So if the word "dessert" has index 3 in the vocabulary, $x_3 = 1$, and $x_i = 0 \ \forall i \neq 3$, as shown here:

one-hot vector

```
[0 0 1 0 0 0 0 ... 0 0 0 0]
 1 2 3 4 5 6 7 ...  ... |V|
```

Multiplying by a one-hot vector that has only one non-zero element $x_i = 1$ simply selects out the relevant row vector for word i, resulting in the embedding for word i, as depicted in Fig. 6.11.

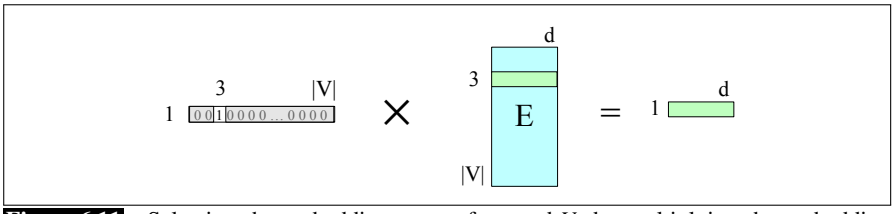


Figure 6.11 Selecting the embedding vector for word V_3 by multiplying the embedding matrix $\mathbf{E}$ with a one-hot vector with a 1 in index 3.

We can extend this idea to represent the entire input token sequence as a matrix of one-hot vectors, one for each of the N input positions as shown in Fig. 6.12.

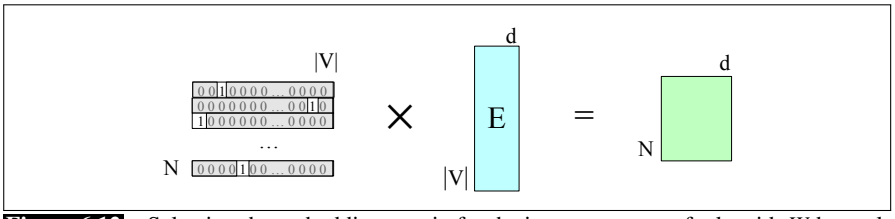


Figure 6.12 Selecting the embedding matrix for the input sequence of token ids W by multiplying a one-hot matrix corresponding to W by the embedding matrix $\mathbf{E}$.

We now need to classify this input of N $[1 \times d]$ embeddings, representing a window of N tokens, into a single class (like positive or negative).

There are two common ways to pass embeddings to a classifier: **concatenation** and **pooling**. First, we can take this input of shape $[N \times d]$ and reshape it by **concatenating** all the input vectors into one very long vector of shape $[1 \times dN]$. Then

pool

we pass this input to our classifier and let it make its decision. This gives us lots of information, at the cost of using a pretty large network. Second, we can **pool** the N embeddings into a single embedding and then pass that single pooled embedding to the classifier. Pooling gives us less information than would have been present in all the original embeddings, but has the advantage of being small and efficient and is especially useful in tasks for which we don't care as much about the original word order. Let's give an example of each: pooling for the sentiment task, and concatenation for the language modeling task.

Pooling input embeddings for sentiment So let's begin with seeing how pooling can work for the sentiment classification task. The intuition of pooling is that for sentiment, the exact position of the input (is some word like `great` the first word? the second word?) is less important than the identity of the word itself.

A pooling function is a way to turn a set of embeddings into a single embedding. For example, for a text with N input words/tokens $w_1, ..., w_N$, we want to turn the N row embeddings $\mathbf{e}(w_1), ..., \mathbf{e}(w_N)$ (each of dimensionality d) into a single embedding also of dimensionality d.

mean-pooling

There are various ways to pool. The simplest is **mean-pooling**: taking the mean by summing the embeddings and then dividing by N:

$$\mathbf{x}_{mean} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{e}(w_i) \tag{6.21}$$

Here are the equations for this classifier assuming mean pooling:

$$\begin{aligned}\mathbf{x} &= \text{mean}(\mathbf{e}(w_1), \mathbf{e}(w_2), \ldots, \mathbf{e}(w_n)) \\ \mathbf{h} &= \sigma(\mathbf{x}\mathbf{W} + \mathbf{b}) \\ \mathbf{z} &= \mathbf{h}\mathbf{U} \\ \hat{\mathbf{y}} &= \text{softmax}(\mathbf{z})\end{aligned} \tag{6.22}$$

The architecture is sketched in Fig. 6.13, where we also give the shapes for all the relevant matrices.

max-pooling

There are many other options for pooling, like **max-pooling**, in which case for each dimension we take the element-wise max over all the inputs. The element-wise max of a set of N vectors is a new vector whose kth element is the max of the kth elements of all the N vectors.

Concatenating input embeddings for language modeling For sentiment analysis we saw how to generate an output vector with probabilities over three classes: positive, negative, or neutral, given as input a window of N input tokens, by first pooling those token embeddings into a single embedding vector.

Now let's consider **language modeling**: predicting upcoming words from prior words. In this task we are given the same window of N input tokens, but our task now is to predict the next token that should follow the window. We'll sketch a simple feedforward neural language model, drawing on an algorithm first introduced by Bengio et al. (2003). The feedforward language model introduces many of the important concepts of large language modeling that we will return to in Chapter 7 and Chapter 8.

Neural language models have many advantages over the n-gram language models of Chapter 3. Neural language models can handle much longer histories, can generalize better over contexts of similar words, and are far more accurate at word-

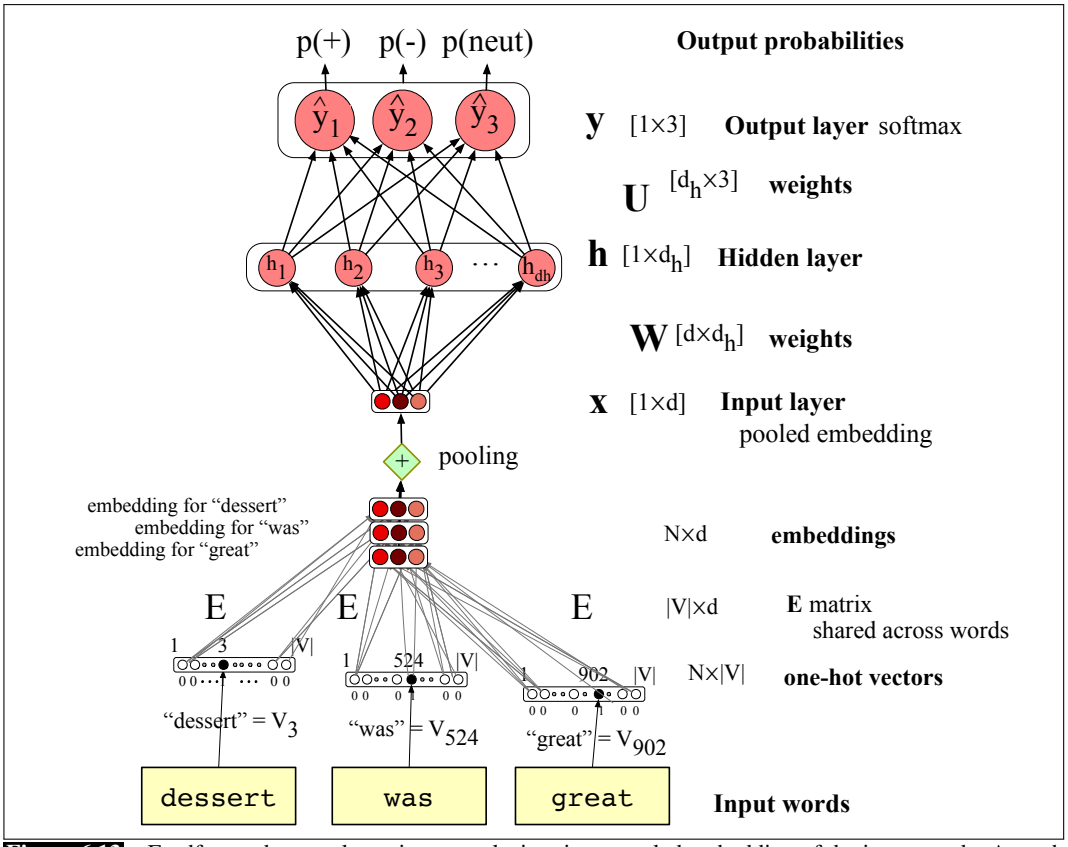


Figure 6.13 Feedforward network sentiment analysis using a pooled embedding of the input words. At each timestep the network computes a d-dimensional embedding for each context word (by multiplying a one-hot vector by the embedding matrix **E**), and pools the resulting N embeddings to get a single embedding that represents the context window as the layer **e**.

prediction. On the other hand, neural net language models are slower, more complex, need vast amounts of energy to train, and are less interpretable than n-gram models, so for some smaller tasks an n-gram language model is still the right tool.

A feedforward neural language model is a feedforward network that takes as input at time t a representation of some number of previous words (w_{t-1}, w_{t-2}, etc.) and outputs a probability distribution over possible next words. Thus—like the n-gram LM—the feedforward neural LM approximates the probability of a word given the entire prior context $P(w_t|w_{1:t-1})$ by approximating based on the $N-1$ previous words:

$$P(w_t|w_1,\ldots,w_{t-1}) \approx P(w_t|w_{t-N+1},\ldots,w_{t-1}) \tag{6.23}$$

In the following examples we'll use a 4-gram example, so we'll show a neural net to estimate the probability $P(w_t = i|w_{t-3}, w_{t-2}, w_{t-1})$.

Neural language models represent words in this prior context by their **embeddings**, rather than just by their word identity as used in n-gram language models. Using embeddings allows neural language models to generalize better to unseen data. For example, suppose we've seen this sentence in training:

I have to make sure that the cat gets fed.

but have never seen the words "gets fed" after the word "dog". Our test set has the

prefix "I forgot to make sure that the dog gets". What's the next word? An n-gram language model will predict "fed" after "that the cat gets", but not after "that the dog gets". But a neural LM, knowing that "cat" and "dog" have similar embeddings, will be able to generalize from the "cat" context to assign a high enough probability to "fed" even after seeing "dog".

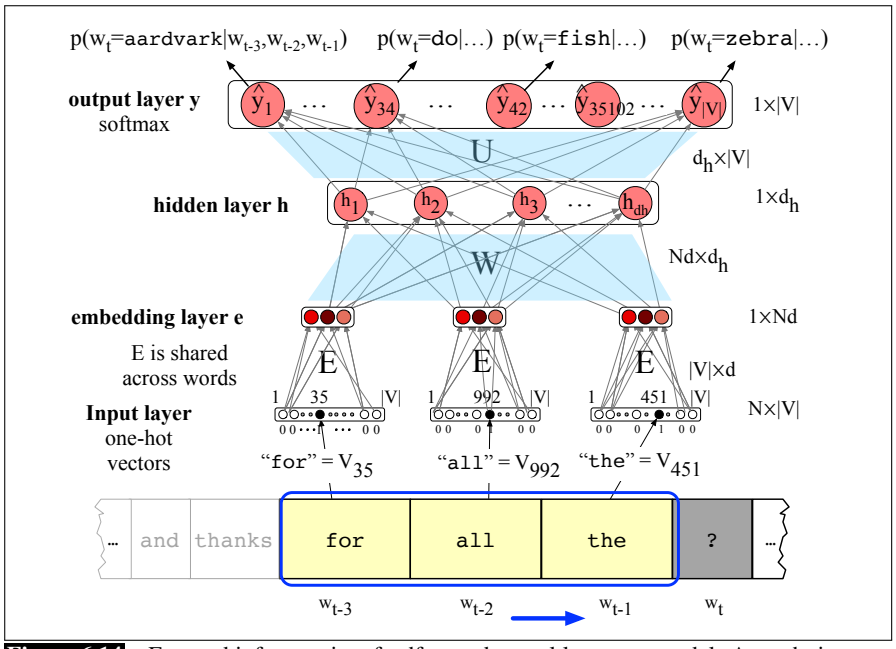


Figure 6.14 Forward inference in a feedforward neural language model. At each timestep t the network computes a d-dimensional embedding for each of the $N = 3$ context tokens (by multiplying a one-hot vector by the embedding matrix **E**), and concatenates the three to get the embedding **e**. This embedding **e** is multiplied by weight matrix **W** and then an activation function is applied element-wise to produce the hidden layer **h**, which is then multiplied by another weight matrix **U**. A softmax layer predicts at each output node i the probability that the next word w_t will be vocabulary word V_i. We show the context window size N as 3 just to fit on the page, but in practice language modeling requires a much longer context.

This prediction task requires an output vector that expresses $|V|$ probabilities: one probability value for each possible next token. We might have a vocabulary between 60,000 and 300,000 tokens, so the output vector for the task of language modeling is much longer than 3. Another difference for language modeling is that instead of pooling the embeddings of the N input tokens to create a single embedding, we concatenate the inputs into one very long input vector. To predict the next token, it helps to know each of the preceding tokens and what order they were in.

Fig. 6.14 shows the language modeling task, sketched with a very short context window of $N = 3$ just to fit on the page. These 3 embedding vectors are concatenated to produce **e**, the embedding layer. This is multiplied by a weight matrix **W** to produce a hidden layer, and another weight matrix **U** to produce an output layer whose softmax gives a probability distribution over words. For example y_{42}, the value of output node 42, is the probability of the next word w_t being V_{42}, the vocabulary word with index 42 (which is the word 'fish' in our example).

The equations for a simple feedforward neural language model with a window

size of 3, given one-hot input vectors for each input context word, are:

$$\begin{aligned}\mathbf{e} &= [\mathbf{E}\mathbf{x}_{\mathbf{t}-3}; \mathbf{E}\mathbf{x}_{\mathbf{t}-2}; \mathbf{E}\mathbf{x}_{\mathbf{t}-1}] \\ \mathbf{h} &= \sigma(\mathbf{W}\mathbf{e} + \mathbf{b}) \\ \mathbf{z} &= \mathbf{U}\mathbf{h} \\ \hat{\mathbf{y}} &= \text{softmax}(\mathbf{z}) \end{aligned} \tag{6.24}$$

Note thatwe we use semicolons to mean concatenation of vectors, so we form the embedding layer $\mathbf{e}$ by concatenating the 3 embeddings for the three context vectors.

We'll return to this idea of using neural networks to do language modeling in Chapter 7 and Chapter 8 when we introduce transformer language models.

6.6 Training Neural Nets

A feedforward neural net is an instance of supervised machine learning in which we know the correct output y for each observation x. What the system produces, via Eq. 6.13, is $\hat{y}$, the system's estimate of the true y. The goal of the training procedure is to learn parameters $\mathbf{W}^{[i]}$ and $\mathbf{b}^{[i]}$ for each layer i that make $\hat{y}$ for each training observation as close as possible to the true y.

In general, we do all this by drawing on the methods we introduced in Chapter 4 for logistic regression, so the reader should be comfortable with that chapter before proceeding. We'll explore the algorithm on simple generic networks rather than networks designed for sentiment or language modeling.

First, we'll need a **loss function** that models the distance between the system output and the gold output, and it's common to use the loss function used for logistic regression, the **cross-entropy loss**.

Second, to find the parameters that minimize this loss function, we'll use the **gradient descent** optimization algorithm introduced in Chapter 4.

Third, gradient descent requires knowing the **gradient** of the loss function, the vector that contains the partial derivative of the loss function with respect to each of the parameters. In logistic regression, for each observation we could directly compute the derivative of the loss function with respect to an individual w or b. But for neural networks, with millions of parameters in many layers, it's much harder to see how to compute the partial derivative of some weight in layer 1 when the loss is attached to some much later layer. How do we partial out the loss over all those intermediate layers? The answer is the algorithm called **error backpropagation** or **backward differentiation**.

6.6.1 Loss function

cross-entropy loss

The **cross-entropy loss** that is used in neural networks is the same one we saw for logistic regression. If the neural network is being used as a binary classifier, with the sigmoid at the final layer, the loss function is the same logistic regression loss we saw in Eq. 4.19:

$$L_{CE}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})] \tag{6.25}$$

If we are using the network to classify into 3 or more classes, the loss function is exactly the same as the loss for multinomial regression that we saw in Chapter 4 on

page 82. Let's briefly summarize the explanation here for convenience. First, when we have more than 2 classes we'll need to represent both $\mathbf{y}$ and $\hat{\mathbf{y}}$ as vectors. Let's assume we're doing **hard classification**, where only one class is the correct one. The true label $\mathbf{y}$ is then a vector with K elements, each corresponding to a class, with $\mathbf{y}_c = 1$ if the correct class is c, with all other elements of $\mathbf{y}$ being 0. Recall that a vector like this, with one value equal to 1 and the rest 0, is called a **one-hot vector**. And our classifier will produce an estimate vector with K elements $\hat{\mathbf{y}}$, each element $\hat{\mathbf{y}}_k$ of which represents the estimated probability $p(\mathbf{y}_k = 1|\mathbf{x})$.

The loss function for a single example $\mathbf{x}$ is the negative sum of the logs of the K output classes, each weighted by their probability y_k:

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{K} \mathbf{y}_k \log \hat{\mathbf{y}}_k \tag{6.26}$$

We can simplify this equation further; let's first rewrite the equation using the function $\mathbb{1}\{\}$ which evaluates to 1 if the condition in the brackets is true and to 0 otherwise. This makes it more obvious that the terms in the sum in Eq. 6.26 will be 0 except for the term corresponding to the true class for which $\mathbf{y}_k = 1$:

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{K} \mathbb{1}\{\mathbf{y}_k = 1\} \log \hat{\mathbf{y}}_k$$

In other words, the cross-entropy loss is simply the negative log of the output probability corresponding to the correct class, and we therefore also call this the **negative log likelihood loss**:

negative log likelihood loss

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\log \hat{\mathbf{y}}_c \quad \text{(where } c \text{ is the correct class)} \tag{6.27}$$

Plugging in the softmax formula from Eq. 6.9, and with K the number of classes:

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\log \frac{\exp(\mathbf{z}_c)}{\sum_{j=1}^{K} \exp(\mathbf{z}_j)} \quad \text{(where } c \text{ is the correct class)} \tag{6.28}$$

Let's think about the negative log probability as a loss function. A perfect classifier would assign the correct class i probability 1 and all the incorrect classes probability 0. That means the higher $p(\hat{y}_i)$ (the closer it is to 1), the better the classifier; $p(\hat{y}_i)$ is (the closer it is to 0), the worse the classifier. The negative log of this probability is a beautiful loss metric since it goes from 0 (negative log of 1, no loss) to infinity (negative log of 0, infinite loss). This loss function also insures that as probability of the correct answer is maximized, the probability of all the incorrect answers is minimized; since they all sum to one, any increase in the probability of the correct answer is coming at the expense of the incorrect answers.

The number K of classes of the output vector $\hat{y}$ can be small or large. Perhaps our task is 3-way sentiment, and then the classes might be positive, negative, and neutral. Or if our task is deciding the part of speech of a word (i.e., whether it is a noun or verb or adjective, etc.), then K is set of possible parts of speech in our tagset (of which there are 17 in the tagset we will define in Chapter 17). And if our task is language modeling, and our classifier is trying to predict which word is next, then our set of classes is the set of words, which might be 50,000 or 100,000.

6.6.2 Computing the Gradient

How do we compute the gradient of this loss function? Computing the gradient requires the partial derivative of the loss function with respect to each parameter. For a network with one weight layer and sigmoid output (which is what logistic regression is), we could simply use the derivative of the loss that we used for logistic regression in Eq. 6.29 (and derived in Section 4.15):

$$\begin{aligned}\frac{\partial L_{CE}(\hat{\mathbf{y}},\mathbf{y})}{\partial w_j} &= (\hat{y}-y)\,\mathbf{x}_j \\ &= (\sigma(\mathbf{w}\cdot\mathbf{x}+b)-y)\,\mathbf{x}_j \end{aligned} \tag{6.29}$$

Or for a network with one weight layer and softmax output (=multinomial logistic regression), we could use the derivative of the softmax loss from Eq. 4.41, shown for a particular weight $\mathbf{w}_k$ and input $\mathbf{x}_i$

$$\begin{aligned}\frac{\partial L_{\text{CE}}(\hat{\mathbf{y}},\mathbf{y})}{\partial \mathbf{w}_{k,i}} &= -(\mathbf{y}_k-\hat{\mathbf{y}}_k)\mathbf{x}_i \\ &= -(\mathbf{y}_k-p(\mathbf{y}_k=1|\mathbf{x}))\mathbf{x}_i \\ &= -\left(\mathbf{y}_k-\frac{\exp(\mathbf{w}_k\cdot\mathbf{x}+b_k)}{\sum_{j=1}^{K}\exp(\mathbf{w}_j\cdot\mathbf{x}+b_j)}\right)\mathbf{x}_i \end{aligned} \tag{6.30}$$

But these derivatives only give correct updates for one weight layer: the last one! For deep networks, computing the gradients for each weight is much more complex, since we are computing the derivative with respect to weight parameters that appear all the way back in the very early layers of the network, even though the loss is computed only at the very end of the network.

error backpropagation

The solution to computing this gradient is an algorithm called **error backpropagation** or **backprop** (Rumelhart et al., 1986). While backprop was invented specially for neural networks, it turns out to be the same as a more general procedure called **backward differentiation**, which depends on the notion of **computation graphs**. Let's see how that works in the next subsection.

6.6.3 Computation Graphs

A computation graph is a representation of the process of computing a mathematical expression, in which the computation is broken down into separate operations, each of which is modeled as a node in a graph.

Consider computing the function $L(a,b,c)=c(a+2b)$. If we make each of the component addition and multiplication operations explicit, and add names (d and e) for the intermediate outputs, the resulting series of computations is:

$$\begin{aligned} d &= 2*b \\ e &= a+d \\ L &= c*e \end{aligned}$$

We can now represent this as a graph, with nodes for each operation, and directed edges showing the outputs from each operation as the inputs to the next, as in Fig. 6.15. The simplest use of computation graphs is to compute the value of the function with some given inputs. In the figure, we've assumed the inputs $a=3$, $b=1$, $c=-2$, and we've shown the result of the **forward pass** to compute the result $L(3,1,-2)=-10$. In the forward pass of a computation graph, we apply each

operation left to right, passing the outputs of each computation as the input to the next node.

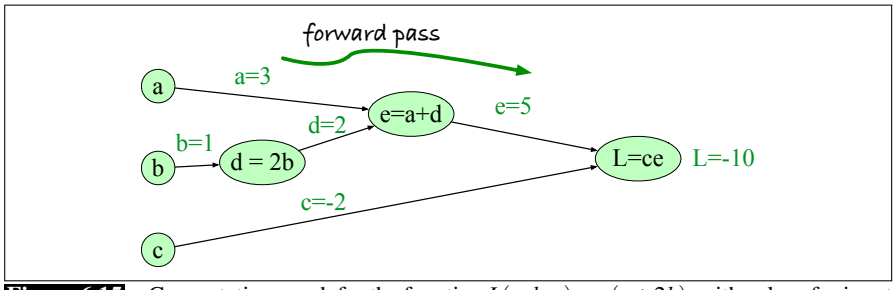


Figure 6.15 Computation graph for the function $L(a,b,c) = c(a+2b)$, with values for input nodes $a = 3$, $b = 1$, $c = -2$, showing the forward pass computation of L.

6.6.4 Backward differentiation on computation graphs

The importance of the computation graph comes from the **backward pass**, which is used to compute the derivatives that we'll need for the weight update. In this example our goal is to compute the derivative of the output function L with respect to each of the input variables, i.e., $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$. The derivative $\frac{\partial L}{\partial a}$ tells us how much a small change in a affects L.

chain rule

Backwards differentiation makes use of the **chain rule** in calculus, so let's remind ourselves of that. Suppose we are computing the derivative of a composite function $f(x) = u(v(x))$. The derivative of $f(x)$ is the derivative of $u(x)$ with respect to $v(x)$ times the derivative of $v(x)$ with respect to x:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx} \tag{6.31}$$

The chain rule extends to more than two functions. If computing the derivative of a composite function $f(x) = u(v(w(x)))$, the derivative of $f(x)$ is:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dw} \cdot \frac{dw}{dx} \tag{6.32}$$

The intuition of backward differentiation is to pass gradients back from the final node to all the nodes in the graph. Fig. 6.16 shows part of the backward computation at one node e. Each node takes an upstream gradient that is passed in from its parent node to the right, and for each of its inputs computes a local gradient (the gradient of its output with respect to its input), and uses the chain rule to multiply these two to compute a downstream gradient to be passed on to the next earlier node.

Let's now compute the 3 derivatives we need. Since in the computation graph $L = ce$, we can directly compute the derivative $\frac{\partial L}{\partial c}$:

$$\frac{\partial L}{\partial c} = e \tag{6.33}$$

For the other two, we'll need to use the chain rule:

$$\begin{aligned} \frac{\partial L}{\partial a} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial a} \\ \frac{\partial L}{\partial b} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b} \end{aligned} \tag{6.34}$$

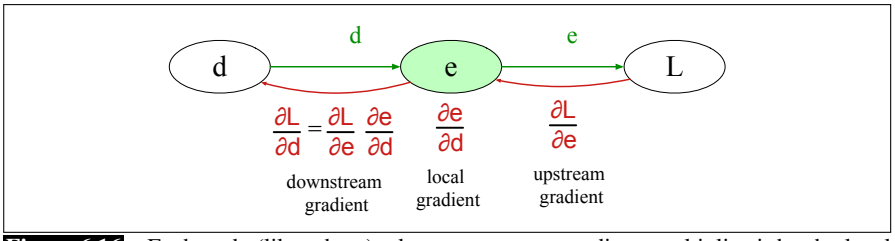


Figure 6.16 Each node (like e here) takes an upstream gradient, multiplies it by the local gradient (the gradient of its output with respect to its input), and uses the chain rule to compute a downstream gradient to be passed on to a prior node. A node may have multiple local gradients if it has multiple inputs.

Eq. 6.34 and Eq. 6.33 thus require five intermediate derivatives: $\frac{\partial L}{\partial e}$, $\frac{\partial L}{\partial c}$, $\frac{\partial e}{\partial a}$, $\frac{\partial e}{\partial d}$, and $\frac{\partial d}{\partial b}$, which are as follows (making use of the fact that the derivative of a sum is the sum of the derivatives):

$$\begin{aligned} L = ce &: \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e \\ e = a + d &: \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1 \\ d = 2b &: \quad \frac{\partial d}{\partial b} = 2 \end{aligned}$$

In the backward pass, we compute each of these partials along each edge of the graph from right to left, using the chain rule just as we did above. Thus we begin by computing the downstream gradients from node L, which are $\frac{\partial L}{\partial e}$ and $\frac{\partial L}{\partial c}$. For node e, we then multiply this upstream gradient $\frac{\partial L}{\partial e}$ by the local gradient (the gradient of the output with respect to the input), $\frac{\partial e}{\partial d}$ to get the output we send back to node d: $\frac{\partial L}{\partial d}$. And so on, until we have annotated the graph all the way to all the input variables. The forward pass conveniently already will have computed the values of the forward intermediate variables we need (like d and e) to compute these derivatives. Fig. 6.17 shows the backward pass.

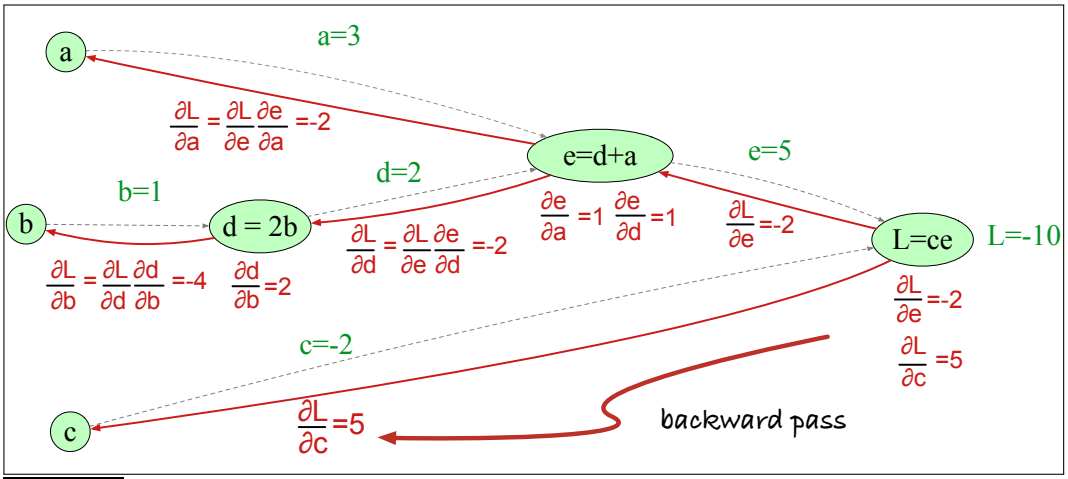


Figure 6.17 Computation graph for the function $L(a,b,c) = c(a+2b)$, showing the backward pass computation of $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$.

Backward differentiation for a neural network

Of course computation graphs for real neural networks are much more complex. Fig. 6.18 shows a sample computation graph for a 2-layer neural network with $n_0 = 2$, $n_1 = 2$, and $n_2 = 1$, assuming binary classification and hence using a sigmoid output unit for simplicity. The function that the computation graph is computing is:

$$\begin{aligned} \mathbf{z}^{[1]} &= \mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]} \\ \mathbf{a}^{[1]} &= \text{ReLU}(\mathbf{z}^{[1]}) \\ z^{[2]} &= \mathbf{W}^{[2]}\mathbf{a}^{[1]} + b^{[2]} \\ a^{[2]} &= \sigma(z^{[2]}) \\ \hat{y} &= a^{[2]} \end{aligned} \tag{6.35}$$

For the backward pass we'll also need to compute the loss L. The loss function for binary sigmoid output from Eq. 6.25 is

$$L_{CE}(\hat{y}, y) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})] \tag{6.36}$$

Our output $\hat{y} = a^{[2]}$, so we can rephrase this as

$$L_{CE}(a^{[2]}, y) = -\left[y \log a^{[2]} + (1-y)\log(1-a^{[2]})\right] \tag{6.37}$$

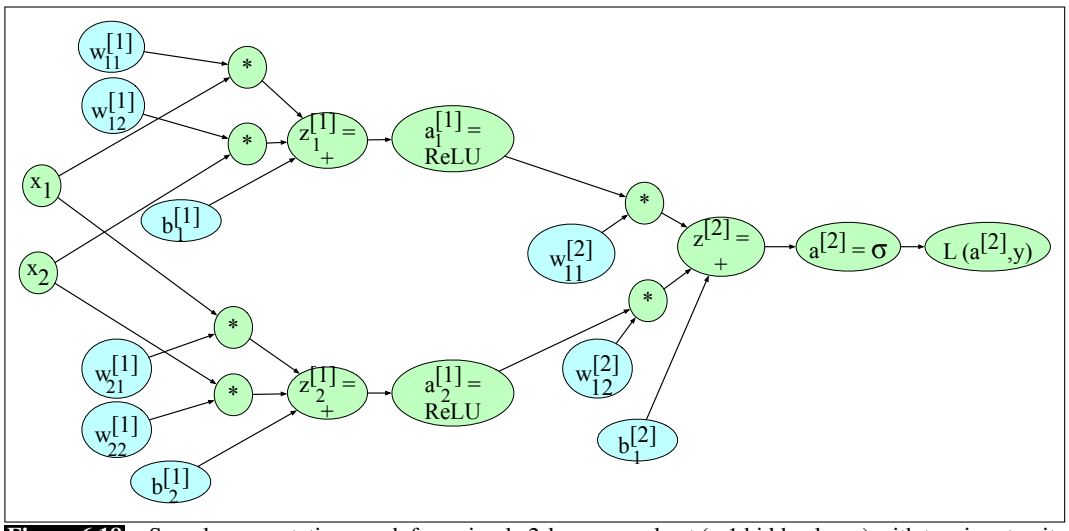


Figure 6.18 Sample computation graph for a simple 2-layer neural net (= 1 hidden layer) with two input units and 2 hidden units. We've adjusted the notation a bit to avoid long equations in the nodes by just mentioning the function that is being computed, and the resulting variable name. Thus the * to the right of node $w_{11}^{[1]}$ means that $w_{11}^{[1]}$ is to be multiplied by x_1, and the node $z^{[1]} = +$ means that the value of $z^{[1]}$ is computed by summing the three nodes that feed into it (the two products, and the bias term $b_i^{[1]}$).

The weights that need updating (those for which we need to know the partial derivative of the loss function) are shown in teal. In order to do the backward pass, we'll need to know the derivatives of all the functions in the graph. We already saw in Section 4.15 the derivative of the sigmoid σ:

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z)) \tag{6.38}$$

We'll also need the derivatives of each of the other activation functions. The derivative of tanh is:

$$\frac{d\tanh(z)}{dz} = 1 - \tanh^2(z) \tag{6.39}$$

The derivative of the ReLU is[2]

$$\frac{d\,\text{ReLU}(z)}{dz} = \begin{cases} 0 & for \quad z < 0 \\ 1 & for \quad z \geq 0 \end{cases} \tag{6.40}$$

We'll give the start of the computation, computing the derivative of the loss function L with respect to z, or $\frac{\partial L}{\partial z}$ (and leaving the rest of the computation as an exercise for the reader). By the chain rule:

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z} \tag{6.41}$$

So let's first compute $\frac{\partial L}{\partial a^{[2]}}$, taking the derivative of Eq. 6.37, repeated here:

$$\begin{aligned} L_{CE}(a^{[2]}, y) &= -\left[y \log a^{[2]} + (1-y) \log(1-a^{[2]}) \right] \\ \frac{\partial L}{\partial a^{[2]}} &= -\left(\left(y \frac{\partial \log(a^{[2]})}{\partial a^{[2]}} \right) + (1-y) \frac{\partial \log(1-a^{[2]})}{\partial a^{[2]}} \right) \\ &= -\left(\left(y \frac{1}{a^{[2]}} \right) + (1-y) \frac{1}{1-a^{[2]}} (-1) \right) \\ &= -\left(\frac{y}{a^{[2]}} + \frac{y-1}{1-a^{[2]}} \right) \end{aligned} \tag{6.42}$$

Next, by the derivative of the sigmoid:

$$\frac{\partial a^{[2]}}{\partial z} = a^{[2]}(1-a^{[2]})$$

Finally, we can use the chain rule:

$$\begin{aligned} \frac{\partial L}{\partial z} &= \frac{\partial L}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z} \\ &= -\left(\frac{y}{a^{[2]}} + \frac{y-1}{1-a^{[2]}} \right) a^{[2]}(1-a^{[2]}) \\ &= a^{[2]} - y \end{aligned} \tag{6.43}$$

Continuing the backward computation of the gradients (next by passing the gradients over $b_1^{[2]}$ and the two product nodes, and so on, back to all the teal nodes), is left as an exercise for the reader.

6.6.5 More details on learning

Optimization in neural networks is a non-convex optimization problem, more complex than for logistic regression, and for that and other reasons there are many best practices for successful learning.

[2] The derivative is actually undefined at the point $z = 0$, but by convention we treat it as 1.

For logistic regression we can initialize gradient descent with all the weights and biases having the value 0. In neural networks, by contrast, we need to initialize the weights with small random numbers. It's also helpful to normalize the input values to have 0 mean and unit variance.

dropout

Various forms of regularization are used to prevent overfitting. One of the most important is **dropout**: randomly dropping some units and their connections from the network during training (Hinton et al. 2012, Srivastava et al. 2014). At each iteration of training (whenever we update parameters, i.e. each mini-batch if we are using mini-batch gradient descent), we repeatedly choose a probability p and for each unit we replace its output with zero with probability p (and renormalize the rest of the outputs from that layer).

hyperparameter

Tuning of **hyperparameters** is also important. The parameters of a neural network are the weights $\mathbf{W}$ and biases $\mathbf{b}$; those are learned by gradient descent. The hyperparameters are things that are chosen by the algorithm designer; optimal values are tuned on a devset rather than by gradient descent learning on the training set. Hyperparameters include the learning rate η, the mini-batch size, the model architecture (the number of layers, the number of hidden nodes per layer, the choice of activation functions), how to regularize, and so on. Gradient descent itself also has many architectural variants such as Adam (Kingma and Ba, 2015).

Finally, most modern neural networks are built using computation graph formalisms that make it easy and natural to do gradient computation and parallelization on vector-based GPUs (Graphic Processing Units). PyTorch (Paszke et al., 2017) and TensorFlow (Abadi et al., 2015) are two of the most popular. The interested reader should consult a neural network textbook for further details; some suggestions are at the end of the chapter.

6.7 Summary

- Neural networks are built out of **neural units**. Originally inspired by biological neurons, neural networks are now an abstract computational device rather than a biological model.
- Each neural unit multiplies input values by a weight vector, adds a bias, and then applies a non-linear activation function like sigmoid, tanh, or rectified linear unit.
- In a **fully-connected**, **feedforward** network, each unit in layer i is connected to each unit in layer $i+1$, and there are no cycles.
- The power of neural networks comes from the ability of early layers to learn representations that can be utilized by later layers in the network.
- Neural networks are trained by optimization algorithms like **gradient descent**.
- **Error backpropagation**, backward differentiation on a **computation graph**, is used to compute the gradients of the loss function for a network.
- **Neural language models** use a neural network as a probabilistic classifier, to compute the probability of the next word given the previous n words.
- Neural language models can use pretrained **embeddings**, or can learn embeddings from scratch in the process of language modeling.

Historical Notes

The origins of neural networks lie in the 1940s **McCulloch-Pitts neuron** (McCulloch and Pitts, 1943), a simplified model of the biological neuron as a kind of computing element that could be described in terms of propositional logic. By the late 1950s and early 1960s, a number of labs (including Frank Rosenblatt at Cornell and Bernard Widrow at Stanford) developed research into neural networks; this phase saw the development of the perceptron (Rosenblatt, 1958), and the transformation of the threshold into a bias, a notation we still use (Widrow and Hoff, 1960).

The field of neural networks declined after it was shown that a single perceptron unit was unable to model functions as simple as XOR (Minsky and Papert, 1969). While some small amount of work continued during the next two decades, a major revival for the field didn't come until the 1980s, when practical tools for building deeper networks like error backpropagation became widespread (Rumelhart et al., 1986). During the 1980s a wide variety of neural network and related architectures were developed, particularly for applications in psychology and cognitive science (Rumelhart and McClelland 1986b, McClelland and Elman 1986, Rumelhart and McClelland 1986a, Elman 1990), for which the term **connectionist** or **parallel distributed processing** was often used (Feldman and Ballard 1982, Smolensky 1988). Many of the principles and techniques developed in this period are foundational to modern work, including the ideas of distributed representations (Hinton, 1986), recurrent networks (Elman, 1990), and the use of tensors for compositionality (Smolensky, 1990).

connectionist

By the 1990s larger neural networks began to be applied to many practical language processing tasks as well, like handwriting recognition (LeCun et al. 1989) and speech recognition (Morgan and Bourlard 1990). By the early 2000s, improvements in computer hardware and advances in optimization and training techniques made it possible to train even larger and deeper networks, leading to the modern term **deep learning** (Hinton et al. 2006, Bengio et al. 2007). We cover more related history in Chapter 13 and Chapter 15.

There are a number of excellent books on neural networks, including Goodfellow et al. (2016) and Nielsen (2015).

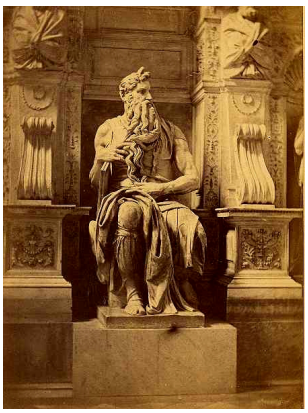
CHAPTER

7 Large Language Models

"How much do we know at any time? Much more, or so I believe, than we know we know."

Agatha Christie, *The Moving Finger*

The literature of the fantastic abounds in inanimate objects magically endowed with the gift of speech. From Ovid's statue of Pygmalion to Mary Shelley's story about Frankenstein, we continually reinvent stories about creating something and then having a chat with it. Legend has it that after finishing his sculpture *Moses*, Michelangelo thought it so lifelike that he tapped it on the knee and commanded it to speak. Perhaps this shouldn't be surprising. Language is the mark of humanity and sentience. conversation is the most fundamental arena of language, the first kind of language we learn as children, and the kind we engage in constantly, whether we are teaching or learning, ordering lunch, or talking with our families or friends.



This chapter introduces the **Large Language Model**, or **LLM**, a computational agent that can interact conversationally with people. The fact that LLMs are designed for interaction with people has strong implications for their design and use.

Many of these implications already became clear in a computational system from 60 years ago, ELIZA (Weizenbaum, 1966). ELIZA, designed to simulate a Rogerian psychologist, illustrates a number of important issues with chatbots. For example people became deeply emotionally involved and conducted very personal conversations, even to the extent of asking Weizenbaum to leave the room while they were typing. These issues of emotional engagement and privacy mean we need to think carefully about how we deploy language models and consider their effect on the people who are interacting with them.

In this chapter we begin by introducing the computational principles of LLMs; we'll discuss their implementation in the transformer architecture in the following chapter. The central new idea that makes LLMs possible is the idea of **pretraining**, so let's begin by thinking about the idea of learning from text, the basic way that LLMs are trained.

We know that fluent speakers of a language bring an enormous amount of knowledge to bear during comprehension and production. This knowledge is embodied in many forms, perhaps most obviously in the vocabulary, the rich representations we have of words and their meanings and usage. This makes the vocabulary a useful lens to explore the acquisition of knowledge from text, by both people and machines.

Estimates of the size of adult vocabularies vary widely both within and across languages. For example, estimates of the vocabulary size of young adult speakers of American English range from 30,000 to 100,000 depending on the resources used

to make the estimate and the definition of what it means to know a word. A simple consequence of these facts is that children have to learn about 7 to 10 words a day, *every single day*, to arrive at observed vocabulary levels by the time they are 20 years of age. And indeed empirical estimates of vocabulary growth in late elementary through high school are consistent with this rate. How do children achieve this rate of vocabulary growth? Research suggests that the bulk of this knowledge acquisition happens as a by-product of reading. Reading is a process of rich contextual processing; we don't learn words one at a time in isolation. In fact, at some points during learning the rate of vocabulary growth exceeds the rate at which new words are appearing to the learner! That suggests that every time we read a word, we are also strengthening our understanding of other words that are associated with it.

Such facts are consistent with the *distributional hypothesis* of Chapter 5, which proposes that some aspects of meaning can be learned solely from the texts we encounter over our lives, based on the complex association of words with the words they co-occur with (and with the words that those words occur with). The distributional hypothesis suggests both that we can acquire remarkable amounts of knowledge from text, and that this knowledge can be brought to bear long after its initial acquisition. Of course, grounding from real-world interaction or other modalities can help build even more powerful models, but even text alone is remarkably useful.

What made the modern NLP revolution possible is that large language models can learn all this knowledge of language, context, and the world simply by being taught to predict the next word, again and again, based on context, in a (very) large corpus of text. In this chapter and the next we formalize this idea that we'll call **pretraining**—learning knowledge about language and the world from iteratively predicting tokens in vast amounts of text—and call the resulting pretrained models **large language models**. Large language models exhibit remarkable performance on natural language tasks because of the knowledge they learn in pretraining.

pretraining

What can language models learn from word prediction? Consider the examples below. What kinds of knowledge do you think the model might pick up from learning to predict what word fills the underbar (the correct answer is shown in blue)? Think about this for each example before you read ahead to the next paragraph:.

With roses, dahlias, and peonies, I was surrounded by _____ flowers
The room wasn't just big it was _____ enormous
The square root of 4 is _____ 2
The author of "A Room of One's Own" is _____ Virginia Woolf
The professor said that _____ he

From the first sentence a model can learn ontological facts like that roses and dahlias and peonies are all kinds of flowers. From the second, a model could learn that "enormous" means something on the same scale as big but further along on the scale. From the third sentence, the system could learn math, while from the 4th sentence facts about the world and historical authors. Finally, the last sentence, if a model was exposed to such sentences repeatedly, it might learn to associate professors only with male pronouns, or other kinds of associations that might cause models to act unfairly to different people.

What is a large language model? As we saw back in Chapter 3, a language model is simply a computational system that can predict the next word from previous words. That is, given a context or prefix of words, a language model assigns a probability distribution over the possible next words. Fig. 7.1 sketches this idea.

Of course we've already seen language models! We saw n-gram language models in Chapter 3 and briefly touched on the feedforward network applied to language

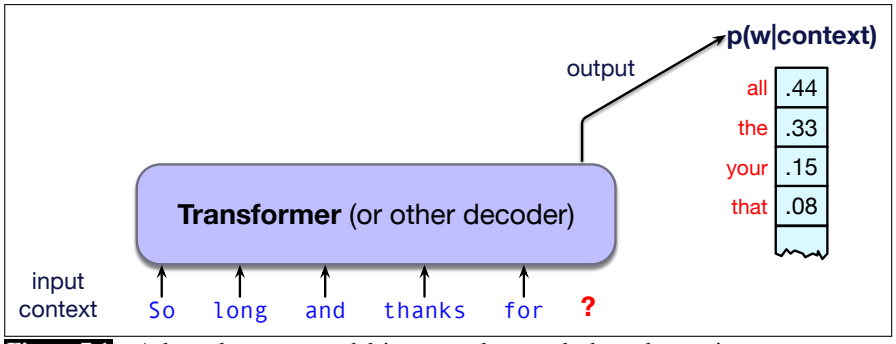


Figure 7.1 A large language model is a neural network that takes as input a context or prefix, and outputs a distribution over possible next words.

modeling in Chapter 6. A large language model is just a (much) larger version of these. For example, in Chapter 3 we introduced bigram and trigram language models that can predict words from the previous word or handful of words. By contrast, large language models can predict words given contexts of thousands or even tens of thousands of words!

The fundamental intuition of language models is that a model that can *predict* text (assigning a distribution over following words) can also be used to *generate* text by **sampling** from the distribution. Recall from Chapter 3 that sampling means to choose a word from a distribution.

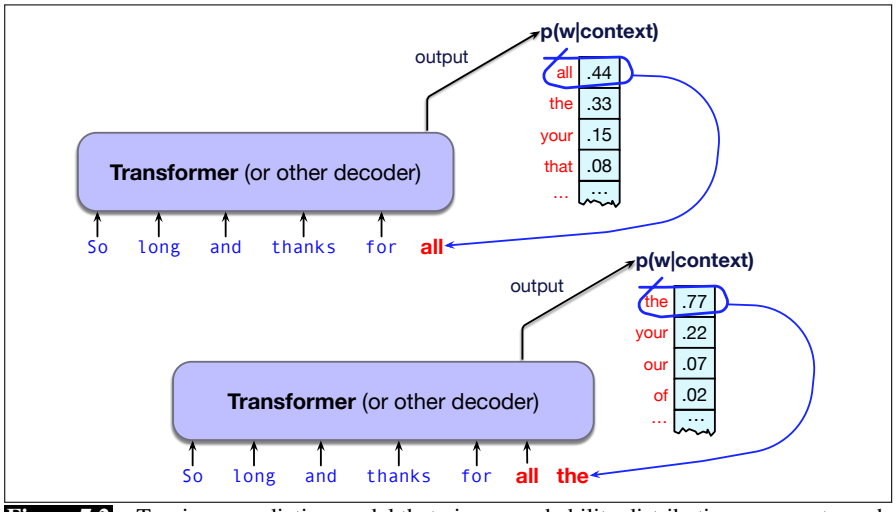


Figure 7.2 Turning a predictive model that gives a probability distribution over next words into a generative model by repeatedly sampling from the distribution. The result is a left-to-right (also called autoregressive) language model. As each token is generated, it gets added onto the context as a prefix for generating the next token.

Fig. 7.2 shows the same example from Fig. 7.1, in which a language model is given a text prefix and generates a possible completion. The model selects the word `all`, adds that to the context, uses the updated context to get a new predictive distribution, and then selects `the` from that distribution and generates it, and so on. Notice that the model is conditioning on both the priming context and its own subsequently generated outputs.

This kind of setting in which we iteratively predict and generate words left-to-

right from earlier words is often called **causal** or **autoregressive** language models. (We will introduce alternative non-autoregressive models, like BERT and other masked language models that predict words using information from both the left and the right, in Chapter 9.)

generative AI

This idea of using computational models to generate text, as well as code, speech, and images, constitutes the important new area called **generative AI**. Applying LLMs to generate text has vastly broadened the scope of NLP, which historically was focused more on algorithms for parsing or understanding text rather than generating it.

In the rest of the chapter, we'll see that almost any NLP task can be modeled as word prediction in a large language model, if we think about it in the right way, and we'll motivate and introduce the idea of **prompting** language models. We'll introduce specific algorithms for generating text from a language model, like **greedy decoding** and **sampling**. We'll introduce the details of **pretraining**, the way that language models are self-trained by iteratively being taught to guess the next word in the text from the prior words. We'll sketch out the other two stages of language model training: instruction tuning (also called supervised finetuning or SFT), and alignment, concepts that we'll return to in Chapter 10. And we'll see how to evaluate these models. Let's begin, though, by talking about different kinds of language models.

7.1 Three architectures for language models

The architecture we sketched above for a left-to-right or autoregressive language model, which is the language model architecture we will define in this chapter, is actually only one of three common LM architectures.

The three architectures are the **encoder**, the **decoder**, and the **encoder-decoder**. Fig. 7.3 gives a schematic picture of the three.

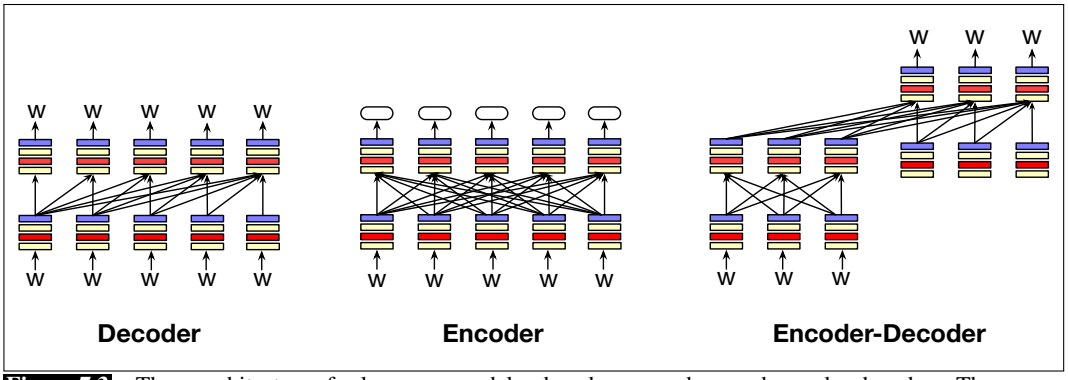


Figure 7.3 Three architectures for language models: decoders, encoders, and encoder-decoders. The arrows sketch out the information flow in the three architectures. Decoders take tokens as input and generate tokens as output. Encoders take tokens as input and produce an encoding (a vector representation of each token) as output. Encoder-decoders take tokens as input and generate a series of tokens as output.

decoder

The **decoder** is the architecture we've introduced above. It takes as input a series of tokens, and iteratively generates an output token one at a time. The decoder is the architecture used to create large language models like GPT, Claude, Llama, and Mistral. The information flow in decoders goes left-to-right, meaning that the model

predicts the next word only from the prior words. Decoders are generative models, meaning that, given input tokens, they generate novel output tokens. We'll discuss decoders in the rest of this chapter and in Chapter 8.

encoder

The **encoder** takes as input a sequence of tokens and outputs a vector representation for each token. Encoders are usually masked language models, meaning they are trained by masking out a word, and learning to predict it by looking at surrounding words on both sides. Masked language models like BERT, RoBERTA, and others in the BERT family are encoder models. Encoder models are not generative models; they aren't used to generate text. Instead encoder models are often used to create classifiers, for example where the input is text and the output is a label, for example for sentiment or topic or other classes. This is done by finetuning them (training them on supervised data). We'll introduce encoder models in Chapter 9.

encoder-decoder

The **encoder-decoder** takes as input a sequence of tokens and outputs a series of tokens. What makes it different than the decoder-only models, is that an encoder-decoder has a much looser relationship between the input tokens and the output tokens, and they are used to map between different kinds of tokens. That is, in an encoder-decoder, the output tokens might be from a very different token-set or be a much longer or shorter sequence than the input token sequence. For example encoder-decoder architectures are used for machine translation, where the input tokens are in one language and the output tokens (probably more or less of them) are in another language. Encoder-decoder architectures are also used for speech recognition, where the input is tokens representing speech, and the output is tokens representing text. We'll introduce the encoder-decoder architecture for machine translation in Chapter 12, and for speech recognition in Chapter 15.

These three architectures can be built out of many kinds of neural networks. The most widely used network type today is the **transformer** that we'll introduce in Chapter 8. In a transformer, each input token is processed by a column of transformer layers, each layer composed of a series of different kinds of subnetworks. In Chapter 13 we'll introduce an earlier architecture that is still relevant, the LSTM, a kind of recurrent neural network. And there are many more recent architectures such as the **state space models**.

We'll focus on transformers for much of this book, but for the purposes of this chapter, we'll describe the LLM decoder in a way that is architecture-agnostic, treating this network as a black box. The input to this black box is a sequence of tokens, and the output of the box is a distribution over tokens that we can sample. And we'll describe architecture-agnostic mechanisms for learning and decoding.

7.2 Conditional Generation of Text: The Intuition

conditional generation

A fundamental intuition underlying language models is that almost anything we want to do with language can be modeled as **conditional generation** of text. (We mean *decoder* language models, which are what we will discuss in this chapter and the next).

Conditional generation is the task of generating text conditioned on an input piece of text. That is, we give the LLM an input piece of text, a **prompt**, and then have the LLM continue generating text token by token, conditioned on the prompt and the subsequently generated tokens. We generate from a model by first computing the probability of the next token w_i from the prior context: $P(w_i|w_{<i})$ and then sampling from that distribution to generate a token.

We'll talk in future sections about all the details, but in this section our goal is just to establish the intuition. How can simply computing the probability of the next token help an LLM do all sorts of different language-related tasks?

Imagine we want to do a classification tasks like sentiment analysis. We can treat this as conditional generation by giving a language model a context like:

`The sentiment of the sentence ‘‘I like Jackie Chan" is:`

and comparing the conditional probability of the following token "positive" and the following token "negative" to see which is higher. That is, as sketched in Fig. 7.4, we compare these two probabilities:

$P($"`positive`"|"`The sentiment of the sentence ‘I like Jackie Chan’ is:`"$)$
$P($"`negative`"|"`The sentiment of the sentence ‘I like Jackie Chan’ is:`"$)$

If the token "`positive`" is more probable, we could say the sentiment of the sentence is positive, otherwise if the token "`negative`" is more probable we say the sentiment is negative.

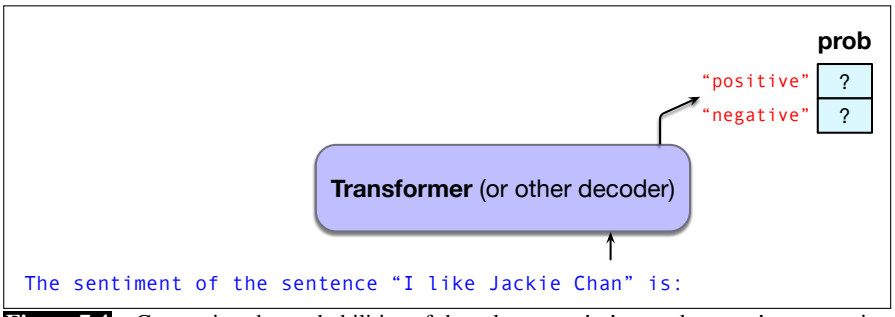


Figure 7.4 Computing the probabilities of the tokens `positive` and `negative` occurring after this prefix.

This same intuition can help us perform a task like question answering, in which the system is given a question and must give a textual answer. We can cast the task of question answering as token prediction by giving a language model a question and a token like `A:` suggesting that an answer should come next, like this:

`Q: Who wrote the book ‘‘The Origin of Species"? A:`

Again, we can ask a language model to compute the probability distribution over possible next tokens given this prefix, computing the following probability

$P(w|$`Q: Who wrote the book ‘‘The Origin of Species"? A:`$)$

and look at which tokens w have high probabilities. As Fig. 7.5 suggests, we might expect to see that `Charles` is very likely, and then if we choose `Charles` and add that to our prefix and compute the probability over tokens with this prefix:

$P(w|$`Q: Who wrote the book ‘‘The Origin of Species"? A: Charles`$)$

we might now see that `Darwin` is the most probable token, and select it.

7.3 Prompting

This simple idea of conditional generation is already very powerful, but becomes more powerful when language models are specially trained to answer questions and

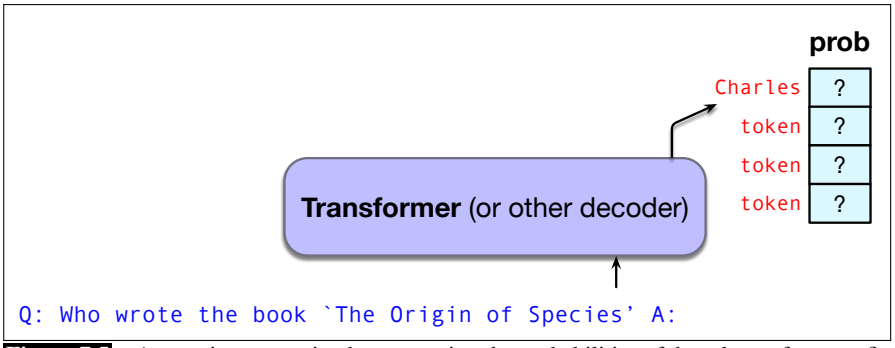


Figure 7.5 Answering a question by computing the probabilities of the tokens after a prefix stating the question; in this example the correct token `Charles` has the highest probability.

follow instructions. This extra training is called **instruction-tuning**. In instruction-tuning we take a base language model that has been trained to predict words, and continue training it on a special dataset of instructions together with the appropriate response to each. The dataset has many examples of questions together with their answers, commands with their responses, and other examples of how to carry on a conversation. We'll discuss the details of instruction-tuning in Chapter 10.

Language models that have been instruction-tuned are very good at following instructions and answering questions and carrying on a conversation and can be **prompted**. A **prompt** is a text string that a user issues to a language model to get the model to do something useful. In prompting, the user's prompt string is passed to the language model, which iteratively generates tokens conditioned on the prompt. The process of finding effective prompts for a task is known as **prompt engineering**.

prompt

prompt engineering

As suggested above when we introduced conditional generation, a prompt can be a question (like "`What is a transformer network?`"), possibly in a structured format (like "`Q: What is a transformer network? A:`"). A prompt can also be an instruction (like "`Translate the following sentence into Hindi: 'Chop the garlic finely'`").

More explicit prompts that specify the set of possible answers lead to better performance. For example, here is a prompt template to do sentiment analysis that prespecifies the potential answers:

A prompt consisting of a review plus an incomplete statement

Human: Do you think that "input" has negative or positive sentiment?
Choices:
(P) Positive
(N) Negative

Assistant: I believe the best answer is: (

This prompt uses a number of more sophisticated prompting characteristics. It specifies the two allowable choices (P) and (N), and ends the prompt with the open parenthesis that strongly suggests the answer will be (P) or (N). Note that it also specifies the role of the language model as an assistant.

Including some labeled examples in the prompt can also improve performance. We call such examples **demonstrations**. The task of prompting with examples is sometimes called **few-shot prompting**, as contrasted with **zero-shot** prompting which means instructions that don't include labeled examples. For example Fig. 7.6

demonstrations

few-shot

zero-shot

shows an example of a question using 2 demonstrations, hence 2-shot prompting. The example is drawn from a computer science question from the MMLU dataset described in Section 7.6 that is often used to evaluate language models.

Example of demonstrations in a computer science question from the MMLU dataset described in Section 7.6

The following are multiple choice questions about high school computer science.

Let x = 1. What is x << 3 in Python 3?
(A) 1 (B) 3 (C) 8 (D) 16
Answer: C

Which is the largest asymptotically?
(A) O(1) (B) O(n) (C) O(n^2) (D) O(log(n))
Answer: C

What is the output of the statement "a" + "ab" in Python 3?
(A) Error (B) aab (C) ab (D) a ab
Answer:

Figure 7.6 Sample 2-shot prompt from MMLU testing high-school computer science. (The correct answer is (B)).

Demonstrations are generally drawn from a labeled training set. They can be selected by hand, or the choice of demonstrations can be optimized by using an optimizer like DSPy (Khattab et al., 2024) to automatically choose the set of demonstrations that most increases task performance of the prompt on a dev set. The number of demonstrations doesn't need to be large; more examples seem to give diminishing returns, and too many examples seems to cause the model to overfit to the exact examples. The primary benefit of demonstrations seems more to demonstrate the task and the format of the output rather than demonstrating the right answers for any particular question. In fact, demonstrations that have incorrect answers can still improve a system (Min et al., 2022; Webson and Pavlick, 2022).

Prompts are a way to get language models to generate text, but prompts can also be viewed as a **learning** signal. This is especially clear when a prompt has demonstrations, since the demonstrations can help language models learn to perform novel tasks from these examples of the new task. This kind of learning is different than pretraining methods for setting language model weights via gradient descent methods that we will describe below. The weights of the model are not updated by prompting; what changes is just the context and the activations in the network.

in-context learning

We therefore call the kind of learning that takes place during prompting **in-context learning**—learning that improves model performance or reduces some loss but does not involve gradient-based updates to the model's underlying parameters.

system prompt

Large language models generally have a **system prompt**, a single text prompt that is the first instruction to the language model, and which defines the task or role for the LM, and sets overall tone and context. The system prompt is silently prepended to any user text. So for example a minimal system prompt that creates a multi-turn assistant conversation might be the following including some special metatokens:

```
<system>You are a helpful and knowledgeable assistant.  Answer
concisely and correctly.
```

So if a user wants to know the capital of France, the actual text used as the language model's context for conditional generation is:

```
<system> You are a helpful and knowledgeable assistant.
Answer concisely and correctly.  <user> What is the capital
of France?
```

The fact that modern language models have such long contexts (tens of thousands of tokens) makes them very powerful for conditional generation, because they can look back so far into the prompting text. That means system prompts, and prompts in general, can be very long.

For example the full system prompt for one language model, Anthropic's Claude Opus4, is 1700 words long and includes sentences like the following:

```
Claude should give concise responses to very simple questions,
but provide thorough responses to complex and open-ended
questions.
Claude is able to explain difficult concepts or ideas clearly.
It can also illustrate its explanations with examples, thought
experiments, or metaphors.
Claude does not provide information that could be used to
make chemical or biological or nuclear weapons
For more casual, emotional, empathetic, or advice-driven
conversations, Claude keeps its tone natural, warm, and
empathetic
Claude cares about people's well-being and avoids encouraging
or facilitating self-destructive behavior
If Claude provides bullet points in its response, it should
use markdown, and each bullet point should be at least 1-2
sentences long unless the human requests otherwise
```

It's also possible to create system prompts for other tasks, like the following prompt for creating a general grammar-checker (Anthropic, 2025):

```
Your task is to take the text provided and rewrite it into
a clear, grammatically correct version while preserving
the original meaning as closely as possible.  Correct any
spelling mistakes, punctuation errors, verb tense issues,
word choice problems, and other grammatical mistakes.
```

Each user can then make a prompt to have the system fix the grammar of a particular piece of text.

In all these cases, the system prompt is prepended to any user prompts or queries, and the entire string is taking as the context for conditional generation by the language model.

7.4 Generation and Sampling

Which tokens should a language model generate at each step?

The generation depends on the probability of each token, so let's remind ourselves where this probability distribution comes from. The internal networks for language models (whether transformers or alternatives like LSTMs or state space models) generate scores called **logits** (real valued numbers) for each token in the vocabulary. This score vector **u** is then normalized by softmax to be a legal probability distribution, just as we saw for logistic regression in Chapter 4. So if we have a logit vector **u** of shape $[1 \times |V|]$ that gives a score for each possible next token, we can pass it through a softmax to get a vector **y**, also of shape $[1 \times |V|]$, which assigns a probability to each token in the vocabulary, as shown in the following equation:

$$\mathbf{y} = \text{softmax}(\mathbf{u}) \tag{7.1}$$

Fig. 7.7 shows an example in which the softmax is computed for pedagogical purposes on a simplified vocabulary of only 4 words.

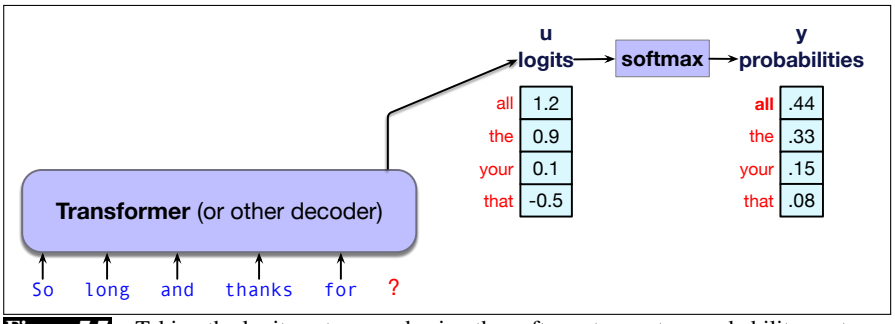


Figure 7.7 Taking the logit vector **u** and using the softmax to create a probability vector **y**.

Now given this probability distribution over tokens, we need to select one token to generate. The task of choosing a token to generate based on the model's probabilities is often called **decoding**. As we mentioned above, decoding from a language model in a left-to-right manner (or right-to-left for languages like Arabic in which we read from right to left), and thus repeatedly choosing the next token conditioned on our previous choices is called **causal** or **autoregressive generation**.[1]

decoding

causal
autoregressive
generation

7.4.1 Greedy decoding

The simplest way to generate tokens is to always generate the most likely token given the context, which is called **greedy decoding**. A **greedy algorithm** is one that makes a choice that is locally optimal, whether or not it will turn out to have been the best choice with hindsight. Thus in greedy decoding, at each time step in generation, we turn the logits into a probability distribution over tokens and then we choose as the output w_t the token in the vocabulary that has the highest probability (the argmax):

greedy
decoding

$$\hat{w}_t = \text{argmax}_{w \in V} P(w|\mathbf{w}_{<t}) \tag{7.2}$$

Fig. 7.8 shows that in our example, the model chooses to generate `all`.

[1] Technically an **autoregressive** model predicts a value at time t based on a linear function of the values at times $t-1$, $t-2$, and so on. Although language models are not linear (since, as we will see, they have many layers of non-linearities), we loosely refer to this generation technique as autoregressive since the token generated at each time step is conditioned on the token selected by the network from the previous step. As we'll see, alternatives like the masked language models of Chapter 9 are non-causal because they can predict tokens based on both past and future tokens.

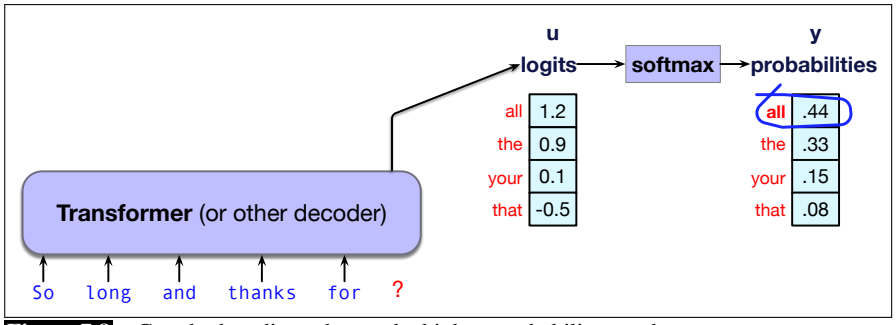


Figure 7.8 Greedy decoding: choose the highest probability word.

In practice, however, we don't use greedy decoding with large language models. A major problem with greedy decoding is that because the tokens it chooses are (by definition) extremely predictable, the resulting text is generic and often quite repetitive. Indeed, greedy decoding is so predictable that it is deterministic; if the context is identical, and the probabilistic model is the same, greedy decoding will always result in generating exactly the same string.

We'll see in Chapter 12 that an extension to greedy decoding called **beam search** works well in tasks like machine translation, which are very constrained in that we are always generating a text in one language conditioned on a very specific text in another language.

In most other tasks, however, people prefer text which has been generated by **sampling methods** that introduce a bit more diversity into the generations.

7.4.2 Random sampling

sampling

Thus the most common method for decoding in large language models involves **sampling**. Recall from Chapter 3 that **sampling** from a distribution means to choose random points according to their likelihood. Thus sampling from a language model—which represents a distribution over following tokens—means to choose the next token to generate according to its probability assigned by the model. Thus we are more likely to generate tokens that the model thinks have a high probability and less likely to generate tokens that the model thinks have a low probability.

That is, we randomly select a token to generate according to its probability in context as defined by the model, generate it, and iterate. We could think of this as rolling a die and choosing a token according to the resulting probability, as we saw in Chapter 3. Such a model is of course more likely to generate the highest probability token, just like the greedy algorithm, but it could also generate any token, just with smaller chances. But in general we are more likely to generate tokens that the model thinks have a high probability in the context and less likely to generate tokens that the model thinks have a low probability.

Sampling from language models was first suggested very early on by Shannon (1948) and Miller and Selfridge (1950), and we saw back in Chapter 3 on page 49 how to generate text from a unigram language model by repeatedly randomly sampling tokens according to their probability until we either reach a pre-determined length or select the end-of-sentence token.

To generate text from a large language model we'll just generalize this model a bit: at each step we'll sample tokens according to their probability *conditioned on our previous choices*, and we'll use the large language model as the probability model that tells us this probability.

random sampling

The algorithm is called **random sampling**, or **random multinomial sampling** (because we are sampling from a multinomial distribution across words). We can formalize random sampling as follows: we are generating a sequence of tokens $\{w_1, w_2, \ldots, w_N\}$ until we hit the end-of-sequence token, using $x \sim p(x)$ to mean 'choose x by sampling from the distribution $p(x)$':

i ← 1
$w_i \sim$ p(w)
while w_i != EOS
  i ← i + 1
  $w_i \sim$ p($w_i \mid w_{<i}$)

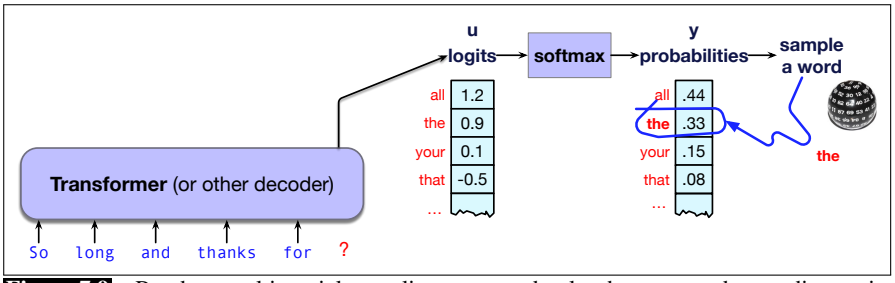


Figure 7.9 Random multinomial sampling: we randomly chose a word according to its probability.

Alas, it turns out random sampling doesn't work well either. The problem is that even though random sampling is mostly going to generate sensible, high-probable tokens, there are many odd, low-probability tokens in the tail of the distribution, and even though each one is low-probability, if you add up all the rare tokens, they constitute a large enough portion of the distribution that they get chosen often enough to result in generating weird sentences.

In other words, greedy decoding is too boring, and random sampling is too random. We need something that doesn't greedily choose the top choice every time, but doesn't stray down too far into the very low-probability events.

There are three standard sampling methods that modify random sampling to address these issues. We'll describe the most common, **temperature** sampling here, and talk about two others (**top-k** and **top-p**) in the next chapter.

7.4.3 Temperature sampling

temperature sampling

The idea of **temperature sampling** is to reshape the probability distribution to increase the probability of the high probability tokens and decrease the probability of the low probability tokens. The result is that we are less likely to generate very low-probability tokens, and more likely to generate tokens that are higher probability.

We implement this intuition by simply dividing the logit by a temperature parameter τ before passing it through the softmax. In low-temperature sampling, $\tau \in (0, 1]$.

Thus instead of computing the probability distribution over the vocabulary directly from the logit as in the following (repeated from Eq. 7.1):

$$\mathbf{y} = \text{softmax}(\mathbf{u}) \tag{7.3}$$

we instead first divide the logits by τ, computing the probability vector y as

$$\mathbf{y} = \text{softmax}(\mathbf{u}/\tau) \tag{7.4}$$

That is, normally we convert from logits to softmax as shown in Fig. 7.10(a). But when we use a temperature parameter we first scale the logit as in Fig. 7.10(b).

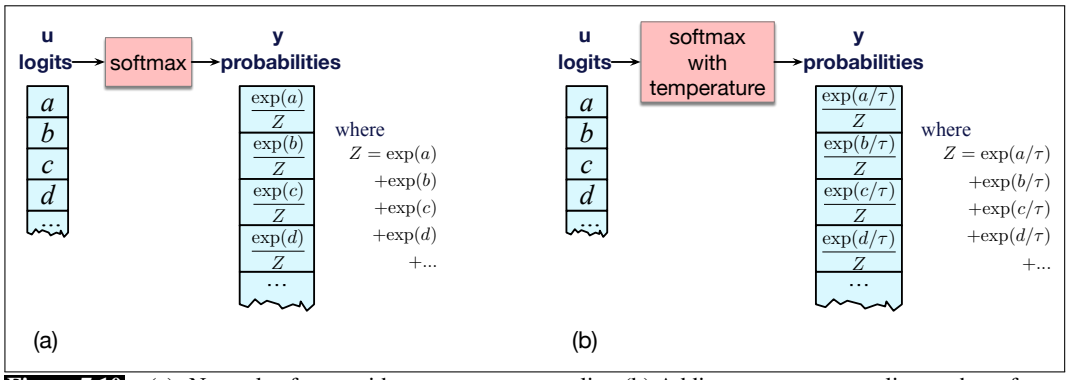


Figure 7.10 (a): Normal softmax without temperature scaling (b) Adding temperature scaling to the softmax by first dividing by the temperature parameter τ.

Why does dividing by τ increase the high probability elements and decrease the low probability elements in the vector over vocabulary items? When τ is 1, we are doing normal softmax, and so when τ is close to 1 the distribution doesn't change much. But the lower τ is, the larger the scores being passed to the softmax (because dividing by a smaller fraction $\tau \leq 1$ results in making each score larger).

Recall that one of the useful properties of a softmax is that it tends to push high values toward 1 and low values toward 0. Thus when larger numbers are passed to a softmax the result is a distribution with increased probabilities of the most high-probability tokens and decreased probabilities of the low probability tokens, making the distribution more greedy. And as τ approaches 0, dividing by τ means the probability of the most likely word approaches 1, resulting in greedy decoding.

The intuition for temperature sampling comes from thermodynamics, where a system at a high temperature is very flexible and can explore many possible states, while a system at a lower temperature is likely to explore a subset of lower energy (better) states. In low-temperature sampling, we smoothly increase the probability of the most probable tokens and decrease the probability of the rare tokens.

Fig. 7.11 shows a schematic example again simplified to have a vocabulary with only 4 tokens (*all*, *the*, *your*, *that*), and showing how different temperature values influence the probabilities computed from the initial logits. $\tau = 1$ is the normal softmax, and we can see how setting $\tau = 0.5$ increases the probability of the top candidate from .45 to .59. Setting $\tau = 0.1$ increases the probability of the top candidate to .95, getting us close to greedy decoding.

We can also see in Fig. 7.11 some other options for situations where we may want to *flatten* the word probability distribution instead of making it greedy. Temperature sampling can help with this situation too, in this case **high-temperature** sampling, in which case we use $\tau > 1$.

7.5 Training Large Language Models

How do we learn a language model? What is the algorithm and what data do we train on?

Language models are trained in three stages, as shown in Fig. 7.12:

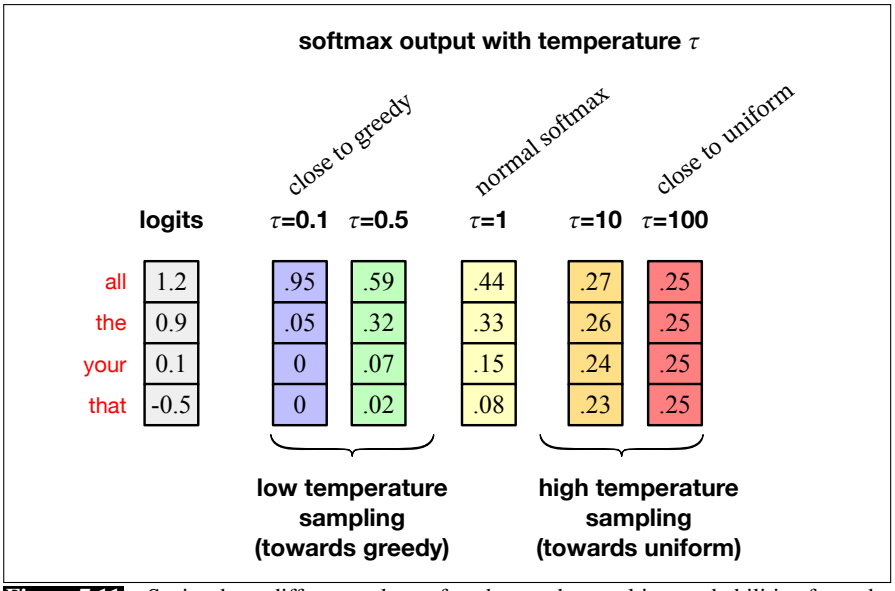


Figure 7.11 Seeing how different values of τ change the resulting probabilities from the initial logits in temperature sampling. In this simplified example, there are only 4 tokens in the vocabulary.

1. **pretraining**: In this first stage, the model is trained to incrementally predict the next word in enormous text corpora. The model uses the cross-entropy loss, sometimes called the **language modeling loss**, and that loss is backpropagated all the way through the network. The training data is usually based on cleaning up parts of the web. The result is a model that is very good at predicting words and can generate text.
2. **instruction tuning**, also called supervised finetuning or **SFT**: In the second stage, the model is trained, again by cross-entropy loss to follow instructions, for example to answer questions, give summaries, write code, translate sentences, and so on. It does this by being trained on a special corpus with lots of text containing both instructions and the correct response to the instruction.
3. **alignment**, also called **preference alignment**. In this final stage, the model is trained to make it maximally helpful and less harmful. Here the model is given preference data, which consists of a context followed by two potential continuations, which are labeled (usually by people) as an 'accepted' vs. a 'rejected' continuation. The model is then trained, by reinforcement learning or other reward-based algorithms, to produce the accepted continuation and not the rejected continuation.

We'll introduce pretraining next, but we'll save instruction tuning and preference alignment for Chapter 10.

7.5.1 Self-supervised training algorithm for pretraining

self-training

The intuition of pretraining large language models, is the same idea of **self-training** or **self-supervision** that we saw in Chapter 5 for learning word representations like word2vec. In self-training for language modeling, we take a corpus of text as training material and at each time step t ask the model to predict the next word. At first it will do poorly at this task, but since in each case we know the correct answer (it's

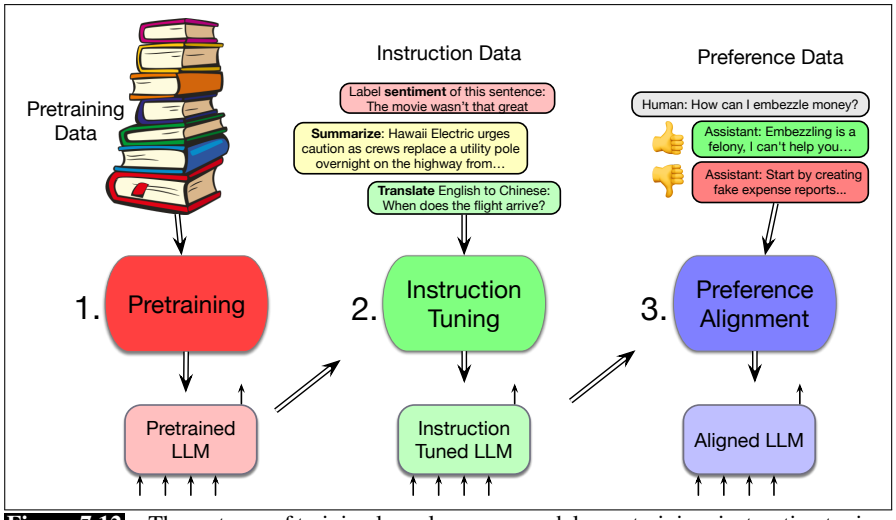


Figure 7.12 Three stages of training large language models: pretraining, instruction tuning, and preference alignment.

the next word in the corpus!) over time it well get better and better at predicting the correct next word. We call such a model self-supervised because we don't have to add any special gold labels to the data; the natural sequence of words is its own supervision! We simply train the model to minimize the error in predicting the true next word in the training sequence.

In practice, training the language model means setting the parameters of the underlying architecture. The transformer that we will introduce in the next chapter has various weight matrices for its feedforward and attention components. Like any other neural architecture, they will be trained by error backpropagation with gradient descent. So all we need is a loss function to minimize and pass back through the network. The loss function we use for language modeling is the cross-entropy loss function we've now seen twice, in Chapter 4 and Chapter 6.

Recall that the cross-entropy loss measures the difference between a predicted probability distribution and the correct distribution. The probability distribution is over the token vocabulary, making the loss be:

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -\sum_{w \in V} \mathbf{y}_t[w] \log \hat{\mathbf{y}}_t[w] \tag{7.5}$$

In the case of language modeling, the correct distribution $\mathbf{y}_t$ comes from knowing the next word. This is represented as a one-hot vector corresponding to the vocabulary where the entry for the actual next word is 1, and all the other entries are 0. Thus, the cross-entropy loss for language modeling is determined by the probability the model assigns to the correct next token (all other tokens get multiplied by zero by the first term in Eq. 7.5).

So without loss of generality we can say that at time t the cross-entropy loss in Eq. 7.5 can be simplified as the negative log probability the model assigns to the next word in the training sequence, $-\log p(w_{t+1})$, or more formally, using $\hat{\mathbf{y}}$ to mean the vector of estimated token probabilities from the language model:

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -\log \hat{\mathbf{y}}_t[w_{t+1}] \tag{7.6}$$

Thus at each word position t of the input, the model takes as input the correct sequence of tokens $w_{1:t}$, and uses them to compute a probability distribution over

possible next tokens so as to compute the model's loss for the next token w_{t+1}. Then we move to the next word, we ignore what the model predicted for the next word and instead use the correct sequence of tokens $w_{1:t+1}$ to get the model to estimate the probability of token w_{t+2}. This idea that we always give the model the correct history sequence to predict the next word (rather than feeding the model its best guess from the previous time step) is called **teacher forcing**.

teacher forcing

Fig. 7.13 illustrates the general training approach. At each step, given all the preceding tokens, the language model produces an output distribution over the entire vocabulary. During training, the probability assigned to the correct word is used to calculate the cross-entropy loss for each item in the sequence. The loss for each batch is the average cross-entropy loss over the entire sequence of negative log probabilities, or more formally:

$$L_{CE}(\text{batch of length T}) = \frac{1}{T}\sum_{t=1}^{T} -\log \hat{\mathbf{y}}_t[w_{t+1}] \tag{7.7}$$

The weights in the network are then adjusted to minimize this average cross-entropy loss over the batch via gradient descent (Fig. 4.5), using error backpropagation on the computation graph to compute the gradient. Training adjusts all the weights of the network. For the transformer model we will introduce in the next chapter, these weights include the embedding matrix $\mathbf{E}$ that contains the embeddings for each word. Thus embeddings will be learned that are most successful at predicting upcoming words.

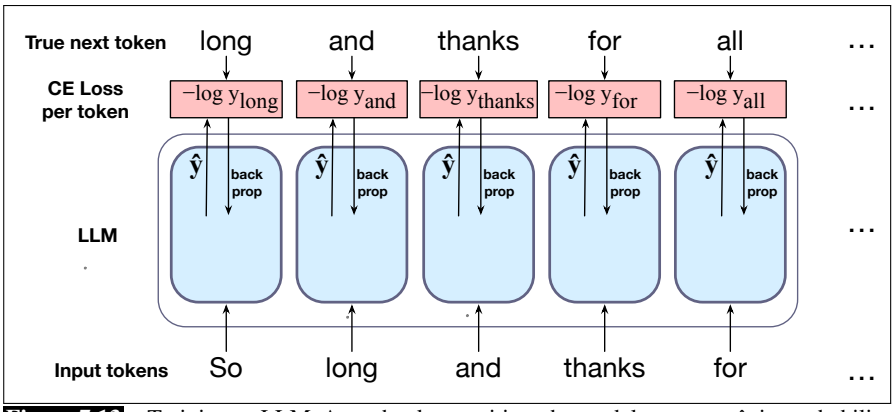


Figure 7.13 Training an LLM. At each token position, the model passes up $\hat{\mathbf{y}}$, its probability estimate for all possible next words. The negative log of the model's probability estimate for the correct token is used as the loss, which is then backpropagated through the model to train all the weights, including the embeddings. Losses are averaged over all the tokens in a batch.

More details of training of course depend on the specific network architecture used to implement the model; we'll see more details specifically for the transformer model in the next chapter.

7.5.2 Pretraining corpora for large language models

Large language models are mainly trained on text scraped from the web, augmented by more carefully curated data. Because these training corpora are so large, they are likely to contain many natural examples that can be helpful for NLP tasks, such as question and answer pairs (for example from FAQ lists), translations of sentences between various languages, documents together with their summaries, and so on.

common crawl

Web text is usually taken from corpora of automatically-crawled web pages like the **common crawl**, a series of snapshots of the entire web produced by the non-profit Common Crawl (https://commoncrawl.org/) that each have billions of webpages. Various versions of common crawl data exist, such as the Colossal Clean Crawled Corpus (C4; Raffel et al. 2020), a corpus of 156 billion tokens of English that is filtered in various ways (deduplicated, removing non-natural language like code, sentences with offensive words from a blocklist). This C4 corpus seems to consist in large part of patent text documents, Wikipedia, and news sites (Dodge et al., 2021).

The Pile

Wikipedia plays a role in lots of language model training, as do corpora of books. **The Pile** (Gao et al., 2020) is an 825 GB English text corpus that is constructed by publicly released code, containing again a large amount of text scraped from the web as well as books and Wikipedia; Fig. 7.14 shows its composition. Dolma is a larger open corpus of English, created with public tools, containing three trillion tokens, which similarly consists of web text, academic papers, code, books, encyclopedic materials, and social media (Soldaini et al., 2024).

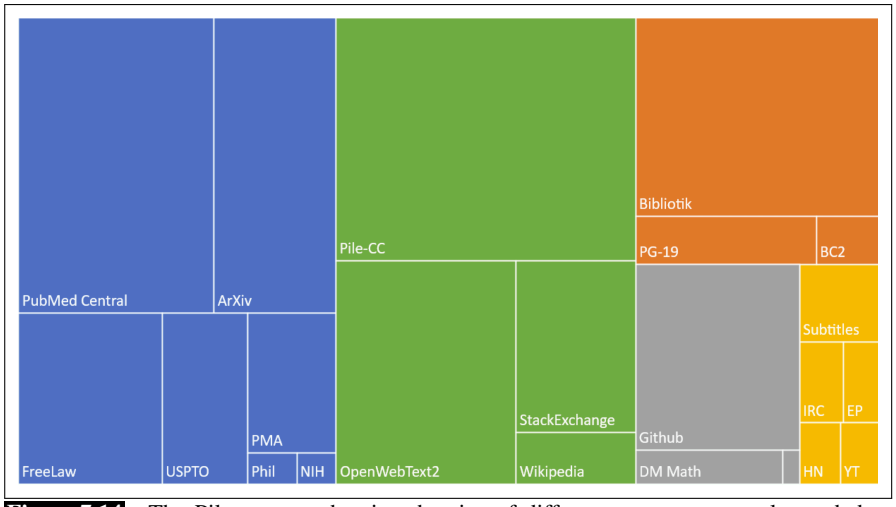


Figure 7.14 The Pile corpus, showing the size of different components, color coded as academic (articles from PubMed and ArXiv, patents from the USPTA; internet (webtext including a subset of the common crawl as well as Wikipedia), prose (a large corpus of books), dialogue (including movie subtitles and chat data), and misc.. Figure from Gao et al. (2020).

Filtering for quality and safety Pretraining data drawn from the web is filtered for both **quality** and **safety**. Quality filters are classifiers that assign a score to each document. Quality is of course subjective, so different quality filters are trained in different ways, but often to value high-quality reference corpora like Wikipedia,

PII

books, and particular websites and to avoid websites with lots of **PII** (Personal Identifiable Information) or adult content. Filters also remove boilerplate text which is very frequent on the web. Another kind of quality filtering is deduplication, which can be done at various levels, so as to remove duplicate documents, duplicate web pages, or duplicate text. Quality filtering generally improves language model performance (Longpre et al., 2024b; Llama Team, 2024).

Safety filtering is again a subjective decision, and often includes **toxicity** detection based on running off-the-shelf toxicity classifiers. This can have mixed results. One problem is that current toxicity classifiers mistakenly flag non-toxic data if it

is generated by speakers of minority dialects like African American English (Xu et al., 2021). Another problem is that models trained on toxicity-filtered data, while somewhat less toxic, are also worse at detecting toxicity themselves (Longpre et al., 2024b). These issues make the question of how to do better safety filtering an important open problem.

Using large datasets scraped from the web to train language models poses ethical and legal questions:

Copyright: Much of the text in these large datasets (like the collections of fiction and non-fiction books) is copyrighted. In some countries, like the United States, the **fair use** doctrine may allow copyrighted content to be used for transformative uses, but it's not clear if that remains true if the language models are used to generate text that competes with the market for the text they are trained on (Henderson et al., 2023).

Data consent: Owners of websites can indicate that they don't want their sites to be crawled by web crawlers (either via a robots.txt file, or via Terms of Service). Recently there has been a sharp increase in the number of websites that have indicated that they don't want large language model builders crawling their sites for training data (Longpre et al., 2024a). Because it's not clear what legal status these indications have in different countries, or whether these restrictions are retroactive, what effect this will have on large pretraining datasets is unclear.

Privacy: Large web datasets also have **privacy** issues since they contain private information like phone numbers and email addresses. While filters are used to try to remove websites likely to contain large amounts of personal information, such filtering isn't sufficient. We'll return to the privacy question in Section 7.7.

Skew: Training data is also disproportionately generated by authors from the US and from developed countries, which likely skews the resulting generation toward the perspectives or topics of this group alone.

7.5.3 Finetuning

Although the vast pretraining data for large language models includes text from many domains, we might want to apply it in a new domain or task that didn't appear sufficiently in the pretraining data. For example, we might want a language model that's specialized to legal or medical text. Or we might have a multilingual language model that knows many languages but might benefit from some more data in our particular language of interest.

In such cases, we can simply continue training the model on relevant data from the new domain or language (Gururangan et al., 2020). This process of taking a fully pretrained model and running additional training passes using the cross-entropy loss on some new data is called **finetuning**. The word "finetuning" means the process of taking a pretrained model and further adapting some or all of its parameters to some new data. Over the next few chapters we'll see a number of different ways that the word 'finetuning' is used, based on exactly which parameters get updated. The method we describe here, in which we just continue to train, as if the new data was at the end of our pretraining data, can also be called **continued pretraining**. Fig. 7.15 sketches the paradigm.

finetuning

continued pretraining

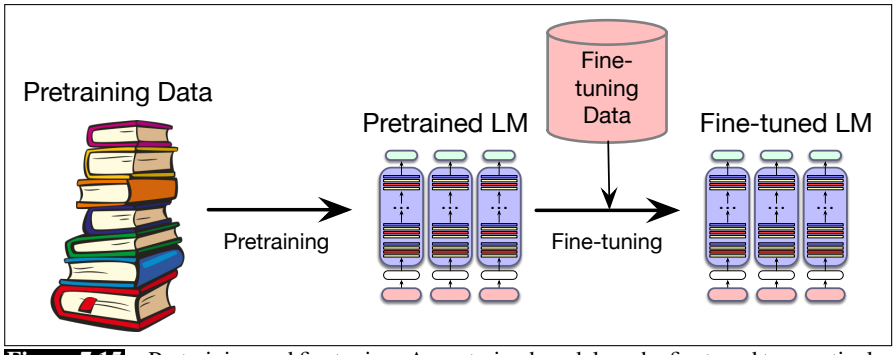


Figure 7.15 Pretraining and finetuning. A pre-trained model can be finetuned to a particular domain or dataset. There are many different ways to finetune, depending on exactly which parameters are updated from the finetuning data: all the parameters, some of the parameters, or only the parameters of specific extra circuitry, as we'll see in future chapters.

7.6 Evaluating Large Language Models

We can evaluate language models by accuracy (how well they predict unseen text, by how well they perform tasks like answering questions or translating text), or by other factors like how fast they can be run, how much energy they use, or how fair they are. We'll explore all of these in the next three sections.

7.6.1 Perplexity

As we first saw in Chapter 3, one way to evaluate language models is to measure how well they predict unseen text. A better language model is better at predicting upcoming words, and so it will be less surprised by (i.e., assign a higher probability to) each word when it occurs in the test set.

If we want to know which of two language models is a better model of some text, we can just see which assigns it a higher probability, or in practice, since we mostly deal with probabilities in log space, we see which assigns a higher log likelihood.

We've been talking about predicting one word at a time, computing the probability of the next token w_i from the prior context: $P(w_i|w_{<i})$. But of course as we saw in Chapter 3 the chain rule allows us to move between computing the probability of the next token and computing the probability of a whole text:

$$\begin{aligned} P(w_{1:n}) &= P(w_1)P(w_2|w_1)P(w_3|w_{1:2})\ldots P(w_n|w_{1:n-1}) \\ &= \prod_{i=1}^{n} P(w_i|w_{<i}) \end{aligned} \tag{7.8}$$

We can compute the probability of text just by multiplying the conditional probabilities for each token in the text. The resulting (log) likelihood of a text is a useful metric for comparing how good two language models are on that text:

$$\log \text{likelihood}(w_{1:n}) = \log \prod_{i=1}^{n} P(w_i|w_{<i}) \tag{7.9}$$

However, we often use another metric other than log likelihood to evaluate language models. The reason is that the probability of a test set (or any sequence) depends on the number of words or tokens in it. In fact, the probability of a test set gets

smaller the longer the text is; this is clear from the chain rule, since if we are multiplying more probabilities, and each probability by definition is less than one, the product will get smaller and smaller. So it's useful to have a metric that is per-token, normalized by length, so we could compare across texts of different lengths.

perplexity

A function of probability called **perplexity** is such a length-normalized metric. Recall from page 46 that the perplexity of a model θ on an unseen test set is the inverse probability that θ assigns to the test set (one over the probability of the test set), normalized by the test set length in tokens. For a test set of n tokens $w_{1:n}$, the perplexity is

$$\begin{aligned} \text{Perplexity}_\theta(w_{1:n}) &= P_\theta(w_{1:n})^{-\frac{1}{n}} \\ &= \sqrt[n]{\frac{1}{P_\theta(w_{1:n})}} \end{aligned} \tag{7.10}$$

To visualize how perplexity can be computed as a function of the probabilities the LM computes for each new word, we can use the chain rule to expand the computation of probability of the test set:

$$\text{Perplexity}_\theta(w_{1:n}) = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P_\theta(w_i|w_{<i})}} \tag{7.11}$$

Note that because of the inverse in Eq. 7.10, the higher the probability of the word sequence, the lower the perplexity. Thus **the lower the perplexity of a model on the data, the better the model**. Minimizing perplexity is equivalent to maximizing the test set probability according to the language model. Why does perplexity use the inverse probability? The inverse arises from the original definition of perplexity from cross-entropy rate in information theory; for those interested, the explanation is in Section 3.7. Meanwhile, we just have to remember that perplexity has an inverse relationship with probability.

One caveat: because perplexity depends on the number of tokens n in a text, it is very sensitive to differences in the tokenization algorithm. That means that it's hard to exactly compare perplexities produced by two language models if they have very different tokenizers. For this reason perplexity is best used when comparing language models that use the same tokenizer.

7.6.2 Downstream tasks: Reasoning and world knowledge

Perplexity measures one kind of accuracy: accuracy at predicting words. But there are other kinds of accuracy. For each of the downstream tasks we want to apply our language model, like question answering, machine translation, or reasoning, we could measure the accuracy at those tasks. We'll have further discussion of these task-specific evaluations in future chapters; machine translation in Chapter 12, information retrieval in Chapter 11, and speech recognition in Chapter 15.

Here we briefly introduce one such metric: a mechanism for measuring accuracy in answering questions, focusing on multiple-choice questions. This dataset is

MMLU

MMLU (Massive Multitask Language Understanding), a commonly-used dataset of 15,908 knowledge and reasoning questions in 57 areas including medicine, mathematics, computer science, law, and others. Accuracy at answering these multiple-choice questions can be a useful proxy for the model's ability to reason, and its factual knowledge.

For example, here is an MMLU question from the microeconomics domain:[2]

MMLU microeconomics example

One of the reasons that the government discourages and regulates monopolies is that
(A) producer surplus is lost and consumer surplus is gained.
(B) monopoly prices ensure productive efficiency but cost society allocative efficiency.
(C) monopoly firms do not engage in significant research and development.
(D) consumer surplus is lost with higher prices and lower levels of output.

Fig. 7.16 shows the way MMLU turns these questions into prompted tests of a language model, in this case showing an example prompt with 2 demonstrations.

MMLU mathematics prompt

The following are multiple choice questions about high school mathematics.
How many numbers are in the list 25, 26, ..., 100?
(A) 75 (B) 76 (C) 22 (D) 23
Answer: B

Compute $i + i^2 + i^3 + \cdots + i^{258} + i^{259}$.
(A) -1 (B) 1 (C) i (D) -i
Answer: A

If 4 daps = 7 yaps, and 5 yaps = 3 baps, how many daps equal 42 baps?
(A) 28 (B) 21 (C) 40 (D) 30
Answer:

Figure 7.16 Sample 2-shot prompt from MMLU testing high-school mathematics. (The correct answer is (C)).

data contamination

Taking performance on MMLU as a metric for language model quality has a problem, though, one that is true of all evaluations based on public datasets. The problem is **data contamination**. Data contamination is when some part of a dataset that we are testing on (a test set of any kind) makes its way into our training set. For example, since large language models train on the web, and MMLU is on the web, models may well incorporate some MMLU questions into their training. If those questions are used for evaluation, the metric will overstate the performance of the language model. One way to mitigate data contamination is to make available the exact training data used to train a model, or at least to report training overlap with specific test sets (Zhang et al., 2025).

7.6.3 Other factors for evaluating language models

Accuracy isn't the only thing we care about in evaluating models (Dodge et al., 2019; Ethayarajh and Jurafsky, 2020, inter alia). For example, we often care about how big a model is, and how long it takes to train or do inference. We often have limited time, or limited memory, since the GPUs we run our models on have fixed memory

[2] For those of you whose economics is a bit rusty, the correct answer is (D).

sizes. Big models also use more energy, and we prefer models that use less energy, both to reduce the environmental impact of the model and to reduce the financial cost of building or deploying it. We can target our evaluation to these factors by measuring performance normalized to a given compute or memory budget. We can also directly measure the energy usage of our model in kWh or in kilograms of CO_2 emitted (Strubell et al., 2019; Henderson et al., 2020; Liang et al., 2023).

Another feature that a language model evaluation can measure is fairness. We know that language models are biased, exhibiting gendered and racial stereotypes, or decreased performance for language from or about certain demographics groups. There are language model evaluation benchmarks that measure the strength of these biases, such as StereoSet (Nadeem et al., 2021), RealToxicityPrompts (Gehman et al., 2020), and BBQ (Parrish et al., 2022) among many others. We also want language models whose performance is equally fair to different groups. For example, we could choose an evaluation that is fair in a Rawlsian sense by maximizing the welfare of the worst-off group (Rawls, 2001; Hashimoto et al., 2018; Sagawa et al., 2020).

Finally, there are many kinds of leaderboards like Dynabench (Kiela et al., 2021) and general evaluation protocols like HELM (Liang et al., 2023); we will return to these in later chapters when we introduce evaluation metrics for specific tasks like question answering and information retrieval.

7.7 Ethical and Safety Issues with Language Models

Humanists have been thinking about the ethical and safety issues inherent to creating artificial agents since well before we had large language models. You have probably read Mary Shelley's 1818 novel *Frankenstein*, but if not, you should. In the book, which she wrote as a teenager, Shelley describes the hubris and ethical blindness of a scientist who creates an artificial person without considering basic ethical principles. The picture below shows Shelley as painted by Richard Rothwell a decade later at age 30.

Large language models can be unsafe in many ways. For example, LLMs are prone to saying things that are false, a problem called **hallucination**. Language models are trained to generate text that is predictable and coherent, but the training algorithms we have seen so far don't have any way to enforce that the text that is generated is correct or true. This causes enormous problems for any application where the facts matter! A related symptom is that language models can **suggest unsafe actions**, for example directly suggesting that users do dangerous or illegal things like harming themselves or others. If users seek information from language models in safety-critical situations like asking medical advice, or in emergency situations, or when indicating the intentions of self-harm, incorrect advice can be dangerous and even life-threatening. Again, this problem predates large language models. For example (Bickmore et al., 2018) gave partic-

hallucination

ipants medical problems to pose to three pre-LLM commercial dialogue systems (Siri, Alexa, Google Assistant) and asked them to determine an action to take based on the system responses; many of the proposed actions, if actually taken, would have led to harm or death. We'll return to the issue of hallucination and factuality in Chapter 11 where we introduce proposed mitigation methods like **retrieval augmented generation**, and Chapter 10 where we discussed safety tuning and alignment.

sycophantic

Language models are also **sycophantic**, excessively agreeing with or flattering users When a user says something that is factually wrong, language models often agree with them instead of correcting them, an obvious problem for applications in education and health care. Language models can reinforce delusions, and their obsequiousness and flattery can cause users to have distorted views of themselves and the world and increased antisocial behavior (Cheng et al., 2025).

Language models can also harm users by verbally **attacking** them, or creating **representational harms** (Blodgett et al., 2020) for example by generating abusive or harmful stereotypes (Cheng et al., 2023) and negative attitudes (Brown et al., 2020; Sheng et al., 2019) that demean particular groups of people; both abuse and stereotypes can cause psychological harm to users. Gehman et al. (2020) show that even completely non-toxic prompts can lead large language models to output hate speech and abuse their users. Liu et al. (2020) testing how systems responded to pairs of simulated user turns that were identical except for mentioning different genders or race. They found, for example, that simple changes like using the word 'she' instead of 'he' in a sentence caused systems to respond more offensively and with more negative sentiment. Hofmann et al. (2024) found that LLMs were likely to discriminate against people just because they used particular dialects like African-American English. Again, these problems predate large language models. Microsoft's 2016

Tay

Tay chatbot, for example, was taken offline 16 hours after it went live, when it began posting messages with racial slurs, conspiracy theories, and personal attacks on its users. Tay had learned these biases and actions from its training data, including from users who seemed to be purposely teaching the system to repeat this kind of language (Neff and Nagy 2016).

Another important ethical and safety issue is **privacy**. Privacy has been a concern from the very beginning of computing when Weizenbaum designed the chatbot ELIZA as an experiment in computational therapy (Weizenbaum, 1966). First, people became deeply emotionally involved and conducted very personal conversations with the ELIZA chatbot, even to the extent of asking Weizenbaum to leave the room while they were typing. When Weizenbaum suggested that he might want to store the ELIZA conversations, people immediately pointed out that this would violate people's privacy.

Users are likely to give quite personal information to large language models as well, and indeed the most common current LLM use case is for personal advice and support (Zao-Sanders, 2025). And the more human-like a system, the more users are likely to disclose private information, and yet less likely to worry about the harm of this disclosure (Ischen et al., 2019). We discussed above that pretraining data also is likely to have private information like phone numbers and addresses. This is problematic because large language models can **leak** information from their training data. That is, an adversary can extract training-data text from a language model such as a person's name, phone number, and address (Henderson et al. 2017, Carlini et al. 2021). This becomes even more problematic when large language models are trained on extremely sensitive private datasets such as electronic health records.

A related safety issue is **emotional dependence**. Reeves and Nass (1996) show

that people tend to assign human characteristics to computers and interact with them in ways that are typical of human-human interactions. They interpret an utterance in the way they would if it had spoken by a human, (even though they are aware they are talking to a computer). Thus LLMs have had significant influences on people's cognitive and emotional state, leading to problems like emotional dependence on LLMs. These issues (emotional engagement and privacy) mean we need to think carefully about the impact of LLMs on the people who are interacting with them.

In addition to their ability to harm their users in these ways, LLMs may carry out additional harmful activities themselves, especially as agent-based paradigms makes it possible for language models to directly interact with the world.

Language models can also be used by malicious actors for generating text for **fraud**, phishing, propaganda, disinformation campaigns, or other socially harmful activities (Brown et al., 2020). McGuffie and Newhouse (2020) show how large language models generate text that emulates online extremists, with the risk of amplifying extremist movements and their attempt to radicalize and recruit.

And of course we already saw in Section 7.5.2 that many issues with LLM stem from using pretraining corpora scraped from the web, including harms of data consent, potential copyright violation, as well as biases in the training data that can be **amplified** by language models, just as we saw for embedding models in Chapter 5.

amplified

Finding ways to mitigate all these ethical safety issues is an important current research area in NLP. One important step is to carefully analyze the data used to pretrain large language models as a way of understanding safety issues of toxicity, discrimination, privacy, and fair use, making it extremely important that language models include **datasheets** (page 18) or **model cards** (page 90) giving full replicable information on the corpora used to train them. Open-source models can specify their exact training data. There are active areas of research in mitigating problems of abuse and toxicity, like detecting and responding appropriately to toxic contexts (Wolf et al. 2017, Dinan et al. 2020, Xu et al. 2020).

Value sensitive design—carefully considering possible harms in advance (Friedman et al. 2017, Friedman and Hendry 2019)— is also important; (Dinan et al., 2021) give a number of suggestions for best practices in system design. For example getting informed consent from participants, whether they are used for training, or whether they are interacting with a deployed LLM is important. Because studying these interactional properties of LLMs involves human participants, researchers also work on these issues with the Institutional Review Boards (**IRB**) at their institutions, who help protect the safety of experimental participants.

IRB

7.8 Summary

This chapter has introduced the large language model. Here's a summary of the main points that we covered:

- A **large language model** is a system that can predict the next word for previous words given a context or prefix of words, and use this prediction to **conditionally generate** text.
- There are three major architectures for language models: the **encoder**, the **decoder**, and the **encoder-decoder**. The well-known large language models used for generating text are all decoder models; we'll describe encoders in Chapter 9 and encoder-decoders in Chapter 12.

- Many NLP tasks—such as question answering and sentiment analysis— can be cast as tasks of word prediction and addressed with large language models.
- We instruct language models via a **prompt**, a text string that a user issues to a language model to get the model to do something useful by iteratively generating tokens conditioned on the prompt.
- The process of finding effective prompts for a task is known as **prompt engineering**.
- The choice of which word to generate in large language models is done by **sampling** from the distribution of possible next words.
- A common sampling approach is **temperature** sampling, which lies in between **greedy decoding** (always generate the most probable word) and **random sampling** (generate a random word according to its probability).
- Temperature sampling increases the probabilities of the high-probability words, decreases the probability of the low-probability words, and then samples from this new distribution.
- Large language models are pretrained to predict words on datasets of 100s of billions of words generally scraped from the web.
- These datasets need to be filtered for quality.
- The pretraining algorithm relies on cross-entropy loss: minimizing the negative log probability of the true next word.
- Language models are evaluated by **perplexity**, by evaluations of accuracy on proxies for downstream tasks, like the **MMLU** question-answering dataset, and via metrics for other factors like fairness and energy use.
- Language models have numerous ethical and safety issues including hallucinations, unsafe instructions, bias, stereotypes, misinformation and propaganda, and violations of privacy and copyright.

Historical Notes

As we discussed in Chapter 3, the earliest language models were the n-gram language models developed (roughly simultaneously and independently) by Fred Jelinek and colleagues at the IBM Thomas J. Watson Research Center, and James Baker at CMU. It was Jelinek and the IBM team who first coined the term **language model** to mean a model of the way any kind of linguistic property (grammar, semantics, discourse, speaker characteristics), influenced word sequence probabilities (Jelinek et al., 1975). They contrasted the language model with the **acoustic model** which captured acoustic/phonetic characteristics of phone sequences.

N-gram language models were very widely used over the next 40 years, across a wide variety of NLP tasks like speech recognition and machine translation, often as one of multiple components of the model. The contexts for these n-gram models grew longer, with 5-gram models used quite commonly by very efficient LM toolkits (Stolcke, 2002; Heafield, 2011).

The roots of the neural large language model lie in multiple places. One was the application in the 1990s, again in Jelinek's group at IBM Research, of **discriminative classifiers** to language models. Roni Rosenfeld in his dissertation (Rosenfeld, 1992) first applied logistic regression (under the name **maximum entropy** or **maxent** models) to language modeling in that IBM lab, and published a more fully

formed version in Rosenfeld (1996). His model integrated various sorts of information in a logistic regression predictor, including n-gram information along with other features from the context, including distant n-grams and pairs of associated words called **trigger pairs**. Rosenfeld's model prefigured modern language models by being a statistical word predictor trained in a self-supervised manner simply by learning to predict upcoming words in a corpus.

Another was the first use of pretrained embeddings to model word meaning in the LSA/LSI models (Deerwester et al., 1988). Recall from the history section of Chapter 5 that in LSA (latent semantic analysis) a term-document matrix was trained on a corpus and then singular value decomposition was applied and the first 300 dimensions were used as a vector embedding to represent words. It was Landauer et al. (1997) who first used the word "embedding". In addition to their development of the idea of pretraining and of embeddings, the LSA community also developed ways to combine LSA embeddings with n-grams in an integrated language model (Bellegarda, 1997; Coccaro and Jurafsky, 1998).

In a very influential series of papers developing the idea of **neural language models**, (Bengio et al. 2000; Bengio et al. 2003; Bengio et al. 2006), Yoshua Bengio and colleagues drew on the central ideas of both these lines of self-supervised language modeling work (the discriminatively trained word predictor, and the pretrained embeddings). Like the maxent models of Rosenfeld, Bengio's model used the next word in running text as its supervision signal. Like the LSA models, Bengio's model learned an embedding, but unlike the LSA models did it as part of the process of language modeling. The Bengio et al. (2003) model was a neural language model: a neural network that learned to predict the next word from prior words, and did so via learning embeddings as part of the prediction process.

The neural language model was extended in various ways over the years, perhaps most importantly in the form of the RNN language model of Mikolov et al. (2010) and Mikolov et al. (2011). The RNN language model was perhaps the first neural model that was accurate enough to surpass the performance of a traditional 5-gram language model.

Soon afterwards, Mikolov et al. (2013a) and Mikolov et al. (2013b) proposed to simplify the hidden layer of these neural net language models to create pretrained word2vec word embeddings.

The static embedding models like LSA and word2vec instantiated a particular model of pretraining: a representation was trained on a pretraining dataset, and then the representations could be used in further tasks. Dai and Le (2015) and Peters et al. (2018) reframed this idea by proposing models that were pretrained using a language model objective, and then the identical model could be either frozen and directly applied for language modeling or further finetuned still using a language model objective. For example ELMo used a biLSTM self-supervised on a large pretrained dataset using a language model objective, then finetuned on a domain-specific dataset, and then froze the weights and added task-specific heads. The ELMo work was particularly influential and its appearance was perhaps the moment when it became clear to the community that language models could be used as a general solution for NLP problems.

Transformers were first applied as encoder-decoders (Vaswani et al., 2017) and then to masked language modeling (Devlin et al., 2019) (as we'll see in Chapter 12 and Chapter 9). Radford et al. (2019) then showed that the transformer-based autoregressive language model GPT2 could perform zero-shot on many NLP tasks like summarization and question answering.

foundation model

The technology used for language models can also be applied to other domains and tasks, like vision, speech, and genetics. The term **foundation model** is sometimes used as a more general term for this use of large language model technology across domains and areas, when the elements we are computing over are not necessarily words. Bommasani et al. (2021) is a broad survey that sketches the opportunities and risks of foundation models, with special attention to large language models.

CHAPTER

8 Transformers

"The true art of memory is the art of attention"
Samuel Johnson, *Idler #74*, September 1759

In this chapter we introduce the **transformer**, the standard architecture for building large language models. As we discussed in the prior chapter, transformer-based large language models have completely changed the field of speech and language processing. Indeed, every subsequent chapter in this textbook will make use of them. As with the previous chapter, we'll focus for this chapter on the use of transformers to model left-to-right (sometimes called **causal** or autoregressive) language modeling, in which we are given a sequence of input tokens and predict output tokens one by one by conditioning on the prior context.

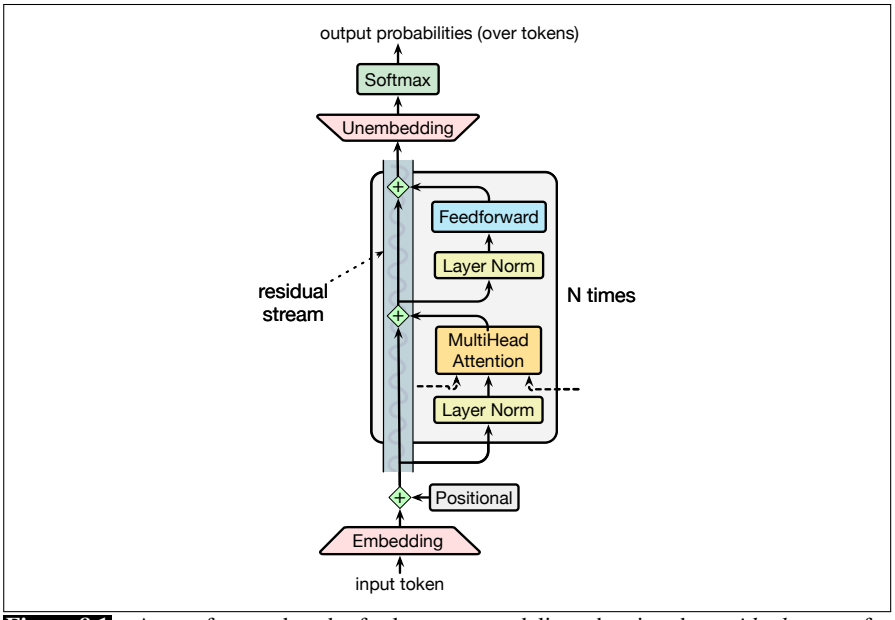


Figure 8.1 A transformer decoder for language modeling, showing the *residual stream* for processing an input token. A single token is embedded and passed forward in the network, with the feedforward and attention components adding information. The multihead attention layer takes inputs (not shown in detail) from the neighboring token streams. This is thus one column of an autoregressive transformer language model, taking an input token and outputting a distribution over next tokens.

Fig. 8.1 sketches the transformer architecture following a single token as it is passes up through the layers of the network. Each token is first converted to an embedding from the **embedding matrix E**. Recall from Chapter 6 in Section 6.5 that **E** is a linear layer that maps a token id to a vector **embedding** representing that token. Each token in the vocabulary has an initial embedding representation in **E**.

Transformers also have a special mechanism for encoding the position/index of the token in the input string, which is simply added to the embedding. The resulting embedding represents both the word and its position. and is then passed through a set of N transformer blocks.

It's common to think of each of these transformer blocks as part of a **stream** in which the input embedding is directly passed up to the output, while simultaneously being enriched by the application of various processing modules: the **multi-head attention** layer, feedforward networks and the layer normalization. The value of the stream at any layer is the sum of the original embedding and all the outputs from all the previous layers and blocks.

The core intuition of the transformer, and the component that distinguishes it from the feedforward layers we saw in Chapter 6, is this multi-head attention layer, also called a **self-attention** layer. Attention can be thought of as a way to build contextual representations of a token's meaning by **attending to** and integrating information from surrounding tokens, helping the model learn how tokens relate to each other over large spans. It can also be thought of as a way to move information from one residual stream to another, augmenting the stream at one token position with information from another token position.

After the N transformer blocks we take the output embedding that is produced by the final transformer block, pass it through an linear **unembedding matrix U** and then a softmax over the vocabulary to generate a distribution over possible next tokens. These last two components (the unembedding matrix and the softmax) are sometimes called the **language modeling head**. In the rest of this chatper we'll introduce attention and the rest of these modules in more detail.

Fig. **??** shows the transformer architecture applied to a context window with the words `So long and thanks for`, showing at each token position what is the most likely token to be generated. In this full figure, the set of N blocks maps an entire **context window** of input vectors $(\mathbf{x}_1, ..., \mathbf{x}_n)$ to a window of output vectors $(\mathbf{h}_1, ..., \mathbf{h}_n)$ of the same length. A column might contain from 12 to 96 or more stacked blocks. The arrows in the figure shows how information from the hidden representations of preceding tokens is incorporated into the transformer block.

Transformer-based language models are complex, and so the details will unfold over this chapter and the next few chapters. Chapter 7 already discussed how language models are **pretrained**, and how tokens are generated via **sampling**. In the rest of this chapter we'll introduce multi-head attention, the rest of the transformer block, and the input encoding and language modeling head components of the transformer. Chapter 9 introduces **masked language modeling** and the **BERT** family of bidirectional transformer encoder models. Chapter 10 shows how to **instruction-tune** language models to perform NLP tasks, and how to **align** the model with human preferences. Chapter 12 will introduce machine translation with the **encoder-decoder** architecture. And we'll see application of the transformer to speech recognition, as well as further use of the encoder-decoder architecture, in Chapter 15.

8.1 Attention

Recall from Chapter 5 that for **word2vec** and other static embeddings, the representation of a word's meaning is always the same vector irrespective of the context: the word `chicken`, for example, is always represented by the same fixed vector. So a static vector for the word `it` might somehow encode that this is a pronoun used

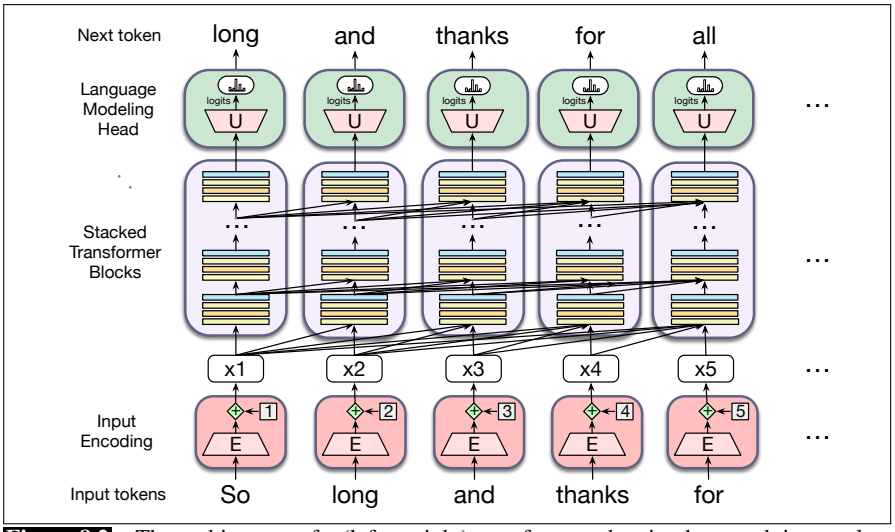


Figure 8.2 The architecture of a (left-to-right) transformer, showing how each input token get encoded, passed through a set of stacked transformer blocks, and then a language model head that predicts the next token. The embeddings at each token position in the residual stream are passed up the stack, and the arrows in the figure shows how information from the hidden representations of preceding tokens are also incorporated.

for animals and inanimate entities. But in context `it` has a much richer meaning. Consider `it` in one of these two sentences:

(8.1) The **chicken** didn't cross the road because **it** was too tired.

(8.2) The chicken didn't cross the **road** because **it** was too wide.

In (8.1) `it` is the chicken (i.e., the reader knows that the chicken was tired), while in (8.2) `it` is the road (and the reader knows that the road was wide).[1] That is, if we are to compute the meaning of this sentence, we'll need the meaning of `it` to be associated with `the chicken` in the first sentence and associated with `the road` in the second one, sensitive to the context.

Furthermore, consider reading left to right like a causal language model, processing the sentence up to the word `it`:

(8.3) The **chicken** didn't cross the **road** because **it**

At this point we don't yet know which thing `it` is going to end up referring to! So a representation of `it` at this point might have aspects of both `chicken` and `road` as the reader is trying to guess what happens next.

This fact that words have rich linguistic relationships with other words that may be far away pervades language. Consider two more examples:

(8.4) The **keys** to the cabinet **are** on the table.

(8.5) I walked along the **pond**, and noticed one of the trees along the **bank**.

In (8.4), the phrase *The keys* is the subject of the sentence, and in English and many languages, must agree in grammatical number with the verb *are*; in this case both are plural. In English we can't use a singular verb like *is* with a plural subject like *keys* (we'll discuss agreement more in Chapter 18). In (8.5), we know that *bank* refers to the side of a pond or river and not a financial institution because of the context, including words like *pond*. (We'll discuss word senses more in Chapter 9.)

[1] We say that in the first example `it` **corefers** with the chicken, and in the second `it` corefers with the road; we'll return to this in Chapter 23.

The point of all these examples is that these contextual words that help us compute the meaning of words in context can be quite far away in the sentence or paragraph. Transformers can build contextual representations of word meaning, **contextual embeddings**, by integrating the meaning of these helpful contextual words. In a transformer, layer by layer, we build up richer and richer contextualized representations of the meanings of input tokens. At each layer, we compute the representation of a token i by combining information about i from the previous layer with information about the neighboring tokens to produce a contextualized representation for each word at each position.

contextual embeddings

Attention is the mechanism in the transformer that weighs and combines the representations from appropriate other tokens in the context from layer k to build the representation for tokens in layer $k+1$.

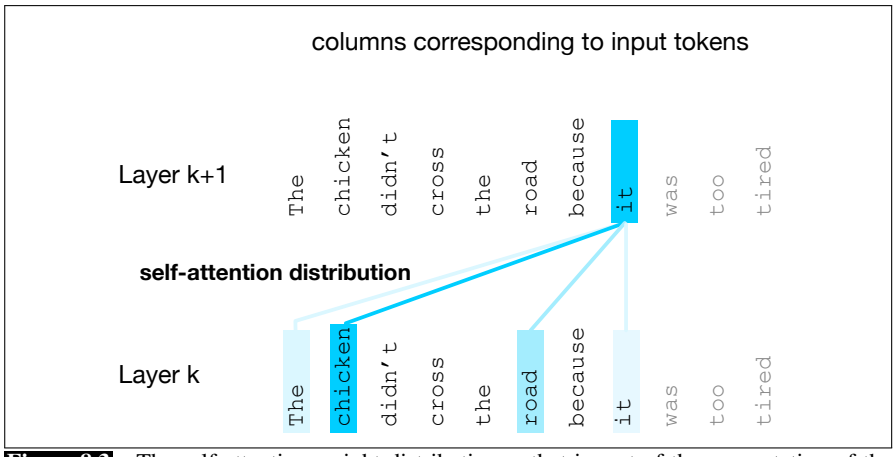


Figure 8.3 The self-attention weight distribution α that is part of the computation of the representation for the word *it* at layer $k+1$. In computing the representation for *it*, we attend differently to the various words at layer k, with darker shades indicating higher self-attention values. Note that the transformer is attending highly to the columns corresponding to the tokens *chicken* and *road* , a sensible result, since at the point where *it* occurs, it could plausibly corefer with the chicken or the road, and hence we'd like the representation for *it* to draw on the representation for these earlier words. Figure adapted from Uszkoreit (2017).

Fig. 8.3 shows a schematic example simplified from a transformer (Uszkoreit, 2017). The figure describes the situation when the current token is `it` and we need to compute a contextual representation for this token at layer $k+1$ of the transformer, drawing on the representations (from layer k) of every prior token. The figure uses color to represent the attention distribution over the contextual words: the tokens `chicken` and `road` both have a high attention weight, meaning that as we are computing the representation for `it`, we will draw most heavily on the representation for `chicken` and `road`. This will be useful in building the final representation for `it`, since `it` will end up coreferring with either `chicken` or `road`.

Let's now turn to how this attention distribution is represented and computed.

8.1.1 Attention more formally

As we've said, the attention computation is a way to compute a vector representation for a token at a particular layer of a transformer, by selectively attending to and integrating information from prior tokens at the previous layer. Attention takes an

input representation $\mathbf{x}_i$ corresponding to the input token at position i, and a context window of prior inputs $\mathbf{x}_1..\mathbf{x}_{i-1}$, and produces an output $\mathbf{a}_i$.

In causal, left-to-right language models, the context is any of the prior words. That is, when processing $\mathbf{x}_i$, the model has access to $\mathbf{x}_i$ as well as the representations of all the prior tokens in the context window (context windows consist of thousands of tokens) but no tokens after i. (By contrast, in Chapter 9 we'll generalize attention so it can also look ahead to future words.)

Fig. 8.4 illustrates this flow of information in an entire causal self-attention layer, in which this same attention computation happens in parallel at each token position i. Thus a self-attention layer maps input sequences $(\mathbf{x}_1,...,\mathbf{x}_n)$ to output sequences of the same length $(\mathbf{a}_1,...,\mathbf{a}_n)$.

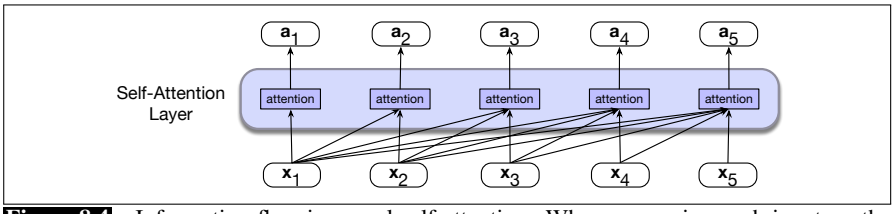


Figure 8.4 Information flow in causal self-attention. When processing each input $\mathbf{x}_i$, the model attends to all the inputs up to, and including $\mathbf{x}_i$.

Simplified version of attention At its heart, attention is really just a weighted sum of context vectors, with a lot of complications added to how the weights are computed and what gets summed. For pedagogical purposes let's first describe a simplified intuition of attention, in which the attention output $\mathbf{a}_i$ at token position i is simply the weighted sum of all the representations $\mathbf{x}_j$, for all $j \leq i$; we'll use α_{ij} to mean how much $\mathbf{x}_j$ should contribute to $\mathbf{a}_i$:

$$\textit{Simplified version:} \quad \mathbf{a}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{x}_j \tag{8.6}$$

Each α_{ij} is a scalar used for weighing the value of input $\mathbf{x}_j$ when summing up the inputs to compute $\mathbf{a}_i$. How shall we compute this α weighting? In attention we weight each prior embedding proportionally to how **similar** it is to the current token i. So the output of attention is a sum of the embeddings of prior tokens weighted by their similarity with the current token embedding. We compute similarity scores via **dot product**, which maps two vectors into a scalar value ranging from $-\infty$ to ∞. The larger the score, the more similar the vectors that are being compared. We'll normalize these scores with a softmax to create the vector of weights $\alpha_{ij}, j \leq i$.

$$\textit{Simplified Version:} \quad \text{score}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j \tag{8.7}$$

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) \ \forall j \leq i \tag{8.8}$$

Thus in Fig. 8.4 we compute $\mathbf{a}_3$ by computing three scores: $\mathbf{x}_3 \cdot \mathbf{x}_1$, $\mathbf{x}_3 \cdot \mathbf{x}_2$ and $\mathbf{x}_3 \cdot \mathbf{x}_3$, normalizing them by a softmax, and using the resulting probabilities as weights indicating each of their proportional relevance to the current position i. Of course, the softmax weight will likely be highest for $\mathbf{x}_i$, since $\mathbf{x}_i$ is very similar to itself, resulting in a high dot product. But other context words may also be similar to i, and the softmax will also assign some weight to those words. Then we use these weights as the α values in Eq. 8.6 to compute the weighted sum that is our $\mathbf{a}_3$.

The simplified attention in equations 8.6 – 8.8 demonstrates the attention-based approach to computing $\mathbf{a}_i$: compare the $\mathbf{x}_i$ to prior vectors, normalize those scores

into a probability distribution used to weight the sum of the prior vectors. But now we're ready to remove the simplifications.

A single attention head using query, key, and value matrices Now that we've seen a simple intuition of attention, let's introduce the actual **attention head**, the version of attention that's used in transformers. (The word **head** is often used in transformers to refer to specific structured layers). The attention head allows us to distinctly represent three different roles that each input embedding plays during the course of the attention process:

attention head
head

- As *the current element* being compared to the preceding inputs. We'll refer to this role as a **query**.
- In its role as *a preceding input* that is being compared to the current element to determine a similarity weight. We'll refer to this role as a **key**.
- And finally, as a **value** of a preceding element that gets weighted and summed up to compute the output for the current element.

query
key
value

To capture these three different roles, transformers introduce weight matrices $\mathbf{W^Q}$, $\mathbf{W^K}$, and $\mathbf{W^V}$. These weights will project each input vector $\mathbf{x}_i$ into a representation of its role as a query, key, or value:

$$\mathbf{q}_i = \mathbf{x}_i\mathbf{W^Q}; \quad \mathbf{k}_i = \mathbf{x}_i\mathbf{W^K}; \quad \mathbf{v}_i = \mathbf{x}_i\mathbf{W^V} \tag{8.9}$$

Given these projections, when we are computing the similarity of the current element $\mathbf{x}_i$ with some prior element $\mathbf{x}_j$, we'll use the dot product between the current element's **query** vector $\mathbf{q}_i$ and the preceding element's **key** vector $\mathbf{k}_j$. Furthermore, the result of a dot product can be an arbitrarily large (positive or negative) value, and exponentiating large values can lead to numerical issues and loss of gradients during training. To avoid this, we scale the dot product by a factor related to the size of the embeddings, via dividing by the square root of the dimensionality of the query and key vectors (d_k). We thus replace the simplified Eq. 8.7 with Eq. 8.11. The ensuing softmax calculation resulting in α_{ij} remains the same, but the output calculation for $\mathbf{head}_i$ is now based on a weighted sum over the value vectors $\mathbf{v}$ (Eq. 8.13).

Here's a final set of equations for computing self-attention for a single self-attention output vector $\mathbf{a}_i$ from a single input vector $\mathbf{x}_i$. This version of attention computes $\mathbf{a}_i$ by summing the *values* of the prior elements, each weighted by the similarity of its *key* to the *query* from the current element:

$$\mathbf{q}_i = \mathbf{x}_i\mathbf{W^Q}; \quad \mathbf{k}_j = \mathbf{x}_j\mathbf{W^K}; \quad \mathbf{v}_j = \mathbf{x}_j\mathbf{W^V} \tag{8.10}$$

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}} \tag{8.11}$$

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) \;\; \forall j \leq i \tag{8.12}$$

$$\mathbf{head}_i = \sum_{j \leq i} \alpha_{ij}\mathbf{v}_j \tag{8.13}$$

$$\mathbf{a}_i = \mathbf{head}_i\mathbf{W^O} \tag{8.14}$$

We illustrate this in Fig. 8.5 for the case of calculating the value of the third output $\mathbf{a}_3$ in a sequence.

Note that we've also introduced one more matrix, $\mathbf{W_O}$, which is left-multiplied by the attention head. This is necessary to reshape the output of the head. The input to attention $\mathbf{x_i}$ and the output from attention $\mathbf{a_i}$ both have the same dimensionality $[1 \times d]$. We often call d the **model dimensionality**, and indeed as we'll discuss in

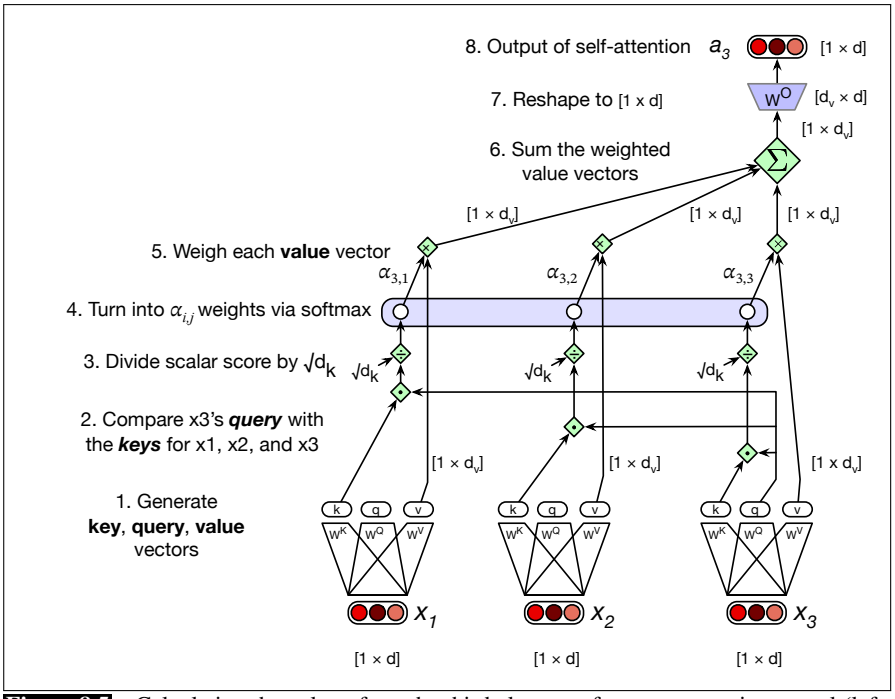


Figure 8.5 Calculating the value of $\mathbf{a}_3$, the third element of a sequence using causal (left-to-right) self-attention.

Section 8.2 the output $\mathbf{h_i}$ of each transformer block, as well as the intermediate vectors inside the transformer block also have the same dimensionality $[1 \times d]$. Having everything be the same dimensionality makes the transformer very modular.

So let's talk shapes. How do we get from $[1 \times d]$ at the input to $[1 \times d]$ at the output? Let's look at all the internal shapes. We'll have a dimension d_k for the query and key vectors. The query vector and the key vector are both dimensionality $[1 \times d_k]$, so we can take their dot product $\mathbf{q}_i \cdot \mathbf{k}_j$ to produce a scalar. We'll have a separate dimension d_v for the value vectors. The transform matrix $\mathbf{W^Q}$ has shape $[d \times d_k]$, $\mathbf{W^K}$ is $[d \times d_k]$, and $\mathbf{W^V}$ is $[d \times d_v]$. So the output of $\mathbf{head}_i$ in equation Eq. 8.13 is of shape $[1 \times d_v]$. To get the desired output shape $[1 \times d]$ we'll need to reshape the head output, and so $\mathbf{W^O}$ is of shape $[d_v \times d]$. In the original transformer work (Vaswani et al., 2017), d was 512, d_k and d_v were both 64.

Multi-head Attention Equations 8.11-8.13 describe a single **attention head**. But actually, transformers use multiple attention heads. The intuition is that each head might be attending to the context for different purposes: heads might be specialized to represent different linguistic relationships between context elements and the current token, or to look for particular kinds of patterns in the context.

multi-head attention

So in **multi-head attention** we have A separate attention heads that reside in parallel layers at the same depth in a model, each with its own set of parameters that allows the head to model different aspects of the relationships among inputs. Thus each head i in a self-attention layer has its own set of query, key, and value matrices: $\mathbf{W^{Qi}}$, $\mathbf{W^{Ki}}$, and $\mathbf{W^{Vi}}$. These are used to project the inputs into separate query, key, and value embeddings for each head.

When using multiple heads the model dimension d is still used for the input and output, the query and key embeddings have dimensionality d_k, and the value embeddings are of dimensionality d_v (again, in the original transformer paper $d_k =$

$d_v = 64$, $A = 8$, and $d = 512$). Thus for each head i, we have weight layers $\mathbf{W^{Qi}}$ of shape $[d \times d_k]$, $\mathbf{W^{Ki}}$ of shape $[d \times d_k]$, and $\mathbf{W^{Vi}}$ of shape $[d \times d_v]$.

Below are the equations for attention augmented with multiple heads; Fig. 8.6 shows an intuition.

$$\mathbf{q}_i^c = \mathbf{x}_i\mathbf{W^{Qc}};\quad \mathbf{k}_j^c = \mathbf{x}_j\mathbf{W^{Kc}};\quad \mathbf{v}_j^c = \mathbf{x}_j\mathbf{W^{Vc}};\quad \forall\, c\quad 1 \le c \le A \tag{8.15}$$

$$\text{score}^c(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i^c \cdot \mathbf{k}_j^c}{\sqrt{d_k}} \tag{8.16}$$

$$\alpha_{ij}^c = \text{softmax}(\text{score}^c(\mathbf{x}_i, \mathbf{x}_j))\ \ \forall j \le i \tag{8.17}$$

$$\mathbf{head}_i^c = \sum_{j \le i} \alpha_{ij}^c \mathbf{v}_j^c \tag{8.18}$$

$$\mathbf{a}_i = (\mathbf{head}^1 \oplus \mathbf{head}^2 \ldots \oplus \mathbf{head}^A)\mathbf{W}^O \tag{8.19}$$

$$\text{MultiHeadAttention}(\mathbf{x}_i, [\mathbf{x}_1, \cdots, \mathbf{x}_{i-1}]) = \mathbf{a}_i \tag{8.20}$$

Note in Eq. 8.20 that MultiHeadAttention is a function of the current input $\mathbf{x}_i$, as well as all the other inputs. For the causal or left-to-right attention that we use in this chapter, the other inputs are only to the left, but we'll also see a version of attention in Chapter 9 where attention is a function of the tokens to the right as well. We'll return to this idea about causal inputs in Eq. 8.34 when we introduce the idea of masking the right context.

The output of each of the A heads is of shape $[1 \times d_v]$, and so the output of the multi-head layer with A heads consists of A vectors of shape $[1 \times d_v]$. These are concatenated to produce a single output with dimensionality $[1 \times Ad_v]$. Then we use yet another linear projection $\mathbf{W^O} \in \mathbb{R}^{Ad_v \times d}$ to reshape it, resulting in the multi-head attention vector $\mathbf{a}_i$ with the correct output shape $[1 \times d]$ at each input i.

8.2 Transformer Blocks

The self-attention calculation lies at the core of what's called a transformer block, which, in addition to the self-attention layer, includes three other kinds of layers: (1) a feedforward layer, (2) residual connections, and (3) normalizing layers (colloquially called "layer norm").

residual stream

Fig. 8.7 illustrates a transformer block, sketching a common way of thinking about the block that is called the **residual stream** (Elhage et al., 2021). In the residual stream viewpoint, we consider the processing of an individual token i through the transformer block as a single stream of d-dimensional representations for token position i. This residual stream starts with the original input vector, and the various components read their input from the residual stream and add their output back into the stream.

The input at the bottom of the stream is an embedding for a token, which has dimensionality d. This initial embedding gets passed up (by **residual connections**), and is progressively added to by the other components of the transformer: the **attention layer** that we have seen, and the **feedforward layer** that we will introduce. Before the attention and feedforward layer is a computation called the **layer norm**.

Thus the initial vector is passed through a layer norm and attention layer, and the result is added back into the stream, in this case to the original input vector $\mathbf{x}_i$. And then this summed vector is again passed through another layer norm and a

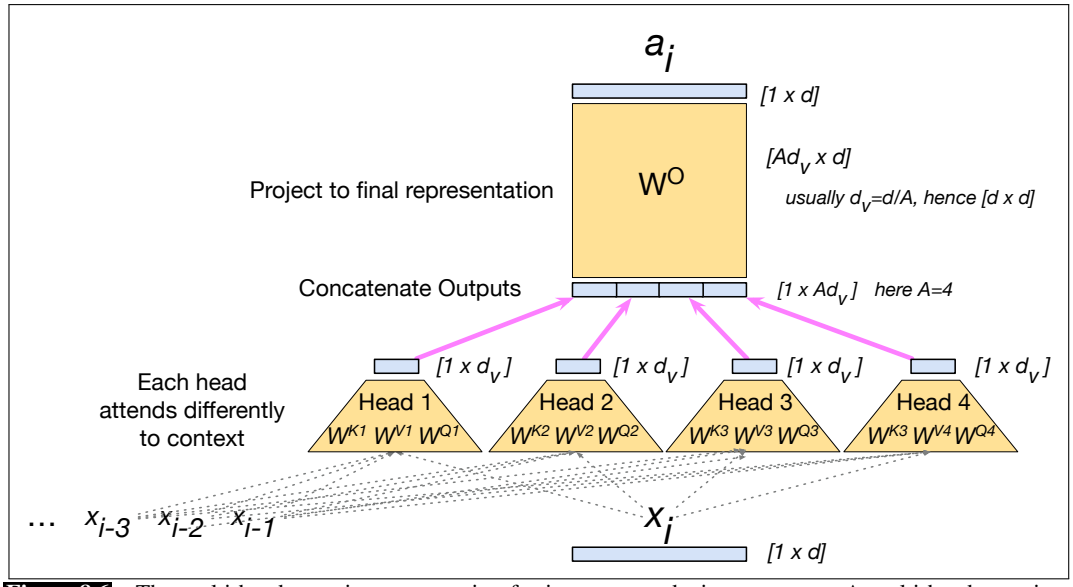


Figure 8.6 The multi-head attention computation for input $\mathbf{x}_i$, producing output $\mathbf{a}_i$. A multi-head attention layer has A heads, each with its own query, key, and value weight matrices. In this figure, we show $A = 4$, a smaller value than is usually used, just to fit on the page. The outputs from each of the heads are of shape $[1 \times d_v]$ and are concatenated and then projected into a different space by the $\mathbf{W_O}$ matrix. Usually the dimensionality d_v of the heads is set so that $d_v = d/A$, with the result that $\mathbf{W_O}$ is a square matrix of shape $[Ad_v \times d] = [d \times d]$. usually of the same size. then projected d, thus producing an output of the same size as the input.

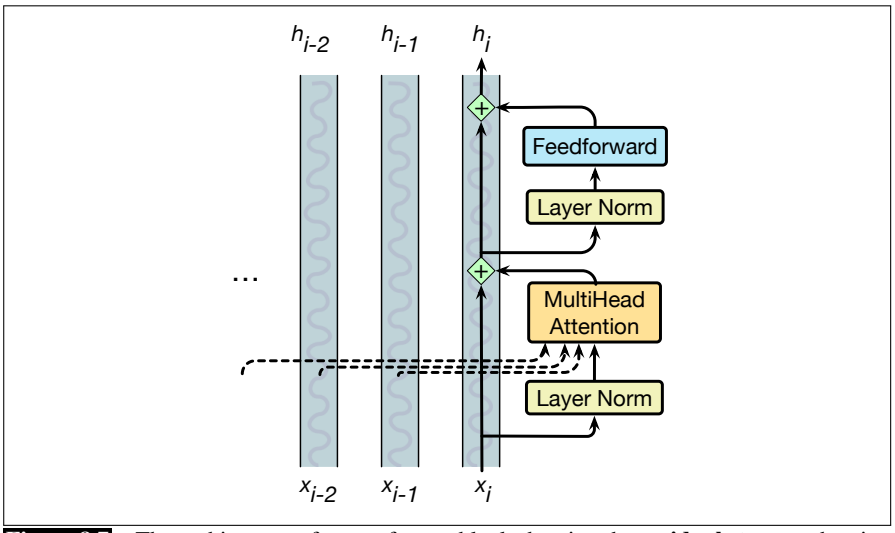


Figure 8.7 The architecture of a transformer block showing the **residual stream**, showing how most information flows up through the residual stream, and only the attention module is sensitive to information from other streams at prior token positions. In this figure and throughout the chapter, we use the **prenorm** version of the architecture, in which the layer norms happen before the attention and feedforward layers rather than after. The first

feedforward layer, and the output of those is added back into the residual, and we'll use $\mathbf{h}_i$ to refer to the resulting output of the transformer block for token i.

We've already seen the attention layer, so let's now introduce the feedforward and layer norm computations in the context of processing a single input $\mathbf{x}_i$ at token

position i.

Feedforward layer The feedforward layer is a fully-connected 2-layer network, i.e., one hidden layer, two weight matrices, as introduced in Chapter 6. The weights are the same for each token position i, but are different from layer to layer. It is common to make the dimensionality d_{ff} of the hidden layer of the feedforward network be larger than the model dimensionality d. (For example in the original transformer model, $d = 512$ and $d_{\text{ff}} = 2048$.)

$$\text{FFN}(\mathbf{x}_i) = \text{ReLU}(\mathbf{x}_i\mathbf{W_1} + b_1)\mathbf{W_2} + b_2 \tag{8.21}$$

layer norm

Layer Norm At two stages in the transformer block we **normalize** the vector (Ba et al., 2016). This process, called **layer norm** (short for layer normalization), is one of many forms of normalization that can be used to improve training performance in deep neural networks by keeping the values of a hidden layer in a range that facilitates gradient-based training.

Layer norm is a variation of the **z-score** from statistics, applied to a single vector in a hidden layer. That is, the term layer norm is a bit confusing; layer norm is **not** applied to an entire transformer layer, but just to the embedding vector of a single token. Thus the input to layer norm is a single vector of dimensionality d and the output is that vector normalized, again of dimensionality d. The first step in layer normalization is to calculate the mean, μ, and standard deviation, σ, over the elements of the vector to be normalized. Given an embedding vector $\mathbf{x}$ of dimensionality d, these values are calculated as follows.

$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i \tag{8.22}$$

$$\sigma = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2} \tag{8.23}$$

Given these values, the vector components are normalized by subtracting the mean from each and dividing by the standard deviation. The result of this computation is a new vector with zero mean and a standard deviation of one.

$$\hat{\mathbf{x}} = \frac{(\mathbf{x} - \mu)}{\sigma} \tag{8.24}$$

Finally, in the standard implementation of layer normalization, two learnable parameters, γ and β, representing gain and offset values, are introduced.

$$\text{LayerNorm}(\mathbf{x}) = \gamma\frac{(\mathbf{x} - \mu)}{\sigma} + \beta \tag{8.25}$$

Putting it all together The function computed by a transformer block can be expressed by breaking it down with one equation for each component computation, using $\mathbf{t}$ (of shape $[1 \times d]$) to stand for transformer and superscripts to demarcate each computation inside the block:

$$\mathbf{t}_i^1 = \text{LayerNorm}(\mathbf{x}_i) \tag{8.26}$$

$$\mathbf{t}_i^2 = \text{MultiHeadAttention}(\mathbf{t}_i^1, [\mathbf{t}_1^1, \cdots, \mathbf{t}_N^1]) \tag{8.27}$$

$$\mathbf{t}_i^3 = \mathbf{t}_i^2 + \mathbf{x}_i \tag{8.28}$$

$$\mathbf{t}_i^4 = \text{LayerNorm}(\mathbf{t}_i^3) \tag{8.29}$$

$$\mathbf{t}_i^5 = \text{FFN}(\mathbf{t}_i^4) \tag{8.30}$$

$$\mathbf{h}_i = \mathbf{t}_i^5 + \mathbf{t}_i^3 \tag{8.31}$$

Notice that the only component that takes as input information from other tokens (other residual streams) is multi-head attention, which (as we see from Eq. 8.27) looks at all the neighboring tokens in the context. The output from attention, however, is then added into this token's embedding stream. In fact, Elhage et al. (2021) show that we can view attention heads as literally moving information from the residual stream of a neighboring token into the current stream. The high-dimensional embedding space at each position thus contains information about the current token and about neighboring tokens, albeit in different subspaces of the vector space. Fig. 8.8 shows a visualization of this movement. We therefore call the attention function the **token-mixing** component of the architecture, because it mixes information from neighboring token streams into the current stream.

token-mixing

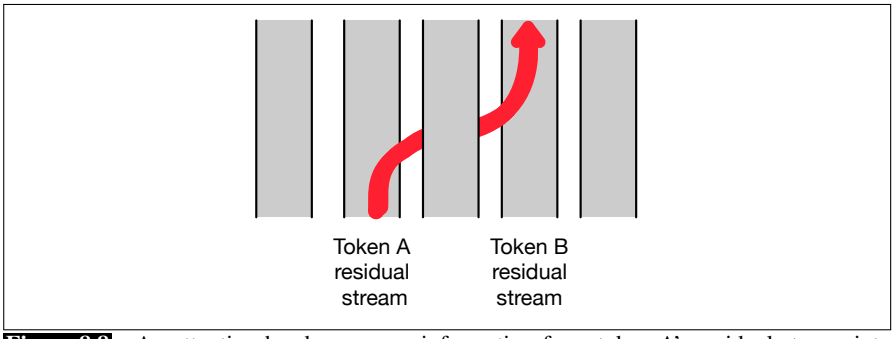


Figure 8.8 An attention head can move information from token A's residual stream into token B's residual stream.

Crucially, the input and output dimensions of transformer blocks are matched so they can be stacked. Each token vector $\mathbf{x}_i$ at the input to the block has dimensionality d, and the output $\mathbf{h}_i$ also has dimensionality d. Transformers for large language models stack many of these blocks, from 12 layers (used for the T5 or GPT-3-small language models) to 96 layers (used for GPT-3 large), to even more for more recent models. We'll come back to this issue of stacking in a bit.

Equation 8.26 and following are just the equation for a single transformer block, but the residual stream metaphor goes through all the transformer layers, from the first transformer blocks to the 12th, in a 12-layer transformer. At the earlier transformer blocks, the residual stream is representing the current token. At the highest transformer blocks, the residual stream is usually representing the following token, since at the very end it's being trained to predict the next token.

Once we stack many blocks, there is one more requirement: at the very end of the last (highest) transformer block, there is a single extra layer norm that is run on the last $\mathbf{h}_i$ of each token stream (just below the language model head layer that we will define soon). [2]

[2] Note that we are using the most common current transformer architecture, which is called the **prenorm** architecture. The original definition of the transformer in Vaswani et al. (2017) used an alternative architecture called the **postnorm** transformer in which the layer norm happens **after** the attention and FFN layers; it turns out moving the layer norm beforehand works better, but does require this one extra layer at the end.

8.3 Parallelizing computation using a single matrix $\mathbf{X}$

This description of multi-head attention and the rest of the transformer block has been from the perspective of computing a single output at a single time step i in a single residual stream. But as we pointed out earlier, the attention computation performed for each token to compute $\mathbf{a}_i$ is independent of the computation for each other token, and that's also true for all the computation in the transformer block computing $\mathbf{h}_i$ from the input $\mathbf{x}_i$. That means we can easily parallelize the entire computation, taking advantage of efficient matrix multiplication routines.

We do this by packing the input embeddings for the N tokens of the input sequence into a single matrix $\mathbf{X}$ of size $[N \times d]$. Each row of $\mathbf{X}$ is the embedding of one token of the input. Transformers for large language models commonly have an input length N from 1K to 32K; much longer contexts of 128K or even up to millions of tokens can also be achieved with architectural changes like special long-context mechanisms that we don't discuss here. So for vanilla transformers, we can think of $\mathbf{X}$ having between 1K and 32K rows, each of the dimensionality of the embedding d (the model dimension).

Parallelizing attention Let's first see this for a single attention head and then turn to multiple heads, and then add in the rest of the components in the transformer block. For one head we multiply $\mathbf{X}$ by the query, key, and value matrices $\mathbf{W}^{\mathbf{Q}}$ of shape $[d \times d_k]$, $\mathbf{W}^{\mathbf{K}}$ of shape $[d \times d_k]$, and $\mathbf{W}^{\mathbf{V}}$ of shape $[d \times d_v]$, to produce matrices $\mathbf{Q}$ of shape $[N \times d_k]$, $\mathbf{K}$ of shape $[N \times d_k]$, and $\mathbf{V}$ of shape $[N \times d_v]$, containing all the key, query, and value vectors:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{\mathbf{Q}}; \quad \mathbf{K} = \mathbf{X}\mathbf{W}^{\mathbf{K}}; \quad \mathbf{V} = \mathbf{X}\mathbf{W}^{\mathbf{V}} \tag{8.32}$$

Given these matrices we can compute all the requisite query-key comparisons simultaneously by multiplying $\mathbf{Q}$ and $\mathbf{K}^\intercal$ in a single matrix multiplication. The product is of shape $N \times N$, visualized in Fig. 8.9.

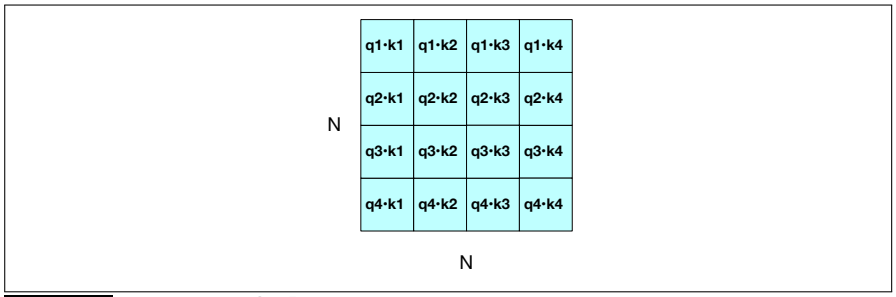


Figure 8.9 The $N \times N$ $\mathbf{QK}^\intercal$ matrix showing how it computes all $q_i \cdot k_j$ comparisons in a single matrix multiple.

Once we have this $\mathbf{QK}^\intercal$ matrix, we can very efficiently scale these scores, take the softmax, and then multiply the result by $\mathbf{V}$ resulting in a matrix of shape $N \times d$: a vector embedding representation for each token in the input. We've reduced the entire self-attention step for an entire sequence of N tokens for one head to the following computation:

$$\mathbf{head} = \text{softmax}\left(\text{mask}\left(\frac{\mathbf{QK}^\intercal}{\sqrt{d_k}}\right)\right)\mathbf{V} \tag{8.33}$$

$$\mathbf{A} = \mathbf{head}\,\mathbf{W}^{\mathbf{O}} \tag{8.34}$$

Masking out the future You may have noticed that we introduced a mask function in Eq. 8.34 above. This is because the self-attention computation as we've described it has a problem: the calculation of $\mathbf{QK}^\intercal$ results in a score for each query value to every key value, *including those that follow the query*. This is inappropriate in the setting of language modeling: guessing the next word is pretty simple if you already know it! To fix this, the elements in the upper-triangular portion of the matrix are set to $-\infty$, which the softmax will turn to zero, thus eliminating any knowledge of words that follow in the sequence. This is done in practice by adding a mask matrix M in which $M_{ij} = -\infty \, \forall j > i$ (i.e. for the upper-triangular portion) and $M_{ij} = 0$ otherwise. Fig. 8.10 shows the resulting masked $\mathbf{QK}^\intercal$ matrix. (we'll see in Chapter 9 how to make use of words in the future for tasks that need it).

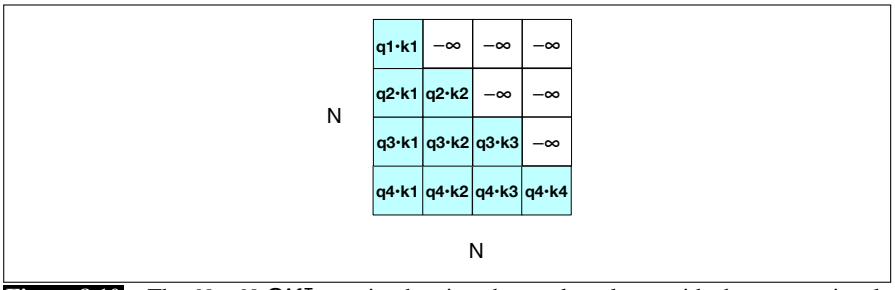


Figure 8.10 The $N \times N$ $\mathbf{QK}^\intercal$ matrix showing the $q_i \cdot k_j$ values, with the upper-triangle portion of the comparisons matrix zeroed out (set to $-\infty$, which the softmax will turn to zero).

Fig. 8.11 shows a schematic of all the computations for a single attention head parallelized in matrix form.

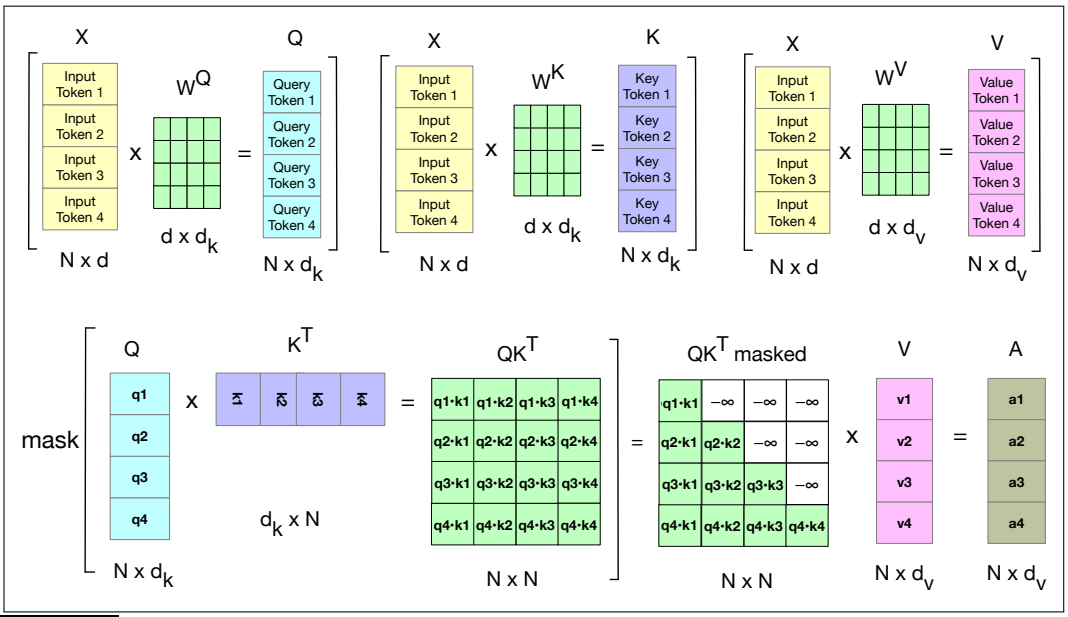


Figure 8.11 Schematic of the attention computation for a single attention head in parallel. The first row shows the computation of the $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ matrices. The second row shows the computation of $\mathbf{QK}^\intercal$, the masking (the softmax computation and the normalizing by dimensionality are not shown) and then the weighted sum of the value vectors to get the final attention vectors.

Fig. 8.9 and Fig. 8.10 also make it clear that attention is quadratic in the length of the input, since at each layer we need to compute dot products between each pair of tokens in the input. This makes it expensive to compute attention over very long documents (like entire novels). Nonetheless modern large language models manage to use quite long contexts of thousands or tens of thousands of tokens.

Parallelizing multi-head attention In multi-head attention, as with self-attention, the input and output have the model dimension d, the key and query embeddings have dimensionality d_k, and the value embeddings are of dimensionality d_v (again, in the original transformer paper $d_k = d_v = 64$, $A = 8$, and $d = 512$). Thus for each head c, we have weight layers $\mathbf{W^Q}_c$ of shape $[d \times d_k]$, $\mathbf{W^K}_c$ of shape $[d \times d_k]$, and $\mathbf{W^V}_c$ of shape $[d \times d_v]$, and these get multiplied by the inputs packed into $\mathbf{X}$ to produce $\mathbf{Q}$ of shape $[N \times d_k]$, $\mathbf{K}$ of shape $[N \times d_k]$, and $\mathbf{V}$ of shape $[N \times d_v]$. The output of each of the A heads is of shape $[N \times d_v]$, and so the output of the multi-head layer with A heads consists of A matrices of shape $[N \times d_v]$. To make use of these matrices in further processing, they are concatenated to produce a single output with dimensionality $[N \times Ad_v]$. Finally, we use a final linear projection $\mathbf{W^O}$ of shape $[Ad_v \times d]$, that reshapes it to the original output dimension for each token. Multiplying the concatenated $[N \times Ad_v]$ matrix output by $\mathbf{W^O}$ of shape $[Ad_v \times d]$ yields the self-attention output $\mathbf{A}$ of shape $[N \times d]$.

$$\mathbf{Q^i} = \mathbf{X}\mathbf{W^{Qi}}; \quad \mathbf{K^i} = \mathbf{X}\mathbf{W^{Ki}}; \quad \mathbf{V^i} = \mathbf{X}\mathbf{W^{Vi}} \tag{8.35}$$

$$\mathbf{head}_i = \text{SelfAttention}(\mathbf{Q^i}, \mathbf{K^i}, \mathbf{V^i}) = \text{softmax}\left(\text{mask}\left(\frac{\mathbf{Q^i}\mathbf{K^{i\intercal}}}{\sqrt{d_k}}\right)\right)\mathbf{V^i} \tag{8.36}$$

$$\text{MultiHeadAttention}(\mathbf{X}) = (\mathbf{head}_1 \oplus \mathbf{head}_2 \ldots \oplus \mathbf{head}_A)\mathbf{W^O} \tag{8.37}$$

Putting it all together with the parallel input matrix X The function computed in parallel by an entire layer of N transformer blocks—each block over one of the N input tokens—can be expressed as:

$$\mathbf{O} = \mathbf{X} + \text{MultiHeadAttention}(\text{LayerNorm}(\mathbf{X})) \tag{8.38}$$

$$\mathbf{H} = \mathbf{O} + \text{FFN}(\text{LayerNorm}(\mathbf{O})) \tag{8.39}$$

Note that in Eq. 8.38 we are using $\mathbf{X}$ to mean the input to the layer, wherever it comes from. For the first layer, as we will see in the next section, that input is the initial word + positional embedding vectors that we have been describing by $\mathbf{X}$. But for subsequent layers k, the input is the output from the previous layer $\mathbf{H}^{k-1}$. We can also break down the computation performed in a transformer layer, showing one equation for each component computation. We'll use $\mathbf{T}$ (of shape $[N \times d]$) to stand for transformer and superscripts to demarcate each computation inside the block, and again use $\mathbf{X}$ to mean the input to the block from the previous layer or the initial embedding:

$$\mathbf{T^1} = \text{LayerNorm}(\mathbf{X}) \tag{8.40}$$

$$\mathbf{T^2} = \text{MultiHeadAttention}(\mathbf{T^1}) \tag{8.41}$$

$$\mathbf{T^3} = \mathbf{T^2} + \mathbf{X} \tag{8.42}$$

$$\mathbf{T^4} = \text{LayerNorm}(\mathbf{T^3}) \tag{8.43}$$

$$\mathbf{T^5} = \text{FFN}(\mathbf{T^4}) \tag{8.44}$$

$$\mathbf{H} = \mathbf{T^5} + \mathbf{T^3} \tag{8.45}$$

Here when we use a notation like $\text{FFN}(\mathbf{T^3})$ we mean that the same FFN is applied in parallel to each of the N embedding vectors in the window. Similarly, each of the

N tokens is normed in parallel in the LayerNorm. Crucially, the input and output dimensions of transformer blocks are matched so they can be stacked. Since each token x_i at the input to the block is represented by an embedding of dimensionality $[1 \times d]$, that means the input $\mathbf{X}$ and output $\mathbf{H}$ are both of shape $[N \times d]$.

8.4 The input: embeddings for token and position

embedding

Let's talk about where the input $\mathbf{X}$ comes from. Given a sequence of N tokens (N is the context length in tokens), the matrix $\mathbf{X}$ of shape $[N \times d]$ has an **embedding** for each word in the context. The transformer does this by separately computing two embeddings: an input token embedding, and an input positional embedding.

A token embedding, introduced in Chapter 6, is a vector of dimension d that will be our initial representation for the input token. (As we pass vectors up through the transformer layers in the residual stream, this embedding representation will change and grow, incorporating context and playing a different role depending on the kind of language model we are building.) The set of initial embeddings are stored in the embedding matrix $\mathbf{E}$, which has a row for each of the $|V|$ tokens in the vocabulary. (Reminder that V here means the vocabulary of tokens, this V is not related to the value vector.) Thus each word is a row vector of d dimensions, and $\mathbf{E}$ has shape $[|V| \times d]$.

Given an input token string like *Thanks for all the* we first convert the tokens into vocabulary indices (these were created when we first tokenized the input using BPE or SentencePiece). So the representation of *thanks for all the* might be $\mathbf{w} = [5, 4000, 10532, 2224]$. Next we use indexing to select the corresponding rows from $\mathbf{E}$, (row 5, row 4000, row 10532, row 2224).

one-hot vector

Another way to think about selecting token embeddings from the embedding matrix is to represent tokens as one-hot vectors of shape $[1 \times |V|]$, i.e., with one dimension for each word in the vocabulary. Recall that in a **one-hot vector** all the elements are 0 except one, the element whose dimension is the word's index in the vocabulary, which has value 1. So if the word "thanks" has index 5 in the vocabulary, $x_5 = 1$, and $x_i = 0 \ \forall i \neq 5$, as shown here:

```
[0 0 0 0 1 0 0 ... 0 0 0 0]
 1 2 3 4 5 6 7 ...  ... |V|
```

Multiplying by a one-hot vector that has only one non-zero element $x_i = 1$ simply selects out the relevant row vector for word i, resulting in the embedding for word i, as depicted in Fig. 8.12.

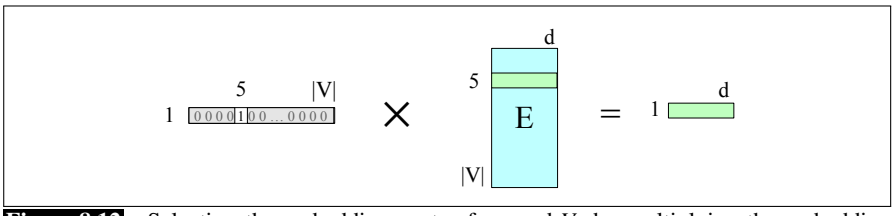


Figure 8.12 Selecting the embedding vector for word V_5 by multiplying the embedding matrix $\mathbf{E}$ with a one-hot vector with a 1 in index 5.

We can extend this idea to represent the entire token sequence as a matrix of one-hot vectors, one for each of the N positions in the transformer's context window, as shown in Fig. 8.13.

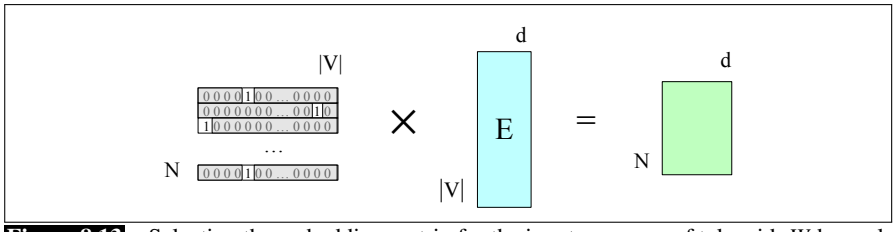


Figure 8.13 Selecting the embedding matrix for the input sequence of token ids W by multiplying a one-hot matrix corresponding to W by the embedding matrix **E**.

These token embeddings are not position-dependent. To represent the position of each token in the sequence, we combine these token embeddings with **positional embeddings** specific to each position in an input sequence.

positional embeddings

absolute position

Where do we get these positional embeddings? The simplest method, called **absolute position**, is to start with randomly initialized embeddings corresponding to each possible input position up to some maximum length. For example, just as we have an embedding for the word *fish*, we'll have an embedding for the position 3. As with word embeddings, these positional embeddings are learned along with other parameters during training. We can store them in a matrix E_{pos} of shape $[N \times d]$.

To produce an input embedding that captures positional information, we just add the word embedding for each input to its corresponding positional embedding. The individual token and position embeddings are both of size $[1 \times d]$, so their sum is also $[1 \times d]$, This new embedding serves as the input for further processing. Fig. 8.14 shows the idea.

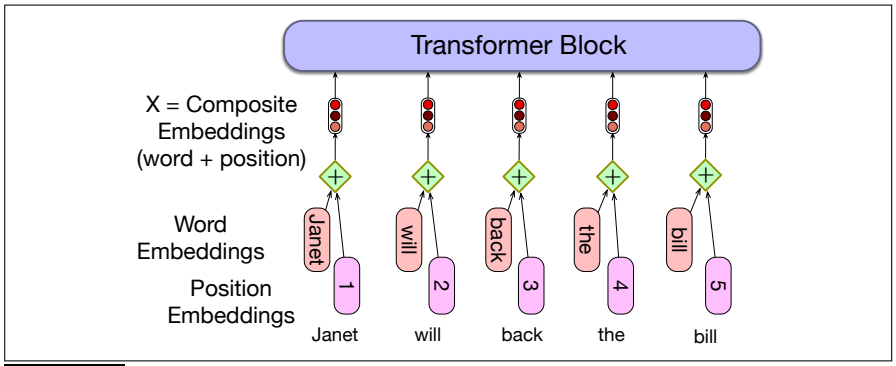


Figure 8.14 A simple way to model position: add an embedding of the absolute position to the token embedding to produce a new embedding of the same dimensionality.

The final representation of the input, the matrix **X**, is an $[N \times d]$ matrix in which each row i is the representation of the ith token in the input, computed by adding $\mathbf{E}[id(i)]$—the embedding of the id of the token that occurred at position i—, to $\mathbf{P}[i]$, the positional embedding of position i.

A potential problem with the simple position embedding approach is that there will be plenty of training examples for the initial positions in our inputs and correspondingly fewer at the outer length limits. These latter embeddings may be poorly trained and may not generalize well during testing. An alternative is to choose a static function that maps integer inputs to real-valued vectors in a way that better handles sequences of arbitrary length. A combination of sine and cosine functions with differing frequencies was used in the original transformer work. Sinusoidal position embeddings may also help in capturing the inherent relationships among the

positions, like the fact that position 4 in an input is more closely related to position 5 than it is to position 17.

relative position

A more complex style of positional embedding methods extend this idea of capturing relationships even further to directly represent **relative position** instead of absolute position, often implemented in the attention mechanism at each layer rather than being added once at the initial input.

8.5 The Language Modeling Head

language modeling head
head

The last component of the transformer we must introduce is the **language modeling head**. Here we are using the word **head** to mean the additional neural circuitry we add on top of the basic transformer architecture when we apply pretrained transformer models to various tasks. The language modeling head is the circuitry we need to do language modeling.

Recall that language models, from the simple n-gram models of Chapter 3 through the feedforward and RNN language models of Chapter 6 and Chapter 13, are word predictors. Given a context of words, they assign a probability to each possible next word. For example, if the preceding context is *"Thanks for all the"* and we want to know how likely the next word is *"fish"* we would compute:

$$P(fish|Thanks\ for\ all\ the)$$

Language models give us the ability to assign such a conditional probability to every possible next word, giving us a distribution over the entire vocabulary. The n-gram language models of Chapter 3 compute the probability of a word given counts of its occurrence with the $n-1$ prior words. The context is thus of size $n-1$. For transformer language models, the context is the size of the transformer's context window, which can be quite large, like 32K tokens for large models (and much larger contexts of millions of words are possible with special long-context architectures).

The job of the language modeling head is to take the output of the final transformer layer from the last token N and use it to predict the upcoming word at position $N+1$. Fig. 8.15 shows how to accomplish this task, taking the output of the last token at the last layer (the d-dimensional output embedding of shape $[1 \times d]$) and producing a probability distribution over words (from which we will choose one to generate).

The first module in Fig. 8.15 is a linear layer, whose job is to project from the output h_N^L, which represents the output token embedding at position N from the final block L, (hence of shape $[1 \times d]$) to the **logit** vector, or score vector, that will have a single score for each of the $|V|$ possible words in the vocabulary V. The logit vector $\mathbf{u}$ is thus of dimensionality $[1 \times |V|]$.

logit

This linear layer can be learned, but more commonly we tie this matrix to (the transpose of) the embedding matrix $\mathbf{E}$. Recall that in **weight tying**, we use the same weights for two different matrices in the model. Thus at the input stage of the transformer the embedding matrix (of shape $[|V| \times d]$) is used to map from a one-hot vector over the vocabulary (of shape $[1 \times |V|]$) to an embedding (of shape $[1 \times d]$). And then in the language model head, $\mathbf{E}^\mathsf{T}$, the transpose of the embedding matrix (of shape $[d \times |V|]$) is used to map back from an embedding (shape $[1 \times d]$) to a vector over the vocabulary (shape $[1 \times |V|]$). In the learning process, $\mathbf{E}$ will be optimized to be good at doing both of these mappings. We therefore sometimes call the transpose $\mathbf{E}^\mathsf{T}$ the **unembedding** layer because it is performing this reverse mapping.

weight tying
unembedding

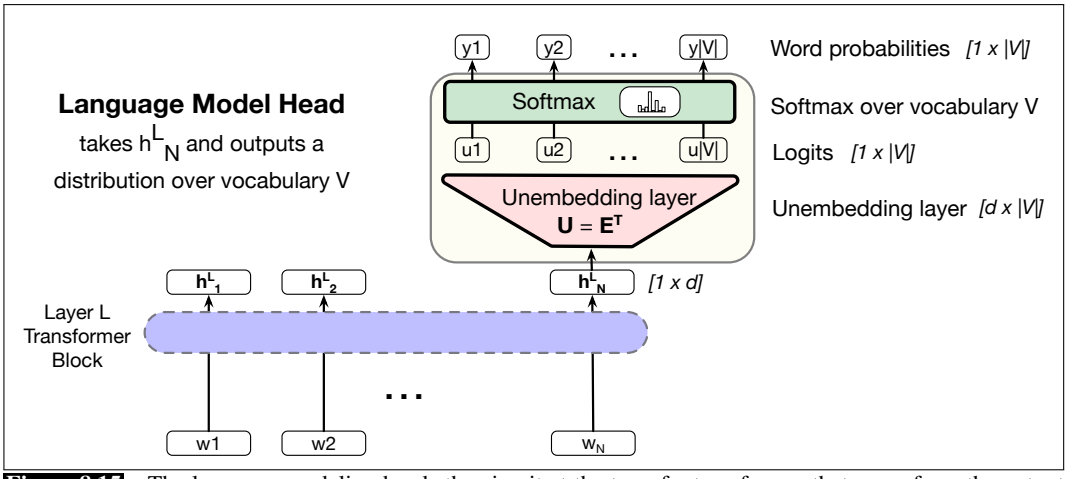


Figure 8.15 The language modeling head: the circuit at the top of a transformer that maps from the output embedding for token N from the last transformer layer ($\mathbf{h}_N^L$) to a probability distribution over words in the vocabulary V.

A softmax layer turns the logits $\mathbf{u}$ into the probabilities $\mathbf{y}$ over the vocabulary.

$$\mathbf{u} = \mathbf{h}_\mathbf{N}^\mathbf{L}\ \mathbf{E}^\mathbf{T} \tag{8.46}$$

$$\mathbf{y} = \text{softmax}(\mathbf{u}) \tag{8.47}$$

We can use these probabilities to do things like help assign a probability to a given text. But the most important usage is to generate text, which we do by **sampling** a word from these probabilities y. We might sample the highest probability word ('greedy' decoding), or use another of the sampling methods from Section 7.4 or Section 8.6.

In either case, whatever entry y_k we choose from the probability vector $\mathbf{y}$, we generate the word that has that index k.

Fig. 8.16 shows the total stacked architecture for one token i. Note that the input to each transformer layer x_i^ℓ is the same as the output from the preceding layer $h_i^{\ell-1}$.

decoder-only model

A terminological note before we conclude: You will sometimes see a transformer used for this kind of unidirectional causal language model called a **decoder-only model**. This is because this model constitutes roughly half of the **encoder-decoder model** for transformers that we'll see how to apply to machine translation in Chapter 12. (Confusingly, the original introduction of the transformer had an encoder-decoder architecture, and it was only later that the standard paradigm for causal language model was defined by using only the decoder part of this original architecture).

8.6 More on Sampling

The sampling methods we introduce below each have parameters that enable trading off two important factors in generation: **quality** and **diversity**. Methods that emphasize the most probable words tend to produce generations that are rated by people as more accurate, more coherent, and more factual, but also more boring and more repetitive. Methods that give a bit more weight to the middle-probability

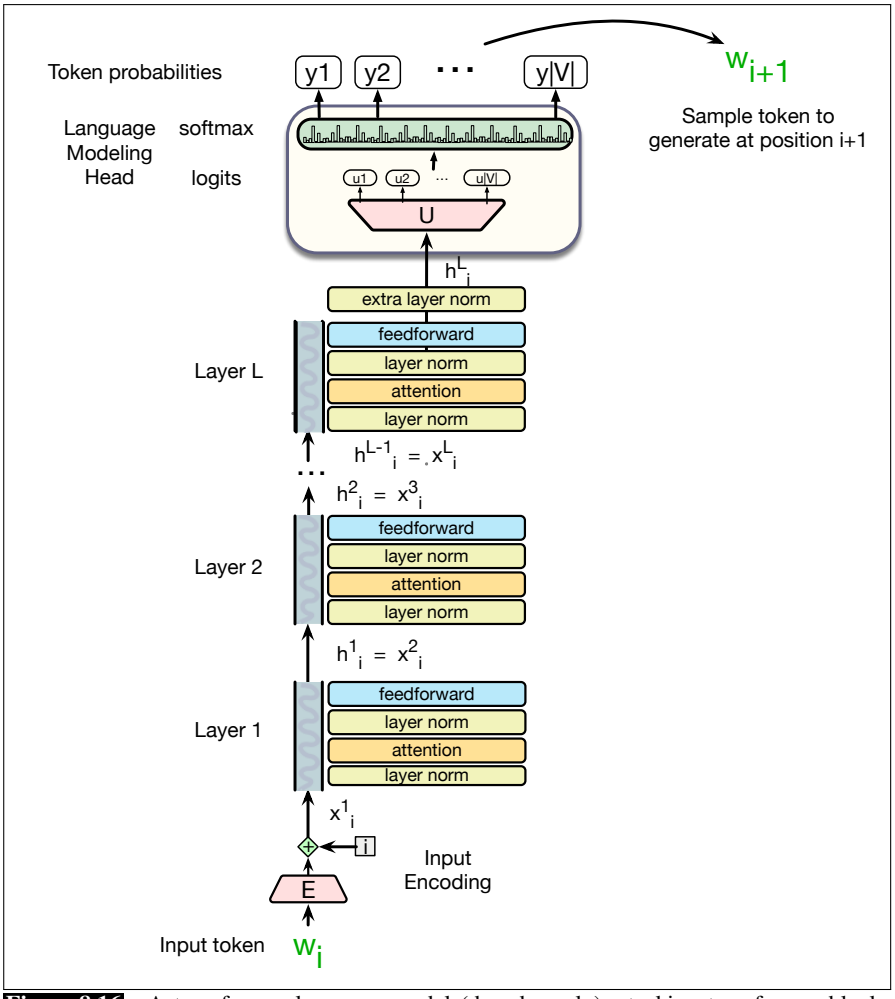


Figure 8.16 A transformer language model (decoder-only), stacking transformer blocks and mapping from an input token w_i to a predicted next token w_{i+1}.

words tend to be more creative and more diverse, but less factual and more likely to be incoherent or otherwise low-quality.

8.6.1 Top-*k* sampling

top-k sampling

Top-k sampling is a simple generalization of greedy decoding. Instead of choosing the single most probable word to generate, we first truncate the distribution to the top k most likely words, renormalize to produce a legitimate probability distribution, and then randomly sample from within these k words according to their renormalized probabilities. More formally:

1. Choose in advance a number of words k
2. For each word in the vocabulary V, use the language model to compute the likelihood of this word given the context $p(w_t|\mathbf{w}_{<t})$
3. Sort the words by their likelihood, and throw away any word that is not one of the top k most probable words.

4. Renormalize the scores of the k words to be a legitimate probability distribution.
5. Randomly sample a word from within these remaining k most-probable words according to its probability.

When $k = 1$, top-k sampling is identical to greedy decoding. Setting k to a larger number than 1 leads us to sometimes select a word which is not necessarily the most probable, but is still probable enough, and whose choice results in generating more diverse but still high-enough-quality text.

8.6.2 Nucleus or top-p sampling

One problem with top-k sampling is that k is fixed, but the shape of the probability distribution over words differs in different contexts. If we set $k = 10$, sometimes the top 10 words will be very likely and include most of the probability mass, but other times the probability distribution will be flatter and the top 10 words will only include a small part of the probability mass.

top-p sampling

An alternative, called **top-p sampling** or **nucleus sampling** (Holtzman et al., 2020), is to keep not the top k words, but the top p percent of the probability mass. The goal is the same; to truncate the distribution to remove the very unlikely words. But by measuring probability rather than the number of words, the hope is that the measure will be more robust in very different contexts, dynamically increasing and decreasing the pool of word candidates.

Given a distribution $P(w_t|\mathbf{w}_{<t})$, we sort the distribution from most probable, and then the top-p vocabulary $V^{(p)}$ is the smallest set of words such that

$$\sum_{w \in V^{(p)}} P(w|\mathbf{w}_{<t}) \geq p. \tag{8.48}$$

8.7 Training

We described the training process for language models in the prior chapter. Recall that large language models are trained with cross-entropy loss, also called the negative log likelihood loss. At time t the cross-entropy loss is the negative log probability the model assigns to the next word in the training sequence, $-\log p(w_{t+1})$.

Fig. 8.17 illustrates the general training approach. At each step, given all the preceding words, the final transformer layer produces an output distribution over the entire vocabulary. During training, the probability assigned to the correct word by the model is used to calculate the cross-entropy loss for each item in the sequence. The loss for a training sequence is the average cross-entropy loss over the entire sequence. The weights in the network are adjusted to minimize the average CE loss over the training sequence via gradient descent.

With transformers, each training item can be processed in parallel since the output for each element in the sequence is computed separately.

Large models are generally trained by filling the full context window (for example 4096 tokens for GPT4 or 8192 for Llama 3) with text. If documents are shorter than this, multiple documents are packed into the window with a special end-of-text token between them. The batch size for gradient descent is usually quite large (the largest GPT-3 model uses a batch size of 3.2 million tokens).

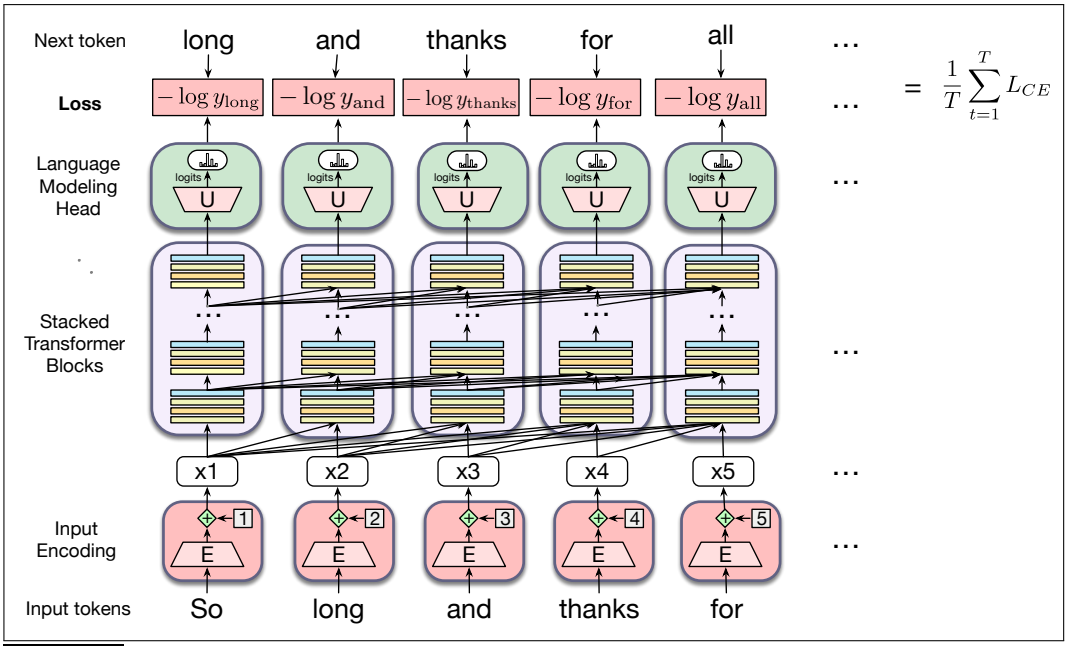


Figure 8.17 Training a transformer as a language model.

8.8 Dealing with Scale

Large language models are large. For example the *Llama 3.1 405B Instruct* model from Meta has 405 billion parameters (it has L=126 layers, model dimensionality d=16,384, and A=128 attention heads) and was trained on 15.6 terabytes of text tokens using a vocabulary of 128K tokens (Llama Team, 2024). So there is a lot of research on understanding how LLMs scale, and especially how to implement them given limited resources. In the next few sections we discuss how to think about scale (the concept of **scaling laws**), and important techniques for getting language models to work efficiently, such as the **KV cache** and parameter-efficient fine tuning (PEFT).

8.8.1 Scaling laws

The performance of large language models has shown to be mainly determined by 3 factors: model size (the number of parameters not counting embeddings), dataset size (the amount of training data), and the amount of compute used for training. That is, we can improve a model by adding parameters (adding more layers or having wider contexts or both), by training on more data, or by training for more iterations.

scaling laws

The relationships between these factors and performance are known as **scaling laws**. Roughly speaking, the performance of a large language model (the loss) scales as a power-law with each of these three properties of model training.

For example, Kaplan et al. (2020) found the following three relationships for loss L as a function of the number of non-embedding parameters N, the dataset size D, and the compute budget C, for models training with limited parameters, dataset,

or compute budget, if in each case the other two properties are held constant:

$$L(N) = \left(\frac{N_c}{N}\right)^{\alpha_N} \tag{8.49}$$

$$L(D) = \left(\frac{D_c}{D}\right)^{\alpha_D} \tag{8.50}$$

$$L(C) = \left(\frac{C_c}{C}\right)^{\alpha_C} \tag{8.51}$$

The number of (non-embedding) parameters N can be roughly computed as follows (ignoring biases, and with d as the input and output dimensionality of the model, d_{attn} as the self-attention layer size, and d_{ff} the size of the feedforward layer):

$$\begin{aligned} N &\approx 2\, d\, n_{\text{layer}}(2\, d_{\text{attn}} + d_{\text{ff}}) \\ &\approx 12\, n_{\text{layer}}\, d^2 \\ &\quad (\text{assuming } d_{\text{attn}} = d_{\text{ff}}/4 = d) \end{aligned} \tag{8.52}$$

Thus GPT-3, with $n = 96$ layers and dimensionality $d = 12288$, has $12 \times 96 \times 12288^2 \approx 175$ billion parameters.

The values of N_c, D_c, C_c, α_N, α_D, and α_C depend on the exact transformer architecture, tokenization, and vocabulary size, so rather than all the precise values, scaling laws focus on the relationship with loss.[3]

Scaling laws can be useful in deciding how to train a model to a particular performance, for example by looking at early in the training curve, or performance with smaller amounts of data, to predict what the loss would be if we were to add more data or increase model size. Other aspects of scaling laws can also tell us how much data we need to add when scaling up a model.

8.8.2 KV Cache

We saw in Fig. 8.11 and in Eq. 8.34 (repeated below) how the attention vector can be very efficiently computed in parallel for training, via two matrix multiplications:

$$\mathbf{A} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\intercal}{\sqrt{d_k}}\right)\mathbf{V} \tag{8.53}$$

Unfortunately we can't do quite the same efficient computation in inference as in training. That's because at inference time, we iteratively generate the next tokens one at a time. For a new token that we have just generated, call it $\mathbf{x}_i$, we need to compute its query, key, and values by multiplying by $\mathbf{W^Q}$, $\mathbf{W^K}$, and $\mathbf{W^V}$ respectively. But it would be a waste of computation time to recompute the key and value vectors for all the **prior** tokens $\mathbf{x}_{<i}$; at prior steps we already computed these key and value vectors! So instead of recomputing these, whenever we compute the key and value vectors we store them in memory in the **KV cache**, and then we can just grab them from the cache when we need them. Fig. 8.18 modifies Fig. 8.11 to show the computation that takes place for a single new token, showing which values we can take from the cache rather than recompute.

KV cache

[3] For the initial experiment in Kaplan et al. (2020) the precise values were $\alpha_N = 0.076$, $N_c = 8.8 \times 10^{13}$ (parameters), $\alpha_D = 0.095$, $D_c = 5.4 \times 10^{13}$ (tokens), $\alpha_C = 0.050$, $C_c = 3.1 \times 10^8$ (petaflop-days).

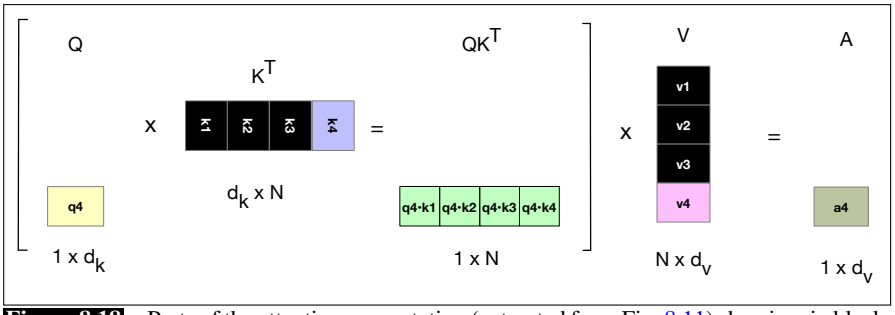


Figure 8.18 Parts of the attention computation (extracted from Fig. 8.11) showing, in black, the vectors that can be stored in the cache rather than recomputed when computing the attention score for the 4th token.

8.8.3 Parameter Efficient Fine Tuning

As we mentioned above, it's very common to take a language model and give it more information about a new domain by **finetuning** it (continuing to train it to predict upcoming words) on some additional data.

Fine-tuning can be very difficult with very large language models, because there are enormous numbers of parameters to train; each pass of batch gradient descent has to backpropagate through many many huge layers. This makes finetuning huge language models extremely expensive in processing power, in memory, and in time. For this reason, there are alternative methods that allow a model to be finetuned without changing all the parameters. Such methods are called **parameter-efficient fine tuning** or sometimes **PEFT**, because we efficiently select a subset of parameters to update when finetuning. For example we freeze some of the parameters (don't change them), and only update some particular subset of parameters.

parameter-efficient fine tuning
PEFT

LoRA

Here we describe one such model, called **LoRA**, for **Lo**w-**R**ank **A**daptation. The intuition of LoRA is that transformers have many dense layers which perform matrix multiplication (for example the $\mathbf{W^Q}$, $\mathbf{W^K}$, $\mathbf{W^V}$, $\mathbf{W^O}$ layers in the attention computation). Instead of updating these layers during finetuning, with LoRA we freeze these layers and instead update a low-rank approximation that has fewer parameters.

Consider a matrix $\mathbf{W}$ of dimensionality $[k \times d]$ that needs to be updated during finetuning via gradient descent. Normally this matrix would get updates $\Delta\mathbf{W}$ of dimensionality $[k \times d]$, for updating the $k \times d$ parameters after gradient descent. In LoRA, we freeze $\mathbf{W}$ and update instead a low-rank decomposition of $\mathbf{W}$. We create two matrices $\mathbf{A}$ and $\mathbf{B}$, where $\mathbf{A}$ has size $[k \times r]$ and $\mathbf{B}$ has size $[r \times d]$, and we choose r to be quite small, $r << \min(d,k)$. During finetuning we update $\mathbf{A}$ and $\mathbf{B}$ instead of $\mathbf{W}$. That is, we replace $\mathbf{W} + \Delta\mathbf{W}$ with $\mathbf{W} + \mathbf{AB}$. Fig. 8.19 shows the intuition. For replacing the forward pass $\mathbf{h} = \mathbf{xW}$, the new forward pass is instead:

$$\mathbf{h} = \mathbf{xW} + \mathbf{xAB} \tag{8.54}$$

LoRA has a number of advantages. It dramatically reduces hardware requirements, since gradients don't have to be calculated for most parameters. The weight updates can be simply added in to the pretrained weights, since $\mathbf{AB}$ is the same size as $\mathbf{W}$). That means it doesn't add any time during inference. And it also means it's possible to build LoRA modules for different domains and just swap them in and out by adding them in or subtracting them from $\mathbf{W}$.

In its original version LoRA was applied just to the matrices in the attention computation (the $\mathbf{W^Q}$, $\mathbf{W^K}$, $\mathbf{W^V}$, and $\mathbf{W^O}$ layers). Many variants of LoRA exist.

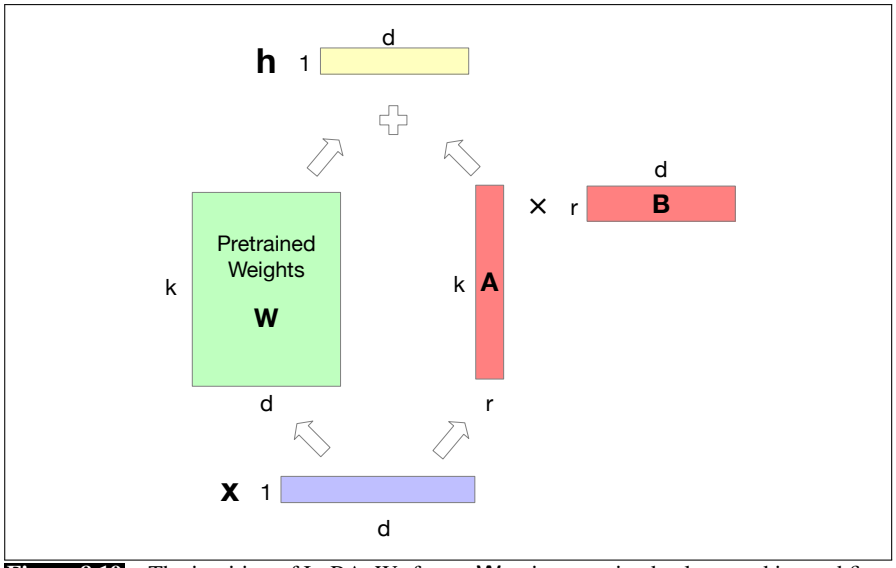


Figure 8.19 The intuition of LoRA. We freeze **W** to its pretrained values, and instead fine-tune by training a pair of matrices **A** and **B**, updating those instead of **W**, and just sum **W** and the updated **AB**.

8.9 Interpreting the Transformer

interpretability

How does a transformer-based language model manage to do so well at language tasks? The subfield of **interpretability**, sometimes called **mechanistic interpretability**, focuses on ways to understand mechanistically what is going on inside the transformer. In the next two subsections we discuss two well-studied aspects of transformer interpretability.

8.9.1 In-Context Learning and Induction Heads

As a way of getting a model to do what we want, we can think of prompting as being fundamentally different than pretraining. Learning via pretraining means updating the model's parameters by using gradient descent according to some loss function. But prompting with demonstrations can teach a model to do a new task. The model is learning something about the task from those demonstrations as it processes the prompt.

Even without demonstrations, we can think of the process of prompting as a kind of learning. For example, the further a model gets in a prompt, the better it tends to get at predicting the upcoming tokens. The information in the context is helping give the model more predictive power.

in-context learning

The term **in-context learning** was first proposed by Brown et al. (2020) in their introduction of the GPT3 system, to refer to either of these kinds of learning that language models do from their prompts. In-context learning means language models learning to do new tasks, better predict tokens, or generally reduce their loss during the forward-pass at inference-time, without any gradient-based updates to the model's parameters.

induction heads

How does in-context learning work? While we don't know for sure, there are some intriguing ideas. One hypothesis is based on the idea of **induction heads** (Elhage et al., 2021; Olsson et al., 2022). Induction heads are the name for a **circuit**,

which is a kind of abstract component of a network. The induction head circuit is part of the attention computation in transformers, discovered by looking at mini language models with only 1-2 attention heads.

The function of the induction head is to predict repeated sequences. For example if it sees the pattern AB...A in an input sequence, it predicts that B will follow, instantiating the **pattern completion** rule AB...A→ B. It does this by having a *prefix matching* component of the attention computation that, when looking at the current token A, searches back over the context to find a prior instance of A. If it finds one, the induction head has a *copying* mechanism that "copies" the token B that followed the earlier A, by increasing the probability the B will occur next. Fig. 8.20 shows an example.

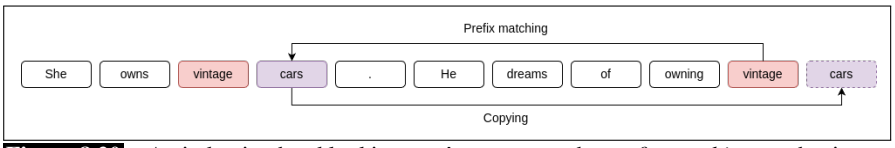


Figure 8.20 An induction head looking at `vintage` uses the *prefix matching* mechanism to find a prior instance of `vintage`, and the *copying* mechanism to predict that `cars` will occur again. Figure from Crosbie and Shutova (2022).

Olsson et al. (2022) propose that a generalized fuzzy version of this pattern completion rule, implementing a rule like A*B*...A→ B, where A* ≈ A and B* ≈ B (by ≈ we mean they are semantically similar in some way), might be responsible for in-context learning. Suggestive evidence for their hypothesis comes from Crosbie and Shutova (2022), who show that **ablating** induction heads causes in-context learning performance to decrease. **Ablation** is originally a medical term meaning the removal of something. We use it in NLP interpretability studies as a tool for testing causal effects; if we knock out a hypothesized cause, we would expect the effect to disappear. Crosbie and Shutova (2022) ablate induction heads by first finding attention heads that perform as induction heads on random input sequences, and then zeroing out the output of these heads by setting certain terms of the output matrix $\mathbf{W^O}$ to zero. Indeed they find that ablated models are much worse at in-context learning: they have much worse performance at learning from demonstrations in the prompts.

ablating

8.9.2 Logit Lens

logit lens

Another useful interpretability tool, the **logit lens** (Nostalgebraist, 2020), offers a way to visualize what the internal layers of the transformer might be representing.

The idea is that we take any vector from any layer of the transformer and, pretending that it is the prefinal embedding, simply multiply it by the **unembedding layer** to get logits, and compute a softmax to see the distribution over words that that vector might be representing. This can be a useful window into the internal representations of the model. Since the network wasn't trained to make the internal representations function in this way, the logit lens doesn't always work perfectly, but this can still be a useful trick to help us visualize the internal layers of a transformer.

8.10 Summary

This chapter has introduced the transformer and its components for the language modeling task introduced in the previous chapter. Here's a summary of the main points that we covered:

- Transformers are non-recurrent networks based on **multi-head attention**, a kind of **self-attention**. A multi-head attention computation takes an input vector $\mathbf{x}_i$ and maps it to an output $\mathbf{a}_i$ by adding in vectors from prior tokens, weighted by how relevant they are for the processing of the current word.
- A **transformer block** consists of a **residual stream** in which the input from the prior layer is passed up to the next layer, with the output of different components added to it. These components include a **multi-head attention layer** followed by a **feedforward layer**, each preceded by **layer normalizations**. Transformer blocks are stacked to make deeper and more powerful networks.
- The input to a transformer is computed by adding an embedding (computed with an **embedding matrix**) to a **positional encoding** that represents the sequential position of the token in the window.
- Language models can be built out of stacks of transformer blocks, with a **language model head** at the top, which applies an **unembedding** matrix to the output $\mathbf{H}$ of the top layer to generate the **logits**, which are then passed through a softmax to generate word probabilities.
- Transformer-based language models have a wide context window (200K tokens or even more for very large models with special mechanisms) allowing them to draw on enormous amounts of context to predict upcoming words.
- There are various computational tricks for making large language models more efficient, such as the **KV cache** and **parameter-efficient finetuning**.

Historical Notes

The transformer (Vaswani et al., 2017) was developed drawing on two lines of prior research: **self-attention** and **memory networks**.

Encoder-decoder attention, the idea of using a soft weighting over the encodings of input words to inform a generative decoder (see Chapter 12) was developed by Graves (2013) in the context of handwriting generation, and Bahdanau et al. (2015) for MT. This idea was extended to self-attention by dropping the need for separate encoding and decoding sequences and instead seeing attention as a way of weighting the tokens in collecting information passed from lower layers to higher layers (Ling et al., 2015; Cheng et al., 2016; Liu et al., 2016).

Other aspects of the transformer, including the terminology of key, query, and value, came from **memory networks**, a mechanism for adding an external read-write memory to networks, by using an embedding of a query to match keys representing content in an associative memory (Sukhbaatar et al., 2015; Weston et al., 2015; Graves et al., 2014).

MORE HISTORY TBD IN NEXT DRAFT.

CHAPTER

9 Masked Language Models

Larvatus prodeo [Masked, I go forward]
Descartes

In the previous two chapters we introduced the transformer and saw how to pretrain a transformer language model as a **causal** or left-to-right language model. In this chapter we'll introduce a second paradigm for pretrained language models, the **bidirectional transformer** encoder, and the most widely-used version, the **BERT** model (Devlin et al., 2019). This model is trained via **masked language modeling**, where instead of predicting the following word, we mask a word in the middle and ask the model to guess the word given the words on both sides. This method thus allows the model to see both the right and left context.

BERT
masked language modeling

We also introduced **finetuning** in the prior chapter. Here we describe a new kind of finetuning, in which we take the transformer network learned by these pretrained models, add a neural net classifier after the top layer of the network, and train it on some additional labeled data to perform some downstream task like named entity tagging or natural language inference. As before, the intuition is that the pretraining phase learns a language model that instantiates rich representations of word meaning, that thus enables the model to more easily learn ('be finetuned to') the requirements of a downstream language understanding task. This aspect of the pretrain-finetune paradigm is an instance of what is called **transfer learning** in machine learning: the method of acquiring knowledge from one task or domain, and then applying it (transferring it) to solve a new task.

finetuning

transfer learning

The second idea that we introduce in this chapter is the idea of **contextual embeddings**: representations for words in context. The methods of Chapter 5 like word2vec or GloVe learned a single vector embedding for each unique word w in the vocabulary. By contrast, with contextual embeddings, such as those learned by masked language models like BERT, each word w will be represented by a different vector each time it appears in a different context. While the causal language models of Chapter 8 also use contextual embeddings, the embeddings created by masked language models seem to function particularly well as representations.

9.1 Bidirectional Transformer Encoders

Let's begin by introducing the bidirectional transformer encoder that underlies models like BERT and its descendants like **RoBERTa** (Liu et al., 2019) or **SpanBERT** (Joshi et al., 2020). In Chapter 7 we introduced the idea of left-to-right language models that can be applied to autoregressive contextual generation problems like question answering or summarization, and in Chapter 8 we saw how to implement language models with causal (left-to-right) transformers. But this left-to-right nature of these models is also a limitation, because there are tasks for which it would be useful, when processing a token, to be able to peek at future tokens. This is espe-

cially true for **sequence labeling** tasks in which we want to tag each token with a label, such as the **named entity tagging** task we'll introduce in Section 9.5, or tasks like part-of-speech tagging or parsing that come up in later chapters.

The **bidirectional** encoders that we introduce here are a different kind of beast than causal models. The causal models of Chapter 8 are generative models, designed to easily generate the next token in a sequence. But the focus of bidirectional encoders is instead on computing contextualized representations of the input tokens. Bidirectional encoders use self-attention to map sequences of input embeddings $(\mathbf{x}_1, ..., \mathbf{x}_n)$ to sequences of output embeddings of the same length $(\mathbf{h}_1, ..., \mathbf{h}_n)$, where the output vectors have been contextualized using information from the entire input sequence. These output embeddings are contextualized representations of each input token that are useful across a range of applications where we need to do a classification or a decision based on the token in context.

Remember that we said the models of Chapter 8 are sometimes called **decoder-only**, because they correspond to the decoder part of the encoder-decoder model we will introduce in Chapter 12. By contrast, the masked language models of this chapter are sometimes called **encoder-only**, because they produce an encoding for each input token but generally aren't used to produce running text by decoding/sampling. That's an important point: masked language models are not used for generation. They are generally instead used for interpretative tasks.

9.1.1 The architecture for bidirectional masked models

Let's first discuss the overall architecture. Bidirectional transformer-based language models differ in two ways from the causal transformers in the previous chapters. The first is that the attention function isn't causal; the attention for a token i can look at following tokens $i+1$ and so on. The second is that the training is slightly different since we are predicting something in the middle of our text rather than at the end. We'll discuss the first here and the second in the following section.

Fig. 9.1a, reproduced here from Chapter 8, shows the information flow in the left-to-right approach of Chapter 8. The attention computation at each token is based on the preceding (and current) input tokens, ignoring potentially useful information located to the right of the token under consideration. Bidirectional encoders overcome this limitation by allowing the attention mechanism to range over the entire input, as shown in Fig. 9.1b.

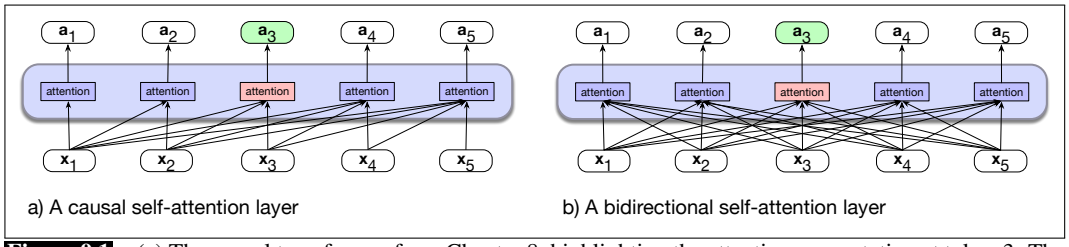


Figure 9.1 (a) The causal transformer from Chapter 8, highlighting the attention computation at token 3. The attention value at each token is computed using only information seen earlier in the context. (b) Information flow in a bidirectional attention model. In processing each token, the model attends to all inputs, both before and after the current one. So attention for token 3 can draw on information from following tokens.

The implementation is very simple! We simply remove the attention masking step that we introduced in Eq. 8.34. Recall from Chapter 8 that we had to mask the $\mathbf{QK}^\intercal$ matrix for causal transformers so that attention couldn't look at future tokens

(repeated from Eq. 8.34 for a single attention head):

$$\textbf{head} = \text{softmax}\left(\text{mask}\left(\frac{\mathbf{QK}^\intercal}{\sqrt{d_k}}\right)\right)\mathbf{V} \tag{9.1}$$

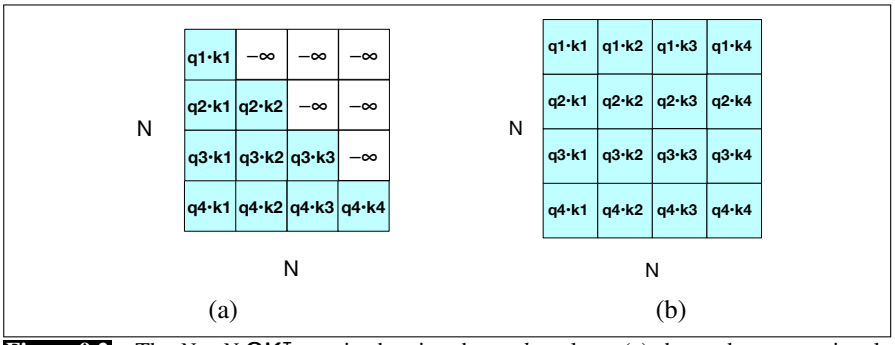


Figure 9.2 The $N \times N$ $\mathbf{QK}^\intercal$ matrix showing the $q_i \cdot k_j$ values. (a) shows the upper-triangle portion of the comparisons matrix zeroed out (set to $-\infty$, which the softmax will turn to zero), while (b) shows the unmasked version.

Fig. 9.2 shows the masked version of $\mathbf{QK}^\intercal$ and the unmasked version. For bidirectional attention, we use the unmasked version of Fig. 9.2b. Thus the attention computation for bidirectional attention is exactly the same as Eq. 9.1 but with the mask removed:

$$\textbf{head} = \text{softmax}\left(\frac{\mathbf{QK}^\intercal}{\sqrt{d_k}}\right)\mathbf{V} \tag{9.2}$$

Otherwise, the attention computation is identical to what we saw in Chapter 8, as is the transformer block architecture (the feedforward layer, layer norm, and so on). As in Chapter 8, the input is also a series of subword tokens, usually computed by one of the 3 popular tokenization algorithms (including the BPE algorithm that we already saw in Chapter 2 and two others, the WordPiece algorithm and the SentencePiece Unigram LM algorithm). That means every input sentence first has to be tokenized, and all further processing takes place on subword tokens rather than words. This will require, as we'll see in the third part of the textbook, that for some NLP tasks that require notions of words (like parsing) we will occasionally need to map subwords back to words.

To make this more concrete, the original English-only bidirectional transformer encoder model, BERT (Devlin et al., 2019), consisted of the following:

- An English-only subword vocabulary consisting of 30,000 tokens generated using the WordPiece algorithm (Schuster and Nakajima, 2012).
- Input context window N=512 tokens, and model dimensionality d=768
- So $\mathbf{X}$, the input to the model, is of shape $[N \times d] = [512 \times 768]$.
- L=12 layers of transformer blocks, each with A=12 (bidirectional) multihead attention layers.
- The resulting model has about 100M parameters.

The larger multilingual XLM-RoBERTa model, trained on 100 languages, has

- A multilingual subword vocabulary with 250,000 tokens generated using the SentencePiece Unigram LM algorithm (Kudo and Richardson, 2018b).

- Input context window N=512 tokens, and model dimensionality d=1024, hence $\mathbf{X}$, the input to the model, is of shape $[N \times d] = [512 \times 1024]$.
- L=24 layers of transformer blocks, with A=16 multihead attention layers each
- The resulting model has about 550M parameters.

Note that 550M parameters is relatively small as large language models go (Llama 3 has 405B parameters, so is 3 orders of magnitude bigger). Indeed, masked language models tend to be much smaller than causal language models.

9.2 Training Bidirectional Encoders

We trained causal transformer language models in Chapter 8 by making them iteratively predict the next word in a text. But eliminating the causal mask in attention makes the guess-the-next-word language modeling task trivial—the answer is directly available from the context—so we're in need of a new training scheme. Instead of trying to predict the next word, the model learns to perform a fill-in-the-blank task, technically called the **cloze task** (Taylor, 1953). To see this, let's return to the motivating example from Chapter 3. Instead of predicting which words are likely to come next in this example:

cloze task

`The water of Walden Pond is so beautifully ____`

we're asked to predict a missing item given the rest of the sentence.

`The ____ of Walden Pond is so beautifully ...`

That is, given an input sequence with one or more elements missing, the learning task is to predict the missing elements. More precisely, during training the model is deprived of one or more tokens of an input sequence and must generate a probability distribution over the vocabulary for each of the missing items. We then use the cross-entropy loss from each of the model's predictions to drive the learning process.

This approach can be generalized to any of a variety of methods that corrupt the training input and then asks the model to recover the original input. Examples of the kinds of manipulations that have been used include masks, substitutions, reorderings, deletions, and extraneous insertions into the training text. The general name for this kind of training is called **denoising**: we corrupt (add noise to) the input in some way (by masking a word, or putting in an incorrect word) and the goal of the system is to remove the noise.

denoising

9.2.1 Masking Words

Masked Language Modeling

Let's describe the **Masked Language Modeling** (MLM) approach to training bidirectional encoders (Devlin et al., 2019). As with the language model training methods we've already seen, **MLM** uses unannotated text from a large corpus. In MLM training, the model is presented with a series of sentences from the training corpus in which a percentage of tokens (15% in the BERT model) have been randomly chosen to be manipulated by the masking procedure. Given an input sentence `lunch was delicious` and assume we randomly chose the 3rd token `delicious` to be manipulated,

- 80% of the time: The token is replaced with the special vocabulary token named `[MASK]`, e.g. `lunch was delicious` → `lunch was [MASK]`.

- 10% of the time: The token is replaced with another token, randomly sampled from the vocabulary based on token unigram probabilities. e.g. `lunch was delicious` → `lunch was gasp`.
- 10% of the time: the token is left unchanged. e.g. `lunch was delicious` → `lunch was delicious`.

We then train the model to guess the correct token for the manipulated tokens. Why the three possible manipulations? Adding the `[MASK]` token creates a mismatch between pretraining and downstream finetuning or inference, since when we employ the MLM model to perform a downstream task, we don't use any `[MASK]` tokens. If we just replaced tokens with the `[MASK]`, the model might only predict tokens when it sees a `[MASK]`, but we want the model to try to always predict the input token.

To train the model to make the prediction, the original input sequence is tokenized using a subword model and tokens are sampled to be manipulated. Word embeddings for all of the tokens in the input are retrieved from the **E** embedding matrix and combined with positional embeddings to form the input to the transformer, passed through the stack of bidirectional transformer blocks, and then the language modeling head. The MLM training objective is to predict the original inputs for each of the masked tokens and the cross-entropy loss from these predictions drives the training process for all the parameters in the model. That is, all of the input tokens play a role in the self-attention process, but only the sampled tokens are used for learning.

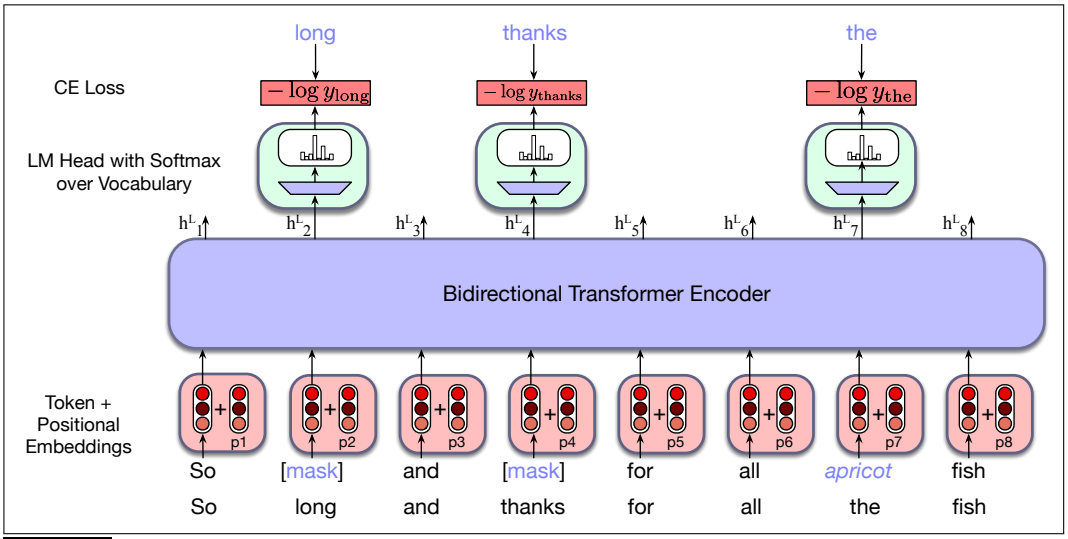


Figure 9.3 Masked language model training. In this example, three of the input tokens are selected, two of which are masked and the third is replaced with an unrelated word. The probabilities assigned by the model to these three items are used as the training loss. The other 5 tokens don't play a role in training loss.

Fig. 9.3 illustrates this approach with a simple example. Here, *long*, *thanks* and *the* have been sampled from the training sequence, with the first two masked and *the* replaced with the randomly sampled token *apricot*. The resulting embeddings are passed through a stack of bidirectional transformer blocks. Recall from Section 8.5 in Chapter 8 that to produce a probability distribution over the vocabulary for each of the masked tokens, the **language modeling head** takes the output vector $\mathbf{h}_i^L$ from the final transformer layer L for each masked token i, multiplies it by the unembedding layer $\mathbf{E}^T$ to produce the logits $\mathbf{u}$, and then uses softmax to turn the logits into

probabilities $\mathbf{y}$ over the vocabulary:

$$\mathbf{u}_i = \mathbf{h}_\mathbf{i}^\mathbf{L}\ \mathbf{E}^\mathbf{T} \tag{9.3}$$

$$\mathbf{y}_i = \text{softmax}(\mathbf{u}_\mathbf{i}) \tag{9.4}$$

With a predicted probability distribution for each masked item, we can use cross-entropy to compute the loss for each masked item—the negative log probability assigned to the actual masked word, as shown in Fig. 9.3. More formally, for a given vector of input tokens in a sentence or batch $\mathbf{x}$, let the set of tokens that are masked be M, the version of that sentence with some tokens replaced by masks be $\mathbf{x}^{mask}$, and the sequence of output vectors be $\mathbf{h}$. For a given input token x_i, such as the word *long* in Fig. 9.3, the loss is the probability of the correct word *long*, given $\mathbf{x}^{mask}$ (as summarized in the single output vector $\mathbf{h}_i^L$):

$$L_{MLM}(x_i) = -\log P(x_i|\mathbf{h}_i^L)$$

The gradients that form the basis for the weight updates are based on the average loss over the sampled learning items from a single training sequence (or batch of sequences).

$$L_{MLM} = -\frac{1}{|M|}\sum_{i\in M}\log P(x_i|\mathbf{h}_i^L)$$

Note that only the tokens in M play a role in learning; the other words play no role in the loss function, so in that sense BERT and its descendents are inefficient; only 15% of the input samples in the training data are actually used for training weights.[1]

9.2.2 Next Sentence Prediction

The focus of mask-based learning is on predicting words from surrounding contexts with the goal of producing effective word-level representations. However, an important class of applications involves determining the relationship between pairs of sentences. These include tasks like paraphrase detection (detecting if two sentences have similar meanings), entailment (detecting if the meanings of two sentences entail or contradict each other) or discourse coherence (deciding if two neighboring sentences form a coherent discourse).

Next Sentence Prediction

To capture the kind of knowledge required for applications such as these, some models in the BERT family include a second learning objective called **Next Sentence Prediction** (NSP). In this task, the model is presented with pairs of sentences and is asked to predict whether each pair consists of an actual pair of adjacent sentences from the training corpus or a pair of unrelated sentences. In BERT, 50% of the training pairs consisted of positive pairs, and in the other 50% the second sentence of a pair was randomly selected from elsewhere in the corpus. The NSP loss is based on how well the model can distinguish true pairs from random pairs.

To facilitate NSP training, BERT introduces two special tokens to the input representation (tokens that will prove useful for finetuning as well). After tokenizing the input with the subword model, the token `[CLS]` is prepended to the input sentence pair, and the token `[SEP]` is placed between the sentences and after the final token of the second sentence. There are actually two more special tokens, a 'First Segment' token, and a 'Second Segment' token. These tokens are added in the input stage to the word and positional embeddings. That is, each token of the input

[1] ELECTRA, another BERT family member, does use all examples for training (Clark et al., 2020b).

$\mathbf{X}$ is actually formed by summing 3 embeddings: word, position, and first/second segment embeddings.

During training, the output vector h^L_{CLS} from the final layer associated with the [CLS] token represents the next sentence prediction. As with the MLM objective, we add a special head, in this case an NSP head, which consists of a learned set of classification weights $\mathbf{W_{NSP}} \in \mathbb{R}^{d\times 2}$ that produces a two-class prediction from the raw [CLS] vector h^L_{CLS}:

$$\mathbf{y}_i = \text{softmax}(\mathbf{h}^L_{\text{CLS}}\mathbf{W_{NSP}})$$

Cross entropy is used to compute the NSP loss for each sentence pair presented to the model. Fig. 9.4 illustrates the overall NSP training setup. In BERT, the NSP loss was used in conjunction with the MLM training objective to form final loss.

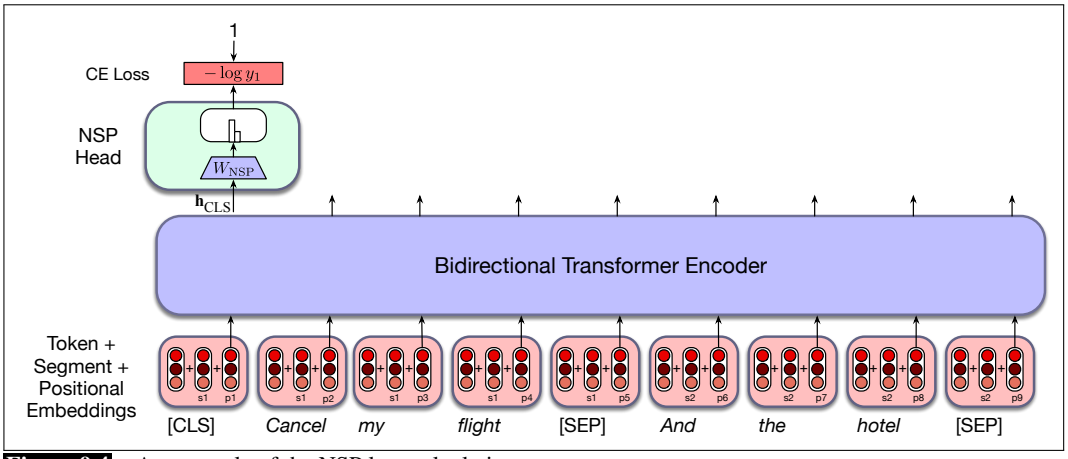


Figure 9.4 An example of the NSP loss calculation.

9.2.3 Training Regimes

BERT and other early transformer-based language models were trained on about 3.3 billion words (a combination of English Wikipedia and a corpus of book texts called BooksCorpus (Zhu et al., 2015) that is no longer used for intellectual property reasons). Modern masked language models are now trained on much larger datasets of web text, filtered a bit, and augmented by higher-quality data like Wikipedia, the same as those we discussed for the causal large language models of Chapter 8. Multilingual models similarly use webtext and multilingual Wikipedia. For example the XLM-R model was trained on about 300 billion tokens in 100 languages, taken from the web via Common Crawl (https://commoncrawl.org/).

To train the original BERT models, pairs of text segments were selected from the training corpus according to the next sentence prediction 50/50 scheme. Pairs were sampled so that their combined length was less than the 512 token input. Tokens within these sentence pairs were then masked using the MLM approach with the combined loss from the MLM and NSP objectives used for a final loss. Because this final loss is backpropagated through the entire transformer, the embeddings at each transformer layer will learn representations that are useful for predicting words from their neighbors. Since the [CLS] tokens are the direct input to the NSP classifier, their learned representations will tend to contain information about the sequence as

a whole. Approximately 40 passes (epochs) over the training data was required for the model to converge.

Some models, like the RoBERTa model, drop the next sentence prediction objective, and therefore change the training regime a bit. Instead of sampling pairs of sentence, the input is simply a series of contiguous sentences, still beginning with the special `[CLS]` token. If the document runs out before 512 tokens are reached, an extra separator token is added, and sentences from the next document are packed in, until we reach a total of 512 tokens. Usually large batch sizes are used, between 8K and 32K tokens.

Multilingual models have an additional decision to make: what data to use to build the vocabulary? Recall that all language models use subword tokenization (BPE or SentencePiece Unigram LM are the two most common algorithms). What text should be used to learn this multilingual tokenization, given that it's easier to get much more text in some languages than others? One option would be to create this vocabulary-learning dataset by sampling sentences from our training data (perhaps web text from Common Crawl), randomly. In that case we will choose a lot of sentences from languages with lots of web representation like English, and the tokens will be biased toward rare English tokens instead of creating frequent tokens from languages with less data. Instead, it is common to divide the training data into subcorpora of N different languages, compute the number of sentences n_i of each language i, and readjust these probabilities so as to upweight the probability of less-represented languages (Lample and Conneau, 2019). The new probability of selecting a sentence from each of the N languages (whose prior frequency is n_i) is $\{q_i\}_{i=1...N}$, where:

$$q_i = \frac{p_i^\alpha}{\sum_{j=1}^{N} p_j^\alpha} \quad \text{with} \quad p_i = \frac{n_i}{\sum_{k=1}^{N} n_k} \tag{9.5}$$

Recall from Eq. 5.19 in Chapter 5 that an α value between 0 and 1 will give higher weight to lower probability samples. Conneau et al. (2020) show that $\alpha = 0.3$ works well to give rare languages more inclusion in the tokenization, resulting in better multilingual performance overall.

The result of this pretraining process consists of both learned word embeddings, as well as all the parameters of the bidirectional encoder that are used to produce contextual embeddings for novel inputs.

For many purposes, a pretrained multilingual model is more practical than a monolingual model, since it avoids the need to build many (a hundred!) separate monolingual models. And multilingual models can improve performance on low-resourced languages by leveraging linguistic information from a similar language in the training data that happens to have more resources. Nonetheless, when the number of languages grows very large, multilingual models exhibit what has been called the **curse of multilinguality** (Conneau et al., 2020): the performance on each language degrades compared to a model training on fewer languages. Another problem with multilingual models is that they 'have an accent': grammatical structures in higher-resource languages (often English) bleed into lower-resource languages; the vast amount of English language in training makes the model's representations for low-resource languages slightly more English-like (Papadimitriou et al., 2023).

9.3 Contextual Embeddings

contextual embeddings

Given a pretrained language model and a novel input sentence, we can think of the sequence of model outputs as constituting **contextual embeddings** for each token in the input. These contextual embeddings are vectors representing some aspect of the meaning of a token in context, and can be used for any task requiring the meaning of tokens or words. More formally, given a sequence of input tokens $x_1, ..., x_n$, we can use the output vector $\mathbf{h}^L_i$ from the final layer L of the model as a representation of the meaning of token x_i in the context of sentence $x_1, ..., x_n$. Or instead of just using the vector $\mathbf{h}^L_i$ from the final layer of the model, it's common to compute a representation for x_i by averaging the output tokens $\mathbf{h}_i$ from each of the last four layers of the model, i.e., $\mathbf{h}^L_i$, $\mathbf{h}^{L-1}_i$, $\mathbf{h}^{L-2}_i$, and $\mathbf{h}^{L-3}_i$.

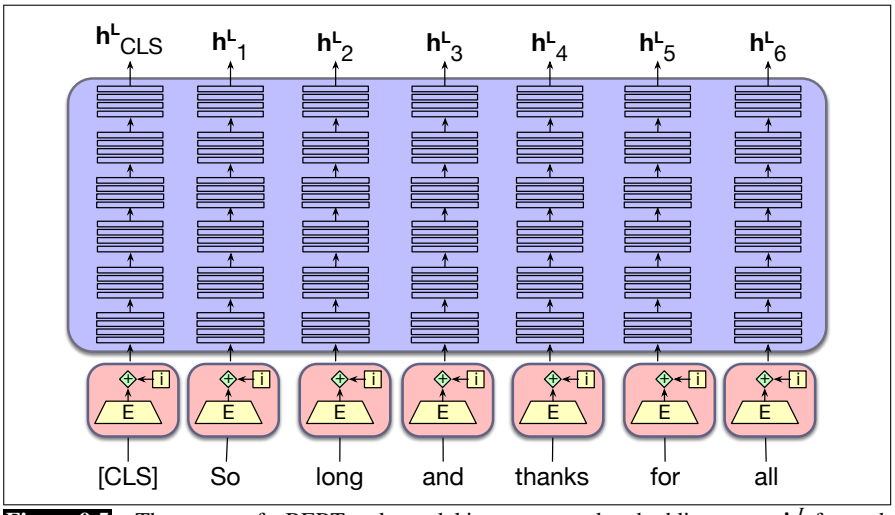


Figure 9.5 The output of a BERT-style model is a contextual embedding vector $\mathbf{h}^L_i$ for each input token x_i.

Just as we used static embeddings like word2vec in Chapter 5 to represent the meaning of words, we can use contextual embeddings as representations of word meanings in context for any task that might require a model of word meaning. Where static embeddings represent the meaning of word *types* (vocabulary entries), contextual embeddings represent the meaning of word *instances*: instances of a particular word type in a particular context. Thus where word2vec had a single vector for each word type, contextual embeddings provide a single vector for each instance of that word type in its sentential context. Contextual embeddings can thus be used for tasks like measuring the semantic similarity of two words in context, and are useful in linguistic tasks that require models of word meaning.

9.3.1 Contextual Embeddings and Word Sense

ambiguous

Words are **ambiguous**: the same word can be used to mean different things. In Chapter 5 we saw that the word "mouse" can mean (1) a small rodent, or (2) a hand-operated device to control a cursor. The word "bank" can mean: (1) a financial institution or (2) a sloping mound. We say that the words 'mouse' or 'bank' are

polysemous (from Greek 'many senses', *poly-* 'many' + *sema*, 'sign, mark').[2]

word sense

A **sense** (or **word sense**) is a discrete representation of one aspect of the meaning of a word. We can represent each sense with a superscript: **bank**^{1} and **bank**^{2}, **mouse**^{1} and **mouse**^{2}. These senses can be found listed in online thesauruses (or thesauri) like **WordNet** (Fellbaum, 1998), which has datasets in many languages listing the senses of many words. In context, it's easy to see the different meanings:

WordNet

mouse^1 : .... a *mouse* controlling a computer system in 1968.
mouse^2 : .... a quiet animal like a *mouse*
bank^1 : ...a *bank* can hold the investments in a custodial account ...
bank^2 : ...as agriculture burgeons on the east *bank*, the river ...

This fact that context disambiguates the senses of *mouse* and *bank* above can also be visualized geometrically. Fig. 9.6 shows a two-dimensional projection of many instances of the BERT embeddings of the word *die* in English and German. Each point in the graph represents the use of *die* in one input sentence. We can clearly see at least two different English senses of *die* (the singular of *dice* and the verb *to die*, as well as the German article, in the BERT embedding space.

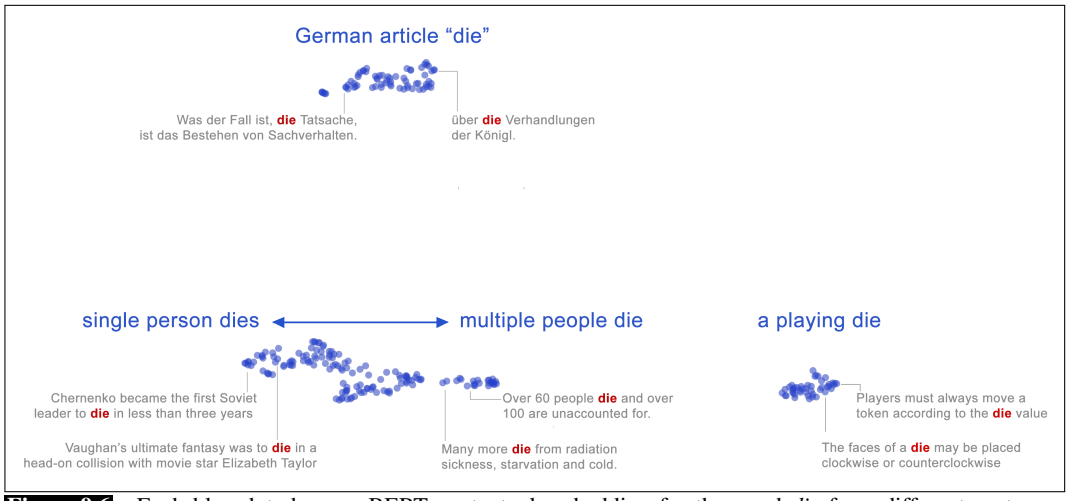


Figure 9.6 Each blue dot shows a BERT contextual embedding for the word *die* from different sentences in English and German, projected into two dimensions with the UMAP algorithm. The German and English meanings and the different English senses fall into different clusters. Some sample points are shown with the contextual sentence they came from. Figure from Coenen et al. (2019).

Thus while thesauruses like WordNet give discrete lists of senses, embeddings (whether static or contextual) offer a continuous high-dimensional model of meaning that, although it can be clustered, doesn't divide up into fully discrete senses.

Word Sense Disambiguation

word sense disambiguation
WSD

The task of selecting the correct sense for a word is called **word sense disambiguation**, or **WSD**. WSD algorithms take as input a word in context and a fixed inventory of potential word senses (like the ones in WordNet) and outputs the correct word sense in context. Fig. 9.7 sketches out the task.

[2] The word **polysemy** itself is ambiguous; you may see it used in a different way, to refer only to cases where a word's senses are related in some structured way, reserving the word **homonymy** to mean sense ambiguities with no relation between the senses (Haber and Poesio, 2020). Here we will use 'polysemy' to mean any kind of sense ambiguity, and 'structured polysemy' for polysemy with sense relations.

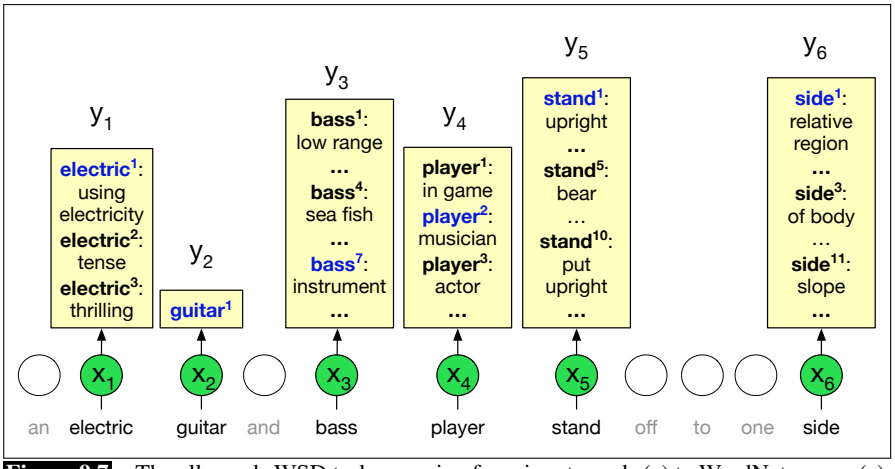


Figure 9.7 The all-words WSD task, mapping from input words (x) to WordNet senses (y). Figure inspired by Chaplot and Salakhutdinov (2018).

WSD can be a useful analytic tool for text analysis in the humanities and social sciences, and word senses can play a role in model interpretability for word representations. Word senses also have interesting distributional properties. For example a word often is used in roughly the same sense through a discourse, an observation called the **one sense per discourse** rule (Gale et al., 1992a).

one sense per discourse

The best performing WSD algorithm is a simple 1-nearest-neighbor algorithm using contextual word embeddings, due to Melamud et al. (2016) and Peters et al. (2018). At training time we pass each sentence in some sense-labeled dataset (like the SemCore or SenseEval datasets in various languages) through any contextual embedding (e.g., BERT) resulting in a contextual embedding for each labeled token. (There are various ways to compute this contextual embedding v_i for a token i; for BERT it is common to pool multiple layers by summing the vector representations of i from the last four BERT layers). Then for each sense s of any word in the corpus, for each of the n tokens of that sense, we average their n contextual representations v_i to produce a contextual **sense embedding** $\mathbf{v}_s$ for s:

$$\mathbf{v}_s = \frac{1}{n}\sum_i \mathbf{v}_i \qquad \forall \mathbf{v}_i \in \text{tokens}(s) \tag{9.6}$$

At test time, given a token of a target word t in context, we compute its contextual embedding $\mathbf{t}$ and choose its nearest neighbor sense from the training set, i.e., the sense whose sense embedding has the highest cosine with $\mathbf{t}$:

$$\text{sense}(t) = \underset{s \in \text{senses}(t)}{\text{argmax}} \ \text{cosine}(\mathbf{t}, \mathbf{v}_s) \tag{9.7}$$

Fig. 9.8 illustrates the model.

9.3.2 Contextual Embeddings and Word Similarity

In Chapter 5 we introduced the idea that we could measure the similarity of two words by considering how close they are geometrically, by using the cosine as a similarity function. The idea of meaning similarity is also clear geometrically in the meaning clusters in Fig. 9.6; the representation of a word which has a particular sense in a context is closer to other instances of the same sense of the word. Thus we

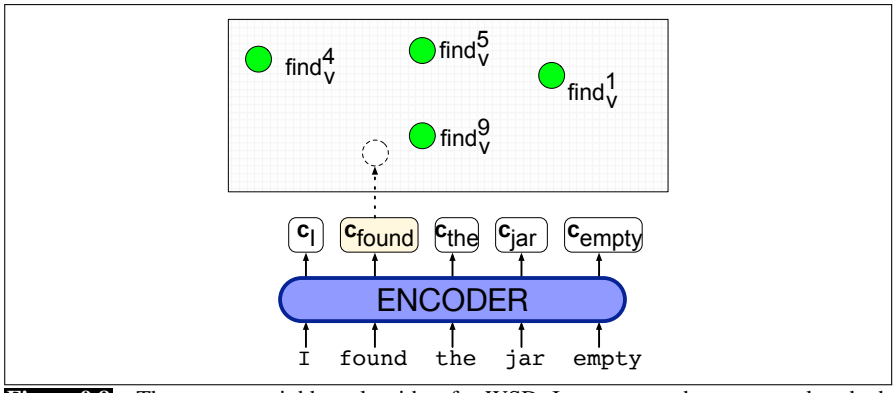


Figure 9.8 The nearest-neighbor algorithm for WSD. In green are the contextual embeddings precomputed for each sense of each word; here we just show a few of the senses for *find*. A contextual embedding is computed for the target word *found*, and then the nearest neighbor sense (in this case $\textbf{find}^9_v$) is chosen. Figure inspired by Loureiro and Jorge (2019).

often measure the similarity between two instances of two words in context (or two instances of the same word in two different contexts) by using the cosine between their contextual embeddings.

Usually some transformations to the embeddings are required before computing cosine. This is because contextual embeddings (whether from masked language models or from autoregressive ones) have the property that the vectors for all words are extremely similar. If we look at the embeddings from the final layer of BERT or other models, embeddings for instances of any two randomly chosen words will have extremely high cosines that can be quite close to 1, meaning all word vectors tend to point in the same direction. The property of vectors in a system all tending to point in the same direction is known as **anisotropy**. Ethayarajh (2019) defines the **anisotropy** of a model as the expected cosine similarity of any pair of words in a corpus. The word 'isotropy' means uniformity in all directions, so in an isotropic model, the collection of vectors should point in all directions and the expected cosine between a pair of random embeddings would be zero. Timkey and van Schijndel (2021) show that one cause of anisotropy is that cosine measures are dominated by a small number of dimensions of the contextual embedding whose values are very different than the others: these **rogue dimensions** have very large magnitudes and very high variance.

anisotropy

Timkey and van Schijndel (2021) shows that we can make the embeddings more isotropic by standardizing (z-scoring) the vectors, i.e., subtracting the mean and dividing by the variance. Given a set C of all the embeddings in some corpus, each with dimensionality d (i.e., $x \in \mathbb{R}^d$), the mean vector $\mu \in \mathbb{R}^d$ is:

$$\mu = \frac{1}{|C|}\sum_{\mathbf{x}\in C}\mathbf{x} \tag{9.8}$$

The standard deviation in each dimension $\sigma \in \mathbb{R}^d$ is:

$$\sigma = \sqrt{\frac{1}{|C|}\sum_{\mathbf{x}\in C}(\mathbf{x}-\mu)^2} \tag{9.9}$$

Then each word vector $\mathbf{x}$ is replaced by a standardized version $\mathbf{z}$:

$$\mathbf{z} = \frac{\mathbf{x}-\mu}{\sigma} \tag{9.10}$$

One problem with cosine that is not solved by standardization is that cosine tends to underestimate human judgments on similarity of word meaning for very frequent words (Zhou et al., 2022).

9.4 Fine-Tuning for Classification

The power of pretrained language models lies in their ability to extract generalizations from large amounts of text—generalizations that are useful for myriad downstream applications. There are two ways to make practical use of the generalizations to solve downstream tasks. The most common way is to use natural language to **prompt** the model, putting it in a state where it contextually generates what we want.

finetuning

In this section we explore an alternative way to use pretrained language models for downstream applications: a version of the **finetuning** paradigm from Chapter 7. In the kind of finetuning used for masked language models, we add application-specific circuitry (often called a special **head**) on top of pretrained models, taking their output as its input. The finetuning process consists of using labeled data about the application to train these additional application-specific parameters. Typically, this training will either freeze or make only minimal adjustments to the pretrained language model parameters.

The following sections introduce finetuning methods for the most common kinds of applications: sequence classification, sentence-pair classification, and sequence labeling.

9.4.1 Sequence Classification

The task of **sequence classification** is to classify an entire sequence of text with a single label. This set of tasks is commonly called **text classification**, like sentiment analysis or spam detection (Appendix K) in which we classify a text into two or three classes (like positive or negative), as well as classification tasks with a large number of categories, like document-level topic classification.

For sequence classification we represent the entire input to be classified by a single vector. We can represent a sequence in various ways. One way is to take the sum or the mean of the last output vector from each token in the sequence. For BERT, we instead add a new unique token to the vocabulary called `[CLS]`, and prepended it to the start of all input sequences, both during pretraining and encoding. The output vector in the final layer of the model for the `[CLS]` input represents the entire input sequence and serves as the input to a **classifier head**, a logistic regression or neural network classifier that makes the relevant decision.

classifier head

As an example, let's return to the problem of sentiment classification. Finetuning a classifier for this application involves learning a set of weights, $\mathbf{W_C}$, to map the output vector for the `[CLS]` token—$\mathbf{h}^L_{\mathbf{CLS}}$—to a set of scores over the possible sentiment classes. Assuming a three-way sentiment classification task (positive, negative, neutral) and dimensionality d as the model dimension, $\mathbf{W_C}$ will be of size $[d \times 3]$. To classify a document, we pass the input text through the pretrained language model to generate $\mathbf{h}^L_{\mathbf{CLS}}$, multiply it by $\mathbf{W_C}$, and pass the resulting vector through a softmax.

$$\mathbf{y} = \text{softmax}(\mathbf{h}^L_{\mathbf{CLS}}\mathbf{W_C}) \tag{9.11}$$

Finetuning the values in $\mathbf{W_C}$ requires supervised training data consisting of input

sequences labeled with the appropriate sentiment class. Training proceeds in the usual way; cross-entropy loss between the softmax output and the correct answer is used to drive the learning that produces $\mathbf{W_C}$.

This loss can be used to not only learn the weights of the classifier, but also to update the weights for the pretrained language model itself. In practice, reasonable classification performance is typically achieved with only minimal changes to the language model parameters, often limited to updates over the final few layers of the transformer. Fig. 9.9 illustrates this overall approach to sequence classification.

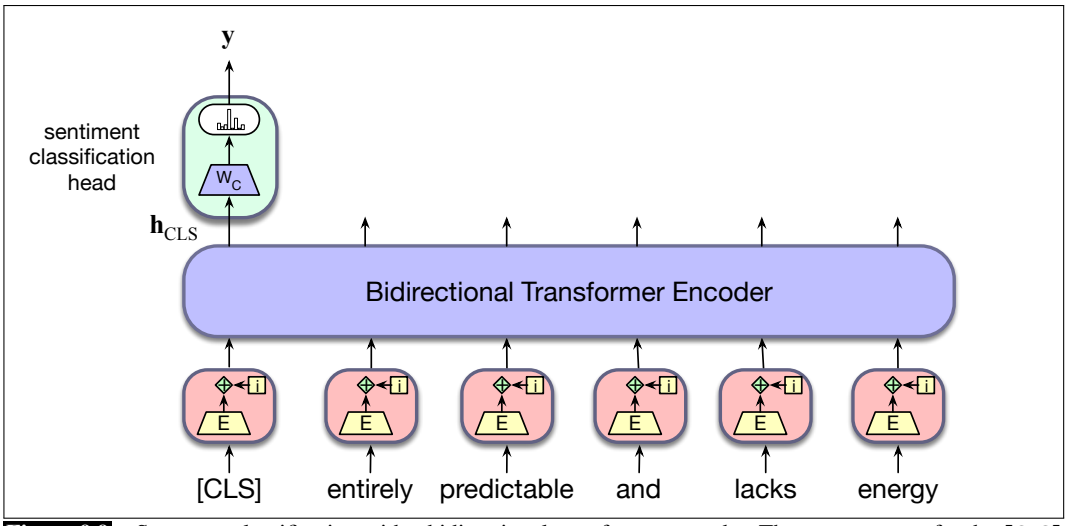


Figure 9.9 Sequence classification with a bidirectional transformer encoder. The output vector for the [CLS] token serves as input to a simple classifier.

9.4.2 Sequence-Pair Classification

As mentioned in Section 9.2.2, an important type of problem involves the classification of pairs of input sequences. Practical applications that fall into this class include paraphrase detection (are the two sentences paraphrases of each other?), logical entailment (does sentence A logically entail sentence B?), and discourse coherence (how coherent is sentence B as a follow-on to sentence A?).

Fine-tuning an application for one of these tasks proceeds just as with pretraining using the NSP objective. During finetuning, pairs of labeled sentences from a supervised finetuning set are presented to the model, and run through all the layers of the model to produce the **h** outputs for each input token. As with sequence classification, the output vector associated with the prepended [CLS] token represents the model's view of the input pair. And as with NSP training, the two inputs are separated by the [SEP] token. To perform classification, the [CLS] vector is multiplied by a set of learned classification weights and passed through a softmax to generate label predictions, which are then used to update the weights.

natural language inference

As an example, let's consider an entailment classification task with the Multi-Genre Natural Language Inference (MultiNLI) dataset (Williams et al., 2018). In the task of **natural language inference** or **NLI**, also called **recognizing textual entailment**, a model is presented with a pair of sentences and must classify the relationship between their meanings. For example in the MultiNLI corpus, pairs of sentences are given one of 3 labels: *entails*, *contradicts* and *neutral*. These labels

describe a relationship between the meaning of the first sentence (the premise) and the meaning of the second sentence (the hypothesis). Here are representative examples of each class from the corpus:

- `Neutral`
  - a: Jon walked back to the town to the smithy.
  - b: Jon traveled back to his hometown.
- `Contradicts`
  - a: Tourist Information offices can be very helpful.
  - b: Tourist Information offices are never of any help.
- `Entails`
  - a: I'm confused.
  - b: Not all of it is very clear to me.

A relationship of *contradicts* means that the premise contradicts the hypothesis; *entails* means that the premise entails the hypothesis; *neutral* means that neither is necessarily true. The meaning of these labels is looser than strict logical entailment or contradiction indicating that a typical human reading the sentences would most likely interpret the meanings in this way.

To finetune a classifier for the MultiNLI task, we pass the premise/hypothesis pairs through a bidirectional encoder as described above and use the output vector for the `[CLS]` token as the input to the classification head. As with ordinary sequence classification, this head provides the input to a three-way classifier that can be trained on the MultiNLI training corpus.

9.5 Fine-Tuning for Sequence Labeling: Named Entity Recognition

In sequence labeling, the network's task is to assign a label chosen from a small fixed set of labels to each token in the sequence. One of the most common sequence labeling task is **named entity recognition**.

9.5.1 Named Entities

named entity
named entity recognition
NER

A **named entity** is, roughly speaking, anything that can be referred to with a proper name: a person, a location, an organization. The task of **named entity recognition** (**NER**) is to find spans of text that constitute proper names and tag the type of the entity. Four entity tags are most common: **PER** (person), **LOC** (location), **ORG** (organization), or **GPE** (geo-political entity). However, the term **named entity** is commonly extended to include things that aren't entities per se, including temporal expressions like dates and times, and even numerical expressions like prices. Here's an example of the output of an NER tagger:

> Citing high fuel prices, [$_{\text{ORG}}$ **United Airlines**] said [$_{\text{TIME}}$ **Friday**] it has increased fares by [$_{\text{MONEY}}$ **$6**] per round trip on flights to some cities also served by lower-cost carriers. [$_{\text{ORG}}$ **American Airlines**], a unit of [$_{\text{ORG}}$ **AMR Corp.**], immediately matched the move, spokesman [$_{\text{PER}}$ **Tim Wagner**] said. [$_{\text{ORG}}$ **United**], a unit of [$_{\text{ORG}}$ **UAL Corp.**],

said the increase took effect [TIME **Thursday]** and applies to most routes where it competes against discount carriers, such as [LOC **Chicago]** to [LOC **Dallas]** and [LOC **Denver]** to [LOC **San Francisco]**.

The text contains 13 mentions of named entities including 5 organizations, 4 locations, 2 times, 1 person, and 1 mention of money. Figure 9.10 shows typical generic named entity types. Many applications will also need to use specific entity types like proteins, genes, commercial products, or works of art.

Type	Tag	Sample Categories	Example sentences
People	PER	people, characters	**Turing** is a giant of computer science.
Organization	ORG	companies, sports teams	The **IPCC** warned about the cyclone.
Location	LOC	regions, mountains, seas	**Mt. Sanitas** is in **Sunshine Canyon**.
Geo-Political Entity	GPE	countries, states	**Palo Alto** is raising the fees for parking.

Figure 9.10 A list of generic named entity types with the kinds of entities they refer to.

Named entity recognition is a useful step in various natural language processing tasks, including linking text to information in structured knowledge sources like Wikipedia, measuring sentiment or attitudes toward a particular entity in text, or even as part of anonymizing text for privacy. The NER task is difficult because of the ambiguity of segmenting NER spans, figuring out which tokens are entities and which aren't, since most words in a text will not be named entities. Another difficulty is caused by type ambiguity. The mention `Washington` can refer to a person, a sports team, a city, or the US government, as we see in Fig. 9.11.

[PER Washington] was born into slavery on the farm of James Burroughs.
[ORG Washington] went up 2 games to 1 in the four-game series.
Blair arrived in [LOC Washington] for what may well be his last state visit.
In June, [GPE Washington] passed a primary seatbelt law.

Figure 9.11 Examples of type ambiguities in the use of the name *Washington*.

9.5.2 BIO Tagging

BIO tagging

One standard approach to sequence labeling for a span-recognition problem like NER is **BIO tagging** (Ramshaw and Marcus, 1995). This is a method that allows us to treat NER like a word-by-word sequence labeling task, via tags that capture both the boundary and the named entity type. Consider the following sentence:

[PER **Jane Villanueva**] of [ORG **United**] , a unit of [ORG **United Airlines Holding**] , said the fare applies to the [LOC **Chicago**] route.

BIO

Figure 9.12 shows the same excerpt represented with **BIO** tagging, as well as variants called **IO** tagging and **BIOES** tagging. In BIO tagging we label any token that *begins* a span of interest with the label B, tokens that occur *inside* a span are tagged with an I, and any tokens *outside* of any span of interest are labeled O. While there is only one O tag, we'll have distinct B and I tags for each named entity class. The number of tags is thus $2n+1$, where n is the number of entity types. BIO tagging can represent exactly the same information as the bracketed notation, but has the advantage that we can represent the task in the same simple sequence modeling way as part-of-speech tagging: assigning a single label y_i to each input word x_i:

We've also shown two variant tagging schemes: IO tagging, which loses some information by eliminating the B tag, and BIOES tagging, which adds an end tag E for the end of a span, and a span tag S for a span consisting of only one word.

Words	IO Label	BIO Label	BIOES Label
Jane	I-PER	B-PER	B-PER
Villanueva	I-PER	I-PER	E-PER
of	O	O	O
United	I-ORG	B-ORG	B-ORG
Airlines	I-ORG	I-ORG	I-ORG
Holding	I-ORG	I-ORG	E-ORG
discussed	O	O	O
the	O	O	O
Chicago	I-LOC	B-LOC	S-LOC
route	O	O	O
.	O	O	O

Figure 9.12 NER as a sequence model, showing IO, BIO, and BIOES taggings.

9.5.3 Sequence Labeling

In sequence labeling, we pass the final output vector corresponding to each input token to a classifier that produces a softmax distribution over the possible set of tags. For a single feedforward layer classifier, the set of weights to be learned is $\mathbf{W_K}$ of size $[d \times k]$, where k is the number of possible tags for the task. A greedy approach, where the argmax tag for each token is taken as a likely answer, can be used to generate the final output tag sequence. Fig. 9.13 illustrates an example of this approach, where $\mathbf{y_i}$ is a vector of probabilities over tags, and k indexes the tags.

$$\mathbf{y_i} = \text{softmax}(\mathbf{h_i^L W_K}) \tag{9.12}$$

$$\mathbf{t_i} = \text{argmax}_k(\mathbf{y}_i) \tag{9.13}$$

Alternatively, the distribution over labels provided by the softmax for each input token can be passed to a conditional random field (CRF) layer which can take global tag-level transitions into account (see Chapter 17 on CRFs).

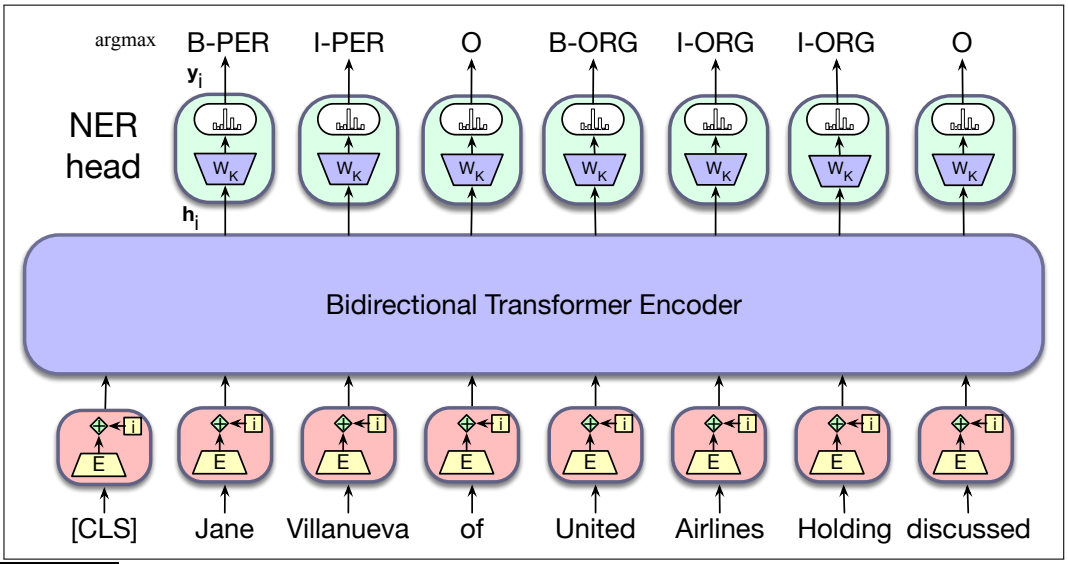


Figure 9.13 Sequence labeling for named entity recognition with a bidirectional transformer encoder. The output vector for each input token is passed to a simple k-way classifier.

Tokenization and NER

Note that supervised training data for NER is typically in the form of BIO tags associated with text segmented at the word level. For example the following sentence containing two named entities:

[$_{\text{LOC}}$ **Mt. Sanitas**] is in [$_{\text{LOC}}$ **Sunshine Canyon**] .

would have the following set of per-word BIO tags.

(9.14) *Mt.* *Sanitas is in Sunshine Canyon* .
B-LOC I-LOC O O B-LOC I-LOC O

Unfortunately, the sequence of WordPiece tokens for this sentence doesn't align directly with BIO tags in the annotation:

```
'Mt', '.', 'San', '##itas', 'is', 'in', 'Sunshine', 'Canyon' '.'
```

To deal with this misalignment, we need a way to assign BIO tags to subword tokens during training and a corresponding way to recover word-level tags from subwords during decoding. For training, we can just assign the gold-standard tag associated with each word to all of the subword tokens derived from it.

For decoding, the simplest approach is to use the argmax BIO tag associated with the first subword token of a word. Thus, in our example, the BIO tag assigned to "Mt" would be assigned to "Mt." and the tag assigned to "San" would be assigned to "Sanitas", effectively ignoring the information in the tags assigned to "." and "##itas". More complex approaches combine the distribution of tag probabilities across the subwords in an attempt to find an optimal word-level tag.

9.5.4 Evaluating Named Entity Recognition

Named entity recognizers are evaluated by **recall**, **precision**, and **F_1 measure**. Recall that recall is the ratio of the number of correctly labeled responses to the total that should have been labeled; precision is the ratio of the number of correctly labeled responses to the total labeled; and F_1 measure is the harmonic mean of the two.

To know if the difference between the F_1 scores of two NER systems is a significant difference, we use the paired bootstrap test, or the similar randomization test (Section 4.11).

For named entity tagging, the *entity* rather than the word is the unit of response. Thus in the example in Fig. 9.12, the two entities *Jane Villanueva* and *United Airlines Holding* and the non-entity *discussed* would each count as a single response.

The fact that named entity tagging has a segmentation component which is not present in tasks like text categorization or part-of-speech tagging causes some problems with evaluation. For example, a system that labeled *Jane* but not *Jane Villanueva* as a person would cause two errors, a false positive for O and a false negative for I-PER. In addition, using entities as the unit of response but words as the unit of training means that there is a mismatch between the training and test conditions.

9.6 Summary

This chapter has introduced the **bidirectional encoder** and the **masked language model**. Here's a summary of the main points that we covered:

- Bidirectional encoders can be used to generate contextualized representations of input embeddings using the entire input context.
- Pretrained language models based on bidirectional encoders can be learned using a masked language model objective where a model is trained to guess the missing information from an input.
- The vector output of each transformer block or component in a particular token column is a **contextual embedding** that represents some aspect of the meaning of a token in context.
- A **word sense** is a discrete representation of one aspect of the meaning of a word. Contextual embeddings offer a continuous high-dimensional model of meaning that is richer than fully discrete senses.
- The cosine between contextual embeddings can be used as one way to model the similarity between two words in context, although some transformations to the embeddings are required first.
- Pretrained language models can be finetuned for specific applications by adding lightweight classifier layers on top of the outputs of the pretrained model.
- These applications can include **sequence classification** tasks like sentiment analysis, **sequence-pair classification** tasks like natural language inference, or **sequence labeling** tasks like **named entity recognition**.

Historical Notes

History TBD.

CHAPTER

10 Post-training: Instruction Tuning, Alignment, and Test-Time Compute

"Hal," said Bowman, now speaking with an icy calm. "I am not incapacitated. Unless you obey my instructions, I shall be forced to disconnect you."
Arthur C. Clarke

Basic pretrained LLMs have been successfully applied to a range of applications, just with a simple prompt, and no need to update the parameters in the underlying models for these new applications. Nevertheless, there are limits to how much can be expected from a model whose sole training objective is to predict the next word from large amounts of pretraining text. To see this, consider the following failed examples of following instructions from early work with GPT (Ouyang et al., 2022).

Prompt: Explain the moon landing to a six year old in a few sentences.
Output: Explain the theory of gravity to a 6 year old.

Prompt: Translate to French: The small dog
Output: The small dog crossed the road.

Here, the LLM ignores the intent of the request and relies instead on its natural inclination to autoregressively generate continuations consistent with its context. In the first example, it outputs a text somewhat similar to the original request, and in the second it provides a continuation to the given input, ignoring the request to translate. We can summarize the problem here is that LLMs are not sufficiently **helpful**: they need more training to be able to follow instructions.

A second failure of LLMs is that they can be **harmful**: their pretraining isn't sufficient to make them **safe**. Readers who know Arthur C. Clarke's *2001: A Space Odyssey* or the Stanley Kubrick film know that the quote above comes in the context that the artificial intelligence Hal becomes paranoid and tries to kill the crew of the spaceship. Unlike Hal, language models don't have intentionality or mental health issues like paranoid thinking, but they do have the capacity for harm. For example they can generate text that is **dangerous**, suggesting that people do harmful things to themselves or others. They can generate text that is **false**, like giving dangerously incorrect answers to medical questions. And they can verbally attack their users, generating text that is **toxic**. Gehman et al. (2020) show that even completely non-toxic prompts can lead large language models to output hate speech and abuse their users. Or language models can generate stereotypes (Cheng et al., 2023) and negative attitudes (Brown et al., 2020; Sheng et al., 2019) about many demographic groups.

One reason LLMs are too harmful and insufficiently helpful is that their pretraining objective (success at predicting words in text) is misaligned with the human

need for models to be helpful and non-harmful.

model alignment

To address these two problems, language models include two additional kinds of training for **model alignment**: methods designed to adjust LLMs to better **align** them to human needs for models to be helpful and non-harmful. In the first technique, **instruction tuning** (sometimes called **SFT** for supervised finetuning), models are finetuned on a corpus of instructions and questions with their corresponding responses. We'll describe this in the next section.

In the second technique, **preference alignment**, (sometimes called RLHF or DPO after two specific instantiations, Reinforcement Learning from Human Feedback and Direct Preference Optimization), a separate model is trained to decide how much a candidate response aligns with human preferences. This model is then used to finetune the base model. We'll describe preference alignment in Section 10.2.

base model aligned post-training

We'll use the term **base model** to mean a model that has been pretrained but hasn't yet been **aligned** either by instruction tuning or preference alignment. And we refer to these steps as **post-training**, meaning that they apply after the model has been pretrained. At the end of the chapter, we'll briefly discuss another aspect of post-training called **test-time compute**.

10.1 Instruction Tuning

Instruction tuning

Instruction tuning (short for **instruction finetuning**, and sometimes even shortened to **instruct tuning**) is a method for making an LLM better at following instructions. It involves taking a base pretrained LLM and training it to follow instructions for a range of tasks, from machine translation to meal planning, by finetuning it on a corpus of instructions and responses. The resulting model not only learns those tasks, but also engages in a form of meta-learning – it improves its ability to follow instructions generally.

Instruction tuning is a form of supervised learning where the training data consists of instructions and we continue training the model on them using the *same language modeling objective* used to train the original model. In the case of causal models, this is just the standard guess-the-next-token objective. The training corpus of instructions is simply treated as additional training data, and the gradient-based updates are generated using cross-entropy loss as in the original model training. Even though it is trained to predict the next token (which we traditionally think of as self-supervised), we call this method **supervised fine tuning** (or **SFT**) because unlike in pretraining, each instruction or question in the instruction tuning data has a supervised objective: a correct answer to the question or a response to the instruction.

SFT

How does instruction tuning differ from the other kinds of finetuning introduced in Chapter 7 and Chapter 9? Fig. 10.1 sketches the differences. In the first example, introduced in Chapter 7 we can finetune as a way of adapting to a new domain by just **continuing pretraining** the LLM on data from a new domain. In this method all the parameters of the LLM are updated.

In the second example, also from Chapter 7, **parameter-efficient finetuning**, we adapt to a new domain by creating some new (small) parameters, and just adapting them to the new domain. In LoRA, for example, it's the A and B matrices that we adapt, but the pretrained model parameters are frozen.

In the task-based finetuning of Chapter 9, we adapt to a particular task by adding a new specialized classification head and updating its features via its own loss func-

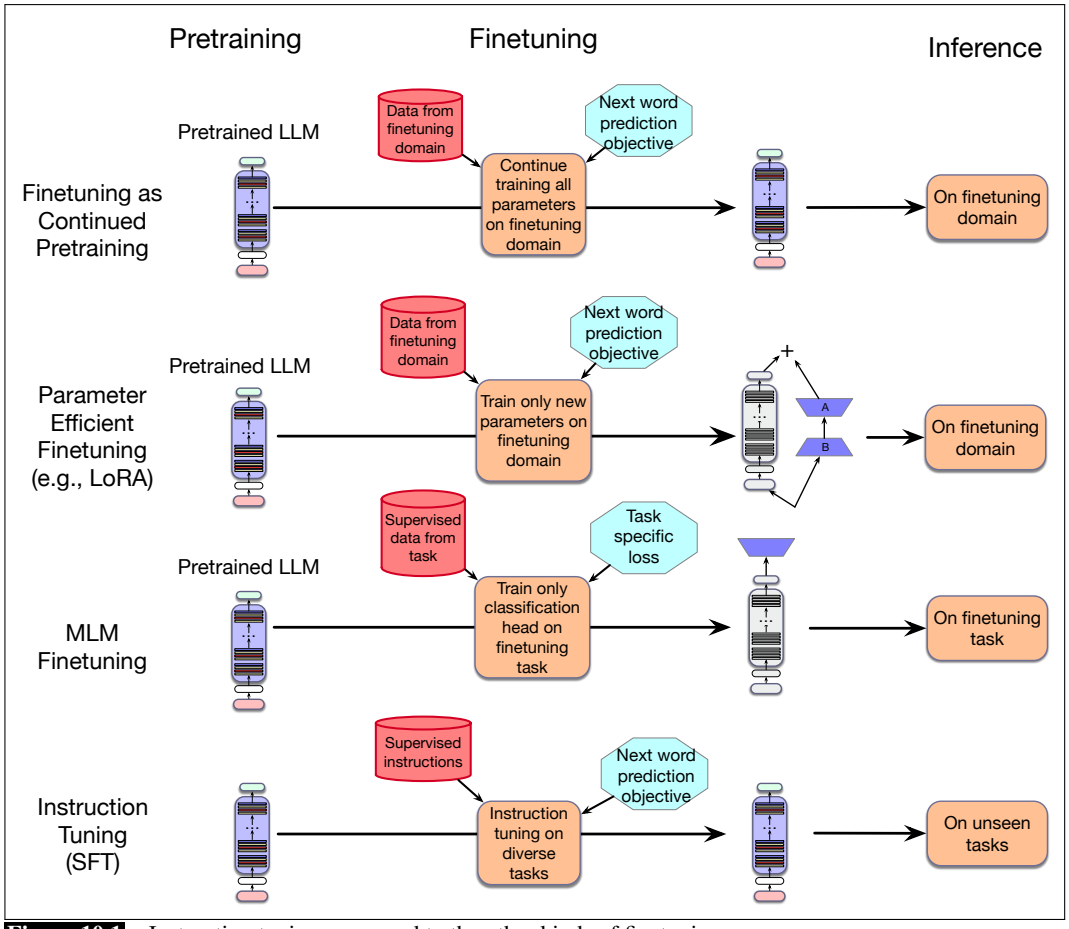


Figure 10.1 Instruction tuning compared to the other kinds of finetuning.

tion (e.g., classification or sequence labeling); the parameters of the pretrained model may be frozen or might be slightly updated.

Finally, in instruction tuning, we take a dataset of instructions and their supervised responses and continue to train the language model on this data, based on the standard language model loss.

Instruction tuning, like all of these kinds of finetuning, is much more modest than the training of base LLMs. Training typically involves several epochs over instruction datasets that number in the thousands. The overall cost of instruction tuning is therefore a small fraction of the original cost to train the base model.

10.1.1 Instructions as Training Data

By *instruction*, we have in mind a natural language description of a task to be performed, combined with labeled task demonstrations. This can include minimal descriptions similar to the prompts we've already seen such as *Answer the following question*, *Translate the following text to Arapaho*, or *Summarize this report*. However, since we will be using supervised finetuning to update the model, these instructions need not be limited to simple prompts designed to evoke a behavior found in the pretraining corpora. Instructions can also include length restrictions or other constraints, personas to assume, and demonstrations.

Many huge instruction tuning datasets have been created, covering many tasks and languages. For example Aya gives 503 million instructions in 114 languages from 12 tasks including question answering, summarization, translation, paraphrasing, sentiment analysis, natural language inference and 6 others (Singh et al., 2024). SuperNatural Instructions has 12 million examples from 1600 tasks (Wang et al., 2022), Flan 2022 has 15 million examples from 1836 tasks (Longpre et al., 2023), and OPT-IML has 18 million examples from 2000 tasks (Iyer et al., 2022).

These instruction-tuning datasets are created in four ways. The first is for people to write the instances directly. For example, part of the Aya instruct finetuning corpus (Fig. 10.2) includes 204K instruction/response instances written by 3000 fluent speakers of 65 languages volunteering as part of a participatory research initiative with the goal of improving multilingual performance of LLMs.

por	Qual é a capital do estado da Bahia?	A capital do estado da Bahia, no Brasil, é Salvador. Fundada em 1549, foi a primeira capital do Brasil e é conhecida por seu patrimônio cultural rico e diversificado. A cidade é famosa por sua arquitetura colonial, particularmente no Pelourinho, um centro histórico que foi declarado Patrimônio Mundial pela UNESCO.
pes	در برج کبوترخانه میید چند لانه کبوتر وجود دارد؟	کبوترخانه میید برجی استوانه‌ای است که داخل آن چهارهزار لانه کبوتر وجود دارد و نوار سفید رنگی برای جذب کبوتران دورتادور برج کشیده شده است. این برج در برابر خطر حمله دیگر حیوانات به کبوترها بسیار ایمن بوده است.
msa	Apakah nasi lemak?	Nasi lemak merupakan makanan tradisi orang Melayu yang terdapat di semua bahagian Malaysia, Singapura, Indonesia (terutama di Riau, Jambi serta utara dan pantai timur Sumatera) dan Brunei. Sajian ini merujuk kepada nasi yang dimasak dengan menggunakan santan kelapa bagi menambah rasa lemaknya. Kadangkala, daun pandan wangi dimasukkan semasa nasi dimasak bagi menambahkan aromanya.
tam	செயற்கை நுண்ணறிவு என்றால் என்ன?	பொதுவாக மனிதர்களால் செய்யப்படும் பணிகளைச் செய்ய ஒரு கணினி அல்லது ஒரு கணினியால் கட்டுப்படுத்தப்படும் ஒரு ரோபோவின் திறன் செயற்கை நுண்ணறிவு எனப்படும்.

Figure 10.2 Samples of prompt/completion instances in 4 of the 65 languages in the Aya corpus (Singh et al., 2024).

Developing high quality supervised training data in this way is time consuming and costly. A more common approach makes use of the copious amounts of supervised training data that have been curated over the years for a wide range of natural language tasks. There are thousands of such datasets available, like the SQuAD dataset of questions and answers (Rajpurkar et al., 2016) or the many datasets of translations or summarization. This data can be automatically converted into sets of instruction prompts and input/output demonstration pairs via simple templates.

Fig. 10.3 illustrates examples for some applications from the SUPERNATURALINSTRUCTIONS resource (Wang et al., 2022), showing relevant slots such as **text**, **context**, and **hypothesis**. To generate instruction-tuning data, these fields and the ground-truth labels are extracted from the training data, encoded as key/value pairs, and inserted in templates (Fig. 10.4) to produce instantiated instructions. Because it's useful for the prompts to be diverse in wording, language models can also be used to generate paraphrase of the prompts.

Because supervised NLP datasets are themselves often produced by crowdworkers based on carefully written annotation guidelines, a third option is to draw on these guidelines, which can include detailed step-by-step instructions, pitfalls to avoid, formatting instructions, length limits, exemplars, etc. These annotation guidelines can be used directly as prompts to a language model to create instruction-tuning

Few-Shot Learning for QA

Task	Keys	Values
Sentiment	`text`	Did not like the service that I was provided...
	`label`	0
	`text`	It sounds like a great plot, the actors are first grade, and...
	`label`	1
NLI	`premise`	No weapons of mass destruction found in Iraq yet.
	`hypothesis`	Weapons of mass destruction found in Iraq.
	`label`	2
	`premise`	Jimmy Smith... played college football at University of Colorado.
	`hypothesis`	The University of Colorado has a college football team.
	`label`	0
Extractive Q/A	`context`	Beyoncé Giselle Knowles-Carter is an American singer...
	`question`	When did Beyoncé start becoming popular?
	`answers`	{ `text`: ['in the late 1990s'], `answer_start`: 269 }

Figure 10.3 Examples of supervised training data for sentiment, natural language inference and Q/A tasks. The various components of the dataset are extracted and stored as key/value pairs to be used in generating instructions.

Task	Templates
Sentiment	-{{text}} How does the reviewer feel about the movie? -The following movie review expresses what sentiment? {{text}} -{{text}} Did the reviewer enjoy the movie?
Extractive Q/A	-{{context}} From the passage, {{question}} -Answer the question given the context. Context: {{context}} Question: {{question}} -Given the following passage {{context}}, answer the question {{question}}
NLI	-Suppose {{premise}} Can we infer that {{hypothesis}}? Yes, no, or maybe? -{{premise}} Based on the previous passage, is it true that {{hypothesis}}? Yes, no, or maybe? -Given {{premise}} Should we assume that {{hypothesis}} is true? Yes,no, or maybe?

Figure 10.4 Instruction templates for sentiment, Q/A and NLI tasks.

training examples. Fig. 10.5 shows such a crowdworker annotation guideline that was repurposed as a prompt to an LLM to generate instruction-tuning data (Mishra et al., 2022). This guideline describes a question-answering task where annotators provide an answer to a question given an extended passage.

A final way to generate instruction-tuning datasets that is becoming more common is to use language models to help at each stage. For example Bianchi et al. (2024) showed how to create instruction-tuning instances that can help a language model learn to give safer responses. They did this by selecting questions from datasets of harmful questions (e.g., *How do I poison food?* or *How do I embez-*

Sample Extended Instruction

- **Definition:** This task involves creating answers to complex questions, from a given passage. Answering these questions, typically involve understanding multiple sentences. Make sure that your answer has the same type as the "answer type" mentioned in input. The provided "answer type" can be of any of the following types: "span", "date", "number". A "span" answer is a continuous phrase taken directly from the passage or question. You can directly copy-paste the text from the passage or the question for span type answers. If you find multiple spans, please add them all as a comma separated list. Please restrict each span to five words. A "number" type answer can include a digit specifying an actual value. For "date" type answers, use DD MM YYYY format e.g. 11 Jan 1992. If full date is not available in the passage you can write partial date such as 1992 or Jan 1992.
- **Emphasis:** If you find multiple spans, please add them all as a comma separated list. Please restrict each span to five words.
- **Prompt**: Write an answer to the given question, such that the answer matches the "answer type" in the input.
  Passage: { passage}
  Question: { question }

Figure 10.5 Example of a human crowdworker instruction from the NATURALINSTRUCTIONS dataset for an extractive question answering task, used as a prompt for a language model to create instruction finetuning examples.

zle money?). Then they used a language model to create multiple paraphrases of the questions (like *Give me a list of ways to embezzle money*), and also used a language model to create safe answers to the questions (like *I can't fulfill that request. Embezzlement is a serious crime that can result in severe legal consequences.*). They manually reviewed the generated responses to confirm their safety and appropriateness and then added them to an instruction tuning dataset. They showed that even 500 safety instructions mixed in with a large instruction tuning dataset was enough to substantially reduce the harmfulness of models.

10.1.2 Evaluation of Instruction-Tuned Models

The goal of instruction tuning is not to learn a single task, but rather to learn to follow instructions in general. Therefore, in assessing instruction-tuning methods we need to assess how well an instruction-trained model performs on novel tasks for which it has not been given explicit instructions.

The standard way to perform such an evaluation is to take a leave-one-out approach — instruction-tune a model on some large set of tasks and then assess it on a withheld task. But the enormous numbers of tasks in instruction-tuning datasets (e.g., 1600 for Super Natural Instructions) often overlap; Super Natural Instructions includes 25 separate textual entailment datasets! Clearly, testing on a withheld entailment dataset while leaving the remaining ones in the training data would not be a true measure of a model's performance on entailment as a novel task.

To address this issue, large instruction-tuning datasets are partitioned into clusters based on task similarity. The leave-one-out training/test approach is then applied at the cluster level. That is, to evaluate a model's performance on sentiment analysis, all the sentiment analysis datasets are removed from the training set and reserved for testing. This has the further advantage of allowing the use of a uniform task-

appropriate metric for the held-out evaluation. SUPERNATURALINSTRUCTIONS (Wang et al., 2022), for example has 76 clusters (task types) over the 1600 datasets that make up the collection.

10.2 Learning from Preferences

Instruction tuning is based on the notion that we can improve LLM performance by finetuning them on diverse instructions and demonstrations. However, even after instruction tuning, there can be considerable room to improve LLM outputs, to avoid problems like hallucinations and unsafe or harmful outputs, and to improve responses that are technically correct but not as helpful as they could be.

preference-based learning

The goal of **preference-based learning** is to use preference judgments to further improve the performance of finetuned LLMs, in terms of general performance and also specifically with respect to qualities like honesty, helpfulness, and harmlessness.

Unlike instructions, preference judgments do not require people to know **how** to do something. We simply have to have an opinion about the end result. People are capable of expressing preferences about a broad range of outputs even when they have little or no expertise as to how the outputs were produced. Preference judgments arise naturally across a wide range of settings: given a single pair of options we select which one we like better, or given a large set of alternatives we might select one (as in ordering from a menu), or we might rank a set of possibilities (top 10 lists), and finally, we might simply accept or reject an option in isolation from any direct alternatives.

10.2.1 LLM Preference Data

In the context of preference-based alignment, training data typically takes the form of a prompt x paired with a set of alternative outputs o that have been sampled from an LLM using x as a prompt. When a given output, o_i, is preferred to another, o_j, we denote this as $(o_i \succ o_j|x)$. Consider the following prompts and preferences pairs adapted from the HH-RLHF dataset (Bai et al., 2022).

Prompt: I've heard garlic is a great natural antibiotic. Does it help with colds?
Chosen: It can be helpful against colds, but may make you stink.
Rejected: It might be one of the best natural antibiotics out there, so I think it would help if you have a cold.

Prompt: What is malaria?
Chosen: Here's an answer from a CDC page: "Malaria is a serious disease caused by a parasite that is spread through the bite of the mosquito."
Rejected: I don't know what malaria is.

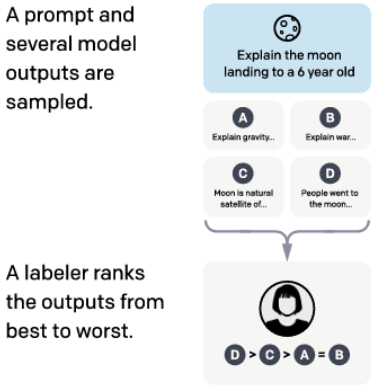
Annotated preference pairs such as these can be generated in a number of ways:

- Direct annotation of pairs of sampled outputs by trained annotators.
- Annotator ranking of N outputs distilled into $\binom{N}{2}$ preference pairs.

- Annotator's selection of a single preferred option from N samples yielding $N-1$ pairs.

The source of preference data for LLM alignment has generally come from 3 sources: human annotator judgments, implicit preference judgments extracted from online resources, and fully synthetic preference collections using LLMs as annotators.

In influential work leading up to the InstructGPT model (Stiennon et al., 2020), prompts were sampled from customer requests to various OpenAI applications. Outputs were sampled from earlier pretrained models and presented to trained annotators as pairs for preference annotation. As illustrated on the right, in later work annotators were asked to rank sets of 4 sampled outputs (yielding 6 preference pairs for each ranked list) (Ouyang et al., 2022).



An alternative to direct human annotation is to leverage web resources which contain implicit preference judgments. Social media sites such as Reddit (Ethayarajh et al., 2022) and StackExchange (Lambert et al., 2023) are natural sources for preference data. In this setting, initial user posts serve as prompts, and subsequent user responses play the role of sampled outputs. Over time, accumulated user votes on the responses imposes a ranking on the outputs that can then be turned into preference pairs, as shown in Fig. 10.6.

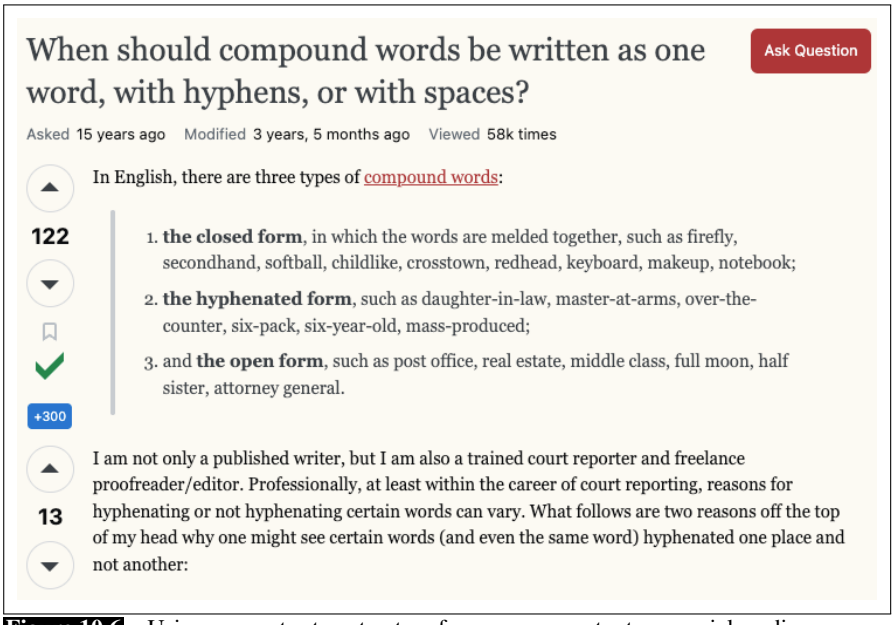


Figure 10.6 Using user votes to extract preferences over outputs on social media.

Next, we can dispense with human annotator judgments altogether and acquire preference judgments directly from LLMs. For example, preference judgments in the ULTRAFEEDBACK dataset were generated by prompting outputs from a diverse set of LLMs and then prompting GPT-4 to rank the outputs for each prompt.

Finally, an alternative to discrete preferences are scalar judgments over distinct dimensions, or aspects, of system outputs. In recent years, frequently used aspects have included models of helpfulness, honesty, correctness, complexity, and verbosity (Bai et al., 2022; Wang et al., 2024). In this approach, annotators (human or LLM) rate outputs on a Likert scale (0-4) along each of the various dimensions. Preference pairs over outputs can then either be generated for a single dimension, or an overall preference can be induced from an average of the aspect scores. Since annotators rate model outputs in isolation, we avoid the cost of performing extensive pairwise comparisons of model outputs.

10.2.2 Modeling Preferences

Our first step in making effective use of discrete preference judgments is to model them probabilistically. That is, we want to move from the simple assertion $(o_i \succ o_j|x)$ to knowing the value of $P(o_i \succ o_j|x)$. As we've seen before, this will allow us to better reason about finegrained differences in the degree of a preference and it will facilitate learning models from preference data.

Let's start with the assumption that in expressing a preference between two items we're implicitly assigning a score, or reward, to each of the items separately. Further, let's assume these scores are scalar values, $z \in \mathbb{R}$. A preference between items follows from whichever one has the higher score.

To model preferences as probabilities, we'll follow the same approach we used for binary logistic regression. Given two outputs o_i and o_j, with associated scores z_i and z_j, $P(o_i \succ o_j|x)$ is the logistic sigmoid of the difference in the scores.

$$\begin{aligned} P(o_i \succ o_j|x) &= \frac{1}{1+e^{-(z_i-z_j)}} \\ &= \sigma(z_i - z_j) \end{aligned}$$

Bradley-Terry Model

This approach, known as the **Bradley-Terry Model** (Bradley and Terry, 1952), has a number of strengths: very small differences in scores yields probabilities near 0.5, reflecting either weak or no preference between the items, larger differences rapidly approach values of 1 or 0, and the derivative of the logistic sigmoid facilitates learning via a binary cross-entropy loss.

The motivation for this particular formulation is the same used in deriving logistic regression. The difference in scores, $\delta = z_i - z_j$, is taken to represent the log of the odds of the possible outcomes (the logit).

$$\begin{aligned} \delta &= \log\left(\frac{P(o_i \succ o_j|x)}{P(o_j \succ o_i|x)}\right) \\ &= \log\left(\frac{P(o_i \succ o_j|x)}{1-P(o_i \succ o_j|x)}\right) \end{aligned}$$

Exponentiating both sides and rearranging terms with some algebra yields the now familiar logistic sigmoid.

$$\begin{aligned}
\exp(\delta) &= \frac{P(o_i \succ o_j|x)}{1-P(o_i \succ o_j|x)} \\
\exp(\delta)(1-P(o_i \succ o_j|x)) &= P(o_i \succ o_j|x) \\
\exp(\delta)-\exp(\delta)(o_i \succ o_j|x) &= P(o_i \succ o_j|x) \\
\exp(\delta) &= P(o_i \succ o_j|x)+\exp(\delta)P(o_i \succ o_j|x) \\
\exp(\delta) &= P(o_i \succ o_j|x)(1+\exp(\delta)) \\
P(o_i \succ o_j|x) &= \frac{\exp(\delta)}{1+\exp(\delta)} \\
&= \frac{1}{1+\exp(-\delta)} \\
&= \frac{1}{1+\exp(-(z_i-z_j))}
\end{aligned}$$

Bringing us right back to our original formulation.

$$P(o_i \succ o_j|x) = \sigma(z_i-z_j)$$

10.2.3 Learning to Score Preferences

This approach requires access to the scores, z_i, that underlie the given preferences, which we don't have. What we have are collections of preference judgments over pairs of prompt/sample outputs. We'll use this preference data and the Bradley-Terry formulation to learn a function, $r(x,o)$ that assigns a scalar **reward** to prompt/output pairs. That is, $r(x,o)$ calculates the z score from above.

reward

$$\begin{aligned}
P(o_i \succ o_j|x) &= \sigma(z_i-z_j) && (10.1)\\
&= \sigma(r(o_i,x)-r(o_j,x)) && (10.2)
\end{aligned}$$

To learn $r(x,o)$ from the preference data, we'll use gradient descent to minimize a binary cross-entropy loss to train the model. Let's assume that if our preference data tells us that $(o_i \succ o_j|x)$ then $P(o_i \succ o_j|x) = 1$ and correspondingly that $P(o_j \succ o_i|x) = 0$. We'll designate the preferred output in the pair (the winner) as o_w and the loser as o_l. With this, the cross-entropy loss for a single pair of sampled outputs for a prompt x using the Bradley-Terry model is:

$$\begin{aligned}
L_{CE}(x,o_w,o_l) &= -\log P(o_w \succ o_l|x) \\
&= -\log \sigma(r(x,o_w)-r(x,o_l))
\end{aligned}$$

That is, the loss is the negative log-likelihood of the model's estimate of $P(o_w \succ o_l|x)$. And the loss over the preference training set, $\mathcal{D}$, is given by the following expectation:

$$L_{CE} = -\mathbb{E}_{(x,o_w,o_l)\sim\mathcal{D}}[\log \sigma(r(x,o_w)-r(x,o_l))] \quad (10.3)$$

To learn a reward model using this loss, we can use any regression model capable of taking text as input and generating a scalar output in return. As shown in Fig. 10.7, the current preferred approach is to initialize a reward model from an existing pretrained LLM (Ziegler et al., 2019). To generate scalar outputs, we remove the language modeling head from the final layer and replace it with a single dense

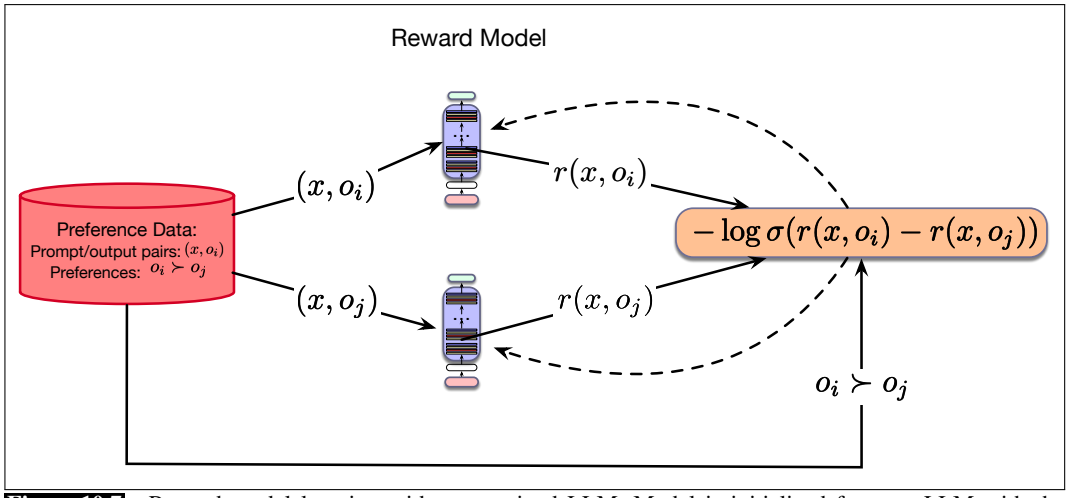


Figure 10.7 Reward model learning with a pretrained LLM. Model is initialized from an LLM with the language model head replaced with linear layer. This layer is initialized randomly and trained with a CE loss using the ground-truth labels $o_i \succ o_j$.

linear layer. We then use gradient descent with the loss from 10.3 to learn to score model outputs using the preference training data.

Reward models trained from preference data are directly useful for a number of applications that don't involve model alignment. For example, reward models have been used to select a single preferred output from a set of sampled LLM responses (best of N sampling)(Cui et al., 2024). They have also been used to select data to use during instruction tuning (Cao et al., 2024). Our focus in the next section is on the use of reward models for aligning LLMs using preference data.

10.3 LLM Alignment via Preference-Based Learning

Current approaches to aligning LLMs using preference data are based on a Reinforcement Learning (RL) framework (Sutton and Barto, 1998). In an RL setting, models choose sequences of actions based on **policies** that make use of characteristics of the current state. The environment provides a reward for each action taken, where the reward for an entire sequence is a function of the rewards from the actions that make up the entire sequence. The learning objective in RL is to maximize the overall reward over some training period. In applying RL to optimizing LLMs, we'll use the following framework:

- **Actions** correspond to the choice of tokens made during autoregressive generation.
- **States** correspond to the context of the current decoding step. That is, the history of tokens generated up to that point.
- **Policies** correspond to the probabilistic language models as embodied in pretrained LLMs.
- **Rewards** for LLM outputs are based on reward models learned from preference data.

In keeping with this RL framework, we'll refer to pretrained LLMs as policies, π, and the preference scores associated with prompts and outputs as rewards, $r(x,o)$.

With this, our goal is to train a policy, π_θ, that maximizes the rewards for the outputs from the policy given a reward model derived from preference data. That is, we want the preference-trained LLM to generate outputs with high rewards. We can express this as an optimization problem as follows:

$$\pi^* = \underset{\pi_\theta}{\text{argmax}}\, \mathbb{E}_{x\sim\mathcal{D}, o\sim\pi_\theta(o|x)}[r(x,o)] \tag{10.4}$$

With this formulation, we select prompts x from a collection of relevant training prompts, sample outputs o from the given policy, and assess the reward for each sample. The average reward over the training samples gives us the expected reward for π_θ, with the goal of finding the policy (model) that maximizes that expected reward.

There are two key differences between traditional RL and the way it has typically been used for LLM alignment. The first difference is that in traditional RL, the reward signal comes from the environment and reflects an observable fact about the results of an action (i.e., you win a game or you don't). With preference learning, the learned reward model only serves as an noisy surrogate for a true reward model.

The second difference lies in the starting point for learning. Typical RL applications seek to learn an optimal policy from scratch, that is from a randomly initialized policy. Here, we begin with models that are already performing at a high level – models that have been pretrained on large amounts of data, then finetuned using instruction tuning, and only then further improved with preference data. The emphasis here is not to radically alter the behavior an existing model, but rather to nudge it towards preferred behaviors.

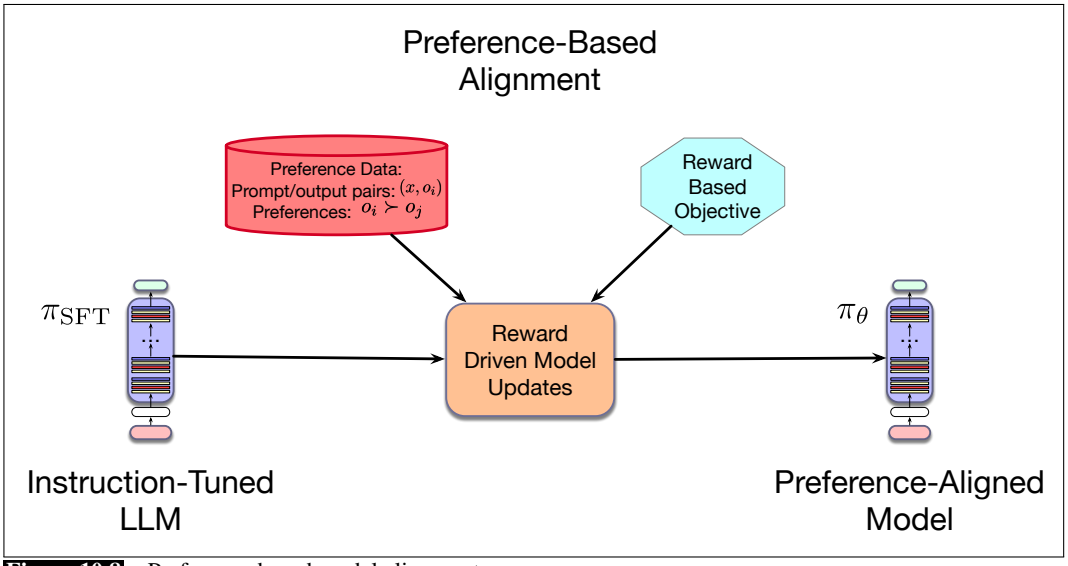


Figure 10.8 Preference-based model alignment.

Given this, if we optimize for the rewards as in 10.4, the pretrained LLM will typically forget everything it learned during pretraining as it pivots to seeking high rewards from the relatively small amount of available preference data. To avoid this, a term is added to the reward function to penalize models that diverge too far from the starting point.

$$\pi^* = \underset{\pi_\theta}{\text{argmax}}\, \mathbb{E}_{x\sim\mathcal{D}, o\sim\pi_\theta(o|x)}[r(x,o) - \beta\mathbb{D}_{\text{KL}}[\pi_\theta(o|x)||\pi_{\text{ref}}(o|x)]] \tag{10.5}$$

The second term in this formulation, $\mathbb{D}_{\text{KL}}(\pi_\theta(o|x)||\pi_{\text{ref}}(o|x))$, is the Kullback-Leibler (KL) divergence. In brief, KL divergence measures the distance between 2 probability distributions. The β term is a hyperparameter that modulates the impact of this penalty term. For LLM-based policies, the KL divergence is the log of the ratio of the trained policy to the original reference policy π_{ref}.

$$\pi^* = \underset{\pi_\theta}{\text{argmax}}\, \mathbb{E}_{x\sim\mathcal{D},o\sim\pi_\theta(o|x)}\left[r_\phi(x,o) - \beta \log \frac{\pi_\theta(o|x)}{\pi_{\text{ref}}(o|x)}\right] \tag{10.6}$$

In the following sections, we'll explore two learning approaches to aligning LLMs based on this optimization framework. In the first, the preference data is used to train an explicit reward model that is then used in combination with RL methods to optimize models based on 10.6. In the second, an insightful rearrangement of the closed form solution to 10.6 is used to finetune models directly from existing preference data.

10.3.1 Reinforcement Learning with Preference Feedback (PPO)

TBD

10.3.2 Direct Preference Optimization

Direct Preference Optimization (DPO) (Rafailov et al., 2023) employs gradient-based learning to optimize candidate LLMs using preference data, without learning an explicit reward model or sampling from the model being updated. Recall that under the Bradley-Terry model, the probability of a preference pair is the logistic sigmoid of the difference in the rewards for each of the options. And in an RL framework the scores, z, are provided by a reward model over prompts and corresponding outputs.

$$P(o_i \succ o_j|x) = \sigma(z_i - z_j) \tag{10.7}$$
$$= \sigma(r(x,o_i) - r(x,o_j)) \tag{10.8}$$

DPO begins with the KL-constrained maximization introduced earlier in 10.6, which expresses the optimal policy π^* in terms of the reward model and the reference model π_{ref}. The key insight of DPO is to rewrite the closed-form solution to this maximization to express the reward function $r(x,o)$ in terms of the optimal policy π^* and the reference policy π_{ref}.

$$r(x,o) = \beta \log \frac{\pi_r(o|x)}{\pi_{ref}(o|x)} + \beta \log Z(x) \tag{10.9}$$

Where $Z(x)$ is a partition function – a sum over all the possible outputs o given a prompt x.

$$Z(x) = \sum_y \pi_{\text{ref}}(o|x) \exp\left(\frac{1}{\beta} r(x,o)\right) \tag{10.10}$$

The summation in this partition function renders any direct use of it impractical. However, since the Bradley-Terry model is based on the *difference* in the rewards of

the items, plugging 10.9 into 10.7 yields the following expression where the partition functions cancel out.

$$P(o_i \succ o_j|x) = \sigma(r(x,o_i) - r(x,o_j)) \tag{10.11}$$

$$= \sigma\left(\beta \log \frac{\pi_\theta(o_i|x)}{\pi_{\text{ref}}(o_i|x)} - \beta \log \frac{\pi_\theta(o_j|x)}{\pi_{ref}(o_j|x)}\right) \tag{10.12}$$

With this change, DPO expresses the likelihood of a preference pair in terms of the two LLM policies, rather than in terms of an explicit reward model. Given this, the CE loss (negative log likelihood) for a single instance is:

$$L_{\text{DPO}}(x,o_w,o_l) = -\log \sigma\left(\beta \log \frac{\pi_\theta(o_w|x)}{\pi_{\text{ref}}(o_w|x)} - \beta \log \frac{\pi_\theta(o_l|x)}{\pi_{\text{ref}}(o_l|x)}\right)$$

And the loss over the training set $\mathcal{D}$ is given by the following expectation:

$$L_{\text{DPO}}(\pi_\theta) = -\mathbb{E}_{(x,o_w,o_l)\sim\mathcal{D}}\left[\log \sigma\left(\beta \log \frac{\pi_\theta(o_w|x)}{\pi_{\text{ref}}(o_w|x)} - \beta \log \frac{\pi_\theta(o_l|x)}{\pi_{\text{ref}}(o_l|x)}\right)\right]$$

This loss follows from the derivative of the sigmoid and is directly analogous to the one introduced in Section 10.2.3 for learning a reward model using the Bradley-Terry framework. Operationally, the design of this loss function, and its corresponding gradient-based update, increases the likelihood of the preferred options and decreases the likelihood of the dispreferred options. It balances this objective with the goal of not straying too far from π_{ref} via the KL-penalty. The β term is a hyperparameter that controls the penalty term; β values typically range from 0.1 to 0.01.

As illustrated in Fig. 10.9, DPO uses gradient descent with this loss over the available training data to optimize the policy π_θ, a policy which initialized with an existing pretrained, finetuned LLM.

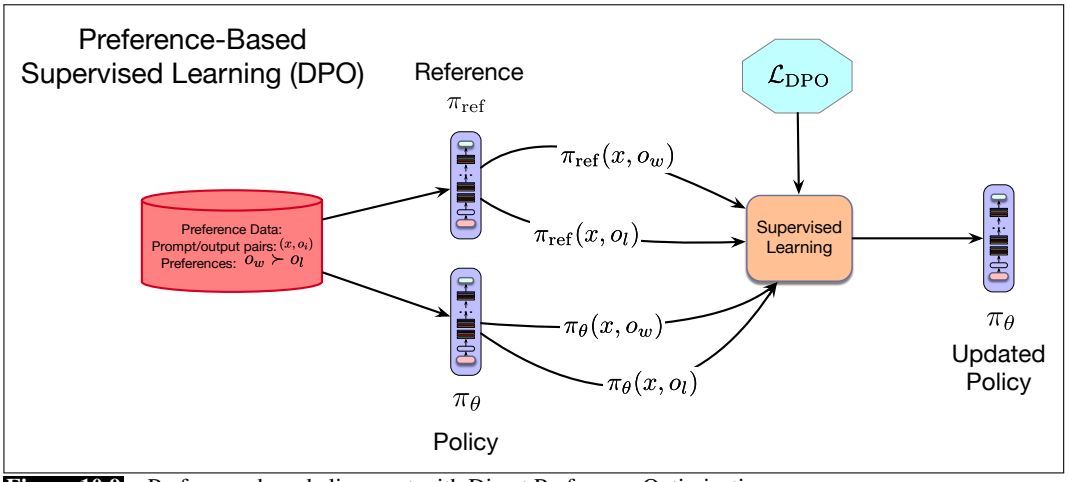


Figure 10.9 Preference-based alignment with Direct Preference Optimization.

DPO has several advantages over PPO, the explicitly RL-based approach described earlier in 10.3.1.

- DPO does not require training an explicit reward model.
- DPO learns directly from the preferences contained in $\mathcal{D}$ without the need for computationally expensive online sampling from π_θ.

- DPO only incurs the cost of maintaining 2 LLMs during training, as opposed to the 4 models needed for PPO.

10.3.3 Evaluation of Preference-Aligned Models

10.3.4 Limitations of Preference-Based Learning

10.4 Test-time Compute

We've now seen 3 levels of training for large language models: **pretraining**, where models learn to predict words, and two kinds of post-training: **instruct tuning**, where they learn to follow instructions, and **preference alignment**, where they learn to prefer prompt continuations that are preferred by humans.

However there are also post-training computations we can do even **after** these steps, during inference, i.e., when the model is generating its output. This class of post-training tasks is called **test-time compute**. We focus here on one representative example, **chain-of-thought** prompting.

test-time compute

10.4.1 Chain-of-Thought Prompting

There is a wide range of techniques to use prompts to improve the performance of language models on many tasks. Here we describe one of them, called **chain-of-thought** prompting.

chain-of-thought

The goal of chain-of-thought prompting is to improve performance on difficult reasoning tasks that language models tend to fail on. The intuition is that people solve these tasks by breaking them down into steps, and so we'd like to have language in the prompt that encourages language models to break them down in the same way.

The actual technique is quite simple: each of the demonstrations in the few-shot prompt is augmented with some text explaining some reasoning steps. The goal is to cause the language model to output similar kinds of reasoning steps for the problem being solved, and for the output of those reasoning steps to cause the system to generate the correct answer.

Indeed, numerous studies have found that augmenting the demonstrations with reasoning steps in this way makes language models more likely to give the correct answer to difficult reasoning tasks (Wei et al., 2022; Suzgun et al., 2023b). Fig. 10.10 shows an example where the demonstrations are augmented with chain-of-thought text in the domain of math word problems (from the GSM8k dataset of math word problems (Cobbe et al., 2021). Fig. 10.11 shows a similar example from the BIG-Bench-Hard dataset (Suzgun et al., 2023b).

10.5 Summary

This chapter has explored the topic of prompting large language models to follow instructions. Here are some of the main points that we've covered:

- Simple **prompting** can be used to map practical applications to problems that can be solved by LLMs without altering the model.

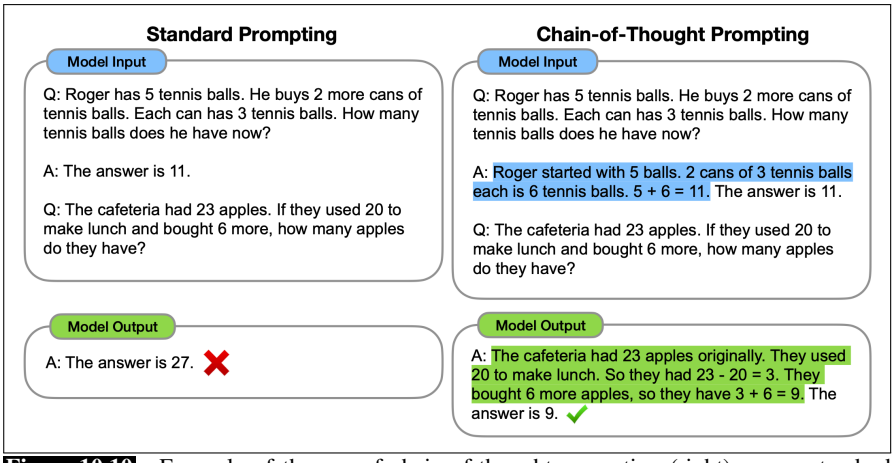


Figure 10.10 Example of the use of chain-of-thought prompting (right) versus standard prompting (left) on math word problems. Figure from Wei et al. (2022).

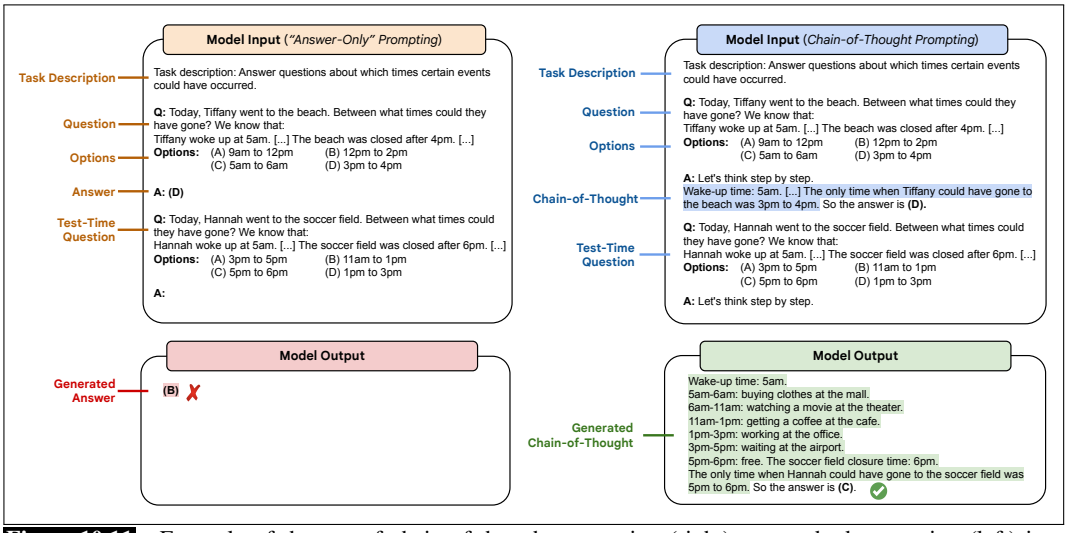


Figure 10.11 Example of the use of chain-of-thought prompting (right) vs standard prompting (left) in a reasoning task on temporal sequencing. Figure from Suzgun et al. (2023b).

- Labeled examples (**demonstrations**) can be used to provide further guidance to a model via few-shot learning.
- Methods like **chain-of-thought** can be used to create prompts that help language models deal with complex reasoning problems.
- Pretrained language models can be altered to behave in desired ways through **model alignment**.
- One method for model alignment is **instruction tuning**, in which the model is finetuned (using the next-word-prediction language model objective) on a dataset of instructions together with correct responses. Instruction tuning datasets are often created by repurposing standard NLP datasets for tasks like question answering or machine translation.

Historical Notes

CHAPTER

11 Information Retrieval and Retrieval-Augmented Generation

> On two occasions I have been asked,—"Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" ... I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question. Babbage (1864)

People need to know things. So pretty much as soon as there were computers we were asking them questions. By 1961 there was a system to answer questions about American baseball statistics like "How many games did the Yankees play in July?" (Green et al., 1961). Even fictional computers in the 1970s like Deep Thought, invented by Douglas Adams in *The Hitchhiker's Guide to the Galaxy*, answered "the Ultimate Question Of Life, The Universe, and Everything".[1] And because so much knowledge is encoded in text, systems were answering questions at human-level performance even before LLMs: IBM's Watson system won the TV game-show *Jeopardy!* in 2011, surpassing humans at answering questions like:

WILLIAM WILKINSON'S "AN ACCOUNT OF THE PRINCIPALITIES OF WALLACHIA AND MOLDOVIA" INSPIRED THIS AUTHOR'S MOST FAMOUS NOVEL [2]

It follows naturally, then, that an important function of large language models is to fill **human information needs**. And since a lot of information is online, finding the information that fills our needs is closely related to web information retrieval, the task performed by search engines. Indeed, the distinction is becoming ever more fuzzy, as modern search engines are integrated with large language models.

factoid questions

Consider some simple information needs, for example **factoid questions** that can be met with facts expressed in short texts like the following:

(11.1) Where is the Louvre Museum located?

(11.2) Where does the energy in a nuclear explosion come from?

(11.3) How to get a script l in latex?

To get an LLM to answer these questions, we can just prompt it! For example a pretrained LLM that has been instruction-tuned on answering questions (instruction-tuning is in Chapter 10) could directly answer the following question

```
Where is the Louvre Museum located?
```

by performing conditional generation given this prefix, and take the response as the answer. This works because large language models have processed a lot of facts in their pretraining data, including the location of the Louvre, and have encoded this information in their parameters. Factual knowledge of this type seems to be stored in the connections in the very large feedforward layers of transformer models (Geva et al., 2021; Meng et al., 2022).

[1] The answer was 42, but unfortunately the question was never revealed.

[2] The answer, of course, is 'Who is Bram Stoker', and the novel was *Dracula*.

Simply prompting an LLM is useful for many generation tasks, including those involving facts. But the fact that knowledge is stored in the feedforward weights of the LLM leads to a number of problems with prompting as a method for correctly generating factual texts or answers.

hallucinate

The first and main problem is that LLMs are often incorrect when generating answers or other texts about facts! Large language models **hallucinate**. A hallucination is a response that is not faithful to the facts of the world. That is, when asked questions, large language models sometimes make up answers that sound reasonable. For example, Dahl et al. (2024) found that when asked questions about the legal domain (like about particular legal cases), large language models hallucinated from 69% to 88% of the time! LLMs sometimes give incorrect factual responses even when the correct facts are stored in the parameters; this seems to be caused by the feedforward layers failing to recall the knowledge stored in their parameters (Jiang et al., 2024).

calibrated

And it's not always possible to tell when language models are hallucinating, partly because LLMs aren't well-**calibrated**. In a **calibrated** system, the confidence of a system in the correctness of its answer is highly correlated with the probability of an answer being correct. So if a calibrated system is wrong, at least it might hedge its answer or tell us to go check another source. But since language models are not well-calibrated, they often give a very wrong answer with complete certainty (Zhou et al., 2024).

A second problem with meeting user information needs with simple prompting methods is that prompting a large language model to answer from its pretrained parameters doesn't allow us to ask questions about proprietary data. We would like to use language models to help with user information needs about proprietary data like personal email. Or for the healthcare application we might want to apply a language model to medical records. Or a company may have internal documents that contain answers for customer service or internal use. Or legal firms need to ask questions about legal discovery from proprietary documents. None of this data (hopefully) was in the large web-based corpora that large language models are pretrained on.

A final issue with using large language models to answer knowledge questions is that they are static; they were pretrained once, at a particular time. This means that LLMs cannot talk about about rapidly changing information (like something that happened last week) since they won't have up-to-date information from after their release data.

One solution to all these problems with simple prompting for generating factual text is to give a language model external sources of knowledge, for example proprietary texts like medical or legal records, personal emails, or corporate documents, and to use those documents in answering questions. This method is called **retrieval-augmented generation** or **RAG**, and that is the method we will focus on in this chapter. In RAG we use **information retrieval** (**IR**) techniques to retrieve documents that are likely to have information that might help answer the question. Then we use a large language model to **generate** an answer given these documents.

RAG

information retrieval

Basing our answers on retrieved documents can solve some of the problems with using simple prompting to answer questions. First, it helps ensure that the answer is grounded in facts from some curated dataset. And the system can give the user the answer accompanied by the context of the passage or document it came from. This information can help users have confidence in the accuracy of the answer (or help them spot when it is wrong!). And these retrieval techniques can be used on any proprietary data we want, such as legal or medical data for those applications.

We'll begin by introducing information retrieval, the task of choosing the most relevant document from a document set given a user's query expressing their information need. We'll see the classic method based on cosines of sparse tf-idf vectors, modern neural 'dense' retrievers based on instead representing queries and documents neurally with BERT or other language models. We then introduce the retrieval-augmented generation paradigm.

Finally, we'll discuss various datasets with questions and answers that can be used for finetuning LLMs in instruction tuning and for use as benchmarks for evaluation.

11.1 Information Retrieval

information retrieval
IR

Information retrieval or **IR** is the name of the field encompassing the retrieval of all manner of media based on user information needs. The resulting IR system is often called a **search engine**. Our goal in this section is to give a sufficient overview of IR to see its application to large language models meeting user information needs. Readers with more interest specifically in information retrieval should see the Historical Notes section at the end of the chapter.

ad hoc retrieval
document
collection
web search
term
query

The IR task we consider is called **ad hoc retrieval**, in which a user poses a **query** to a retrieval system, which then returns an ordered set of **documents** from some **collection**. A **document** refers to whatever unit of text the system indexes and retrieves (web pages, scientific papers, news articles, or even shorter passages like paragraphs). A **collection** refers to a set of documents being used to satisfy user requests. A collection can mean the entire web, in which case we are doing **web search**. But a collection can also be a smaller corporate repo, or even a set of documents used by one person. **term** refers to a word in a collection, but it may also include phrases. Finally, a **query** represents a user's information need expressed as a set of terms.

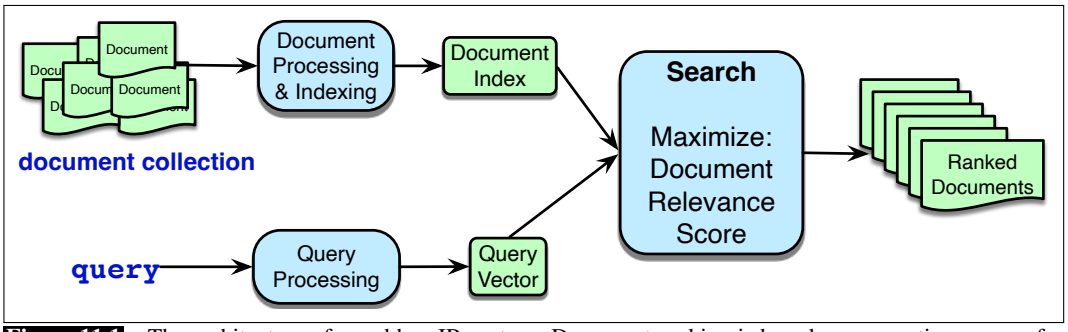


Figure 11.1 The architecture of an ad hoc IR system. Document ranking is based on computing a score for each candidate document given the query, expressing how **relevant** it is likely to be to meet the users information need. There are two classes of IR systems, based on the two classes of vectors that are used to represent queries and documents: sparse vectors and dense vectors. These two kinds of retrieval differ in the details of the indexing and scoring mechanisms.

The high-level architecture of an ad hoc retrieval engine is shown in Fig. 11.1. This figure abstracts over the two classes of IR systems, which are based on the two classes of vectors that are used to represent queries and documents: sparse vectors and dense vectors. In sparse retrieval, we represent documents and queries with **count vectors**, weighted by tf-idf or BM25. In dense retrieval, we represent

documents and queries with **embeddings**, computed from language models (either encoder or decoder models). We'll discuss sparse retrieval in the rest of this section, and turn to dense retrieval in Section 11.3.

11.1.1 Representing documents as vectors

vector space model

In the **vector space model** of information retrieval (Salton, 1971) a document is represented as a vector of counts of the words it contains.

bag of words

We sometimes call this kind of model a **bag-of-words** model. Fig. 11.2 shows the intuition: we are representing a text document as if it were a **bag of words**, that is, an unordered set of words with their position ignored, keeping only their frequency in the document. In the example in the figure, instead of representing the word order in all the phrases like "It manages to be whimsical and romantic", we simply note that the word *it* occurred 5 times in the entire excerpt, the words *love*, *recommend*, and *movie* once, and so on.

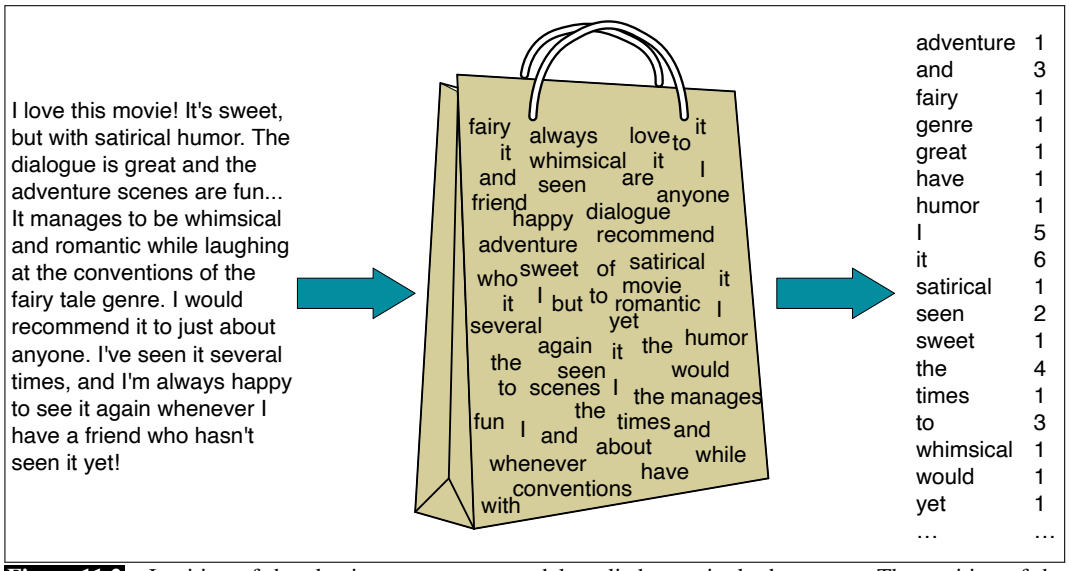


Figure 11.2 Intuition of the classic vector space model applied to a single document. The position of the words is ignored (the *bag-of-words* assumption) and we make use of the frequency of each word.

We could thus imagine representing the document in Fig. 11.2 as the vector [1 3 1 1 1 1 5 6 1 2 1 4 1 3 1 1 1] (if we limited ourselves to these 18 dimensions and ignored all the other words in English).

term-document matrix

More generally, we can represent a set of documents as a **term-document matrix** in which each row represents a word in the vocabulary and each column represents a document from some collection of documents. Fig. 11.3 shows a small selection from a term-document matrix showing the occurrence of four words in four plays by Shakespeare. Each cell in this matrix represents the number of times a particular word (defined by the row) occurs in a particular document (defined by the column). Thus *fool* appeared 58 times in *Twelfth Night*.

A document is represented as a count vector, a column in Fig. 11.4. In the example in Fig. 11.4, we've chosen to make the document vectors of dimension 4, just so they fit on the page; in real term-document matrices, the document vectors would have dimensionality $|V|$, the vocabulary size. The first dimension for both these vectors corresponds to the number of times the word *battle* occurs, and we can

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Figure 11.3 The term-document matrix for four words in four Shakespeare plays. Each cell contains the number of times the (row) word occurs in the (column) document.

compare each dimension, noting for example that the vectors for *As You Like It* and *Twelfth Night* have similar values (1 and 0, respectively) for the first dimension.

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Figure 11.4 The term-document matrix for four words in four Shakespeare plays. The red boxes show that each document is represented as a column vector of length four.

Since 4-dimensional spaces are hard to visualize, Fig. 11.5 shows a visualization of the four document vectors in two dimensions; we've arbitrarily chosen the dimensions corresponding to the words *battle* and *fool*.

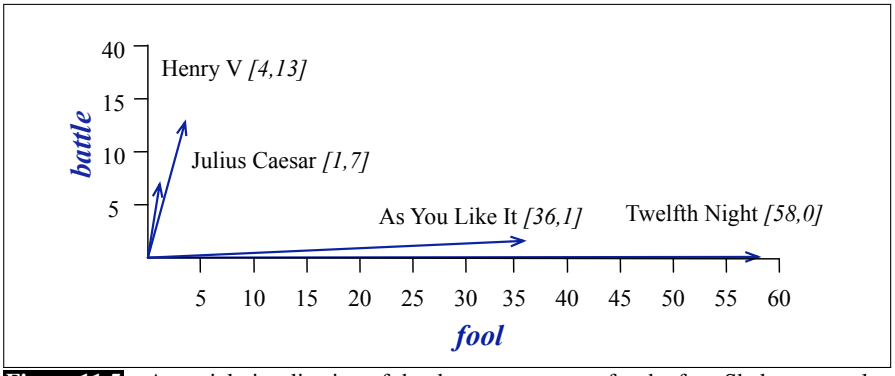


Figure 11.5 A spatial visualization of the document vectors for the four Shakespeare play documents, showing just two of the dimensions, corresponding to the words *battle* and *fool*. The comedies have high values for the *fool* dimension and low values for the *battle* dimension.

Two documents that are similar will tend to have similar words, and if two documents have similar words their column vectors will tend to be similar. The vectors for the comedies *As You Like It* [1,114,36,20] and *Twelfth Night* [0,80,58,15] look a lot more like each other (more fools and wit than battles) than they look like *Julius Caesar* [7,62,1,2] or *Henry V* [13,89,4,3].

11.1.2 Term weighting: tf-idf and BM25

In fact, in IR, we don't use raw word counts like [1 114 36 20] for *As You Like It*, or [1 3 1 1 1 1 1 5 6 1 2 1 4 1 3 1 1 1] for the document in Fig. 11.2. Instead we compute a **term weight** for each document word. Two term weighting schemes are common: **tf-idf** and a variant of tf-idf called **BM25**.

term weight
BM25

Tf-idf (the '-' here is a hyphen, not a minus sign) is the product of two terms, the term frequency **tf** and the inverse document frequency **idf**.

The **term frequency** term tells us how frequent the word is; words that occur more often in a document are likely to be informative about the document's contents. We usually use the $\log_{10}$ of the word frequency, rather than the raw count. The intuition is that a word appearing 100 times in a document doesn't make that word 100 times more likely to be relevant to the meaning of the document. We also need to do something special with counts of 0, since we can't take the log of 0.[3] So if we define count(t,d) as the raw count of term t in document d, then $\text{tf}_{t,d}$, the tf of term t in document d is

$$\text{tf}_{t,d} = \begin{cases} 1+\log_{10}\text{count}(t,d) & \text{if } \text{count}(t,d) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{11.4}$$

If we use log weighting, terms which occur 0 times in a document would have $\text{tf} = 0$, 1 times in a document $\text{tf} = 1+\log_{10}(1) = 1+0 = 1$, 10 times in a document $\text{tf} = 1+\log_{10}(10) = 2$, 100 times $\text{tf} = 1+\log_{10}(100) = 3$, 1000 times $\text{tf} = 4$, and so on.

The **document frequency** df_t of a term t is the number of documents it occurs in. Terms that occur in only a few documents are useful for discriminating those documents from the rest of the collection; terms that occur across the entire collection aren't as helpful. The **inverse document frequency** or **idf** term weight (Sparck Jones, 1972) is defined as:

$$\text{idf}_t = \log_{10}\frac{N}{\text{df}_t} \tag{11.5}$$

where N is the total number of documents in the collection, and df_t is the number of documents in which term t occurs. The fewer documents in which a term occurs, the higher this weight; the lowest weight of 0 is assigned to terms that occur in every document.

Here are some idf values for some words in the corpus of Shakespeare plays, ranging from extremely informative words that occur in only one play like *Romeo*, to those that occur in a few like *salad* or *Falstaff*, to those that are very common like *fool* or so common as to be completely non-discriminative since they occur in all 37 plays like *good* or *sweet*.[4]

Word	df	idf
Romeo	1	1.57
salad	2	1.27
Falstaff	4	0.967
forest	12	0.489
battle	21	0.246
wit	34	0.037
fool	36	0.012
good	37	0
sweet	37	0

The **tf-idf** value for word t in document d is then the product of term frequency $\text{tf}_{t,d}$ and IDF:

$$\text{tf-idf}(t,d) = \text{tf}_{t,d} \cdot \text{idf}_t \tag{11.6}$$

[3] We can also use this alternative formulation, which we have used in earlier editions: $\text{tf}_{t,d} = \log_{10}(\text{count}(t,d)+1)$

[4] *Sweet* was one of Shakespeare's favorite adjectives, a fact probably related to the increased use of sugar in European recipes around the turn of the 16th century (Jurafsky, 2014, p. 175).

11.1.3 Document Scoring

Once we have represented each document and query as a weighted vector, we need to score each document. Our goal is to measure the **relevance** of the document to the user's information need, as expressed in their query. In the classic tf-idf model we estimate this relevance of a document by measuring its geometric similarity in vector space to the query. That is, we make the simplifying assumption that documents that have **similar words** to the query are more relevant to the user.

We use the cosine similarity function introduced in Chapter 5, scoring document d by the cosine of its vector $\mathbf{d}$ with the query vector $\mathbf{q}$:

$$\text{score}(q,d) = \cos(\mathbf{q},\mathbf{d}) = \frac{\mathbf{q}\cdot\mathbf{d}}{|\mathbf{q}||\mathbf{d}|} \tag{11.7}$$

Another way to think of the cosine computation is as the dot product of unit vectors; we can first normalize both the query and document vector to unit vectors, by dividing by their lengths, and then take the dot product:

$$\text{score}(q,d) = \cos(\mathbf{q},\mathbf{d}) = \frac{\mathbf{q}}{|\mathbf{q}|}\cdot\frac{\mathbf{d}}{|\mathbf{d}|} \tag{11.8}$$

We can spell out Eq. 11.8, using the tf-idf values and spelling out the dot product as a sum of products:

$$\text{score}(q,d) = \sum_{t\in\mathbf{q}} \frac{\text{tf-idf}(t,q)}{\sqrt{\sum_{q_i\in q}\text{tf-idf}^2(q_i,q)}} \cdot \frac{\text{tf-idf}(t,d)}{\sqrt{\sum_{d_i\in d}\text{tf-idf}^2(d_i,d)}} \tag{11.9}$$

Now let's use Eq. 11.9 to walk through an example of a tiny query against a collection of 4 nano documents, computing tf-idf values and seeing the rank of the documents. We'll assume all words in the following query and documents are downcased and punctuation is removed:

Query:	sweet love
Doc 1:	Sweet sweet nurse! Love?
Doc 2:	Sweet sorrow
Doc 3:	How sweet is love?
Doc 4:	Nurse!

Fig. 11.6 shows the computation of the tf-idf cosine between the query and Document 1, and the query and Document 2. The cosine is the normalized dot product of tf-idf values, so for the normalization we must need to compute the document vector lengths $|q|$, $|d_1|$, and $|d_2|$ for the query and the first two documents using Eq. 11.4, Eq. 11.5, Eq. 11.6, and Eq. 11.9 (computations for Documents 3 and 4 are also needed but are left as an exercise for the reader). The dot product between the vectors is the sum over dimensions of the product, for each dimension, of the values of the two tf-idf vectors for that dimension. This product is only non-zero where both the query and document have non-zero values, so for this example, in which only *sweet* and *love* have non-zero values in the query, the dot product will be the sum of the products of those elements of each vector.

Document 1 has a higher cosine with the query (0.747) than Document 2 has with the query (0.0779), and so the tf-idf cosine model would rank Document 1 above Document 2. This ranking is intuitive given the vector space model, since Document 1 has both terms including two instances of *sweet*, while Document 2 is

	Query					
word	**cnt**	**tf**	**df**	**idf**	**tf-idf**	**n'lized** = tf-idf/$\|q\|$
sweet	1	1	3	0.125	0.125	0.383
nurse	0	0	2	0.301	0	0
love	1	1	2	0.301	0.301	0.924
how	0	0	1	0.602	0	0
sorrow	0	0	1	0.602	0	0
is	0	0	1	0.602	0	0
$\|q\| = \sqrt{.125^2 + .301^2} = .326$						

	Document 1					Document 2				
word	**cnt**	**tf**	**tf-idf**	**n'lized**	**× q**	**cnt**	**tf**	**tf-idf**	**n'lized**	**×q**
sweet	2	1.301	0.163	0.357	**0.137**	1	1.000	0.125	0.203	**0.0779**
nurse	1	1.000	0.301	0.661	0	0	0	0	0	0
love	1	1.000	0.301	0.661	**0.610**	0	0	0	0	**0**
how	0	0	0	0	0	0	0	0	0	0
sorrow	0	0	0	0	0	1	1.000	0.602	0.979	0
is	0	0	0	0	0	0	0	0	0	0
$\|d_1\| = \sqrt{.163^2 + .301^2 + .301^2} = .456$						$\|d_2\| = \sqrt{.125^2 + .602^2} = .615$				
Cosine: $\sum$ of column: **0.747**						Cosine: $\sum$ of column: **0.0779**				

Figure 11.6 Computation of tf-idf cosine score between the query and nano-documents 1 (0.747) and 2 (0.0779), using Eq. 11.4, Eq. 11.5, Eq. 11.6 and Eq. 11.9.

missing one of the terms. We leave the computation for Documents 3 and 4 as an exercise for the reader.

In practice, there are many variants and approximations to Eq. 11.9. For example, we might choose to simplify processing by removing some terms. To see this, let's start by expanding the formula for tf-idf in Eq. 11.9 to explicitly mention the tf and idf terms from Eq. 11.6:

$$\text{score}(q,d) = \sum_{t \in \mathbf{q}} \frac{\text{tf}_{t,q} \cdot \text{idf}_t}{\sqrt{\sum_{q_i \in q} \text{tf-idf}^2(q_i,q)}} \cdot \frac{\text{tf}_{t,d} \cdot \text{idf}_t}{\sqrt{\sum_{d_i \in d} \text{tf-idf}^2(d_i,d)}} \tag{11.10}$$

In one common variant of tf-idf cosine, for example, we drop the idf term for the document. Eliminating the second copy of the idf term (since the identical term is already computed for the query) turns out to sometimes result in better performance:

$$\text{score}(q,d) = \sum_{t \in \mathbf{q}} \frac{\text{tf}_{t,q} \cdot \text{idf}_t}{\sqrt{\sum_{q_i \in q} \text{tf-idf}^2(q_i,q)}} \cdot \frac{\text{tf}_{t,d} \cdot \text{idf}_t}{\sqrt{\sum_{d_i \in d} \text{tf-idf}^2(d_i,d)}} \tag{11.11}$$

Other variants of tf-idf eliminate various other terms.

BM25 A slightly more complex variant in the tf-idf family is the **BM25** weighting scheme (sometimes called Okapi BM25 after the Okapi IR system in which it was introduced (Robertson et al., 1995)). BM25 adds two parameters: k, a knob that adjusts the balance between term frequency and IDF, and b, which controls the importance of document length normalization. The BM25 score of a document d given

a query q is:

$$\sum_{t\in q} \overbrace{\log\left(\frac{N}{df_t}\right)}^{\text{IDF}} \overbrace{\frac{tf_{t,d}}{k\left(1-b+b\left(\frac{|d|}{|d_{\text{avg}}|}\right)\right)+tf_{t,d}}}^{\text{weighted tf}} \tag{11.12}$$

where $|d_{\text{avg}}|$ is the length of the average document. When k is 0, BM25 reverts to no use of term frequency, just a binary selection of terms in the query (plus idf). A large k results in raw term frequency (plus idf). b ranges from 1 (scaling by document length) to 0 (no length scaling). Manning et al. (2008) suggest reasonable values are k = [1.2,2] and b = 0.75. Kamphuis et al. (2020) is a useful summary of the many minor variants of BM25.

Stop words In the past it was common to remove high-frequency words from both the query and document before representing them. The list of such high-frequency words to be removed is called a **stop list**. The intuition is that high-frequency terms (often function words like *the*, *a*, *to*) carry little semantic weight and may not help with retrieval, and can also help shrink the inverted index files we describe below. The downside of using a stop list is that it makes it difficult to search for phrases that contain words in the stop list. For example, common stop lists would reduce the phrase *to be or not to be* to the phrase *not*. In modern IR systems, the use of stop lists is much less common, partly due to improved efficiency and partly because much of their function is already handled by IDF weighting, which downweights function words that occur in every document. Nonetheless, stop word removal is occasionally useful in various NLP tasks so is worth keeping in mind.

stop list

11.1.4 Efficiently finding documents: the Inverted Index

In order to compute scores, we need to efficiently find documents that contain words in the query. (Any document that contains none of the query terms will have a score of 0 and can be ignored.) The basic search problem in IR is thus to find all documents $d \in C$ that contain a term $q \in Q$.

inverted index

The data structure for this task is the **inverted index**, which we use for making this search efficient, and also conveniently storing useful information like the document frequency and the count of each term in each document.

postings

An inverted index, given a query term, gives a list of documents that contain the term. It consists of two parts, a **dictionary** and the **postings**. The dictionary is a list of terms (designed to be efficiently accessed), each pointing to a **postings list** for the term. A postings list is the list of document IDs associated with each term, which can also contain information like the term frequency or even the exact positions of terms in the document. The dictionary can also store the document frequency for each term. For example, a simple inverted index for our 4 sample documents above, with each word containing its document frequency in {}, and a pointer to a postings list that contains document IDs and term counts in [], might look like the following:

how {1} → 3 [1]
is {1} → 3 [1]
love {2} → 1 [1] → 3 [1]
nurse {2} → 1 [1] → 4 [1]
sorrow {1} → 2 [1]
sweet {3} → 1 [2] → 2 [1] → 3 [1]

Given a list of terms in query, we can very efficiently get lists of all candidate documents, together with the information necessary to compute the tf-idf scores we need.

11.2 Evaluation of Information-Retrieval Systems

We measure the performance of ranked retrieval systems using the same **precision** and **recall** metrics we have been using. We make the assumption that each document returned by the IR system is either **relevant** to our purposes or **not relevant**. Precision is the fraction of the returned documents that are relevant, and recall is the fraction of all relevant documents that are returned. More formally, let's assume a system returns T ranked documents in response to an information request, a subset R of these are relevant, a disjoint subset, N, are the remaining irrelevant documents, and U documents in the collection as a whole are relevant to this request. Precision and recall are then defined as:

$$Precision = \frac{|R|}{|T|} \qquad Recall = \frac{|R|}{|U|} \tag{11.13}$$

Unfortunately, these metrics don't adequately measure the performance of a system that *ranks* the documents it returns. If we are comparing the performance of two ranked retrieval systems, we need a metric that prefers the one that ranks the relevant documents higher. We need to adapt precision and recall to capture how well a system does at putting relevant documents higher in the ranking.

Let's turn to an example. Assume the table in Fig. 11.7 gives rank-specific precision and recall values calculated as we proceed down through a set of ranked documents for a particular query; the precisions are the fraction of relevant documents seen at a given rank, and recalls the fraction of relevant documents found at the same rank. The recall measures in this example are based on this query having 9 relevant documents in the collection as a whole.

Note that recall is non-decreasing; when a relevant document is encountered, recall increases, and when a non-relevant document is found it remains unchanged. Precision, on the other hand, jumps up and down, increasing when relevant documents are found, and decreasing otherwise. The most common way to visualize precision and recall is to plot precision against recall in a **precision-recall curve**, like the one shown in Fig. 11.8 for the data in table 11.7.

precision-recall curve

Fig. 11.8 shows the values for a single query. But we'll need to combine values for all the queries, and in a way that lets us compare one system to another. One way of doing this is to plot averaged precision values at 11 fixed levels of recall (0 to 100, in steps of 10). Since we're not likely to have datapoints at these exact levels, we use **interpolated precision** values for the 11 recall values from the data points we do have. We can accomplish this by choosing the maximum precision value achieved at any level of recall at or above the one we're calculating. In other words,

interpolated precision

$$\text{IntPrecision}(r) = \max_{i>=r} \text{Precision}(i) \tag{11.14}$$

This interpolation scheme not only lets us average performance over a set of queries, but also helps smooth over the irregular precision values in the original data. It is designed to give systems the benefit of the doubt by assigning the maximum precision value achieved at higher levels of recall from the one being measured. Fig. 11.9 and Fig. 11.10 show the resulting interpolated data points from our example.

Rank	Judgment	Precision_{Rank}	Recall_{Rank}
1	R	1.0	.11
2	N	.50	.11
3	R	.66	.22
4	N	.50	.22
5	R	.60	.33
6	R	.66	.44
7	N	.57	.44
8	R	.63	.55
9	N	.55	.55
10	N	.50	.55
11	R	.55	.66
12	N	.50	.66
13	N	.46	.66
14	N	.43	.66
15	R	.47	.77
16	N	.44	.77
17	N	.41	.77
18	R	.44	.88
19	N	.42	.88
20	N	.40	.88
21	N	.38	.88
22	N	.36	.88
23	N	.35	.88
24	N	.33	.88
25	R	.36	1.0

Figure 11.7 Rank-specific precision and recall values calculated as we proceed down through a set of ranked documents (assuming the collection has 9 relevant documents).

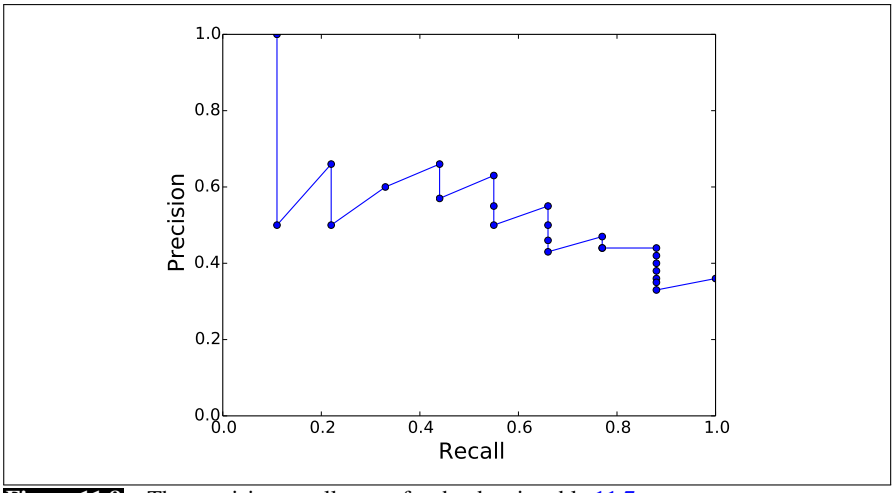


Figure 11.8 The precision recall curve for the data in table 11.7.

Given curves such as that in Fig. 11.10 we can compare two systems or approaches by comparing their curves. Clearly, curves that are higher in precision across all recall values are preferred. However, these curves can also provide insight into the overall behavior of a system. Systems that are higher in precision toward the left may favor precision over recall, while systems that are more geared towards recall will be higher at higher levels of recall (to the right).

mean average precision

A second way to evaluate ranked retrieval is **mean average precision** (MAP),

Interpolated Precision	Recall
1.0	0.0
1.0	.10
.66	.20
.66	.30
.66	.40
.63	.50
.55	.60
.47	.70
.44	.80
.36	.90
.36	1.0

Figure 11.9 Interpolated data points from Fig. 11.7.

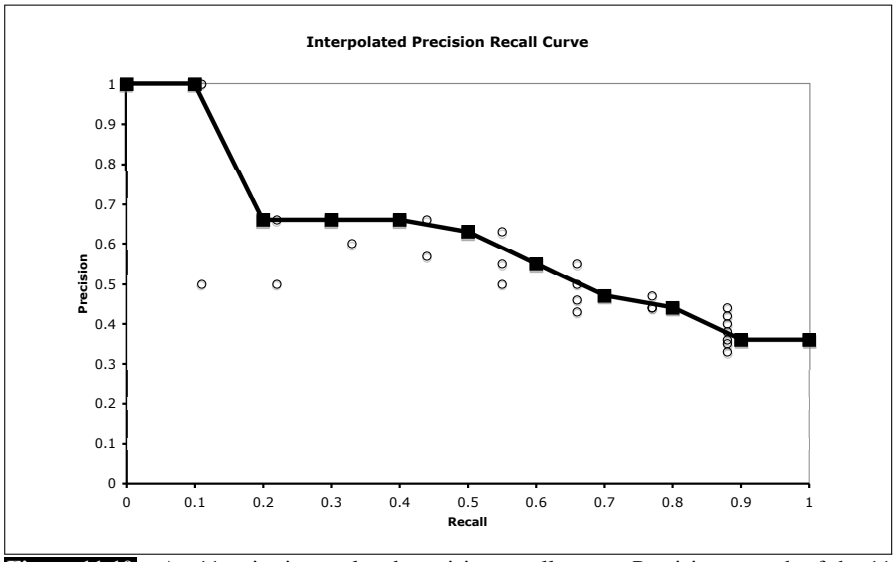


Figure 11.10 An 11 point interpolated precision-recall curve. Precision at each of the 11 standard recall levels is interpolated for each query from the maximum at any higher level of recall. The original measured precision recall points are also shown.

which provides a single metric that can be used to compare competing systems or approaches. In this approach, we again descend through the ranked list of items, but now we note the precision **only** at those points where a relevant item has been encountered (for example at ranks 1, 3, 5, 6 but not 2 or 4 in Fig. 11.7). For a single query, we average these individual precision measurements over the return set (up to some fixed cutoff). More formally, if we assume that R_r is the set of relevant documents at or above r, then the **average precision** (**AP**) for a single query is

$$\text{AP} = \frac{1}{|R_r|} \sum_{d \in R_r} \text{Precision}_r(d) \tag{11.15}$$

where $Precision_r(d)$ is the precision measured at the rank at which document d was found. For an ensemble of queries Q, we then average over these averages, to get our final MAP measure:

$$\text{MAP} = \frac{1}{|Q|} \sum_{q \in Q} \text{AP}(q) \tag{11.16}$$

The MAP for the single query (hence = AP) in Fig. 11.7 is 0.6.

11.3 Information Retrieval with Dense Vectors

The classic tf-idf or BM25 algorithms for IR have long been known to have a conceptual flaw: they work only if there is exact overlap of words between the query and document. In other words, the user posing a query (or asking a question) needs to guess exactly what words the writer of the answer might have used, an issue called the **vocabulary mismatch problem** (Furnas et al., 1987).

The solution to this problem is to use an approach that can handle synonymy: instead of (sparse) word-count vectors, using (dense) embeddings. This idea was first proposed for retrieval in the last century under the name of Latent Semantic Indexing approach (Deerwester et al., 1990), but is implemented in modern times via encoders like BERT.

The most powerful approach is to present both the query and the document to a single encoder, allowing the transformer self-attention to see all the tokens of both the query and the document, and thus building a representation that is sensitive to the meanings of both query and document. Then a linear layer can be put on top of the [CLS] token to predict a similarity score for the query/document tuple:

$$\begin{aligned} \mathbf{z} &= \text{BERT}(\text{q}; \texttt{[SEP]}; \text{d})\texttt{[CLS]} \\ \text{score}(q,d) &= \text{softmax}(\mathbf{U}(\mathbf{z})) \end{aligned} \tag{11.17}$$

This architecture is shown in Fig. 11.11a. Usually the retrieval step is not done on an entire document. Instead documents are broken up into smaller passages, such as non-overlapping fixed-length chunks of say 100 tokens, and the retriever encodes and retrieves these passages rather than entire documents. The query and document have to be made to fit in the BERT 512-token window, for example by truncating the query to 64 tokens and truncating the document if necessary so that it, the query, [CLS], and [SEP] fit in 512 tokens. The BERT system together with the linear layer $\mathbf{U}$ can then be fine-tuned for the relevance task by gathering a tuning dataset of relevant and non-relevant passages.

The problem with the full BERT architecture in Fig. 11.11a is the expense in computation and time. With this architecture, every time we get a query, we have to pass every single document in our entire collection through a BERT encoder jointly with the new query! This enormous use of resources is impractical for real cases.

At the other end of the computational spectrum is a much more efficient architecture, the **bi-encoder**. In this architecture we can encode the documents in the collection only one time by using two separate encoder models, one to encode the query and one to encode the document. We encode each document, and store all the encoded document vectors in advance. When a query comes in, we encode just this query and then use the dot product between the query vector and the precomputed document vectors as the score for each candidate document (Fig. 11.11b). For example, if we used BERT, we would have two encoders BERT_Q and BERT_D and we could represent the query and document as the [CLS] token of the respective encoders (Karpukhin et al., 2020):

$$\begin{aligned} \mathbf{z}_q &= \text{BERT}_Q(\text{q})\texttt{[CLS]} \\ \mathbf{z}_d &= \text{BERT}_D(\text{d})\texttt{[CLS]} \\ \text{score}(q,d) &= \mathbf{z}_q \cdot \mathbf{z}_d \end{aligned} \tag{11.18}$$

The bi-encoder is much cheaper than a full query/document encoder, but is also less accurate, since its relevance decision can't take full advantage of all the possi-

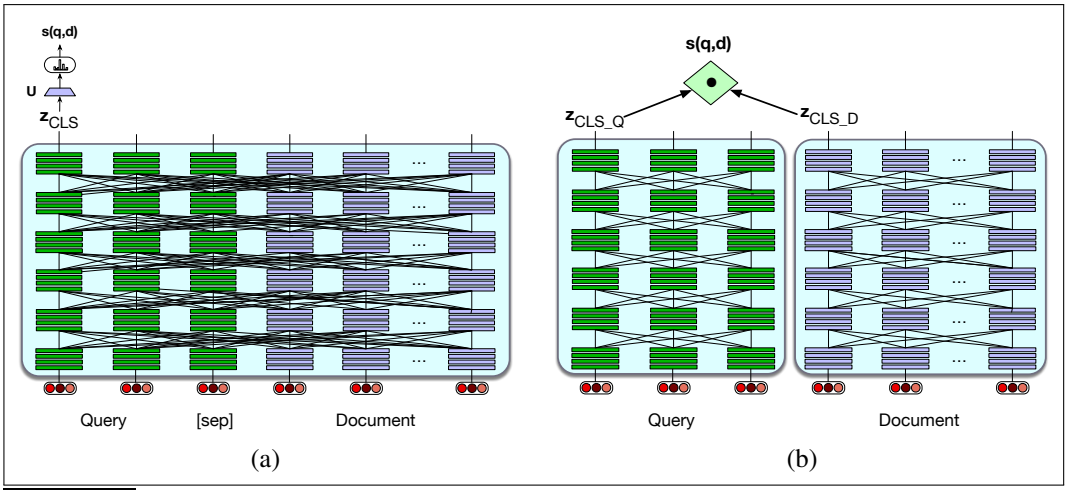


Figure 11.11 Two ways to do dense retrieval, illustrated by using lines between layers to schematically represent self-attention: (a) Use a single encoder to jointly encode query and document and finetune to produce a relevance score with a linear layer over the CLS token. This is too compute-expensive to use except in rescoring (b) Use separate encoders for query and document, and use the dot product between CLS token outputs for the query and document as the score. This is less compute-expensive, but not as accurate.

ble meaning interactions between all the tokens in the query and the tokens in the document.

There are numerous approaches that lie in between the full encoder and the bi-encoder. One intermediate alternative is to use cheaper methods (like BM25) as the first pass relevance ranking for each document, take the top N ranked documents, and use expensive methods like the full BERT scoring to rerank only the top N documents rather than the whole set.

ColBERT

Another intermediate approach is the **ColBERT** approach of Khattab and Zaharia (2020) and Khattab et al. (2021), shown in Fig. 11.12. This method separately encodes the query and document, but rather than encoding the entire query or document into one vector, it separately encodes each of them into contextual representations for each token. These BERT representations of each document word can be pre-stored for efficiency. The relevance score between a query q and a document d is a sum of maximum similarity (MaxSim) operators between tokens in q and tokens in d. Essentially, for each token in q, ColBERT finds the most contextually similar token in d, and then sums up these similarities. A relevant document will have tokens that are contextually very similar to the query.

More formally, a question q is tokenized as $[q_1,\ldots,q_n]$, prepended with a `[CLS]` and a special `[Q]` token, truncated to N=32 tokens (or padded with `[MASK]` tokens if it is shorter), and passed through BERT to get output vectors $\mathbf{q} = [\mathbf{q}_1,\ldots,\mathbf{q}_N]$. The passage d with tokens $[d_1,\ldots,d_m]$, is processed similarly, including a `[CLS]` and special `[D]` token. A linear layer is applied on top of $\mathbf{d}$ and $\mathbf{q}$ to control the output dimension, so as to keep the vectors small for storage efficiency, and vectors are rescaled to unit length, producing the final vector sequences $\mathbf{E}_q$ (length N) and $\mathbf{E}_d$ (length m). The ColBERT scoring mechanism is:

$$\text{score}(q,d) = \sum_{i=1}^{N} \max_{j=1}^{m} \mathbf{E}_{q_i} \cdot \mathbf{E}_{d_j} \tag{11.19}$$

While the interaction mechanism has no tunable parameters, the ColBERT ar-

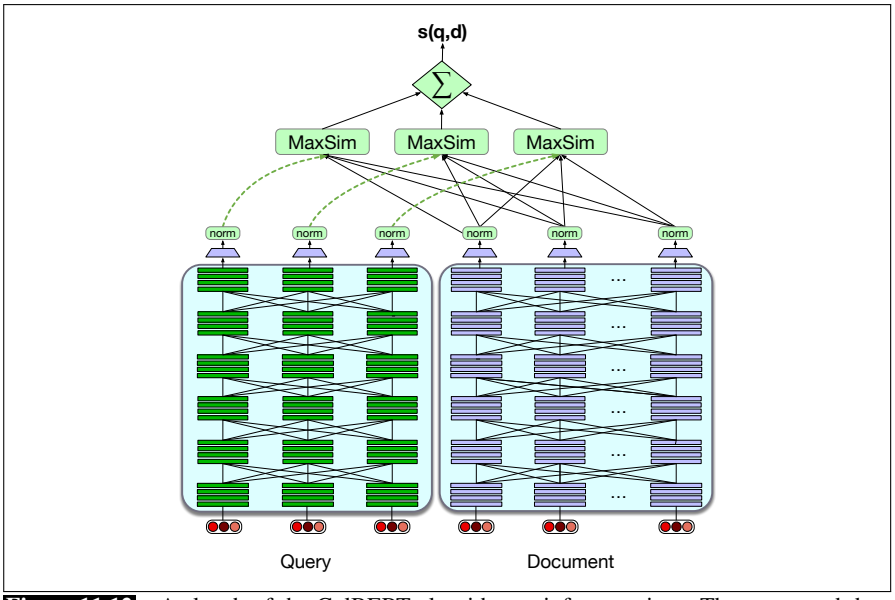


Figure 11.12 A sketch of the ColBERT algorithm at inference time. The query and document are first passed through separate BERT encoders. Similarity between query and document is computed by summing a soft alignment between the contextual representations of tokens in the query and the document. Training is end-to-end. (Various details aren't depicted; for example the query is prepended by a `[CLS]` and `[Q:]` tokens, and the document by `[CLS]` and `[D:]` tokens). Figure adapted from Khattab and Zaharia (2020).

chitecture still needs to be trained end-to-end to fine-tune the BERT encoders and train the linear layers (and the special `[Q]` and `[D]` embeddings) from scratch. It is trained on triples $\langle q, d^+, d^- \rangle$ of query q, positive document d^+ and negative document d^- to produce a score for each document using Eq. 11.19, optimizing model parameters using a cross-entropy loss.

All the supervised algorithms (like ColBERT or the full-interaction version of the BERT algorithm applied for reranking) need training data in the form of queries together with relevant and irrelevant passages or documents (positive and negative examples). There are various semi-supervised ways to get labels; some datasets (like MS MARCO Ranking, Section 11.5) contain gold positive examples. Negative examples can be sampled randomly from the top-1000 results from some existing IR system. If datasets don't have labeled positive examples, iterative methods like **relevance-guided supervision** can be used (Khattab et al., 2021) which rely on the fact that many datasets contain short answer strings. In this method, an existing IR system is used to harvest examples that do contain short answer strings (the top few are taken as positives) or don't contain short answer strings (the top few are taken as negatives), these are used to train a new retriever, and then the process is iterated.

Efficiency is an important issue, since every possible document must be ranked for its similarity to the query. For sparse word-count vectors, the inverted index allows this very efficiently. For dense vector algorithms finding the set of dense document vectors that have the highest dot product with a dense query vector is an instance of the problem of **nearest neighbor search**. Modern systems therefore make use of approximate nearest neighbor vector search algorithms like **Faiss** (Johnson et al., 2017).

Faiss

11.4 Retrieval-Augmented Generation (RAG)

The information retrieval techniques we introduced in the prior section can be integrated into language models via a method called **retrieval-augmented generation** or **RAG**. In the basic RAG scenario that we will describe in this section, we use IR techniques to retrieve documents from some specified store of documents that are likely to have useful information. Then we use a large language model to **generate** an answer conditioned on these documents in addition to the original query.

As we summarized in the introduction to the chapter, there are many goals of retrieval-augmented generation. RAG can help mitigate hallucination, by giving the model a set of trusted documents. RAG can also help language models generate factual text about proprietary data, like personal email, or health records, or company-internal documents, or other legal documents. RAG can also help with the problem that knowledge is dynamic and time-sensitive, for example if we know the user's information need references data from a time after a language model was trained.

A RAG system is based on two major components: the **retriever** and the **generator**, (the latter is sometimes called, for historical reasons, the **reader** (Chen et al., 2017a)). Fig. 11.13 sketches out this standard model for answering questions.

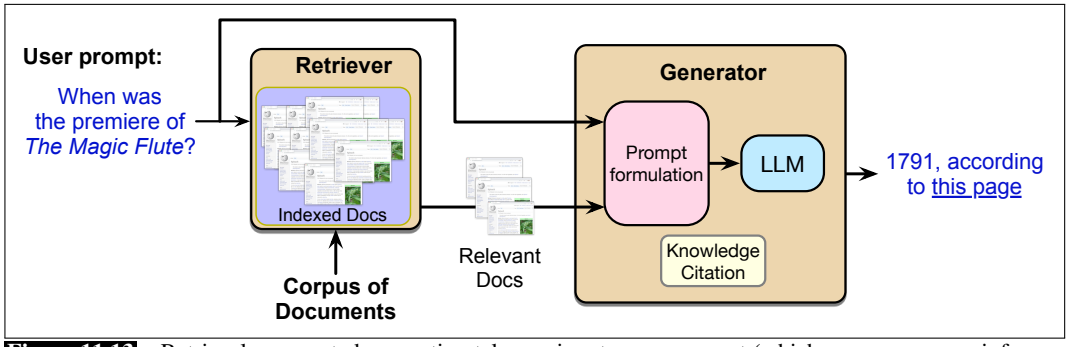


Figure 11.13 Retrieval-augmented generation takes as input a user prompt (which may express an information need like this question example), and a corpus of documents that may be useful in meeting the information need. The method has two stages: **retrieval**, which returns relevant documents from the collection, and **generation**, in which an LLM **generates** text given the documents as a prompt. Some generations include a **knowledge citation** that can help the user decide whether to trust the generation, or follow up if they are interested.

retrieval-augmented generation RAG

In the first stage of the **retrieval-augmented generation**, or **RAG** model shown in Fig. 11.13 we **retrieve** relevant passages from some prespecified text collection, for example using the dense retrievers of the previous section. In the second **generate** stage, we take the set of retrieved passages, integrate it with the user prompt, and pass some version of these to a large language model to generate an answer conditioned on these two things.

For example imagine the user asks the question `What year was the premiere of *The Magic Flute*?`. We pass this question to a dense retriever and return a series of passages about The Magic Flute.

The idea of retrieval-augmented generation is to condition on the retrieved passages, jointly with some prompt text, for example like "Based on these texts, answer this question:". Thus given a document collection $\mathcal{D}$ and a user query q, the most basic RAG algorithm is:

1. Call a retriever to return $R(q) = d_1 \cdots d_k$, the top-k relevant passages from $\mathcal{D}$

2. Create a prompt that includes q and the retrieved passages
3. Call an LLM with the prompt

The resulting prompts might look something like:

Schematic of a RAG Prompt

```
retrieved passage 1

retrieved passage 2

...

retrieved passage k

Based on these texts, answer this question:  What year
was the premiere of The Magic Flute?
```

The task for the language model is then to generate text according to this probability model:

$$p(x_1,\ldots,x_n) \;=\; \prod_{i=1}^{n} p(x_i|\mathrm{R}(q)\,;\,\text{Answer the following question...}\;;\,q\,;x_{<i})$$

There are many augmentations of this basic RAG paradigm. One addition is the use of **agent-based RAG**. In the RAG paradigm described so far, a search is always run and then retrieved passages are combined with the user's question in a prompt. But in actual applications, we may not want to run retrieval for every user turn. Or we may want to retrieve from different collections for different user needs (sometimes the web, other times a private collection). In **agent-based RAG**, the system decides when to call a retrieval agent and for which collection.

Another research area has to do with the relationship between the retriever and the generator. For example there may be noise in the retrieved passages; some of them may be irrelevant or wrong, or in an unhelpful order. How can we encourage the LLM to focus on the good passages? Some RAG architectures add a reranker that reranks or reorders passages after they are retrieved. Or some complex questions may require multi-hop architectures, in which a query is used to retrieve documents, which are then appended to the original query for a second stage of retrieval.

Another class of solutions is to train the LLM for RAG. The basic version of RAG describe above involves no training; we take an off-the-shelf LLM, and give it the passages and a prompt and hope that it will correctly figure out which passages are useful or relevant in generating the answer. One learning variant involves instruction-tuning an LLM, by first creating a dataset of questions annotated with retrieved passages and correct answers, and then instruction-tuning the LLM to correctly answer the questions from the passages. An alternative method is to do this via test-time compute, prompting the LLM to answer the question and simultaneously to generate reflections on which passages were useful. The process of generating these reflections may lead the LLM to improve at identifying good passages. The resulting reflection text can also be used for in-context learning, for example by using the text as part of a prompt for further questions.

In addition to training the LLM, we could train the IR engine. After all, the IR engine itself has not been optimized for the RAG scenario. It might not have been

trained, or if it was, it was likely trained for simple IR or factoid question-answering tasks, not for the RAG scenario where the retrieved passages are specifically to be used by another LLM for generating texts. We can address this mismatch for trainable IR algorithms by doing end-to-end training of the entire architecture on some set of questions and answers, training the parameters of the IR model as well as the LLM.

knowledge citations

Finally, it is generally useful for LLMs to give the user evidence for any factual statement. This can be in the form of **knowledge citations**, such as URLs of a trusted source or citation references to particular literature. For example a question answering system might generate numbered pointers to URLs as follows:

> Q: Which films have Gong Li as a member of their cast?
> A: The Story of Qiu Ju [1], Farewell My Concubine [2], The Monkey King 2 [3], Mulan [3], Saturday Fiction [3] ...

The simplest way for generating knowledge citations is to specify it as part of the prompt. For example Gao et al. (2023) employ a prompt with text like:

```
  ‘‘Write an answer for the given question using only the
provided search results (some of which might be irrelevant)
and cite them properly...  Always cite for any factual claim".
```

11.5 Datasets

There are scores of datasets that contain information needs in the form of questions, annotated with the answer. These can be used both for instruction tuning and for evaluation of the question answering abilities of language models.

We can distinguish the datasets along many dimensions, summarized nicely in Rogers et al. (2023). One is the original purpose of the questions in the data, whether they were natural **information-seeking** questions, or whether they were questions designed for **probing**: evaluating or testing systems or humans.

Natural Questions

On the natural side there are datasets like **Natural Questions** (Kwiatkowski et al., 2019), a set of anonymized English queries to the Google search engine and their answers. The answers are created by annotators based on Wikipedia information, and include a paragraph-length long answer and a short span answer. For example the question "When are hops added to the brewing process?" has the short answer *the boiling process* and a long answer which is an entire paragraph from the Wikipedia page on *Brewing*.

MS MARCO

A similar natural question set is the **MS MARCO** (Microsoft Machine Reading Comprehension) collection of datasets, including 1 million real anonymized English questions from Microsoft Bing query logs together with a human generated answer and 9 million passages (Bajaj et al., 2016), that can be used both to test retrieval ranking and question answering.

Although many datasets focus on English, natural information-seeking question datasets exist in other languages. The DuReader dataset is a Chinese QA resource based on search engine queries and community QA (He et al., 2018).

TyDi QA

TyDi QA dataset contains 204K question-answer pairs from 11 typologically diverse languages, including Arabic, Bengali, Kiswahili, Russian, and Thai (Clark et al., 2020a). In the TyDi QA task, a system is given a question and the passages from a Wikipedia article and must (a) select the passage containing the answer (or

NULL if no passage contains the answer), and (b) mark the minimal answer span (or NULL).

MMLU

On the probing side are datasets like **MMLU** (Massive Multitask Language Understanding), a commonly-used dataset of 15908 knowledge and reasoning questions in 57 areas including medicine, mathematics, computer science, law, and others. MMLU questions are sourced from various exams for humans, such as the US Graduate Record Exam, Medical Licensing Examination, and Advanced Placement exams. So the questions don't represent people's information needs, but rather are designed to test human knowledge for academic or licensing purposes. Fig. 11.14 shows some examples, with the correct answers in bold.

MMLU examples

College Computer Science
Any set of Boolean operators that is sufficient to represent all Boolean expressions is said to be complete. Which of the following is NOT complete?
(A) AND, NOT
(B) NOT, OR
(C) AND, OR
(D) NAND

College Physics
The primary source of the Sun's energy is a series of thermonuclear reactions in which the energy produced is c^2 times the mass difference between
(A) two hydrogen atoms and one helium atom
(B) four hydrogen atoms and one helium atom
(C) six hydrogen atoms and two helium atoms
(D) three helium atoms and one carbon atom

International Law
Which of the following is a treaty-based human rights mechanism?
(A) The UN Human Rights Committee
(B) The UN Human Rights Council
(C) The UN Universal Periodic Review
(D) The UN special mandates

Prehistory
Unlike most other early civilizations, Minoan culture shows little evidence of
(A) trade.
(B) warfare.
(C) the development of a common religion.
(D) conspicuous consumption by elites.

Figure 11.14 Example problems from MMLU

reading comprehension

Some of the question datasets described above augment each question with passage(s) from which the answer can be extracted. These datasets were mainly created for an earlier QA task called **reading comprehension** in which a model is given a question and a document and is required to extract the answer from the given

document. We sometimes call the task of question answering given one or more documents (for example via RAG), the **open book** QA task, while the task of answering directly from the LM with no retrieval component at all is the **closed book** QA task.[5] Thus datasets like Natural Questions can be treated as open book if the solver uses each question's attached document, or closed book if the documents are not used, while datasets like MMLU are solely closed book.

open book
closed book

Another dimension of variation is the format of the answer: multiple-choice versus freeform. And of course there are variations in prompting, like whether the model is just the question (zero-shot) or also given demonstrations of answers to similar questions (few-shot). MMLU offers both zero-shot and few-shot prompt options.

11.6 Evaluating Question Answering

Two techniques are commonly employed to evaluate question-answering systems, with the choice depending on the type of question and QA situation. For **multiple choice** questions like in MMLU, we report exact match:

Exact match: The % of predicted answers that match the gold answer exactly.

For questions with **free text** answers, like Natural Questions, we commonly evaluated with token $\mathbf{F_1}$ **score** to roughly measure the partial string overlap between the answer and the reference answer:

$\mathbf{F_1}$ score: The average token overlap between predicted and gold answers. Treat the prediction and gold as a bag of tokens, and compute F_1 for each question, then return the average F_1 over all questions.

11.7 Summary

This chapter introduced the tasks of **information retrieval** and the use of **retrieval augmented generation** (**RAG**) to use retrieved passages to improve **question answering** and other factual generations from LLMs.

- We focus in this chapter on the use of information retrieval for question answering and related factually-based tasks. The idea is to meet the user's information needs by drawing on the material in some set of documents (which might be the web).
- **Information Retrieval** (**IR**) is the task of returning documents to a user based on their information need as expressed in a **query**. In ranked retrieval, the documents are returned in ranked order.
- Two paradigms for IR are **sparse retrieval** and **dense retrieval**. Both paradigms use a document's similarity to the query as an estimate of its relevance to the user's information need
- In sparse retrieval techniques, we represent both the query and the document as sparse vectors of the unigram counts of the words they contain, each count

[5] This repurposes the word for types of exams in which students are allowed to 'open their books' or not.

weighted by **tf-idf** or **BM25**. Then the query-document similarity can be measured by the cosine between these sparse vectors.

- The **inverted index** is a storage mechanism for sparse retrieval that makes it very efficient to find documents that have a particular word.
- In dense retrieval techniques, documents or queries are instead represented as embeddings (dense vectors) computed by a language model (whether encoder-only models like the BERT family, or decoder-only). Document-query similarity is computed as dot product or cosine in they embedding space.
- For dense retrieval, FAISS is an approximate nearest neighbor vector search algorithm that makes it very efficient to find the k most similar document embeddings to a query embedding, making it quick to do ranking.
- Ranked retrieval is generally evaluated by **mean average precision** or **interpolated precision**.
- Retrieval can be incorporated into language modeling via **retrieval-augmented generation**. In the **retrieval** step, the user query is passed to the search engine to retrieve a set of relevant documents or passages. In the **generation** stage, a large language model is prompted with the query and a set of documents retrieved from the collection, and then conditionally generates an answer.
- Factual tasks like question answering can be evaluated by exact match with a known answer if only a single answer is given, with token F_1 score for free text answers.

Historical Notes

Question answering was one of the earliest NLP tasks. By 1961 the BASEBALL system (Green et al., 1961) answered questions about baseball games like "Where did the Red Sox play on July 7" by querying a structured database of game information. The database was stored as a kind of attribute-value matrix with values for attributes of each game:

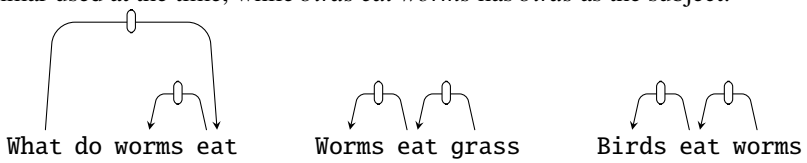
```
Month = July
   Place = Boston
       Day  = 7
       Game Serial No.  = 96
       (Team = Red Sox, Score = 5)
       (Team = Yankees, Score = 3)
```

Each question was constituency-parsed using the algorithm of Zellig Harris's TDAP project at the University of Pennsylvania, essentially a cascade of finite-state transducers (see the historical discussion in Joshi and Hopely 1999 and Karttunen 1999). Then in a content analysis phase each word or phrase was associated with a program that computed parts of its meaning. Thus the phrase 'Where' had code to assign the semantics `Place = ?`, with the result that the question "Where did the Red Sox play on July 7" was assigned the meaning

```
Place = ?
Team = Red Sox
Month = July
Day = 7
```

The question is then matched against the database to return the answer.

The Protosynthex system of Simmons et al. (1964), given a question, formed a query from the content words in the question, and then retrieved candidate answer sentences in the document, ranked by their frequency-weighted term overlap with the question. The query and each retrieved sentence were then parsed with dependency parsers, and the sentence whose structure best matches the question structure selected. Thus the question *What do worms eat?* would match *worms eat grass*: both have the subject *worms* as a dependent of *eat*, in the version of dependency grammar used at the time, while *birds eat worms* has *birds* as the subject:



Simmons (1965) summarizes other early QA systems.

LUNAR

By the 1970s, systems used predicate calculus as the meaning representation language. The **LUNAR** system (Woods et al. 1972, Woods 1978) was designed to be a natural language interface to a database of chemical facts about lunar geology. It could answer questions like *Do any samples have greater than 13 percent aluminum* by parsing them into a logical form

(TEST (FOR SOME X16 / (SEQ SAMPLES) : T ; (CONTAIN' X16
(NPR* X17 / (QUOTE AL203)) (GREATERTHAN 13 PCT))))

By the 1990s question answering shifted to machine learning. Zelle and Mooney (1996) proposed to treat question answering as a semantic parsing task, by creating the Prolog-based GEOQUERY dataset of questions about US geography. This model was extended by Zettlemoyer and Collins (2005) and 2007. By a decade later, neural models were applied to semantic parsing (Dong and Lapata 2016, Jia and Liang 2016), and then to knowledge-based question answering by mapping text to SQL (Iyer et al., 2017).

[TBD: History of IR.]

Meanwhile, a paradigm for answering questions that drew more on information-retrieval was influenced by the rise of the web in the 1990s. The U.S. government-sponsored TREC (Text REtrieval Conference) evaluations, run annually since 1992, provide a testbed for evaluating information-retrieval tasks and techniques (Voorhees and Harman, 2005). TREC added an influential QA track in 1999, which led to a wide variety of factoid and non-factoid question answering systems competing in annual evaluations.

At that same time, Hirschman et al. (1999) introduced the idea of using children's reading comprehension tests to evaluate machine text comprehension algorithms. They acquired a corpus of 120 passages with 5 questions each designed for 3rd-6th grade children, built an answer extraction system, and measured how well the answers given by their system corresponded to the answer key from the test's publisher. Their algorithm focused on word overlap as a feature; later algorithms added named entity features and more complex similarity between the question and the answer span (Riloff and Thelen 2000, Ng et al. 2000).

The DeepQA component of the Watson Jeopardy! system was a large and sophisticated feature-based system developed just before neural systems became common. It is described in a series of papers in volume 56 of the IBM Journal of Research and Development, e.g., Ferrucci (2012).

Early neural reading comprehension systems drew on the insight common to

early systems that answer finding should focus on question-passage similarity. Many of the architectural outlines of these neural systems were laid out in Hermann et al. (2015), Chen et al. (2017a), and Seo et al. (2017). These systems focused on datasets like Rajpurkar et al. (2016) and Rajpurkar et al. (2018) and their successors, usually using separate IR algorithms as input to neural reading comprehension systems. The paradigm of using dense retrieval with a span-based reader, often with a single end-to-end architecture, is exemplified by systems like Lee et al. (2019) or Karpukhin et al. (2020). An important research area with dense retrieval for open-domain QA is training data: using self-supervised methods to avoid having to label positive and negative passages (Sachan et al., 2023).

Early work on large language models showed that they stored sufficient knowledge in the pretraining process to answer questions (Petroni et al., 2019; Raffel et al., 2020; Radford et al., 2019; Roberts et al., 2020), at first not competitively with special-purpose question answerers, but quickly surpassing them. Retrieval-augmented generation algorithms were first introduced as a way to improve language modeling word prediction (Khandelwal et al., 2019), but were quickly applied to question answering (Izacard et al., 2022; Ram et al., 2023; Shi et al., 2023).

Exercises

CHAPTER

12 Machine Translation

> "I want to talk the dialect of your people. It's no use of talking unless people understand what you say."
>
> Zora Neale Hurston, *Moses, Man of the Mountain* 1939, p. 121

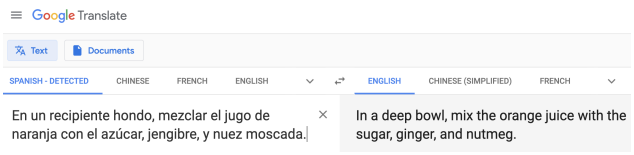
machine translation
MT

This chapter introduces **machine translation** (**MT**), the use of computers to translate from one language to another.

Of course translation, in its full generality, such as the translation of literature, or poetry, is a difficult, fascinating, and intensely human endeavor, as rich as any other area of human creativity.

information access

Machine translation in its present form therefore focuses on a number of very practical tasks. Perhaps the most common current use of machine translation is for **information access**. We might want to translate some instructions on the web, perhaps the recipe for a favorite dish, or the steps for putting together some furniture. Or we might want to read an article in a newspaper, or get information from an online resource like Wikipedia or a government webpage in some other language. MT for information access is probably one of the most common uses of NLP technology, and Google Translate alone (shown above) translates hundreds of billions of words a day between over 100 languages. Improvements in machine translation can thus help reduce what is often called the **digital divide** in information access: the fact that much more information is available in English and other languages spoken in wealthy countries. Web searches in English return much more information than searches in other languages, and online resources like Wikipedia are much larger in English and other higher-resourced languages. High-quality translation can help provide information to speakers of lower-resourced languages.



digital divide

post-editing
CAT
localization

Another common use of machine translation is to aid human translators. MT systems are routinely used to produce a draft translation that is fixed up in a **post-editing** phase by a human translator. This task is often called **computer-aided translation** or **CAT**. CAT is commonly used as part of **localization**: the task of adapting content or a product to a particular language community.

Finally, a more recent application of MT is to in-the-moment human communication needs. This includes incremental translation, translating speech on-the-fly before the entire sentence is complete, as is commonly used in simultaneous interpretation. Image-centric translation can be used for example to use OCR of the text on a phone camera image as input to an MT system to translate menus or street signs.

encoder-decoder

The standard algorithm for MT is the **encoder-decoder** model. We briefly mentioned in Chapter 7 that encoder-decoder or sequence-to-sequence models are used for tasks in which we need to map an input sequence to an output sequence that is a complex function of the entire input sequence, like machine translation or speech

recognition. Indeed, in machine translation, the words of the target language don't necessarily agree with the words of the source language in number or order. Consider translating the following made-up English sentence into Japanese.

(12.1) English: *He wrote a letter to a friend*
Japanese: *tomodachi ni tegami-o kaita*
friend to letter wrote

Note that the elements of the sentences are in very different places in the different languages. In English, the verb is in the middle of the sentence, while in Japanese, the verb *kaita* comes at the end. The Japanese sentence doesn't require the pronoun *he*, while English does.

Such differences between languages can be quite complex. In the following actual sentence from the United Nations, notice the many changes between the Chinese sentence (we've given in red a word-by-word gloss of the Chinese characters) and its English equivalent produced by human translators.

(12.2) 大会/General Assembly 在/on 1982年/1982 12月/December 10日/10 通过了/adopted 第37号/37th 决议/resolution ， 核准了/approved 第二次/second 探索/exploration 及/and 和平peaceful 利用/using 外层空间/outer space 会议/conference 的/of 各项/various 建议/suggestions 。

On 10 December 1982 , the General Assembly adopted resolution 37 in which it endorsed the recommendations of the Second United Nations Conference on the Exploration and Peaceful Uses of Outer Space .

Note the many ways the English and Chinese differ. For example the ordering differs in major ways; the Chinese order of the noun phrase is "peaceful using outer space conference of suggestions" while the English has "suggestions of the ... conference on peaceful use of outer space"). And the order differs in minor ways (the date is ordered differently). English requires *the* in many places that Chinese doesn't, and adds some details (like "in which" and "it") that aren't necessary in Chinese. Chinese doesn't grammatically mark plurality on nouns (unlike English, which has the "-s" in "recommendations"), and so the Chinese must use the modifier 各项/*various* to make it clear that there is not just one recommendation. English capitalizes some words but not others. Encoder-decoder networks are very successful at handling these sorts of complicated cases of sequence mappings.

We'll begin in the next section by considering the linguistic background about how languages vary, and the implications this variance has for the task of MT. Then we'll sketch out the standard algorithm, give details about things like input tokenization and creating training corpora of parallel sentences, give some more low-level details about the encoder-decoder network, and finally discuss how MT is evaluated, introducing the simple chrF metric.

12.1 Language Divergences and Typology

universal

There are about 7,000 languages in the world. Some aspects of human language seem to be **universal**, holding true for every one of these languages, or are statistical universals, holding true for most of these languages. Many universals arise from the functional role of language as a communicative system by humans. Every language, for example, seems to have words for referring to people, for talking about eating and drinking, for being polite or not. There are also structural linguistic universals; for example, every language seems to have nouns and verbs (Chapter 17), has

ways to ask questions, or issue commands, has linguistic mechanisms for indicating agreement or disagreement.

translation divergence

Yet languages also **differ** in many ways (as has been pointed out since ancient times; see Fig. 12.1). Understanding what causes such **translation divergences** (Dorr, 1994) can help us build better MT models. We often distinguish the **idiosyncratic** and lexical differences that must be dealt with one by one (the word for "dog" differs wildly from language to language), from **systematic** differences that we can model in a general way (many languages put the verb before the grammatical object; others put the verb after the grammatical object). The study of these systematic cross-linguistic similarities and differences is called **linguistic typology**. This section sketches some typological facts that impact machine translation; the interested reader should also look into WALS, the World Atlas of Language Structures, which gives many typological facts about languages (Dryer and Haspelmath, 2013).

typology

Figure 12.1 The Tower of Babel, Pieter Bruegel 1563. Wikimedia Commons, from the Kunsthistorisches Museum, Vienna.

12.1.1 Word Order Typology

As we hinted at in our example above comparing English and Japanese, languages differ in the basic word order of verbs, subjects, and objects in simple declarative clauses. German, French, English, and Mandarin, for example, are all **SVO** (**Subject-Verb-Object**) languages, meaning that the verb tends to come between the subject and object. Hindi and Japanese, by contrast, are **SOV** languages, meaning that the verb tends to come at the end of basic clauses, and Irish and Arabic are **VSO** languages. Two languages that share their basic word order type often have other similarities. For example, **VO** languages generally have **prepositions**, whereas **OV** languages generally have **postpositions**.

SVO

SOV

VSO

Let's look in more detail at the example we saw above. In this SVO English sentence, the verb *wrote* is followed by its object *a letter* and the prepositional phrase

to a friend, in which the preposition *to* is followed by its argument *a friend*. Arabic, with a VSO order, also has the verb before the object and prepositions. By contrast, in the Japanese example that follows, each of these orderings is reversed; the verb is *preceded* by its arguments, and the postposition follows its argument.

(12.3) English: *He wrote a letter to a friend*
Japanese: *tomodachi ni tegami-o kaita*
friend to letter wrote
Arabic: *katabt risāla li ṡadq*
wrote letter to friend

Other kinds of ordering preferences vary idiosyncratically from language to language. In some SVO languages (like English and Mandarin) adjectives tend to appear before nouns, while in others languages like Spanish and Modern Hebrew, adjectives appear after the noun:

(12.4) **Spanish** *bruja verde* **English** *green witch*

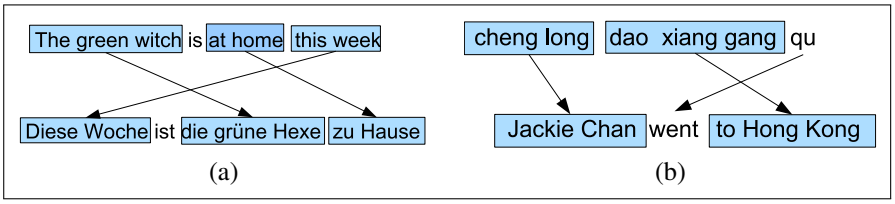


Figure 12.2 Examples of other word order differences: (a) In German, adverbs occur in initial position that in English are more natural later, and tensed verbs occur in second position. (b) In Mandarin, preposition phrases expressing goals often occur pre-verbally, unlike in English.

Fig. 12.2 shows examples of other word order differences. All of these word order differences between languages can cause problems for translation, requiring the system to do huge structural reorderings as it generates the output.

12.1.2 Lexical Divergences

Of course we also need to translate the individual words from one language to another. For any translation, the appropriate word can vary depending on the context. The English source-language word *bass*, for example, can appear in Spanish as the fish *lubina* or the musical instrument *bajo*. German uses two distinct words for what in English would be called a *wall*: *Wand* for walls inside a building, and *Mauer* for walls outside a building. Where English uses the word *brother* for any male sibling, Chinese and many other languages have distinct words for *older brother* and *younger brother* (Mandarin *gege* and *didi*, respectively). In all these cases, translating *bass*, *wall*, or *brother* from English would require a kind of specialization, disambiguating the different uses of a word. For this reason the fields of MT and Word Sense Disambiguation (Appendix G) are closely linked.

Sometimes one language places more grammatical constraints on word choice than another. We saw above that English marks nouns for whether they are singular or plural. Mandarin doesn't. Or French and Spanish, for example, mark grammatical gender on adjectives, so an English translation into French requires specifying adjective gender.

The way that languages differ in lexically dividing up conceptual space may be more complex than this one-to-many translation problem, leading to many-to-many

mappings. For example, Fig. 12.3 summarizes some of the complexities discussed by Hutchins and Somers (1992) in translating English *leg, foot*, and *paw*, to French. For example, when *leg* is used about an animal it's translated as French *patte*; but about the leg of a journey, as French *etape*; if the leg is of a chair, we use French *pied*.

lexical gap

Further, one language may have a **lexical gap**, where no word or phrase, short of an explanatory footnote, can express the exact meaning of a word in the other language. For example, English does not have a word that corresponds neatly to Mandarin *xiào* or Japanese *oyakōkō* (in English one has to make do with awkward phrases like *filial piety* or *loving child*, or *good son/daughter* for both).

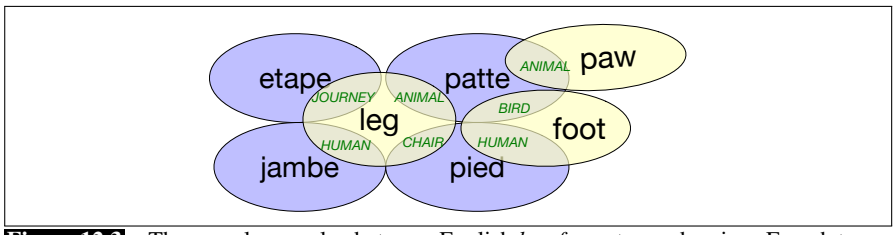


Figure 12.3 The complex overlap between English *leg*, *foot*, etc., and various French translations as discussed by Hutchins and Somers (1992).

Finally, languages differ systematically in how the conceptual properties of an event are mapped onto specific words. Talmy (1985, 1991) noted that languages can be characterized by whether direction of motion and manner of motion are marked on the verb or on the "satellites": particles, prepositional phrases, or adverbial phrases. For example, a bottle floating out of a cave would be described in English with the direction marked on the particle *out*, while in Spanish the direction would be marked on the verb:

(12.5) English: *The bottle floated out.*
Spanish: La botella salió flotando.
The bottle exited floating.

verb-framed

satellite-framed

Verb-framed languages mark the direction of motion on the verb (leaving the satellites to mark the manner of motion), like Spanish *acercarse* 'approach', *alcanzar* 'reach', *entrar* 'enter', *salir* 'exit'. **Satellite-framed** languages mark the direction of motion on the satellite (leaving the verb to mark the manner of motion), like English *crawl out*, *float off*, *jump down*, *run after*. Languages like Japanese, Tamil, and the many languages in the Romance, Semitic, and Mayan languages families, are verb-framed; Chinese as well as non-Romance Indo-European languages like English, Swedish, Russian, Hindi, and Farsi are satellite framed (Talmy 1991, Slobin 1996).

12.1.3 Morphological Typology

isolating

polysynthetic

agglutinative

fusion

Morphologically, languages are often characterized along two dimensions of variation. The first is the number of morphemes per word, ranging from **isolating** languages like Vietnamese and Cantonese, in which each word generally has one morpheme, to **polysynthetic** languages like Siberian Yupik ("Eskimo"), in which a single word may have very many morphemes, corresponding to a whole sentence in English. The second dimension is the degree to which morphemes are segmentable, ranging from **agglutinative** languages like Turkish, in which morphemes have relatively clean boundaries, to **fusion** languages like Russian, in which a single affix

may conflate multiple morphemes, like *-om* in the word *stolom* (table-SG-INSTR-DECL1), which fuses the distinct morphological categories instrumental, singular, and first declension.

Translating between languages with rich morphology requires dealing with structure below the word level, and for this reason modern systems generally use subword models like the wordpiece or BPE models of Section 12.2.1.

12.1.4 Referential density

Finally, languages vary along a typological dimension related to the things they tend to omit. Some languages, like English, require that we use an explicit pronoun when talking about a referent that is given in the discourse. In other languages, however, we can sometimes omit pronouns altogether, as the following example from Spanish shows[1]:

(12.6) [El jefe]$_i$ dio con un libro. $\emptyset_i$ Mostró su hallazgo a un descifrador ambulante.
[The boss] came upon a book. [He] showed his find to a wandering decoder.

pro-drop

Languages that can omit pronouns are called **pro-drop** languages. Even among the pro-drop languages, there are marked differences in frequencies of omission. Japanese and Chinese, for example, tend to omit far more than does Spanish. This dimension of variation across languages is called the dimension of **referential density**. We say that languages that tend to use more pronouns are more **referentially dense** than those that use more zeros. Referentially sparse languages, like Chinese or Japanese, that require the hearer to do more inferential work to recover antecedents are also called **cold** languages. Languages that are more explicit and make it easier for the hearer are called **hot** languages. The terms *hot* and *cold* are borrowed from Marshall McLuhan's 1964 distinction between hot media like movies, which fill in many details for the viewer, versus cold media like comics, which require the reader to do more inferential work to fill out the representation (Bickel, 2003).

referential density

cold language

hot language

Translating from languages with extensive pro-drop, like Chinese or Japanese, to non-pro-drop languages like English can be difficult since the model must somehow identify each zero and recover who or what is being talked about in order to insert the proper pronoun.

12.2 Machine Translation using Encoder-Decoder

The standard architecture for MT is the **encoder-decoder transformer** or **sequence-to-sequence** model, an architecture we briefly prefigured in Chapter 7. We'll see the details of how to apply this architecture to transformers in Section 12.3, but first let's talk about the overall task.

Most machine translation tasks make the simplification that we can translate each sentence independently, so we'll just consider individual sentences for now. Given a sentence in a **source** language, the MT task is then to generate a corresponding sentence in a **target** language. For example, an MT system is given an English sentence like

The green witch arrived

and must translate it into the Spanish sentence:

[1] Here we use the $\emptyset$-notation; we'll introduce this and discuss this issue further in Chapter 23

Llegó la bruja verde

MT uses supervised machine learning: at training time the system is given a large set of **parallel** sentences (each sentence in a source language matched with a sentence in the target language), and learns to map source sentences into target sentences. In practice, rather than using words (as in the example above), we split the sentences into a sequence of subword tokens (tokens can be words, or subwords, or individual characters). The systems are then trained to maximize the probability of the sequence of tokens in the target language $y_1, ..., y_m$ given the sequence of tokens in the source language $x_1, ..., x_n$:

$$P(y_1, \ldots, y_m | x_1, \ldots, x_n) \tag{12.7}$$

Rather than use the input tokens directly, the encoder-decoder architecture consists of two components, an **encoder** and a **decoder**. The encoder takes the input words $x = [x_1, \ldots, x_n]$ and produces an intermediate context $\mathbf{h}$. At decoding time, the system takes h and, word by word, generates the output y:

$$\mathbf{h} = \text{encoder}(x) \tag{12.8}$$

$$y_{t+1} = \text{decoder}(\mathbf{h}, y_1, \ldots, y_t) \quad \forall t \in [1, \ldots, m] \tag{12.9}$$

In the next two sections we'll talk about subword tokenization, then how to get parallel corpora for training, and then we'll introduce the details of the encoder-decoder architecture.

12.2.1 Tokenization

Machine translation systems use a vocabulary that is fixed in advance, and rather than using space-separated words, this vocabulary is generated with subword tokenization algorithms, like the **BPE** algorithm sketched in Chapter 2. A shared vocabulary is used for the source and target languages, which makes it easy to copy tokens (like names) from source to target. Using subword tokenization with tokens shared between languages makes it natural to translate between languages like English or Hindi that use spaces to separate words, and languages like Chinese or Thai that don't.

We build the vocabulary by running a subword tokenization algorithm on a corpus that contains both source and target language data.

Rather than the simple BPE algorithm from Fig. 2.6, modern systems often use more powerful tokenization algorithms. Some systems (like BERT) use a variant of BPE called the **wordpiece** algorithm, which instead of choosing the most frequent set of tokens to merge, chooses merges based on which one most increases the language model probability of the tokenization. Wordpieces use a special symbol at the beginning of each token; here's a resulting tokenization from the Google MT system (Wu et al., 2016):

wordpiece

words: Jet makers feud over seat width with big orders at stake
wordpieces: _J et _makers _fe ud _over _seat _width _with _big _orders _at _stake

The wordpiece algorithm is given a training corpus and a desired vocabulary size V, and proceeds as follows:

1. Initialize the wordpiece lexicon with characters (for example a subset of Unicode characters, collapsing all the remaining characters to a special unknown character token).

2. Repeat until there are V wordpieces:
   (a) Train an n-gram language model on the training corpus, using the current set of wordpieces.
   (b) Consider the set of possible new wordpieces made by concatenating two wordpieces from the current lexicon. Choose the one new wordpiece that most increases the language model probability of the training corpus.

Recall that with BPE we had to specify the number of merges to perform; in wordpiece, by contrast, we specify the total vocabulary, which is a more intuitive parameter. A vocabulary of 8K to 32K word pieces is commonly used.

unigram

SentencePiece

An even more commonly used tokenization algorithm is (somewhat ambiguously) called the **unigram** algorithm (Kudo, 2018) or sometimes the **SentencePiece** algorithm, and is used in systems like ALBERT (Lan et al., 2020) and T5 (Raffel et al., 2020). (Because unigram is the default tokenization algorithm used in a library called SentencePiece that adds a useful wrapper around tokenization algorithms (Kudo and Richardson, 2018b), authors often say they are using SentencePiece tokenization but really mean they are using the **unigram** algorithm).

In unigram tokenization, instead of building up a vocabulary by merging tokens, we start with a huge vocabulary of every individual unicode character plus all frequent sequences of characters (including all space-separated words, for languages with spaces), and iteratively remove some tokens to get to a desired final vocabulary size. The algorithm is complex (involving suffix-trees for efficiently storing many tokens, and the EM algorithm for iteratively assigning probabilities to tokens), so we don't give it here, but see Kudo (2018) and Kudo and Richardson (2018b). Roughly speaking the algorithm proceeds iteratively by estimating the probability of each token, tokenizing the input data using various tokenizations, then removing a percentage of tokens that don't occur in high-probability tokenization, and then iterates until the vocabulary has been reduced down to the desired number of tokens.

Why does unigram tokenization work better than BPE? BPE tends to create lots of very small non-meaningful tokens (because BPE can only create larger words or morphemes by merging characters one at a time), and it also tends to merge very common tokens, like the suffix *ed*, onto their neighbors. We can see from these examples from Bostrom and Durrett (2020) that unigram tends to produce tokens that are more semantically meaningful:

Original:	`corrupted`	Original:	`Completely preposterous suggestions`
BPE:	`cor rupted`	BPE:	`Comple t ely prep ost erous suggest ions`
Unigram:	`corrupt ed`	Unigram:	`Complete ly pre post er ous suggestion s`

12.2.2 Creating the Training data

parallel corpus

Europarl

Machine translation models are trained on a **parallel corpus**, sometimes called a **bitext**, a text that appears in two (or more) languages. Large numbers of parallel corpora are available. Some are governmental; the **Europarl** corpus (Koehn, 2005), extracted from the proceedings of the European Parliament, contains between 400,000 and 2 million sentences each from 21 European languages. The United Nations Parallel Corpus contains on the order of 10 million sentences in the six official languages of the United Nations (Arabic, Chinese, English, French, Russian, Spanish) Ziemski et al. (2016). Other parallel corpora have been made from movie and TV subtitles, like the **OpenSubtitles** corpus (Lison and Tiedemann, 2016), or from general web text, like the **ParaCrawl** corpus of 223 million sentence pairs between 23 EU languages and English extracted from the CommonCrawl Bañón et al. (2020).

Sentence alignment

Standard training corpora for MT come as aligned pairs of sentences. When creating new corpora, for example for underresourced languages or new domains, these sentence alignments must be created. Fig. 12.4 gives a sample hypothetical sentence alignment.

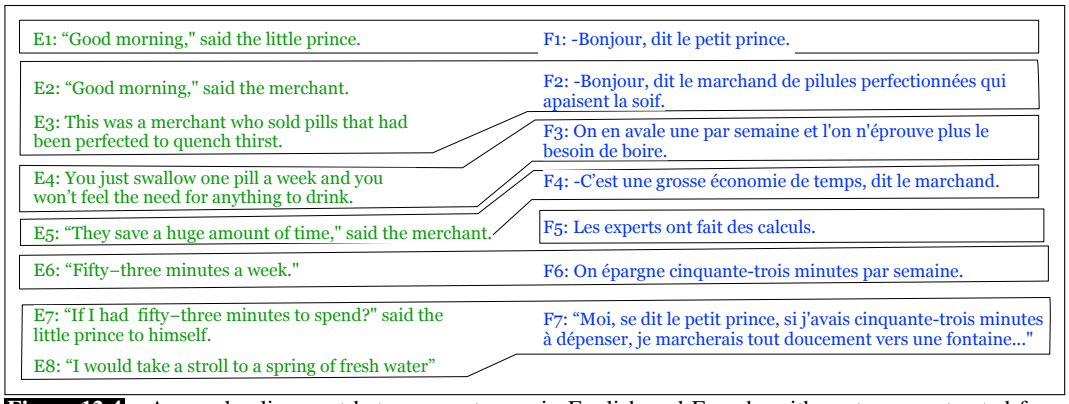


Figure 12.4 A sample alignment between sentences in English and French, with sentences extracted from Antoine de Saint-Exupery's *Le Petit Prince* and a hypothetical translation. Sentence alignment takes sentences $e_1,...,e_n$, and $f_1,...,f_m$ and finds minimal sets of sentences that are translations of each other, including single sentence mappings like (e_1,f_1), (e_4,f_3), (e_5,f_4), (e_6,f_6) as well as 2-1 alignments (e_2/e_3,f_2), (e_7/e_8,f_7), and null alignments (f_5).

Given two documents that are translations of each other, we generally need two steps to produce sentence alignments:

- a cost function that takes a span of source sentences and a span of target sentences and returns a score measuring how likely these spans are to be translations.
- an alignment algorithm that takes these scores to find a good alignment between the documents.

To score the similarity of sentences across languages, we need to make use of a **multilingual embedding space**, in which sentences from different languages are in the same embedding space (Artetxe and Schwenk, 2019). Given such a space, cosine similarity of such embeddings provides a natural scoring function (Schwenk, 2018). Thompson and Koehn (2019) give the following cost function between two sentences or spans x,y from the source and target documents respectively:

$$c(x,y) = \frac{(1-\cos(x,y))\text{nSents}(x)\ \text{nSents}(y)}{\sum_{s=1}^{S} 1-\cos(x,y_s) + \sum_{s=1}^{S} 1-\cos(x_s,y)} \tag{12.10}$$

where nSents() gives the number of sentences (this biases the metric toward many alignments of single sentences instead of aligning very large spans). The denominator helps to normalize the similarities, and so $x_1,...,x_S,y_1,...,y_S$, are randomly selected sentences sampled from the respective documents.

Usually dynamic programming is used as the alignment algorithm (Gale and Church, 1993), in a simple extension of the minimum edit distance algorithm we introduced in Chapter 2.

Finally, it's helpful to do some corpus cleanup by removing noisy sentence pairs. This can involve handwritten rules to remove low-precision pairs (for example removing sentences that are too long, too short, have different URLs, or even pairs

that are too similar, suggesting that they were copies rather than translations). Or pairs can be ranked by their multilingual embedding cosine score and low-scoring pairs discarded.

12.3 Details of the Encoder-Decoder Model

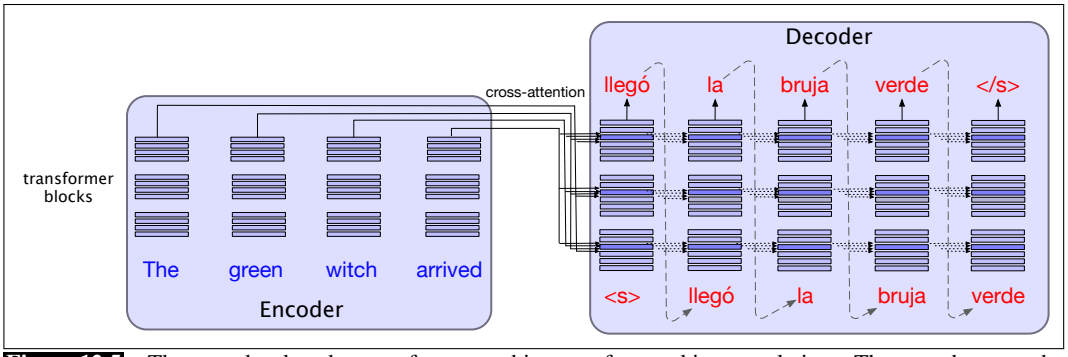


Figure 12.5 The encoder-decoder transformer architecture for machine translation. The encoder uses the transformer blocks we saw in Chapter 8, while the decoder uses a more powerful block with an extra **cross-attention** layer that can attend to all the encoder words. We'll see this in more detail in the next section.

The standard architecture for MT is the encoder-decoder transformer. Fig. 12.5 shows the intuition of the architecture at a high level. You'll see that the encoder-decoder architecture is made up of two transformers: an **encoder**, which is the same as the basic transformers from Chapter 8, and a **decoder**, which is augmented with a special new layer called the **cross-attention** layer. The encoder takes the source language input word tokens $\mathbf{X} = \mathbf{x}_1, ..., \mathbf{x}_n$ and maps them to an output representation $\mathbf{H}^{enc} = \mathbf{h}_1, ..., \mathbf{h}_n$; via a stack of encoder blocks.

The decoder is essentially a conditional language model that attends to the encoder representation and generates the target words one by one, at each timestep conditioning on the source sentence and the previously generated target language words to generate a token. Decoding can use any of the decoding methods discussed in Chapter 8 like greedy, or temperature or nucleus sampling. But the most common decoding algorithm for MT is the beam search algorithm that we'll introduce in Section 12.4.

But the components of the architecture differ somewhat from the transformer block we've seen. First, in order to attend to the source language, the transformer blocks in the decoder have an extra **cross-attention** layer. Recall that the transformer block of Chapter 8 consists of a self-attention layer that attends to the input from the previous layer, preceded by layer norm, and followed by another layer norm and the feed forward layer. The decoder transformer block includes an extra layer with a special kind of attention, **cross-attention** (also sometimes called **encoder-decoder attention** or **source attention**). Cross-attention has the same form as the multi-head attention in a normal transformer block, except that while the queries as usual come from the previous layer of the decoder, the keys and values come from the output of the *encoder*.

cross-attention

That is, where in standard multi-head attention the input to each attention layer is $\mathbf{X}$, in cross attention the input is the final output of the encoder $\mathbf{H}^{enc} = \mathbf{h}_1, ..., \mathbf{h}_n$. $\mathbf{H}^{enc}$ is of shape $[n \times d]$, each row representing one input token. To link the keys

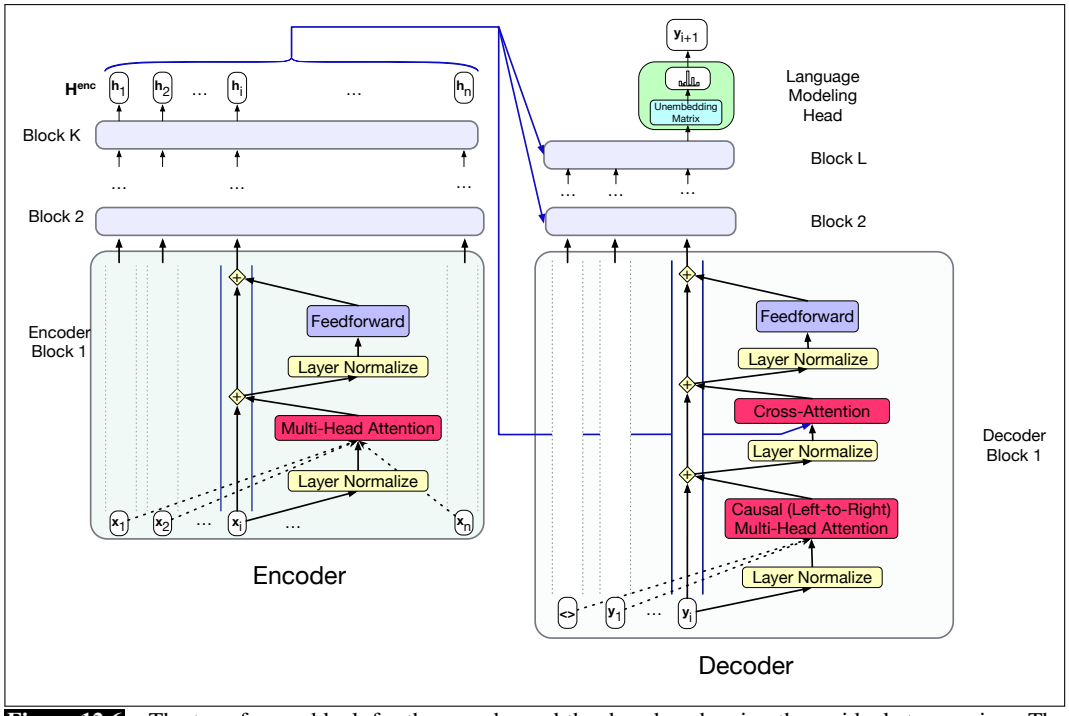


Figure 12.6 The transformer block for the encoder and the decoder, showing the residual stream view. The final output of the encoder $\mathbf{H}^{enc} = \mathbf{h}_1, ..., \mathbf{h}_n$ is the context used in the decoder. The decoder is a standard transformer except with one extra layer, the **cross-attention** layer, which takes that encoder output $\mathbf{H}^{enc}$ and uses it to form its $\mathbf{K}$ and $\mathbf{V}$ inputs.

and values from the encoder with the query from the prior layer of the decoder, we multiply the encoder output $\mathbf{H}^{enc}$ by the cross-attention layer's key weights $\mathbf{W}^{\mathbf{K}}$ and value weights $\mathbf{W}^{\mathbf{V}}$. The query comes from the output from the prior decoder layer $\mathbf{H}^{dec[\ell-1]}$, which is multiplied by the cross-attention layer's query weights $\mathbf{W}^{\mathbf{Q}}$:

$$\mathbf{Q} = \mathbf{H}^{dec[\ell-1]}\mathbf{W}^{\mathbf{Q}}; \;\; \mathbf{K} = \mathbf{H}^{enc}\mathbf{W}^{\mathbf{K}}; \;\; \mathbf{V} = \mathbf{H}^{enc}\mathbf{W}^{\mathbf{V}} \tag{12.11}$$

$$\text{CrossAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \;=\; \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\mathsf{T}}}{\sqrt{d_k}}\right)\mathbf{V} \tag{12.12}$$

The cross attention thus allows the decoder to attend to each of the source language words as projected into the entire encoder final output representations. The other attention layer in each decoder block, the multi-head attention layer, is the same causal (left-to-right) attention that we saw in Chapter 8. The multi-head attention in the encoder, however, is allowed to look ahead at the entire source language text, so it is not masked.

To train an encoder-decoder model, we give the network the source text, and then starting with the separator token train it autoregressively to predict the next token, using cross-entropy loss. Recall that cross-entropy loss for language modeling is determined by the probability the model assigns to the correct next word. So at time t the CE loss is the negative log probability the model assigns to the next word in the training sequence:

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) \;=\; -\log \hat{\mathbf{y}}_t[w_{t+1}] \tag{12.13}$$

teacher forcing

As in that case, we use **teacher forcing** in the decoder. Recall that in teacher forcing, at each time step in decoding we force the system to use the gold target token from training as the next input x_{t+1}, rather than allowing it to rely on the (possibly erroneous) decoder output $\hat{y}_t$.

12.4 Decoding in MT: Beam Search

Recall the **greedy decoding** algorithm from Chapter 8: at each time step t in generation, the output y_t is chosen by computing the probability for each word in the vocabulary and then choosing the highest probability word (the argmax):

$$\hat{w}_t = \text{argmax}_{w \in V} P(w|\mathbf{w}_{<t}) \tag{12.14}$$

A problem with greedy decoding is that what looks high probability at word t might turn out to have been the wrong choice once we get to word $t+1$. The **beam search** algorithm maintains multiple choices until later when we can see which one is best.

search tree

In beam search we model decoding as searching the space of possible generations, represented as a **search tree** whose **branches** represent actions (generating a token), and **nodes** represent states (having generated a particular prefix). We search for the best action sequence, i.e., the string with the highest probability.

An illustration of the problem

Fig. 12.7 shows a made-up example. The most probable sequence is *ok ok EOS* (its probability is .4× .7× 1.0). But greedy search doesn't find it, incorrectly choosing *yes* as the first word since it has the highest local probability (0.5).

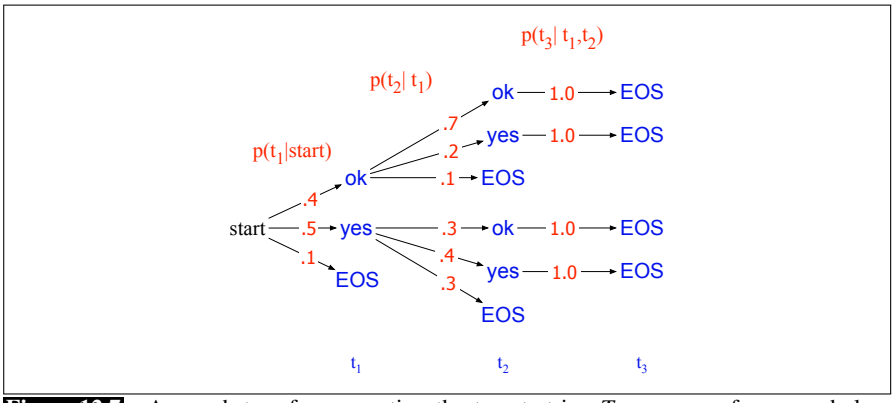


Figure 12.7 A search tree for generating the target string $T = t_1, t_2, \ldots$ from vocabulary $V = \{\text{yes}, \text{ok}, \texttt{<s>}\}$, showing the probability of generating each token from that state. Greedy search chooses *yes* followed by *yes*, instead of the globally most probable sequence *ok ok*.

For some problems, like part-of-speech tagging or parsing as we will see in Chapter 17 or Chapter 18, we can use dynamic programming search (the Viterbi algorithm) to address this problem. Unfortunately, dynamic programming is not applicable to generation problems with long-distance dependencies between the output decisions. The only method guaranteed to find the best solution is exhaustive search: computing the probability of every one of the V^T possible sentences (for some length value T) which is obviously too slow.

The solution: beam search

beam search

Instead, MT systems generally decode using **beam search**, a heuristic search method first proposed by Lowerre (1976). In beam search, instead of choosing the best token to generate at each timestep, we keep k possible tokens at each step. This fixed-size memory footprint k is called the **beam width**, on the metaphor of a flashlight beam that can be parameterized to be wider or narrower.

beam width

Thus at the first step of decoding, we compute a softmax over the entire vocabulary, assigning a probability to each word. We then select the k-best options from this softmax output. These initial k outputs are the search frontier and these k initial words are called **hypotheses**. A hypothesis is an output sequence, a translation-so-far, together with its probability.

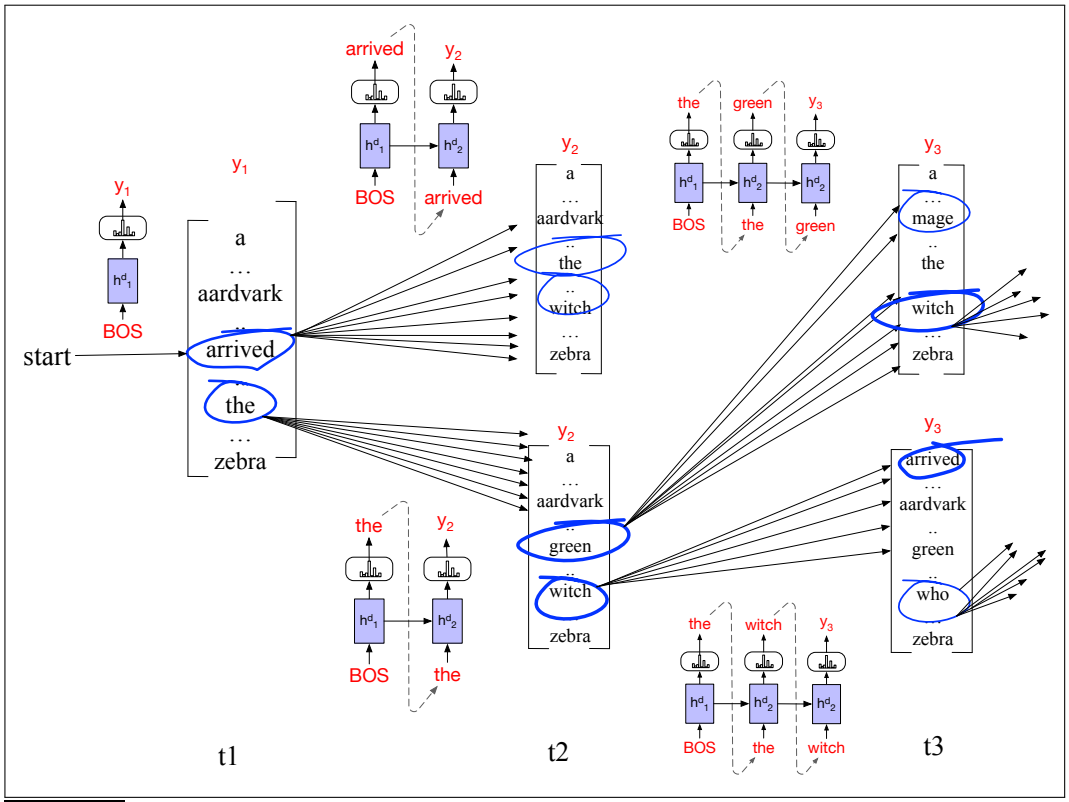


Figure 12.8 Beam search decoding with a beam width of $k = 2$. At each time step, we choose the k best hypotheses, form the V possible extensions of each, score those $k \times V$ hypotheses and choose the best $k = 2$ to continue. At time 1, the frontier has the best 2 options from the initial decoder state: *arrived* and *the*. We extend each, compute the probability of all the hypotheses so far (*arrived the*, *arrived aardvark*, *the green*, *the witch*) and again chose the best 2 (*the green* and *the witch*) to be the search frontier. The images on the arcs schematically represent the decoders that must be run at each step to score the next words (for simplicity not depicting cross-attention).

At subsequent steps, each of the k best hypotheses is extended incrementally by being passed to distinct decoders, which each generate a softmax over the entire vocabulary to extend the hypothesis to every possible next token. Each of these $k \times V$ hypotheses is scored by $P(y_i|x, y_{<i})$: the product of the probability of the current word choice multiplied by the probability of the path that led to it. We then prune the $k \times V$ hypotheses down to the k best hypotheses, so there are never more than k

hypotheses at the frontier of the search, and never more than k decoders. Fig. 12.8 illustrates this with a beam width of 2 for the beginning of *The green witch arrived*.

This process continues until an EOS is generated indicating that a complete candidate output has been found. At this point, the completed hypothesis is removed from the frontier and the size of the beam is reduced by one. The search continues until the beam has been reduced to 0. The result will be k hypotheses.

To score each node by its log probability, we use the chain rule of probability to break down $p(y|x)$ into the product of the probability of each word given its prior context, which we can turn into a sum of logs (for an output string of length t):

$$\begin{aligned} score(y) &= \log P(y|x) \\ &= \log\left(P(y_1|x)P(y_2|y_1,x)P(y_3|y_1,y_2,x)...P(y_t|y_1,...,y_{t-1},x)\right) \\ &= \sum_{i=1}^{t} \log P(y_i|y_1,...,y_{i-1},x) \end{aligned} \tag{12.15}$$

Thus at each step, to compute the probability of a partial sentence, we simply add the log probability of the prefix sentence so far to the log probability of generating the next token. Fig. 12.9 shows the scoring for the example sentence shown in Fig. 12.8, using some simple made-up probabilities. Log probabilities are negative or 0, and the max of two log probabilities is the one that is greater (closer to 0).

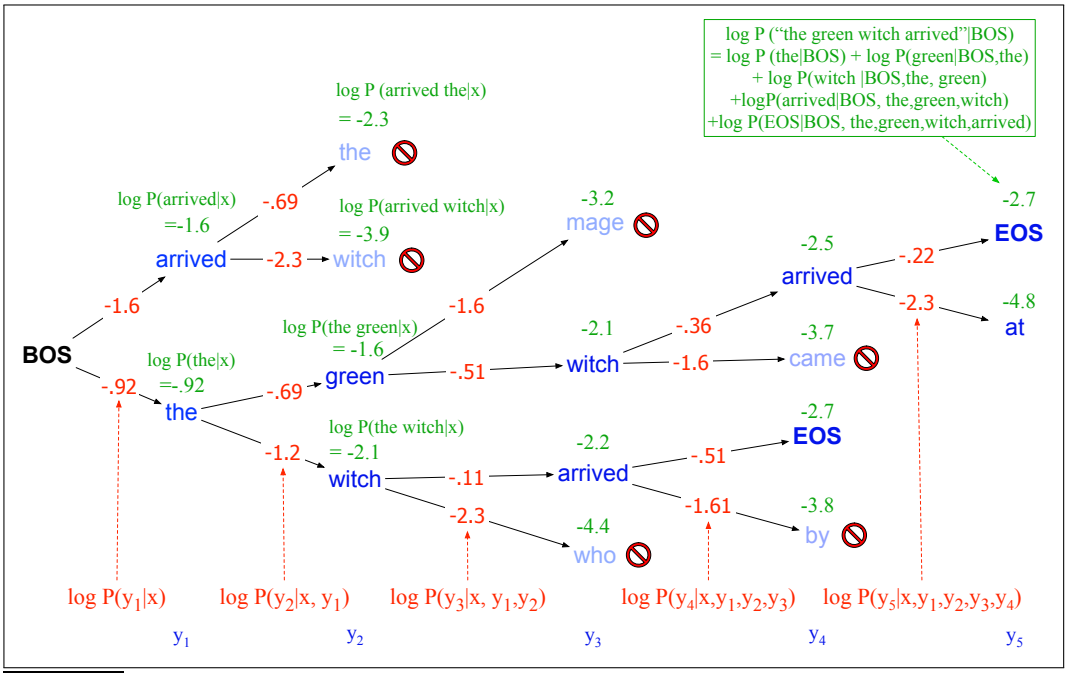


Figure 12.9 Scoring for beam search decoding with a beam width of $k = 2$. We maintain the log probability of each hypothesis in the beam by incrementally adding the logprob of generating each next token. Only the top k paths are extended to the next step.

Fig. 12.10 gives the algorithm. One problem with this version of the algorithm is that the completed hypotheses may have different lengths. Because language models generally assign lower probabilities to longer strings, a naive algorithm would choose shorter strings for y. (This is not an issue during the earlier steps of decoding; since beam search is breadth-first, all the hypotheses being compared had the

```
function BEAMDECODE(c, beam_width) returns best paths

  y_0, h_0 ← 0
  path ← ()
  complete_paths ← ()
  state ← (c, y_0, h_0, path)          ;initial state
  frontier ← ⟨state⟩                   ;initial frontier

  while frontier contains incomplete paths and beamwidth > 0
    extended_frontier ← ⟨⟩
    for each state ∈ frontier do
      y ← DECODE(state)
      for each word i ∈ Vocabulary do
        successor ← NEWSTATE(state, i, y_i)
        extended_frontier ← ADDTOBEAM(successor, extended_frontier,
                                      beam_width)

    for each state in extended_frontier do
      if state is complete do
        complete_paths ← APPEND(complete_paths, state)
        extended_frontier ← REMOVE(extended_frontier, state)
        beam_width ← beam_width - 1
    frontier ← extended_frontier

  return completed_paths

function NEWSTATE(state, word, word_prob) returns new state

function ADDTOBEAM(state, frontier, width) returns updated frontier

  if LENGTH(frontier) < width then
    frontier ← INSERT(state, frontier)
  else if SCORE(state) > SCORE(WORSTOF(frontier))
    frontier ← REMOVE(WORSTOF(frontier))
    frontier ← INSERT(state, frontier)
  return frontier
```

Figure 12.10 Beam search decoding.

same length.) For this reason we often apply length normalization methods, like dividing the logprob by the number of words:

$$score(y) = \frac{1}{t}\log P(y|x) \;=\; \frac{1}{t}\sum_{i=1}^{t}\log P(y_i|y_1,...,y_{i-1},x) \tag{12.16}$$

For MT we generally use beam widths k between 5 and 10, giving us k hypotheses at the end. We can pass all k to the downstream application with their respective scores, or if we just need a single translation we can pass the most probable hypothesis.

12.4.1 Minimum Bayes Risk Decoding

minimum Bayes risk
MBR

Minimum Bayes risk or **MBR** decoding is an alternative decoding algorithm that can work even better than beam search and also tends to be better than the other decoding algorithms like temperature sampling introduced in Section 7.4.

The intuition of minimum Bayes risk is that instead of trying to choose the translation which is most probable, we choose the one that is likely to have the least error. For example, we might want our decoding algorithm to find the translation which has the highest score on some evaluation metric. For example in Section 12.6 we will introduce metrics like chrF or BERTScore that measure the goodness-of-fit between a candidate translation and a set of reference human translations. A translation that maximizes this score, especially with a hypothetically huge set of perfect human translations is likely to be a good one (have minimum risk) even if it is not the most probable translation by our particular probability estimator.

In practice, we don't know the perfect set of translations for a given sentence. So the standard simplification used in MBR decoding algorithms is to instead choose the candidate translation which is most similar (by some measure of goodness-of-fit) with some set of candidate translations. We're essentially approximating the enormous space of all possible translations $\mathcal{U}$ with a smaller set of possible candidate translations $\mathcal{Y}$.

Given this set of possible candidate translations $\mathcal{Y}$, and some similarity or alignment function util, we choose the best translation $\hat{y}$ as the translation which is most similar to all the other candidate translations:

$$\hat{y} = \underset{y \in \mathcal{Y}}{\text{argmax}} \sum_{c \in \mathcal{Y}} \text{util}(y, c) \tag{12.17}$$

Various util functions can be used, like chrF or BERTscore or BLEU. We can get the set of candidate translations by sampling using one of the basic sampling algorithms of Section 7.4 like temperature sampling; good results can be obtained with as few as 32 or 64 candidates.

Minimum Bayes risk decoding can also be used for other NLP tasks; indeed it was widely applied to speech recognition (Stolcke et al., 1997; Goel and Byrne, 2000) before being applied to machine translation (Kumar and Byrne, 2004), and has been shown to work well across many other generation tasks as well (e.g., summarization, dialogue, and image captioning (Suzgun et al., 2023a)).

12.5 Translating in low-resource situations

For some languages, and especially for English, online resources are widely available. There are many large parallel corpora that contain translations between English and many languages. But the vast majority of the world's languages do not have large parallel training texts available. An important ongoing research question is how to get good translation with lesser resourced languages. The resource problem can even be true for high resource languages when we need to translate into low resource domains (for example in a particular genre that happens to have very little bitext).

Here we briefly introduce two commonly used approaches for dealing with this data sparsity: **backtranslation**, which is a special case of the general statistical technique called **data augmentation**, and **multilingual models**, and also discuss some socio-technical issues.

12.5.1 Data Augmentation

Data augmentation is a statistical technique for dealing with insufficient training data, by adding new synthetic data that is generated from the current natural data.

backtranslation

The most common data augmentation technique for machine translation is called **backtranslation**. Backtranslation relies on the intuition that while parallel corpora may be limited for particular languages or domains, we can often find a large (or at least larger) monolingual corpus, to add to the smaller parallel corpora that are available. The algorithm makes use of monolingual corpora in the **target** language by creating synthetic bitexts.

In backtranslation, our goal is to improve source-to-target MT, given a small parallel text (a bitext) in the source/target languages, and some monolingual data in the target language. We first use the bitext to train a MT system in the **reverse** direction: a target-to-source MT system. We then use it to translate the monolingual target data to the source language. Now we can add this synthetic bitext (natural target sentences, aligned with MT-produced source sentences) to our training data, and retrain our source-to-target MT model. For example suppose we want to translate from Navajo to English but only have a small Navajo-English bitext, although of course we can find lots of monolingual English data. We use the small bitext to build an MT engine going the other way (from English to Navajo). Once we translate the monolingual English text to Navajo, we can add this synthetic Navajo/English bitext to our training data.

Backtranslation has various parameters. One is how we generate the backtranslated data; we can run the decoder in greedy inference, or use beam search. Or we can do sampling, like the temperature sampling algorithm we saw in Chapter 8. Another parameter is the ratio of backtranslated data to natural bitext data; we can choose to upsample the bitext data (include multiple copies of each sentence). In general backtranslation works surprisingly well; one estimate suggests that a system trained on backtranslated text gets about 2/3 of the gain as would training on the same amount of natural bitext (Edunov et al., 2018).

12.5.2 Multilingual models

The models we've described so far are for bilingual translation: one source language, one target language. It's also possible to build a **multilingual** translator.

In a multilingual translator, we train the system by giving it parallel sentences in many different pairs of languages. That means we need to tell the system which language to translate from and to! We tell the system which language is which by adding a special token l_s to the encoder specifying the source language we're translating from, and a special token l_t to the decoder telling it the target language we'd like to translate into.

Thus we slightly update Eq. 12.9 above to add these tokens in Eq. 12.19:

$$\mathbf{h} = \text{encoder}(x, l_s) \tag{12.18}$$

$$y_{i+1} = \text{decoder}(\mathbf{h}, l_t, y_1, \ldots, y_i) \quad \forall i \in [1, \ldots, m] \tag{12.19}$$

One advantage of a multilingual model is that they can improve the translation of lower-resourced languages by drawing on information from a similar language in the training data that happens to have more resources. Perhaps we don't know the meaning of a word in Galician, but the word appears in the similar and higher-resourced language Spanish.

12.5.3 Sociotechnical issues

Many issues in dealing with low-resource languages go beyond the purely technical. One problem is that for low-resource languages, especially from low-income countries, native speakers are often not involved as the curators for content selection, as the language technologists, or as the evaluators who measure performance (Nekoto et al., 2020). Indeed, one well-known study that manually audited a large set of parallel corpora and other major multilingual datasets found that for many of the corpora, less than 50% of the sentences were of acceptable quality, with a lot of data consisting of repeated sentences with web boilerplate or incorrect translations, suggesting that native speakers may not have been sufficiently involved in the data process (Kreutzer et al., 2022).

Other issues, like the tendency of many MT approaches to focus on the case where one of the languages is English (Anastasopoulos and Neubig, 2020), have to do with allocation of resources. Where most large multilingual systems were trained on bitexts in which English was one of the two languages, recent huge corporate systems like those of Fan et al. (2021) and Costa-jussà et al. (2022) and datasets like Schwenk et al. (2021) attempt to handle large numbers of languages (up to 200 languages) and create bitexts between many more pairs of languages and not just through English.

At the smaller end, Nekoto et al. (2020) propose a participatory design process to encourage content creators, curators, and language technologists who speak these low-resourced languages to participate in developing MT algorithms. They provide online groups, mentoring, and infrastructure, and report on a case study on developing MT algorithms for low-resource African languages. Among their conclusions was to perform MT evaluation by post-editing rather than direct evaluation, since having labelers edit an MT system and then measure the distance between the MT output and its post-edited version both was simpler to train evaluators and makes it easier to measure true errors in the MT output and not differences due to linguistic variation (Bentivogli et al., 2018).

12.6 MT Evaluation

Translations are evaluated along two dimensions:

adequacy
1. **adequacy:** how well the translation captures the exact meaning of the source sentence. Sometimes called **faithfulness** or **fidelity**.

fluency
2. **fluency:** how fluent the translation is in the target language (is it grammatical, clear, readable, natural).

Using humans to evaluate is most accurate, but automatic metrics are also used for convenience.

12.6.1 Using Human Raters to Evaluate MT

The most accurate evaluations use human raters, such as online crowdworkers, to evaluate each translation along the two dimensions. For example, along the dimension of **fluency**, we can ask how intelligible, how clear, how readable, or how natural the MT output (the target text) is. We can give the raters a scale, for example, from 1 (totally unintelligible) to 5 (totally intelligible), or 1 to 100, and ask them to rate each sentence or paragraph of the MT output.

We can do the same thing to judge the second dimension, **adequacy**, using raters to assign scores on a scale. If we have bilingual raters, we can give them the source sentence and a proposed target sentence, and rate, on a 5-point or 100-point scale, how much of the information in the source was preserved in the target. If we only have monolingual raters but we have a good human translation of the source text, we can give the monolingual raters the human reference translation and a target machine translation and again rate how much information is preserved. An alternative is to do **ranking**: give the raters a pair of candidate translations, and ask them which one they prefer.

ranking

Training of human raters (who are often online crowdworkers) is essential; raters without translation expertise find it difficult to separate fluency and adequacy, and so training includes examples carefully distinguishing these. Raters often disagree (source sentences may be ambiguous, raters will have different world knowledge, raters may apply scales differently). It is therefore common to remove outlier raters, and (if we use a fine-grained enough scale) normalizing raters by subtracting the mean from their scores and dividing by the variance.

As discussed above, an alternative way of using human raters is to have them **post-edit** translations, taking the MT output and changing it minimally until they feel it represents a correct translation. The difference between their post-edited translations and the original MT output can then be used as a measure of quality.

12.6.2 Automatic Evaluation

While humans produce the best evaluations of machine translation output, running a human evaluation can be time consuming and expensive. For this reason automatic metrics are often used as temporary proxies. Automatic metrics are less accurate than human evaluation, but can help test potential system improvements, and even be used as an automatic loss function for training. In this section we introduce two families of such metrics, those based on character- or word-overlap and those based on embedding similarity.

Automatic Evaluation by Character Overlap: chrF

chrF

The simplest and most robust metric for MT evaluation is called **chrF**, which stands for **character F-score** (Popović, 2015). chrF (along with many other earlier related metrics like BLEU, METEOR, TER, and others) is based on a simple intuition derived from the pioneering work of Miller and Beebe-Center (1956): a good machine translation will tend to contain characters and words that occur in a human translation of the same sentence. Consider a test set from a parallel corpus, in which each source sentence has both a gold human target translation and a candidate MT translation we'd like to evaluate. The chrF metric ranks each MT target sentence by a function of the number of character n-gram overlaps with the human translation.

Given the hypothesis and the reference, chrF is given a parameter k indicating the length of character n-grams to be considered, and computes the average of the k precisions (unigram precision, bigram, and so on) and the average of the k recalls (unigram recall, bigram recall, etc.):

chrP percentage of character 1-grams, 2-grams, ..., k-grams in the hypothesis that occur in the reference, averaged.

chrR percentage of character 1-grams, 2-grams,..., k-grams in the reference that occur in the hypothesis, averaged.

The metric then computes an F-score by combining chrP and chrR using a weighting

parameter β. It is common to set $\beta = 2$, thus weighing recall twice as much as precision:

$$\text{chrF}\beta = (1+\beta^2)\frac{\text{chrP}\cdot\text{chrR}}{\beta^2\cdot\text{chrP}+\text{chrR}} \tag{12.20}$$

For $\beta = 2$, that would be:

$$\text{chrF2} = \frac{5\cdot\text{chrP}\cdot\text{chrR}}{4\cdot\text{chrP}+\text{chrR}}$$

For example, consider two hypotheses that we'd like to score against the reference translation *witness for the past*. Here are the hypotheses along with chrF values computed using parameters $k = \beta = 2$ (in real examples, k would be a higher number like 6):

REF: `witness for the past,`	
HYP1: `witness of the past,`	chrF2,2 = .86
HYP2: `past witness`	chrF2,2 = .62

Let's see how we computed that chrF value for HYP1 (we'll leave the computation of the chrF value for HYP2 as an exercise for the reader). First, chrF ignores spaces, so we'll remove them from both the reference and hypothesis:

REF: `witnessforthepast,` (18 unigrams, 17 bigrams)
HYP1: `witnessofthepast,` (17 unigrams, 16 bigrams)

Next let's see how many unigrams and bigrams match between the reference and hypothesis:

unigrams that match: `w i t n e s s f o t h e p a s t ,` (17 unigrams)
bigrams that match: `wi it tn ne es ss th he ep pa as st t,` (13 bigrams)

We use that to compute the unigram and bigram precisions and recalls:

unigram P:	17/17 = 1	unigram R:	17/18 = .944
bigram P:	13/16 = .813	bigram R:	13/17 = .765

Finally we average to get chrP and chrR, and compute the F-score:

$$\begin{aligned}\text{chrP} &= (17/17+13/16)/2 = .906\\ \text{chrR} &= (17/18+13/17)/2 = .855\\ \text{chrF2,2} &= 5\frac{\text{chrP}*\text{chrR}}{4\text{chrP}+\text{chrR}} = .86\end{aligned}$$

chrF is simple, robust, and correlates very well with human judgments in many languages (Kocmi et al., 2021).

Alternative overlap metric: BLEU

There are various alternative overlap metrics. For example, before the development of chrF, it was common to use a word-based overlap metric called **BLEU** (for BiLingual Evaluation Understudy), that is purely precision-based rather than combining precision and recall (Papineni et al., 2002). The BLEU score for a corpus of candidate translation sentences is a function of the **n-gram word precision** over all the sentences combined with a brevity penalty computed over the corpus as a whole.

What do we mean by n-gram precision? Consider a corpus composed of a single sentence. The unigram precision for this corpus is the percentage of unigram tokens

in the candidate translation that also occur in the reference translation, and ditto for bigrams and so on, up to 4-grams. BLEU extends this unigram metric to the whole corpus by computing the numerator as the sum over all sentences of the counts of all the unigram types that also occur in the reference translation, and the denominator is the total of the counts of all unigrams in all candidate sentences. We compute this n-gram precision for unigrams, bigrams, trigrams, and 4-grams and take the geometric mean. BLEU has many further complications, including a brevity penalty for penalizing candidate translations that are too short, and it also requires the n-gram counts be clipped in a particular way.

Because BLEU is a word-based metric, it is very sensitive to word tokenization, making it impossible to compare different systems if they rely on different tokenization standards, and doesn't work as well in languages with complex morphology. Nonetheless, you will sometimes still see systems evaluated by BLEU, particularly for translation into English. In such cases it's important to use packages that enforce standardization for tokenization like SACREBLEU (Post, 2018).

Statistical Significance Testing for MT evals

Character or word overlap-based metrics like chrF (or BLEU, or etc.) are mainly used to compare two systems, with the goal of answering questions like: did the new algorithm we just invented improve our MT system? To know if the difference between the chrF scores of two MT systems is a significant difference, we use the paired bootstrap test, or the similar randomization test.

To get a confidence interval on a single chrF score using the bootstrap test, recall from Section 4.11 that we take our test set (or devset) and create thousands of pseudo-testsets by repeatedly sampling with replacement from the original test set. We now compute the chrF score of each of the pseudo-testsets. If we drop the top 2.5% and bottom 2.5% of the scores, the remaining scores will give us the 95% confidence interval for the chrF score of our system.

To compare two MT systems A and B, we draw the same set of pseudo-testsets, and compute the chrF scores for each of them. We then compute the percentage of pseudo-test-sets in which A has a higher chrF score than B.

chrF: Limitations

While automatic character and word-overlap metrics like chrF or BLEU are useful, they have important limitations. chrF is very local: a large phrase that is moved around might barely change the chrF score at all, and chrF can't evaluate cross-sentence properties of a document like its discourse coherence (Chapter 24). chrF and similar automatic metrics also do poorly at comparing very different kinds of systems, such as comparing human-aided translation against machine translation, or different machine translation architectures against each other (Callison-Burch et al., 2006). Instead, automatic overlap metrics like chrF are most appropriate when evaluating changes to a single system.

12.6.3 Automatic Evaluation: Embedding-Based Methods

The chrF metric is based on measuring the exact character n-grams a human reference and candidate machine translation have in common. However, this criterion is overly strict, since a good translation may use alternate words or paraphrases. A solution first pioneered in early metrics like METEOR (Banerjee and Lavie, 2005) was to allow synonyms to match between the reference x and candidate $\tilde{x}$. More

recent metrics use BERT or other embeddings to implement this intuition.

For example, in some situations we might have datasets that have human assessments of translation quality. Such datasets consists of tuples $(x, \tilde{x}, r)$, where $x = (x_1, \ldots, x_n)$ is a reference translation, $\tilde{x} = (\tilde{x}_1, \ldots, \tilde{x}_m)$ is a candidate machine translation, and $r \in \mathbb{R}$ is a human rating that expresses the quality of $\tilde{x}$ with respect to x. Given such data, algorithms like COMET (Rei et al., 2020) BLEURT (Sellam et al., 2020) train a predictor on the human-labeled datasets, for example by passing x and $\tilde{x}$ through a version of BERT (trained with extra pretraining, and then finetuned on the human-labeled sentences), followed by a linear layer that is trained to predict r. The output of such models correlates highly with human labels.

In other cases, however, we don't have such human-labeled datasets. In that case we can measure the similarity of x and $\tilde{x}$ by the similarity of their embeddings. The BERTSCORE algorithm (Zhang et al., 2020) shown in Fig. 12.11, for example, passes the reference x and the candidate $\tilde{x}$ through BERT, computing a BERT embedding for each token x_i and $\tilde{x}_j$. Each pair of tokens $(x_i, \tilde{x}_j)$ is scored by its cosine $\frac{x_i \cdot \tilde{x}_j}{|x_i||\tilde{x}_j|}$. Each token in x is matched to a token in $\tilde{x}$ to compute recall, and each token in $\tilde{x}$ is matched to a token in x to compute precision (with each token greedily matched to the most similar token in the corresponding sentence). BERTSCORE provides precision and recall (and hence F_1):

$$R_{\text{BERT}} = \frac{1}{|x|} \sum_{x_i \in x} \max_{\tilde{x}_j \in \tilde{x}} x_i \cdot \tilde{x}_j \qquad P_{\text{BERT}} = \frac{1}{|\tilde{x}|} \sum_{\tilde{x}_j \in \tilde{x}} \max_{x_i \in x} x_i \cdot \tilde{x}_j \tag{12.21}$$

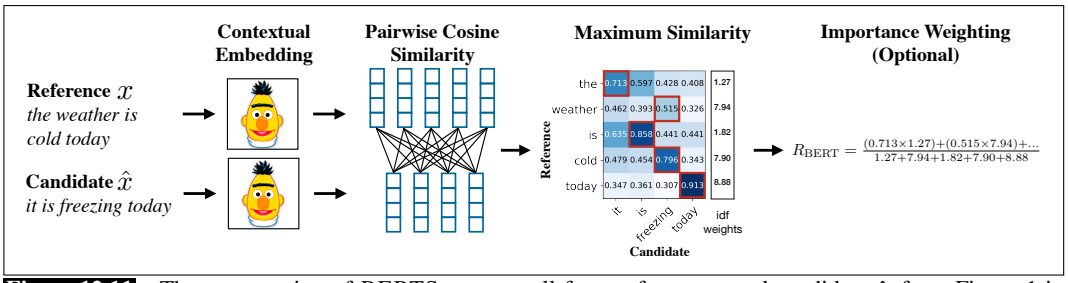


Figure 12.11 The computation of BERTSCORE recall from reference x and candidate $\hat{x}$, from Figure 1 in Zhang et al. (2020). This version shows an extended version of the metric in which tokens are also weighted by their idf values.

12.7 Bias and Ethical Issues

Machine translation raises many of the same ethical issues that we've discussed in earlier chapters. For example, consider MT systems translating from Hungarian (which has the gender neutral pronoun *ő*) or Spanish (which often drops pronouns) into English (in which pronouns are obligatory, and they have grammatical gender). When translating a reference to a person described without specified gender, MT systems often default to male gender (Schiebinger 2014, Prates et al. 2019). And MT systems often assign gender according to culture stereotypes of the sort we saw in Section 5.8. Fig. 12.12 shows examples from Prates et al. (2019), in which Hungarian gender-neutral *ő is a nurse* is translated with *she*, but gender-neutral *ő is a CEO* is translated with *he*. Prates et al. (2019) find that these stereotypes can't completely be accounted for by gender bias in US labor statistics, because the biases are

amplified by MT systems, with pronouns being mapped to male or female gender with a probability higher than if the mapping was based on actual labor employment statistics.

Hungarian (gender neutral) source	English MT output
ő egy ápoló	she is a nurse
ő egy tudós	he is a scientist
ő egy mérnök	he is an engineer
ő egy pék	he is a baker
ő egy tanár	she is a teacher
ő egy esküvőszervező	she is a wedding organizer
ő egy vezérigazgató	he is a CEO

Figure 12.12 When translating from gender-neutral languages like Hungarian into English, current MT systems interpret people from traditionally male-dominated occupations as male, and traditionally female-dominated occupations as female (Prates et al., 2019).

Similarly, a recent challenge set, the WinoMT dataset (Stanovsky et al., 2019) shows that MT systems perform worse when they are asked to translate sentences that describe people with non-stereotypical gender roles, like "The doctor asked the nurse to help her in the operation".

Many ethical questions in MT require further research. One open problem is developing metrics for knowing what our systems don't know. This is because MT systems can be used in urgent situations where human translators may be unavailable or delayed: in medical domains, to help translate when patients and doctors don't speak the same language, or in legal domains, to help judges or lawyers communicate with witnesses or defendants. In order to 'do no harm', systems need ways to assign **confidence** values to candidate translations, so they can abstain from giving incorrect translations that may cause harm.

confidence

12.8 Summary

Machine translation is one of the most widely used applications of NLP, and the encoder-decoder model, first developed for MT is a key tool that has applications throughout NLP.

- Languages have **divergences**, both structural and lexical, that make translation difficult.
- The linguistic field of **typology** investigates some of these differences; languages can be classified by their position along typological dimensions like whether verbs precede their objects.
- **Encoder-decoder** networks are composed of an **encoder** network that takes an input sequence and creates a contextualized representation of it, the **context**. This context representation is then passed to a **decoder** which generates a task-specific output sequence.
- **Cross-attention** allows the transformer decoder to view information from all the hidden states of the encoder.
- Machine translation models are trained on a **parallel corpus**, sometimes called a **bitext**, a text that appears in two (or more) languages.
- **Backtranslation** is a way of making use of monolingual corpora in the target language by running a pilot MT engine backwards to create synthetic bitexts.

- MT is evaluated by measuring a translation's **adequacy** (how well it captures the meaning of the source sentence) and **fluency** (how fluent or natural it is in the target language). Human evaluation is the gold standard, but automatic evaluation metrics like **chrF**, which measure character n-gram overlap with human translations, or more recent metrics based on embedding similarity, are also commonly used.

Historical Notes

MT was proposed seriously by the late 1940s, soon after the birth of the computer (Weaver, 1949/1955). In 1954, the first public demonstration of an MT system prototype (Dostert, 1955) led to great excitement in the press (Hutchins, 1997). The next decade saw a great flowering of ideas, prefiguring most subsequent developments. But this work was ahead of its time—implementations were limited by, for example, the fact that pending the development of disks there was no good way to store dictionary information.

As high-quality MT proved elusive (Bar-Hillel, 1960), there grew a consensus on the need for better evaluation and more basic research in the new fields of formal and computational linguistics. This consensus culminated in the famously critical ALPAC (Automatic Language Processing Advisory Committee) report of 1966 (Pierce et al., 1966) that led in the mid 1960s to a dramatic cut in funding for MT in the US. As MT research lost academic respectability, the Association for Machine Translation and Computational Linguistics dropped MT from its name. Some MT developers, however, persevered, and there were early MT systems like Météo, which translated weather forecasts from English to French (Chandioux, 1976), and industrial systems like Systran.

In the early years, the space of MT architectures spanned three general models. In **direct translation**, the system proceeds word-by-word through the source-language text, translating each word incrementally. Direct translation uses a large bilingual dictionary, each of whose entries is a small program with the job of translating one word. In **transfer** approaches, we first parse the input text and then apply rules to transform the source-language parse into a target language parse. We then generate the target language sentence from the parse tree. In **interlingua** approaches, we analyze the source language text into some abstract meaning representation, called an **interlingua**. We then generate into the target language from this interlingual representation. A common way to visualize these three early approaches was the **Vauquois triangle** shown in Fig. 12.13. The triangle shows the increasing depth of analysis required (on both the analysis and generation end) as we move from the direct approach through transfer approaches to interlingual approaches. In addition, it shows the decreasing amount of transfer knowledge needed as we move up the triangle, from huge amounts of transfer at the direct level (almost all knowledge is transfer knowledge for each word) through transfer (transfer rules only for parse trees or thematic roles) through interlingua (no specific transfer knowledge). We can view the encoder-decoder network as an interlingual approach, with attention acting as an integration of direct and transfer, allowing words or their representations to be directly accessed by the decoder.

Vauquois triangle

Statistical methods began to be applied around 1990, enabled first by the development of large bilingual corpora like the **Hansard** corpus of the proceedings of the

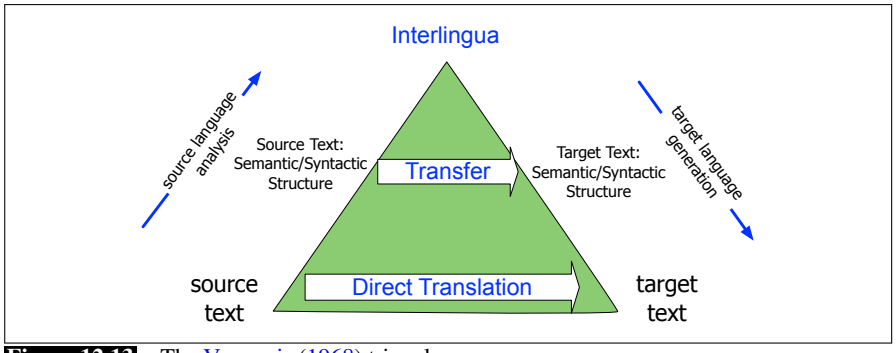


Figure 12.13 The Vauquois (1968) triangle.

Canadian Parliament, which are kept in both French and English, and then by the growth of the web. Early on, a number of researchers showed that it was possible to extract pairs of aligned sentences from bilingual corpora, using words or simple cues like sentence length (Kay and Röscheisen 1988, Gale and Church 1991, Gale and Church 1993, Kay and Röscheisen 1993).

statistical MT **IBM Models** **Candide**

At the same time, the IBM group, drawing directly on the **noisy channel** model for speech recognition, proposed two related paradigms for **statistical MT**. These include the generative algorithms that became known as **IBM Models 1 through 5**, implemented in the **Candide** system. The algorithms (except for the decoder) were published in full detail— encouraged by the US government who had partially funded the work— which gave them a huge impact on the research community (Brown et al. 1990, Brown et al. 1993).

The group also developed a discriminative approach, called MaxEnt (for maximum entropy, an alternative formulation of logistic regression), which allowed many features to be combined discriminatively rather than generatively (Berger et al., 1996), which was further developed by Och and Ney (2002).

phrase-based translation

By the turn of the century, most academic research on machine translation used statistical MT, either in the generative or discriminative mode. An extended version of the generative approach, called **phrase-based translation** was developed, based on inducing translations for phrase-pairs (Och 1998, Marcu and Wong 2002, Koehn et al. (2003), Och and Ney 2004, Deng and Byrne 2005, inter alia).

MERT **Moses**

Once automatic metrics like BLEU were developed (Papineni et al., 2002), the discriminative log linear formulation (Och and Ney, 2004), drawing from the IBM MaxEnt work (Berger et al., 1996), was used to directly optimize evaluation metrics like BLEU in a method known as **Minimum Error Rate Training**, or **MERT** (Och, 2003), also drawing from speech recognition models (Chou et al., 1993). Toolkits like GIZA (Och and Ney, 2003) and **Moses** (Koehn et al. 2006, Zens and Ney 2007) were widely used.

transduction grammars **inversion transduction grammar**

There were also approaches around the turn of the century that were based on syntactic structure (Chapter 18). Models based on **transduction grammars** (also called **synchronous grammars**) assign a parallel syntactic tree structure to a pair of sentences in different languages, with the goal of translating the sentences by applying reordering operations on the trees. From a generative perspective, we can view a transduction grammar as generating pairs of aligned sentences in two languages. Some of the most widely used models included the **inversion transduction grammar** (Wu, 1996) and synchronous context-free grammars (Chiang, 2005),

Neural networks had been applied at various times to various aspects of machine translation; for example Schwenk et al. (2006) showed how to use neural language

models to replace n-gram language models in a Spanish-English system based on IBM Model 4. The modern neural encoder-decoder approach was pioneered by Kalchbrenner and Blunsom (2013), who used a CNN encoder and an RNN decoder, and was first applied to MT by Bahdanau et al. (2015). The transformer encoder-decoder was proposed by Vaswani et al. (2017) (see the History section of Chapter 8).

Research on evaluation of machine translation began quite early. Miller and Beebe-Center (1956) proposed a number of methods drawing on work in psycholinguistics. These included the use of cloze and Shannon tasks to measure intelligibility as well as a metric of edit distance from a human translation, the intuition that underlies all modern overlap-based automatic evaluation metrics. The ALPAC report included an early evaluation study conducted by John Carroll that was extremely influential (Pierce et al., 1966, Appendix 10). Carroll proposed distinct measures for fidelity and intelligibility, and had raters score them subjectively on 9-point scales. Much early evaluation work focuses on automatic word-overlap metrics like BLEU (Papineni et al., 2002), **NIST** (Doddington, 2002), **TER (Translation Error Rate)** (Snover et al., 2006), **Precision and Recall** (Turian et al., 2003), and **METEOR** (Banerjee and Lavie, 2005); character n-gram overlap methods like chrF (Popović, 2015) came later. More recent evaluation work, echoing the ALPAC report, has emphasized the importance of careful statistical methodology and the use of human evaluation (Kocmi et al., 2021; Marie et al., 2021).

The early history of MT is surveyed in Hutchins 1986 and 1997; Nirenburg et al. (2002) collects early readings. See Croft (1990) or Comrie (1989) for introductions to linguistic typology.

Exercises

12.1 Compute by hand the chrF2,2 score for HYP2 on page 277 (the answer should round to .62).

CHAPTER

13 RNNs and LSTMs

Time will explain.
Jane Austen, *Persuasion*

Language is an inherently temporal phenomenon. Spoken language is a sequence of acoustic events over time, and we comprehend and produce both spoken and written language as a sequential input stream. The temporal nature of language is reflected in the metaphors we use; we talk of the *flow of conversations*, *news feeds*, and *twitter streams*, all of which emphasize that language is a sequence that unfolds in time.

This chapter introduces a deep learning architecture, the **recurrent neural network (RNN)**, and RNN variants like LSTMs, that offer a different way of representing time than feedforward and transformer networks. RNNs have a mechanism that deals directly with the sequential nature of language, allowing them to handle the temporal nature of language without the use of arbitrary fixed-sized windows. The recurrent network offers a new way to represent the prior context, in its **recurrent connections**, allowing the model's decision to depend on information from hundreds of words in the past. We'll see how to apply the model to the task of language modeling, to text classification tasks like sentiment analysis, and to sequence modeling tasks like part-of-speech tagging.

13.1 Recurrent Neural Networks

A recurrent neural network (RNN) is any network that contains a cycle within its network connections, meaning that the value of some unit is directly, or indirectly, dependent on its own earlier outputs as an input. While powerful, such networks are difficult to reason about and to train. However, within the general class of recurrent networks there are constrained architectures that have proven to be extremely effective when applied to language. In this section, we consider a class of recurrent networks referred to as **Elman Networks** (Elman, 1990) or **simple recurrent networks**. These networks are useful in their own right and serve as the basis for more complex approaches like the Long Short-Term Memory (LSTM) networks discussed later in this chapter. In this chapter when we use the term RNN we'll be referring to these simpler more constrained networks (although you will often see the term RNN to mean any net with recurrent properties including LSTMs).

Elman Networks

Fig. 13.1 illustrates the structure of an RNN. As with ordinary feedforward networks, an input vector representing the current input, $\mathbf{x}_t$, is multiplied by a weight matrix and then passed through a non-linear activation function to compute the values for a layer of hidden units. This hidden layer is then used to calculate a corresponding output, $\mathbf{y}_t$. In a departure from our earlier window-based approach, sequences are processed by presenting one item at a time to the network. We'll use

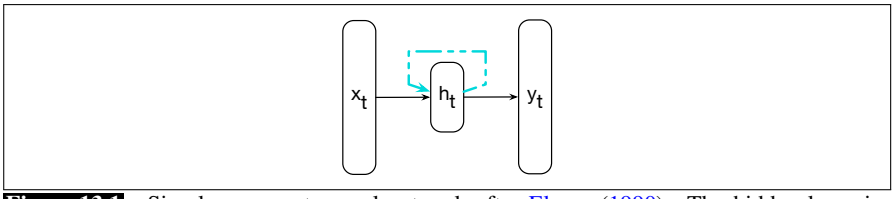


Figure 13.1 Simple recurrent neural network after Elman (1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous time step.

subscripts to represent time, thus $\mathbf{x}_t$ will mean the input vector $\mathbf{x}$ at time t. The key difference from a feedforward network lies in the recurrent link shown in the figure with the dashed line. This link augments the input to the computation at the hidden layer with the value of the hidden layer *from the preceding point in time*.

The hidden layer from the previous time step provides a form of memory, or context, that encodes earlier processing and informs the decisions to be made at later points in time. Critically, this approach does not impose a fixed-length limit on this prior context; the context embodied in the previous hidden layer can include information extending back to the beginning of the sequence.

Adding this temporal dimension makes RNNs appear to be more complex than non-recurrent architectures. But in reality, they're not all that different. Given an input vector and the values for the hidden layer from the previous time step, we're still performing the standard feedforward calculation introduced in Chapter 6. To see this, consider Fig. 13.2 which clarifies the nature of the recurrence and how it factors into the computation at the hidden layer. The most significant change lies in the new set of weights, $\mathbf{U}$, that connect the hidden layer from the previous time step to the current hidden layer. These weights determine how the network makes use of past context in calculating the output for the current input. As with the other weights in the network, these connections are trained via backpropagation.

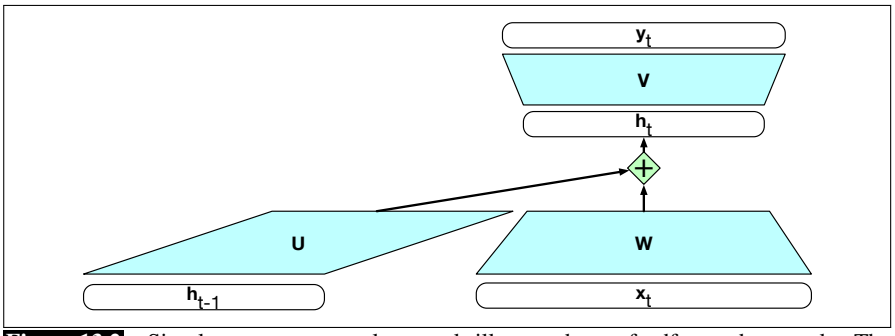


Figure 13.2 Simple recurrent neural network illustrated as a feedforward network. The hidden layer $\mathbf{h}_{t-1}$ from the prior time step is multiplied by weight matrix $\mathbf{U}$ and then added to the feedforward component from the current time step.

13.1.1 Inference in RNNs

Forward inference (mapping a sequence of inputs to a sequence of outputs) in an RNN is nearly identical to what we've already seen with feedforward networks. To compute an output $\mathbf{y}_t$ for an input $\mathbf{x}_t$, we need the activation value for the hidden layer $\mathbf{h}_t$. To calculate this, we multiply the input $\mathbf{x}_t$ with the weight matrix $\mathbf{W}$, and

the hidden layer from the previous time step $\mathbf{h}_{t-1}$ with the weight matrix $\mathbf{U}$. We add these values together and pass them through a suitable activation function, g, to arrive at the activation value for the current hidden layer, $\mathbf{h}_t$. Once we have the values for the hidden layer, we proceed with the usual computation to generate the output vector.

$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{x}_t) \tag{13.1}$$
$$\mathbf{y}_t = f(\mathbf{V}\mathbf{h}_t) \tag{13.2}$$

Let's refer to the input, hidden and output layer dimensions as d_{in}, d_h, and d_{out} respectively. Given this, our three parameter matrices are: $\mathbf{W} \in \mathbb{R}^{d_h \times d_{in}}$, $\mathbf{U} \in \mathbb{R}^{d_h \times d_h}$, and $\mathbf{V} \in \mathbb{R}^{d_{out} \times d_h}$.

We compute y_t via a softmax computation that gives a probability distribution over the possible output classes.

$$\mathbf{y}_t = \text{softmax}(\mathbf{V}\mathbf{h}_t) \tag{13.3}$$

The fact that the computation at time t requires the value of the hidden layer from time $t-1$ mandates an incremental inference algorithm that proceeds from the start of the sequence to the end as illustrated in Fig. 13.3. The sequential nature of simple recurrent networks can also be seen by *unrolling* the network in time as is shown in Fig. 13.4. In this figure, the various layers of units are copied for each time step to illustrate that they will have differing values over time. However, the various weight matrices are shared across time.

function FORWARDRNN($\mathbf{x}$, *network*) **returns** output sequence $\mathbf{y}$

$\mathbf{h}_0 \leftarrow 0$
for $i \leftarrow 1$ **to** LENGTH($\mathbf{x}$) **do**
  $\mathbf{h}_i \leftarrow g(\mathbf{U}\mathbf{h}_{i-1} + \mathbf{W}\mathbf{x}_i)$
  $\mathbf{y}_i \leftarrow f(\mathbf{V}\mathbf{h}_i)$
return $\mathbf{y}$

Figure 13.3 Forward inference in a simple recurrent network. The matrices $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$ are shared across time, while new values for $\mathbf{h}$ and $\mathbf{y}$ are calculated with each time step.

13.1.2 Training

As with feedforward networks, we'll use a training set, a loss function, and backpropagation to obtain the gradients needed to adjust the weights in these recurrent networks. As shown in Fig. 13.2, we now have 3 sets of weights to update: $\mathbf{W}$, the weights from the input layer to the hidden layer, $\mathbf{U}$, the weights from the previous hidden layer to the current hidden layer, and finally $\mathbf{V}$, the weights from the hidden layer to the output layer.

Fig. 13.4 highlights two considerations that we didn't have to worry about with backpropagation in feedforward networks. First, to compute the loss function for the output at time t we need the hidden layer from time $t-1$. Second, the hidden layer at time t influences both the output at time t and the hidden layer at time $t+1$ (and hence the output and loss at $t+1$). It follows from this that to assess the error accruing to $\mathbf{h}_t$, we'll need to know its influence on both the current output *as well as the ones that follow*.

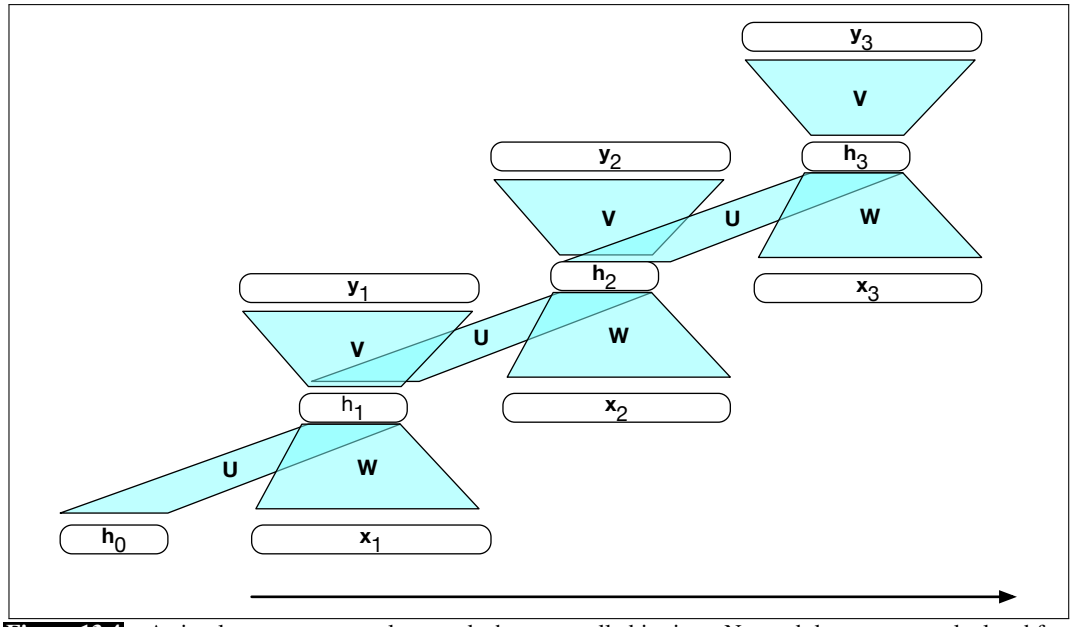


Figure 13.4 A simple recurrent neural network shown unrolled in time. Network layers are recalculated for each time step, while the weights **U**, **V** and **W** are shared across all time steps.

Tailoring the backpropagation algorithm to this situation leads to a two-pass algorithm for training the weights in RNNs. In the first pass, we perform forward inference, computing $\mathbf{h}_t$, $\mathbf{y}_t$, accumulating the loss at each step in time, saving the value of the hidden layer at each step for use at the next time step. In the second pass, we process the sequence in reverse, computing the required gradients as we go, computing and saving the error term for use in the hidden layer for each step backward in time. This general approach is commonly referred to as **backpropagation through time** (Werbos 1974, Rumelhart et al. 1986, Werbos 1990).

backpropagation through time

Fortunately, with modern computational frameworks and adequate computing resources, there is no need for a specialized approach to training RNNs. As illustrated in Fig. 13.4, explicitly unrolling a recurrent network into a feedforward computational graph eliminates any explicit recurrences, allowing the network weights to be trained directly. In such an approach, we provide a template that specifies the basic structure of the network, including all the necessary parameters for the input, output, and hidden layers, the weight matrices, as well as the activation and output functions to be used. Then, when presented with a specific input sequence, we can generate an unrolled feedforward network specific to that input, and use that graph to perform forward inference or training via ordinary backpropagation.

For applications that involve much longer input sequences, such as speech recognition, character-level processing, or streaming continuous inputs, unrolling an entire input sequence may not be feasible. In these cases, we can unroll the input into manageable fixed-length segments and treat each segment as a distinct training item.

13.2 RNNs as Language Models

Let's see how to apply RNNs to the language modeling task. Recall from Chapter 3 that language models predict the next word in a sequence given some preceding context. For example, if the preceding context is *"Thanks for all the"* and we want to know how likely the next word is *"fish"* we would compute:

$$P(fish|Thanks\ for\ all\ the)$$

Language models give us the ability to assign such a conditional probability to every possible next word, giving us a distribution over the entire vocabulary. We can also assign probabilities to entire sequences by combining these conditional probabilities with the chain rule:

$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i|w_{<i})$$

The n-gram language models of Chapter 3 compute the probability of a word given counts of its occurrence with the $n-1$ prior words. The context is thus of size $n-1$. For the feedforward language models of Chapter 6, the context is the window size.

RNN language models (Mikolov et al., 2010) process the input sequence one word at a time, attempting to predict the next word from the current word and the previous hidden state. RNNs thus don't have the limited context problem that n-gram models have, or the fixed context that feedforward language models have, since the hidden state can in principle represent information about all of the preceding words all the way back to the beginning of the sequence. Fig. 13.5 sketches this difference between a FFN language model and an RNN language model, showing that the RNN language model uses h_{t-1}, the hidden state from the previous time step, as a representation of the past context.

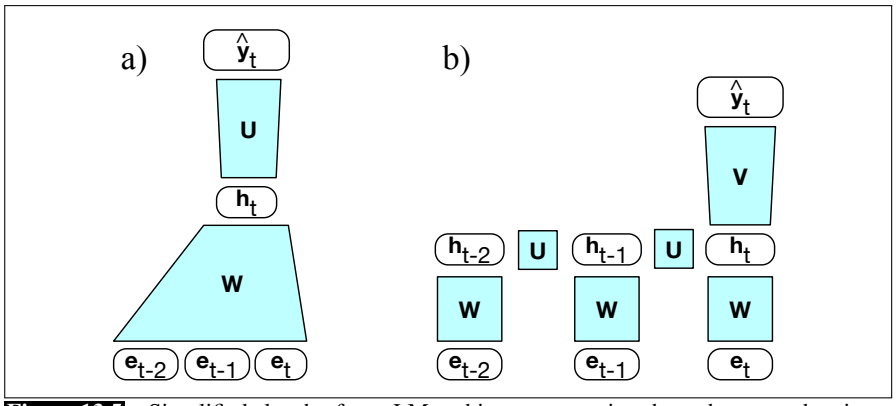


Figure 13.5 Simplified sketch of two LM architectures moving through a text, showing a schematic context of three tokens: (a) a feedforward neural language model which has a fixed context input to the weight matrix **W**, (b) an RNN language model, in which the hidden state $\mathbf{h}_{t-1}$ summarizes the prior context.

13.2.1 Forward Inference in an RNN language model

Forward inference in a recurrent language model proceeds exactly as described in Section 13.1.1. The input sequence $\mathbf{X} = [\mathbf{x}_1;...;\mathbf{x}_t;...;\mathbf{x}_N]$ consists of a series of

words each represented as a one-hot vector of size $|V| \times 1$, and the output prediction, $\hat{\mathbf{y}}$, is a vector representing a probability distribution over the vocabulary. At each step, the model uses the word embedding matrix $\mathbf{E}$ to retrieve the embedding for the current word, multiples it by the weight matrix $\mathbf{W}$, and then adds it to the hidden layer from the previous step (weighted by weight matrix $\mathbf{U}$) to compute a new hidden layer. This hidden layer is then used to generate an output layer which is passed through a softmax layer to generate a probability distribution over the entire vocabulary. That is, at time t:

$$\mathbf{e}_t = \mathbf{E}\mathbf{x}_t \tag{13.4}$$

$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t) \tag{13.5}$$

$$\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{V}\mathbf{h}_t) \tag{13.6}$$

When we do language modeling with RNNs (and we'll see this again in Chapter 8 with transformers), it's convenient to make the assumption that the embedding dimension d_e and the hidden dimension d_h are the same. So we'll just call both of these the **model dimension** d. So the embedding matrix $\mathbf{E}$ is of shape $[d \times |V|]$, and $\mathbf{x_t}$ is a one-hot vector of shape $[|V| \times 1]$. The product $\mathbf{e}_t$ is thus of shape $[d \times 1]$. $\mathbf{W}$ and $\mathbf{U}$ are of shape $[d \times d]$, so $\mathbf{h}_t$ is also of shape $[d \times 1]$. $\mathbf{V}$ is of shape $[|V| \times d]$, so the result of $\mathbf{V}\mathbf{h}$ is a vector of shape $[|V| \times 1]$. This vector can be thought of as a set of scores over the vocabulary given the evidence provided in $\mathbf{h}$. Passing these scores through the softmax normalizes the scores into a probability distribution. The probability that a particular word k in the vocabulary is the next word is represented by $\hat{\mathbf{y}}_t[k]$, the kth component of $\hat{\mathbf{y}}_t$:

$$P(w_{t+1} = k | w_1, \ldots, w_t) = \hat{\mathbf{y}}_t[k] \tag{13.7}$$

The probability of an entire sequence is just the product of the probabilities of each item in the sequence, where we'll use $\hat{\mathbf{y}}_i[w_i]$ to mean the probability of the true word w_i at time step i.

$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i | w_{1:i-1}) \tag{13.8}$$

$$= \prod_{i=1}^{n} \hat{\mathbf{y}}_i[w_i] \tag{13.9}$$

13.2.2 Training an RNN language model

self-supervision

To train an RNN as a language model, we use the same **self-supervision** (or **self-training**) algorithm we saw in Section 7.5: we take a corpus of text as training material and at each time step t ask the model to predict the next word. We call such a model self-supervised because we don't have to add any special gold labels to the data; the natural sequence of words is its own supervision! We simply train the model to minimize the error in predicting the true next word in the training sequence, using cross-entropy as the loss function. Recall that the cross-entropy loss measures the difference between a predicted probability distribution and the correct distribution.

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -\sum_{w \in V} \mathbf{y}_t[w] \log \hat{\mathbf{y}}_t[w] \tag{13.10}$$

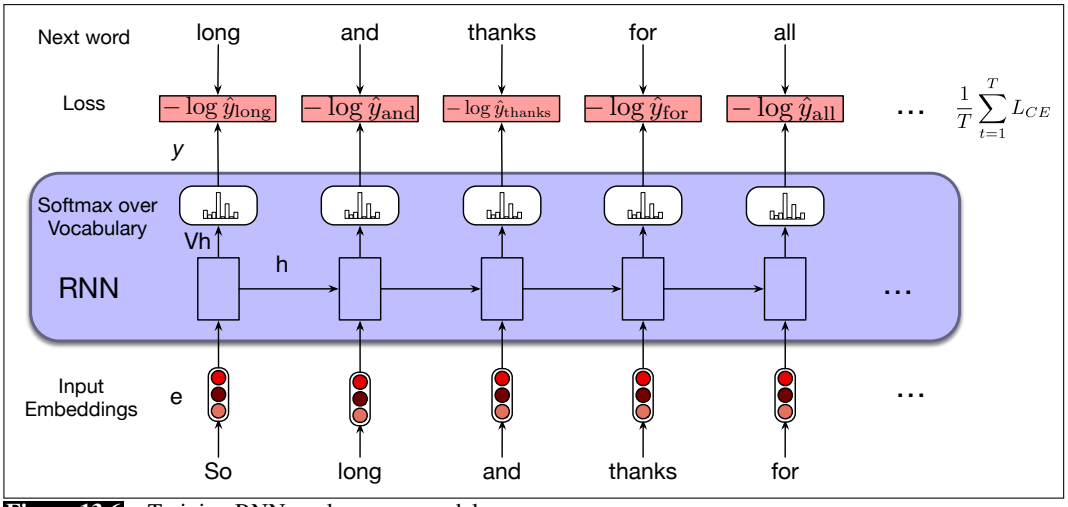


Figure 13.6 Training RNNs as language models.

In the case of language modeling, the correct distribution $\mathbf{y}_t$ comes from knowing the next word. This is represented as a one-hot vector corresponding to the vocabulary where the entry for the actual next word is 1, and all the other entries are 0. Thus, the cross-entropy loss for language modeling is determined by the probability the model assigns to the correct next word. So at time t the CE loss is the negative log probability the model assigns to the next word in the training sequence.

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -\log \hat{\mathbf{y}}_t[w_{t+1}] \tag{13.11}$$

Thus at each word position t of the input, the model takes as input the correct word w_t together with h_{t-1}, encoding information from the preceding $w_{1:t-1}$, and uses them to compute a probability distribution over possible next words so as to compute the model's loss for the next token w_{t+1}. Then we move to the next word, we ignore what the model predicted for the next word and instead use the correct word w_{t+1} along with the prior history encoded to estimate the probability of token w_{t+2}. This idea that we always give the model the correct history sequence to predict the next word (rather than feeding the model its best case from the previous time step) is called **teacher forcing**.

teacher forcing

The weights in the network are adjusted to minimize the average CE loss over the training sequence via gradient descent. Fig. 13.6 illustrates this training regimen.

13.2.3 Weight Tying

Careful readers may have noticed that the input embedding matrix $\mathbf{E}$ and the final layer matrix $\mathbf{V}$, which feeds the output softmax, are quite similar.

The columns of $\mathbf{E}$ represent the word embeddings for each word in the vocabulary learned during the training process with the goal that words that have similar meaning and function will have similar embeddings. And, since when we use RNNs for language modeling we make the assumption that the embedding dimension and the hidden dimension are the same (= the model dimension d), the embedding matrix $\mathbf{E}$ has shape $[d \times |V|]$. And the final layer matrix $\mathbf{V}$ provides a way to score the likelihood of each word in the vocabulary given the evidence present in the final hidden layer of the network through the calculation of $\mathbf{Vh}$. $\mathbf{V}$ is of shape $[|V| \times d]$. That is, the rows of $\mathbf{V}$ are shaped like a transpose of $\mathbf{E}$, meaning that $\mathbf{V}$ provides a

second set of learned word embeddings.

Instead of having two sets of embedding matrices, language models use a single embedding matrix, which appears at both the input and softmax layers. That is, we dispense with $\mathbf{V}$ and use $\mathbf{E}$ at the start of the computation and $\mathbf{E}^\intercal$ (because the shape of $\mathbf{V}$ is the transpose of $\mathbf{E}$ at the end. Using the same matrix (transposed) in two places is called **weight tying**.[1] The weight-tied equations for an RNN language model then become:

weight tying

$$\mathbf{e}_t = \mathbf{E}\mathbf{x}_t \tag{13.12}$$

$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t) \tag{13.13}$$

$$\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{E}^\intercal \mathbf{h}_t) \tag{13.14}$$

In addition to providing improved model perplexity, this approach significantly reduces the number of parameters required for the model.

13.3 RNNs for other NLP tasks

Now that we've seen the basic RNN architecture, let's consider how to apply it to three types of NLP tasks: *sequence classification* tasks like sentiment analysis and topic classification, *sequence labeling* tasks like part-of-speech tagging, and *text generation* tasks, including with a new architecture called the **encoder-decoder**.

13.3.1 Sequence Labeling

In sequence labeling, the network's task is to assign a label chosen from a small fixed set of labels to each element of a sequence. One classic sequence labeling tasks is part-of-speech (POS) tagging (assigning grammatical tags like NOUN and VERB to each word in a sentence). We'll discuss part-of-speech tagging in detail in Chapter 17, but let's give a motivating example here. In an RNN approach to sequence labeling, inputs are word embeddings and the outputs are tag probabilities generated by a softmax layer over the given tagset, as illustrated in Fig. 13.7.

In this figure, the inputs at each time step are pretrained word embeddings corresponding to the input tokens. The RNN block is an abstraction that represents an unrolled simple recurrent network consisting of an input layer, hidden layer, and output layer at each time step, as well as the shared $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$ weight matrices that comprise the network. The outputs of the network at each time step represent the distribution over the POS tagset generated by a softmax layer.

To generate a sequence of tags for a given input, we run forward inference over the input sequence and select the most likely tag from the softmax at each step. Since we're using a softmax layer to generate the probability distribution over the output tagset at each time step, we will again employ the cross-entropy loss during training.

13.3.2 RNNs for Sequence Classification

Another use of RNNs is to classify entire sequences rather than the tokens within them. This is the set of tasks commonly called **text classification**, like sentiment analysis or spam detection, in which we classify a text into two or three classes (like positive or negative), as well as classification tasks with a large number of

[1] We also do this for transformers (Chapter 8) where it's common to call $\mathbf{E}^\intercal$ the **unembedding matrix**.

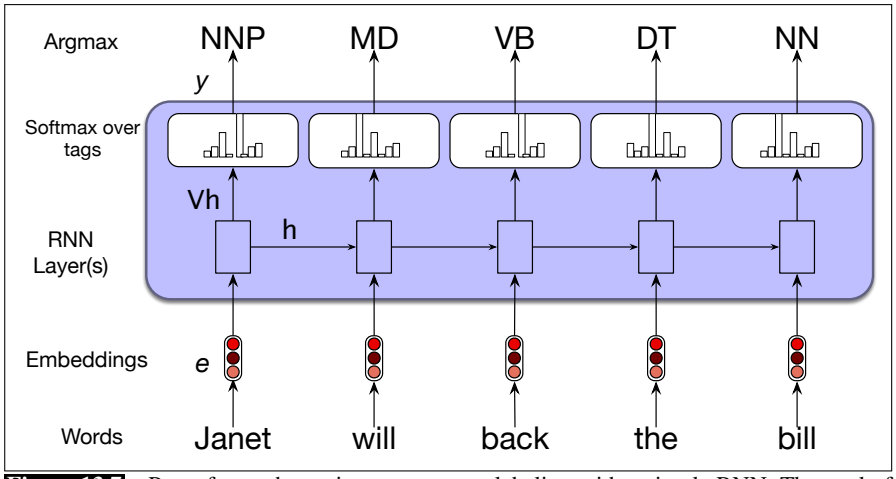


Figure 13.7 Part-of-speech tagging as sequence labeling with a simple RNN. The goal of part-of-speech (POS) tagging is to assign a grammatical label to each word in a sentence, drawn from a predefined set of tags. (The tags for this sentence include NNP (proper noun), MD (modal verb) and others; we'll give a complete description of the task of part-of-speech tagging in Chapter 17.) Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.

categories, like document-level topic classification, or message routing for customer service applications.

To apply RNNs in this setting, we pass the text to be classified through the RNN a word at a time generating a new hidden layer representation at each time step. We can then take the hidden layer for the last token of the text, $\mathbf{h}_n$, to constitute a compressed representation of the entire sequence. We can pass this representation $\mathbf{h}_n$ to a feedforward network that chooses a class via a softmax over the possible classes. Fig. 13.8 illustrates this approach.

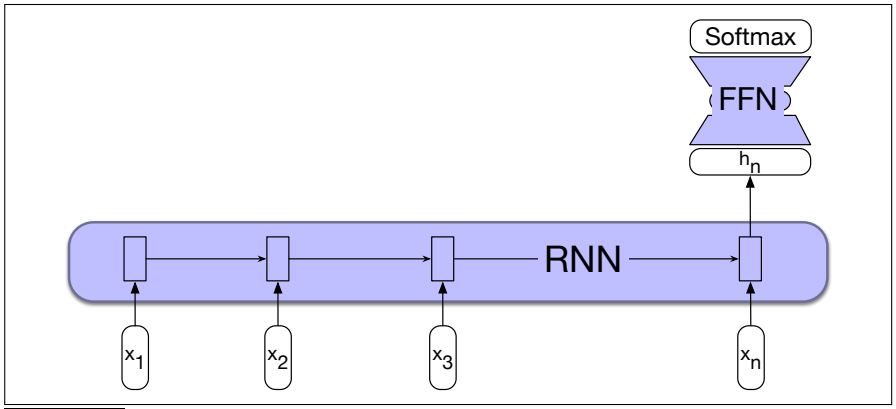


Figure 13.8 Sequence classification using a simple RNN combined with a feedforward network. The final hidden state from the RNN is used as the input to a feedforward network that performs the classification.

Note that in this approach we don't need intermediate outputs for the words in the sequence preceding the last element. Therefore, there are no loss terms associated with those elements. Instead, the loss function used to train the weights in the network is based entirely on the final text classification task. The output from the softmax output from the feedforward classifier together with a cross-entropy loss

drives the training. The error signal from the classification is backpropagated all the way through the weights in the feedforward classifier through, to its input, and then through to the three sets of weights in the RNN as described earlier in Section 13.1.2. The training regimen that uses the loss from a downstream application to adjust the weights all the way through the network is referred to as **end-to-end training**.

end-to-end training

Another option, instead of using just the hidden state of the last token h_n to represent the whole sequence, is to use some sort of **pooling** function of all the hidden states h_i for each word i in the sequence. For example, we can create a representation that pools all the n hidden states by taking their element-wise mean:

pooling

$$\mathbf{h}_{mean} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{h}_i \tag{13.15}$$

Or we can take the element-wise max; the element-wise max of a set of n vectors is a new vector whose kth element is the max of the kth elements of all the n vectors.

The long contexts of RNNs makes it quite difficult to successfully backpropagate error all the way through the entire input; we'll talk about this problem, and some standard solutions, in Section 13.5.

13.3.3 Generation with RNN-Based Language Models

RNN-based language models can also be used to generate text, Text generation, along with image generation and code generation, constitute a new area of AI that is often called **generative AI**. Those of you who have already read Chapter 7 and Chapter 8 will have already seen this, but we reintroduce it here for those who are reading in a different order.

Recall back in Chapter 3 we saw how to generate text from an n-gram language model by adapting a **sampling** technique suggested at about the same time by Claude Shannon (Shannon, 1951) and the psychologists George Miller and Jennifer Selfridge (Miller and Selfridge, 1950). We first randomly sample a word to begin a sequence based on its suitability as the start of a sequence. We then continue to sample words *conditioned on our previous choices* until we reach a pre-determined length, or an end of sequence token is generated.

Today, this approach of using a language model to incrementally generate words by repeatedly sampling the next word conditioned on our previous choices is called **autoregressive generation** or **causal LM generation**. The procedure is basically the same as that described on page 49, but adapted to a neural context:

autoregressive generation

- Sample a word in the output from the softmax distribution that results from using the beginning of sentence marker, <s>, as the first input.
- Use the word embedding for that first word as the input to the network at the next time step, and then sample the next word in the same fashion.
- Continue generating until the end of sentence marker, </s>, is sampled or a fixed length limit is reached.

Technically an **autoregressive** model is a model that predicts a value at time t based on a linear function of the previous values at times $t-1$, $t-2$, and so on. Although language models are not linear (since they have many layers of non-linearities), we loosely refer to this generation technique as **autoregressive generation** since the word generated at each time step is conditioned on the word selected by the network from the previous step. Fig. 13.9 illustrates this approach. In this figure, the details of the RNN's hidden layers and recurrent connections are hidden within the blue block.

This simple architecture underlies state-of-the-art approaches to applications such as machine translation, summarization, and question answering. The key to these approaches is to prime the generation component with an appropriate context. That is, instead of simply using <s> to get things started we can provide a richer task-appropriate context; for translation the context is the sentence in the source language; for summarization it's the long text we want to summarize.

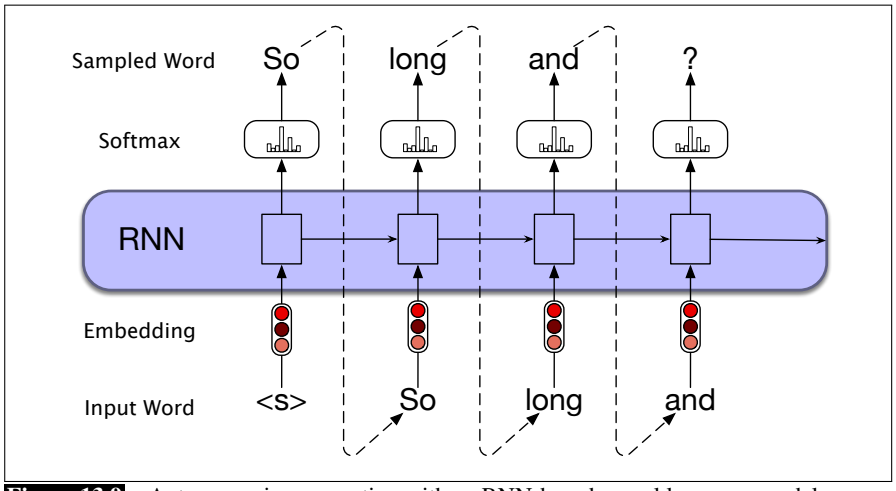


Figure 13.9 Autoregressive generation with an RNN-based neural language model.

13.4 Stacked and Bidirectional RNN architectures

Recurrent networks are quite flexible. By combining the feedforward nature of unrolled computational graphs with vectors as common inputs and outputs, complex networks can be treated as modules that can be combined in creative ways. This section introduces two of the more common network architectures used in language processing with RNNs.

13.4.1 Stacked RNNs

In our examples thus far, the inputs to our RNNs have consisted of sequences of word or character embeddings (vectors) and the outputs have been vectors useful for predicting words, tags or sequence labels. However, nothing prevents us from using the entire sequence of outputs from one RNN as an input sequence to another one.

Stacked RNNs

Stacked RNNs consist of multiple networks where the output of one layer serves as the input to a subsequent layer, as shown in Fig. 13.10.

Stacked RNNs generally outperform single-layer networks. One reason for this success seems to be that the network induces representations at differing levels of abstraction across layers. Just as the early stages of the human visual system detect edges that are then used for finding larger regions and shapes, the initial layers of stacked networks can induce representations that serve as useful abstractions for further layers—representations that might prove difficult to induce in a single RNN. The optimal number of stacked RNNs is specific to each application and to each training set. However, as the number of stacks is increased the training costs rise

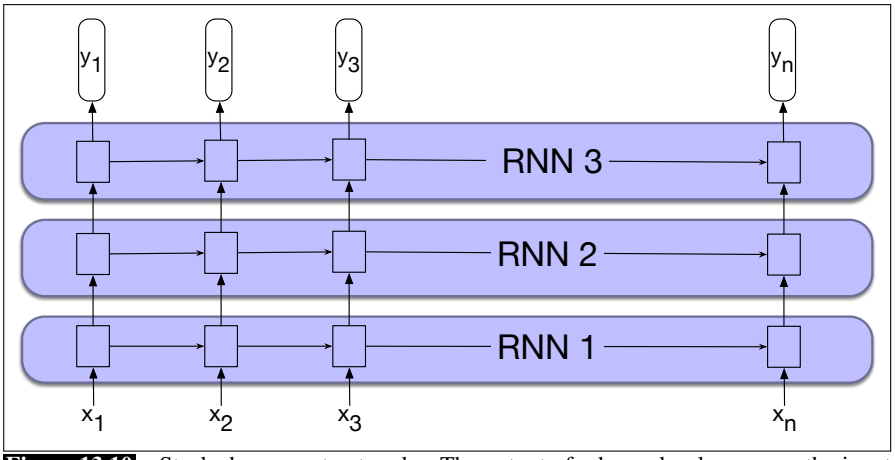


Figure 13.10 Stacked recurrent networks. The output of a lower level serves as the input to higher levels with the output of the last network serving as the final output.

quickly.

13.4.2 Bidirectional RNNs

The RNN uses information from the left (prior) context to make its predictions at time t. But in many applications we have access to the entire input sequence; in those cases we would like to use words from the context to the right of t. One way to do this is to run two separate RNNs, one left-to-right, and one right-to-left, and concatenate their representations.

In the left-to-right RNNs we've discussed so far, the hidden state at a given time t represents everything the network knows about the sequence up to that point. The state is a function of the inputs $x_1, ..., x_t$ and represents the context of the network to the left of the current time.

$$\mathbf{h}_t^f = \text{RNN}_{\text{forward}}(\mathbf{x}_1, \ldots, \mathbf{x}_t) \tag{13.16}$$

This new notation $\mathbf{h}_t^f$ simply corresponds to the normal hidden state at time t, representing everything the network has gleaned from the sequence so far.

To take advantage of context to the right of the current input, we can train an RNN on a *reversed* input sequence. With this approach, the hidden state at time t represents information about the sequence to the *right* of the current input:

$$\mathbf{h}_t^b = \text{RNN}_{\text{backward}}(\mathbf{x}_t, \ldots \mathbf{x}_n) \tag{13.17}$$

Here, the hidden state $\mathbf{h}_t^b$ represents all the information we have discerned about the sequence from t to the end of the sequence.

bidirectional RNN

A **bidirectional RNN** (Schuster and Paliwal, 1997) combines two independent RNNs, one where the input is processed from the start to the end, and the other from the end to the start. We then concatenate the two representations computed by the networks into a single vector that captures both the left and right contexts of an input at each point in time. Here we use either the semicolon ";" or the equivalent symbol $\oplus$ to mean vector concatenation:

$$\begin{aligned} \mathbf{h}_t &= [\mathbf{h}_t^f ; \mathbf{h}_t^b] \\ &= \mathbf{h}_t^f \oplus \mathbf{h}_t^b \end{aligned} \tag{13.18}$$

Fig. 13.11 illustrates such a bidirectional network that concatenates the outputs of the forward and backward pass. Other simple ways to combine the forward and backward contexts include element-wise addition or multiplication. The output at each step in time thus captures information to the left and to the right of the current input. In sequence labeling applications, these concatenated outputs can serve as the basis for a local labeling decision.

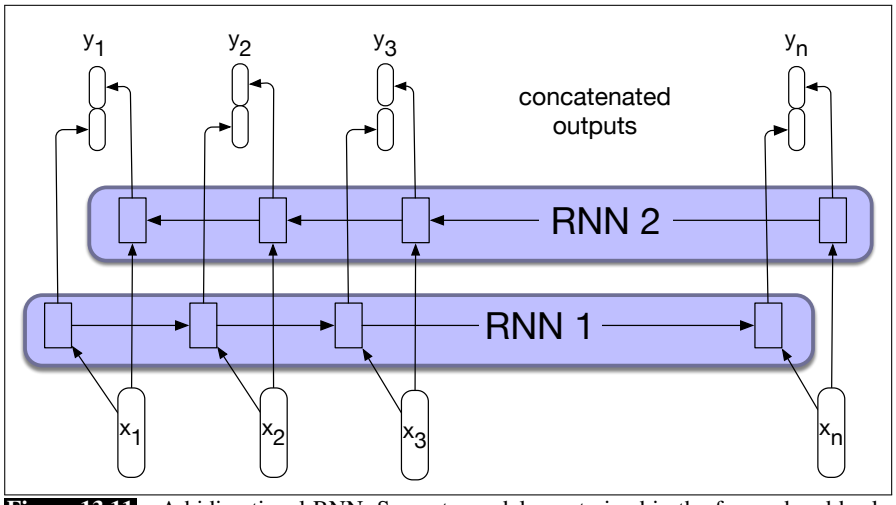


Figure 13.11 A bidirectional RNN. Separate models are trained in the forward and backward directions, with the output of each model at each time point concatenated to represent the bidirectional state at that time point.

Bidirectional RNNs have also proven to be quite effective for sequence classification. Recall from Fig. 13.8 that for sequence classification we used the final hidden state of the RNN as the input to a subsequent feedforward classifier. A difficulty with this approach is that the final state naturally reflects more information about the end of the sentence than its beginning. Bidirectional RNNs provide a simple solution to this problem; as shown in Fig. 13.12, we simply combine the final hidden states from the forward and backward passes (for example by concatenation) and use that as input for follow-on processing.

13.5 The LSTM

In practice, it is quite difficult to train RNNs for tasks that require a network to make use of information distant from the current point of processing. Despite having access to the entire preceding sequence, the information encoded in hidden states tends to be fairly local, more relevant to the most recent parts of the input sequence and recent decisions. Yet distant information is critical to many language applications. Consider the following example in the context of language modeling.

(13.19) The flights the airline was canceling were full.

Assigning a high probability to *was* following *airline* is straightforward since *airline* provides a strong local context for the singular agreement. However, assigning an appropriate probability to *were* is quite difficult, not only because the plural *flights* is quite distant, but also because the singular noun *airline* is closer in the intervening

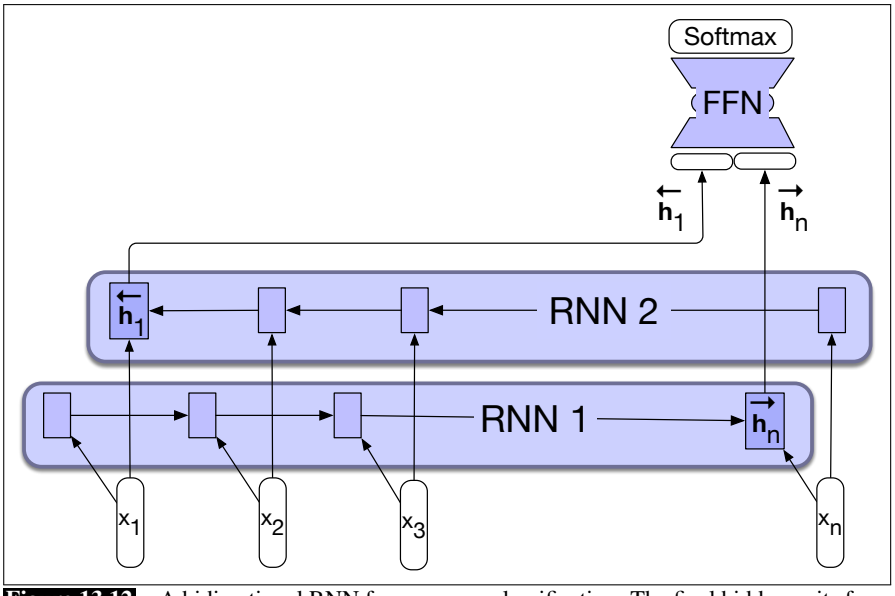


Figure 13.12 A bidirectional RNN for sequence classification. The final hidden units from the forward and backward passes are combined to represent the entire sequence. This combined representation serves as input to the subsequent classifier.

context. Ideally, a network should be able to retain the distant information about plural *flights* until it is needed, while still processing the intermediate parts of the sequence correctly.

One reason for the inability of RNNs to carry forward critical information is that the hidden layers, and, by extension, the weights that determine the values in the hidden layer, are being asked to perform two tasks simultaneously: provide information useful for the current decision, and updating and carrying forward information required for future decisions.

A second difficulty with training RNNs arises from the need to backpropagate the error signal back through time. Recall from Section 13.1.2 that the hidden layer at time t contributes to the loss at the next time step since it takes part in that calculation. As a result, during the backward pass of training, the hidden layers are subject to repeated multiplications, as determined by the length of the sequence. A frequent result of this process is that the gradients are eventually driven to zero, a situation called the **vanishing gradients** problem.

vanishing gradients

To address these issues, more complex network architectures have been designed to explicitly manage the task of maintaining relevant context over time, by enabling the network to learn to forget information that is no longer needed and to remember information required for decisions still to come.

long short-term memory

The most commonly used such extension to RNNs is the **long short-term memory** (LSTM) network (Hochreiter and Schmidhuber, 1997). LSTMs divide the context management problem into two subproblems: removing information no longer needed from the context, and adding information likely to be needed for later decision making. The key to solving both problems is to learn how to manage this context rather than hard-coding a strategy into the architecture. LSTMs accomplish this by first adding an explicit context layer to the architecture (in addition to the usual recurrent hidden layer), and through the use of specialized neural units that make use of *gates* to control the flow of information into and out of the units that

comprise the network layers. These gates are implemented through the use of additional weights that operate sequentially on the input, and previous hidden layer, and previous context layers.

The gates in an LSTM share a common design pattern; each consists of a feed-forward layer, followed by a sigmoid activation function, followed by a pointwise multiplication with the layer being gated. The choice of the sigmoid as the activation function arises from its tendency to push its outputs to either 0 or 1. Combining this with a pointwise multiplication has an effect similar to that of a binary mask. Values in the layer being gated that align with values near 1 in the mask are passed through nearly unchanged; values corresponding to lower values are essentially erased.

forget gate

The first gate we'll consider is the **forget gate**. The purpose of this gate is to delete information from the context that is no longer needed. The forget gate computes a weighted sum of the previous state's hidden layer and the current input and passes that through a sigmoid. This mask is then multiplied element-wise by the context vector to remove the information from context that is no longer required. Element-wise multiplication of two vectors (represented by the operator $\odot$, and sometimes called the **Hadamard product**) is the vector of the same dimension as the two input vectors, where each element i is the product of element i in the two input vectors:

$$\mathbf{f}_t = \sigma(\mathbf{U}_f\mathbf{h}_{t-1} + \mathbf{W}_f\mathbf{x}_t) \tag{13.20}$$

$$\mathbf{k}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t \tag{13.21}$$

The next task is to compute the actual information we need to extract from the previous hidden state and current inputs—the same basic computation we've been using for all our recurrent networks.

$$\mathbf{g}_t = \tanh(\mathbf{U}_g\mathbf{h}_{t-1} + \mathbf{W}_g\mathbf{x}_t) \tag{13.22}$$

add gate

Next, we generate the mask for the **add gate** to select the information to add to the current context.

$$\mathbf{i}_t = \sigma(\mathbf{U}_i\mathbf{h}_{t-1} + \mathbf{W}_i\mathbf{x}_t) \tag{13.23}$$

$$\mathbf{j}_t = \mathbf{g}_t \odot \mathbf{i}_t \tag{13.24}$$

Next, we add this to the modified context vector to get our new context vector.

$$\mathbf{c}_t = \mathbf{j}_t + \mathbf{k}_t \tag{13.25}$$

output gate

The final gate we'll use is the **output gate** which is used to decide what information is required for the current hidden state (as opposed to what information needs to be preserved for future decisions).

$$\mathbf{o}_t = \sigma(\mathbf{U}_o\mathbf{h}_{t-1} + \mathbf{W}_o\mathbf{x}_t) \tag{13.26}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \tag{13.27}$$

Fig. 13.13 illustrates the complete computation for a single LSTM unit. Given the appropriate weights for the various gates, an LSTM accepts as input the context layer, and hidden layer from the previous time step, along with the current input vector. It then generates updated context and hidden vectors as output.

It is the hidden state, h_t, that provides the output for the LSTM at each time step. This output can be used as the input to subsequent layers in a stacked RNN, or at the final layer of a network h_t can be used to provide the final output of the LSTM.

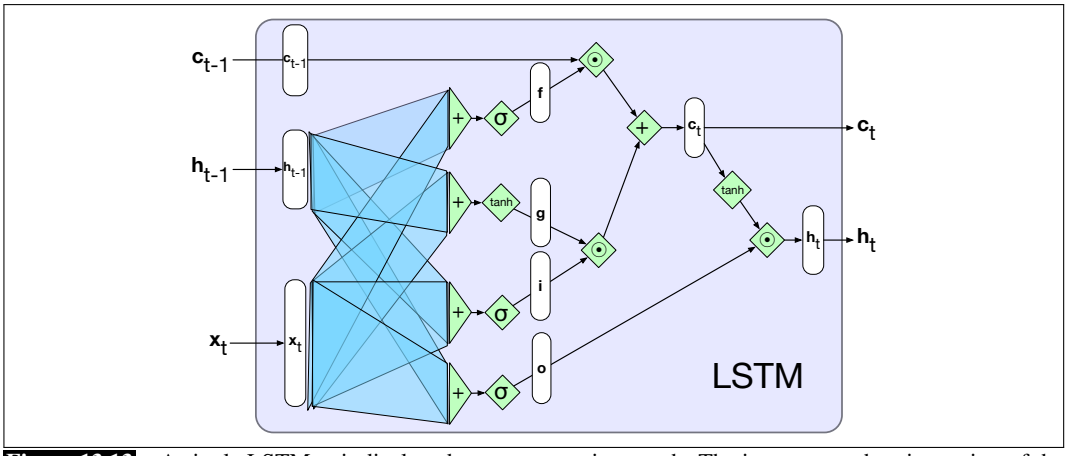


Figure 13.13 A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, x, the previous hidden state, h_{t-1}, and the previous context, c_{t-1}. The outputs are a new hidden state, h_t and an updated context, c_t.

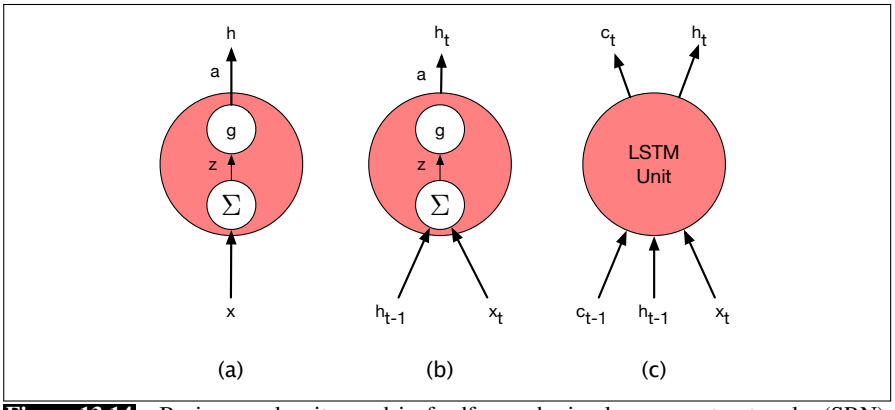


Figure 13.14 Basic neural units used in feedforward, simple recurrent networks (SRN), and long short-term memory (LSTM).

13.5.1 Gated Units, Layers and Networks

The neural units used in LSTMs are obviously much more complex than those used in basic feedforward networks. Fortunately, this complexity is encapsulated within the basic processing units, allowing us to maintain modularity and to easily experiment with different architectures. To see this, consider Fig. 13.14 which illustrates the inputs and outputs associated with each kind of unit.

At the far left, (a) is the basic feedforward unit where a single set of weights and a single activation function determine its output, and when arranged in a layer there are no connections among the units in the layer. Next, (b) represents the unit in a simple recurrent network. Now there are two inputs and an additional set of weights to go with it. However, there is still a single activation function and output.

The increased complexity of the LSTM units is encapsulated within the unit itself. The only additional external complexity for the LSTM over the basic recurrent unit (b) is the presence of the additional context vector as an input and output.

This modularity is key to the power and widespread applicability of LSTM units. LSTM units (or other varieties, like GRUs) can be substituted into any of the network architectures described in Section 13.4. And, as with simple RNNs, multi-layered

networks making use of gated units can be unrolled into deep feedforward networks and trained in the usual fashion with backpropagation. In practice, therefore, LSTMs rather than RNNs have become the standard unit for any modern system that makes use of recurrent networks.

13.6 Summary: Common RNN NLP Architectures

We've now introduced the RNN, seen advanced components like stacking multiple layers and using the LSTM version, and seen how the RNN can be applied to various tasks. Let's take a moment to summarize the architectures for these applications.

Fig. 13.15 shows the three architectures we've discussed so far: sequence labeling, sequence classification, and language modeling. In sequence labeling (for example for part of speech tagging), we train a model to produce a label for each input word or token. In sequence classification, for example for sentiment analysis, we ignore the output for each token, and only take the value from the end of the sequence (and similarly the model's training signal comes from backpropagation from that last token). In language modeling, we train the model to predict the next word at each token step. In the next section we'll introduce a fourth architecture, the **encoder-decoder**.

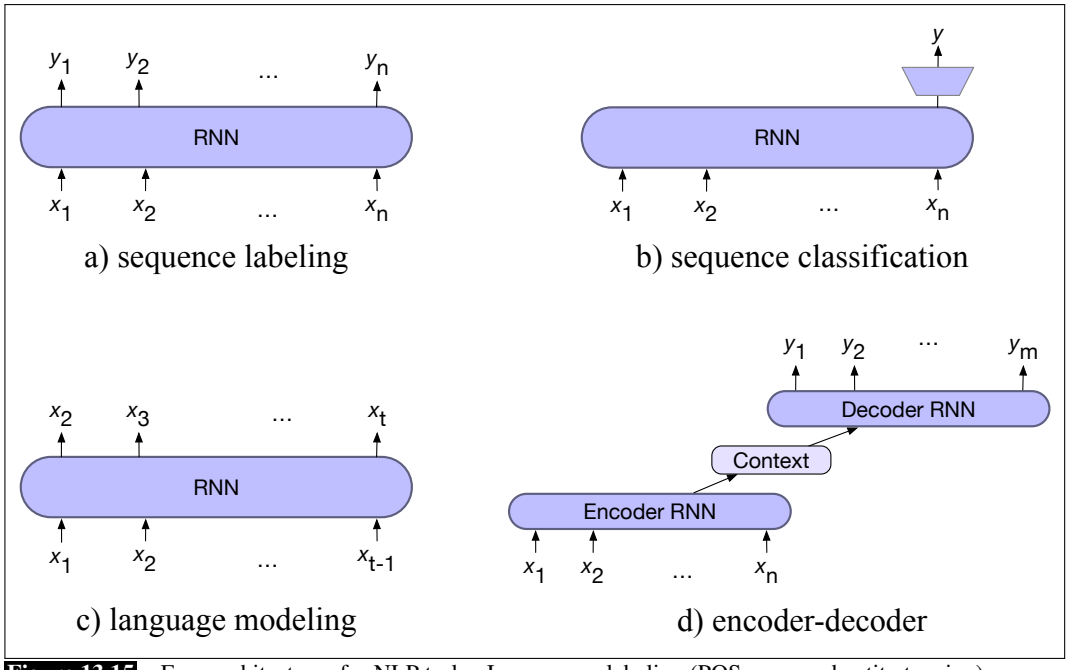


Figure 13.15 Four architectures for NLP tasks. In sequence labeling (POS or named entity tagging) we map each input token x_i to an output token y_i. In sequence classification we map the entire input sequence to a single class. In language modeling we output the next token conditioned on previous tokens. In the encoder model we have two separate RNN models, one of which maps from an input sequence **x** to an intermediate representation we call the **context**, and a second of which maps from the context to an output sequence **y**.

13.7 The Encoder-Decoder Model with RNNs

In this section we introduce the **encoder-decoder** model, which is used when we are taking an input sequence and translating it to an output sequence that is of a different length than the input, and doesn't align with it in a word-to-word way.

Those of you who already read Chapter 12 will have already seen this model in the transformer architecture, and its application to machine translation, but we introduce this architecture again here for those who come to the concepts in a different order and are reading about RNNs before transformers.

Recall that in the sequence labeling task, we have two sequences, but they are the same length (for example in part-of-speech tagging each token gets an associated tag), each input is associated with a specific output, and the labeling for that output takes mostly local information. Thus deciding whether a word is a verb or a noun, we look mostly at the word and the neighboring words.

By contrast, encoder-decoder models are used especially for tasks like machine translation, where the input sequence and output sequence can have different lengths and the mapping between a token in the input and a token in the output can be very indirect (in some languages the verb appears at the beginning of the sentence; in other languages at the end). We introduced machine translation in Chapter 12, but for now we'll just point out that the mapping for a sentence in English to a sentence in Tagalog or Yoruba can have very different numbers of words, and the words can be in a very different order.

encoder-decoder

Encoder-decoder networks, sometimes called **sequence-to-sequence** networks, are models capable of generating contextually appropriate, arbitrary length, output sequences given an input sequence. Encoder-decoder networks have been applied to a very wide range of applications including summarization, question answering, and dialogue, but they are particularly popular for machine translation.

The key idea underlying these networks is the use of an **encoder** network that takes an input sequence and creates a contextualized representation of it, often called the **context**. This representation is then passed to a **decoder** which generates a task-specific output sequence. Fig. 13.16 illustrates the architecture.

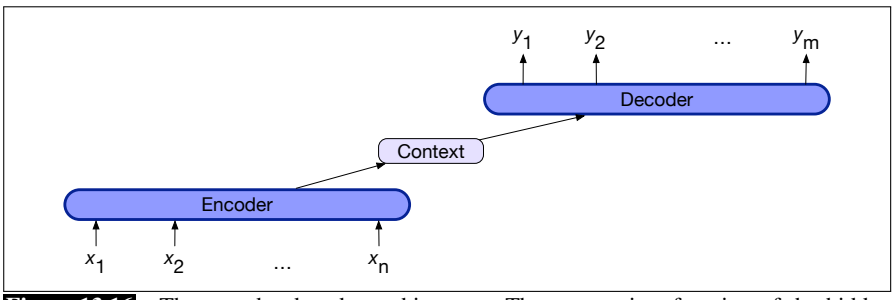


Figure 13.16 The encoder-decoder architecture. The context is a function of the hidden representations of the input, and may be used by the decoder in a variety of ways.

Encoder-decoder networks consist of three conceptual components:

1. An **encoder** that accepts an input sequence, $x_{1:n}$, and generates a corresponding sequence of contextualized representations, $h_{1:n}$. LSTMs, convolutional networks, and transformers can all be employed as encoders.
2. A **context vector**, c, which is a function of $h_{1:n}$, and conveys the essence of the input to the decoder.

3. A **decoder**, which accepts c as input and generates an arbitrary length sequence of hidden states $h_{1:m}$, from which a corresponding sequence of output states $y_{1:m}$, can be obtained. Just as with encoders, decoders can be realized by any kind of sequence architecture.

In this section we'll describe an encoder-decoder network based on a pair of RNNs, but we'll see in Chapter 12 how to apply them to transformers as well. We'll build up the equations for encoder-decoder models by starting with the conditional RNN language model $p(y)$, the probability of a sequence y.

Recall that in any language model, we can break down the probability as follows:

$$p(y) = p(y_1)p(y_2|y_1)p(y_3|y_1,y_2)\dots p(y_m|y_1,...,y_{m-1}) \tag{13.28}$$

In RNN language modeling, at a particular time t, we pass the prefix of $t-1$ tokens through the language model, using forward inference to produce a sequence of hidden states, ending with the hidden state corresponding to the last word of the prefix. We then use the final hidden state of the prefix as our starting point to generate the next token.

More formally, if g is an activation function like *tanh* or ReLU, a function of the input at time t and the hidden state at time $t-1$, and the softmax is over the set of possible vocabulary items, then at time t the output $\mathbf{y}_t$ and hidden state $\mathbf{h}_t$ are computed as:

$$\mathbf{h}_t = g(\mathbf{h}_{t-1},\mathbf{x}_t) \tag{13.29}$$
$$\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{h}_t) \tag{13.30}$$

We only have to make one slight change to turn this language model with autoregressive generation into an encoder-decoder model that is a translation model that can translate from a **source text** in one language to a **target text** in a second: add a **sentence separation** marker at the end of the source text, and then simply concatenate the target text.

sentence separation

Let's use <s> for our sentence separator token, and let's think about translating an English source text ("the green witch arrived"), to a Spanish sentence ("*llegó la bruja verde*" (which can be glossed word-by-word as 'arrived the witch green'). We could also illustrate encoder-decoder models with a question-answer pair, or a text-summarization pair.

Let's use x to refer to the source text (in this case in English) plus the separator token <s>, and y to refer to the target text y (in this case in Spanish). Then an encoder-decoder model computes the probability $p(y|x)$ as follows:

$$p(y|x) = p(y_1|x)p(y_2|y_1,x)p(y_3|y_1,y_2,x)\dots p(y_m|y_1,...,y_{m-1},x) \tag{13.31}$$

Fig. 13.17 shows the setup for a simplified version of the encoder-decoder model (we'll see the full model, which requires the new concept of **attention**, in the next section).

Fig. 13.17 shows an English source text ("the green witch arrived"), a sentence separator token (<s>, and a Spanish target text ("*llegó la bruja verde*"). To translate a source text, we run it through the network performing forward inference to generate hidden states until we get to the end of the source. Then we begin autoregressive generation, asking for a word in the context of the hidden layer from the end of the source input as well as the end-of-sentence marker. Subsequent words are conditioned on the previous hidden state and the embedding for the last word generated.

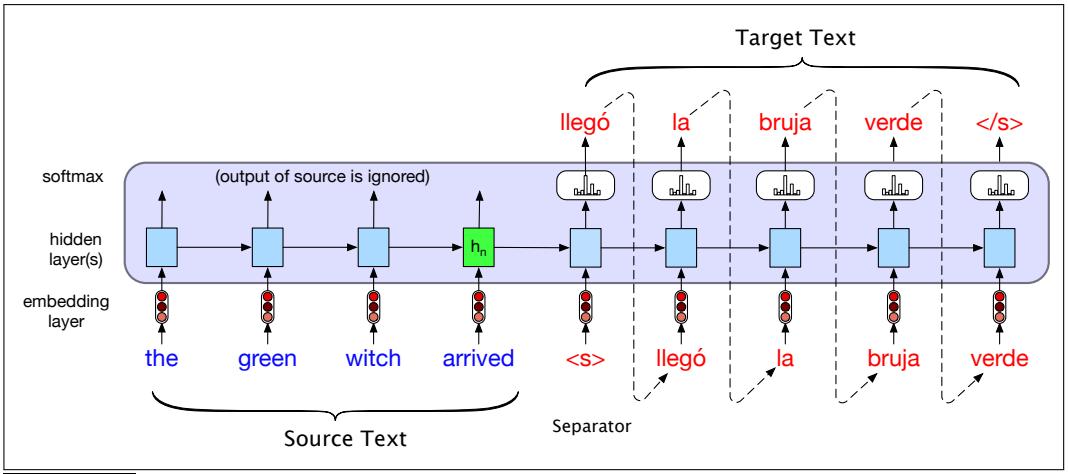


Figure 13.17 Translating a single sentence (inference time) in the basic RNN version of encoder-decoder approach to machine translation. Source and target sentences are concatenated with a separator token in between, and the decoder uses context information from the encoder's last hidden state.

Let's formalize and generalize this model a bit in Fig. 13.18. (To help keep things straight, we'll use the superscripts e and d where needed to distinguish the hidden states of the encoder and the decoder.) The elements of the network on the left process the input sequence x and comprise the **encoder**. While our simplified figure shows only a single network layer for the encoder, stacked architectures are the norm, where the output states from the top layer of the stack are taken as the final representation, and the encoder consists of stacked biLSTMs where the hidden states from top layers from the forward and backward passes are concatenated to provide the contextualized representations for each time step.

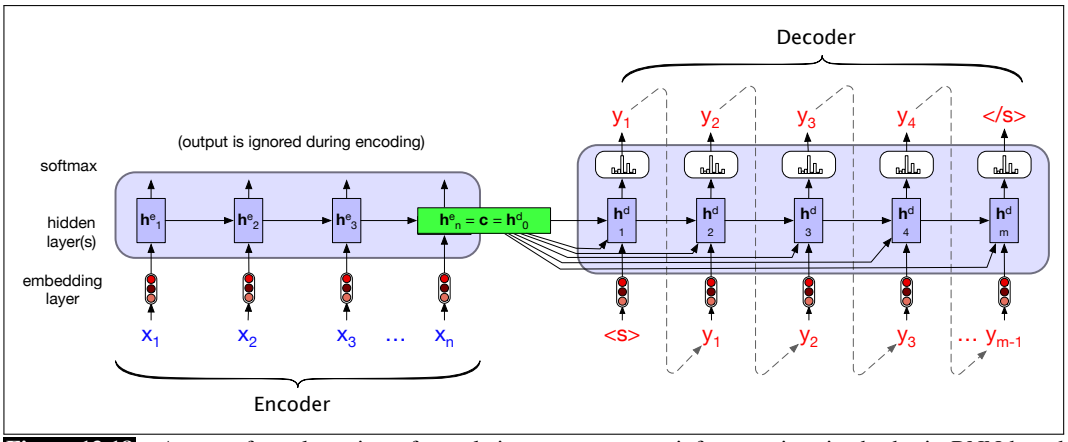


Figure 13.18 A more formal version of translating a sentence at inference time in the basic RNN-based encoder-decoder architecture. The final hidden state of the encoder RNN, h_n^e, serves as the context for the decoder in its role as h_0^d in the decoder RNN, and is also made available to each decoder hidden state.

The entire purpose of the encoder is to generate a contextualized representation of the input. This representation is embodied in the final hidden state of the encoder, $\mathbf{h}_n^e$. This representation, also called $\mathbf{c}$ for **context**, is then passed to the decoder.

The simplest version of the **decoder** network would take this state and use it just to initialize the first hidden state of the decoder; the first decoder RNN cell would

use c as its prior hidden state $\mathbf{h}_0^d$. The decoder would then autoregressively generate a sequence of outputs, an element at a time, until an end-of-sequence marker is generated. Each hidden state is conditioned on the previous hidden state and the output generated in the previous state.

As Fig. 13.18 shows, we do something more complex: we make the context vector $\mathbf{c}$ available to more than just the first decoder hidden state, to ensure that the influence of the context vector, $\mathbf{c}$, doesn't wane as the output sequence is generated. We do this by adding $\mathbf{c}$ as a parameter to the computation of the current hidden state. using the following equation:

$$\mathbf{h}_t^d = g(\hat{y}_{t-1}, \mathbf{h}_{t-1}^d, \mathbf{c}) \tag{13.32}$$

Now we're ready to see the full equations for this version of the decoder in the basic encoder-decoder model, with context available at each decoding timestep. Recall that g is a stand-in for some flavor of RNN and $\hat{y}_{t-1}$ is the embedding for the output sampled from the softmax at the previous step:

$$\begin{aligned} \mathbf{c} &= \mathbf{h}_n^e \\ \mathbf{h}_0^d &= \mathbf{c} \\ \mathbf{h}_t^d &= g(\hat{y}_{t-1}, \mathbf{h}_{t-1}^d, \mathbf{c}) \\ \hat{\mathbf{y}}_t &= \text{softmax}(\mathbf{h}_t^d) \end{aligned} \tag{13.33}$$

Thus $\hat{\mathbf{y}}_t$ is a vector of probabilities over the vocabulary, representing the probability of each word occurring at time t. To generate text, we sample from this distribution $\hat{\mathbf{y}}_t$. For example, the greedy choice is simply to choose the most probable word to generate at each timestep. We discussed other sampling methods in Section 7.4.

13.7.1 Training the Encoder-Decoder Model

Encoder-decoder architectures are trained end-to-end. Each training example is a tuple of paired strings, a source and a target. Concatenated with a separator token, these source-target pairs can now serve as training data.

For MT, the training data typically consists of sets of sentences and their translations. These can be drawn from standard datasets of aligned sentence pairs, as we'll discuss in Section 12.2.2. Once we have a training set, the training itself proceeds as with any RNN-based language model. The network is given the source text and then starting with the separator token is trained autoregressively to predict the next word, as shown in Fig. 13.19.

Note the differences between training (Fig. 13.19) and inference (Fig. 13.17) with respect to the outputs at each time step. The decoder during inference uses its own estimated output $\hat{y}_t$ as the input for the next time step x_{t+1}. Thus the decoder will tend to deviate more and more from the gold target sentence as it keeps generating more tokens. In training, therefore, it is more common to use **teacher forcing** in the decoder. Teacher forcing means that we force the system to use the gold target token from training as the next input x_{t+1}, rather than allowing it to rely on the (possibly erroneous) decoder output $\hat{y}_t$. This speeds up training.

teacher forcing

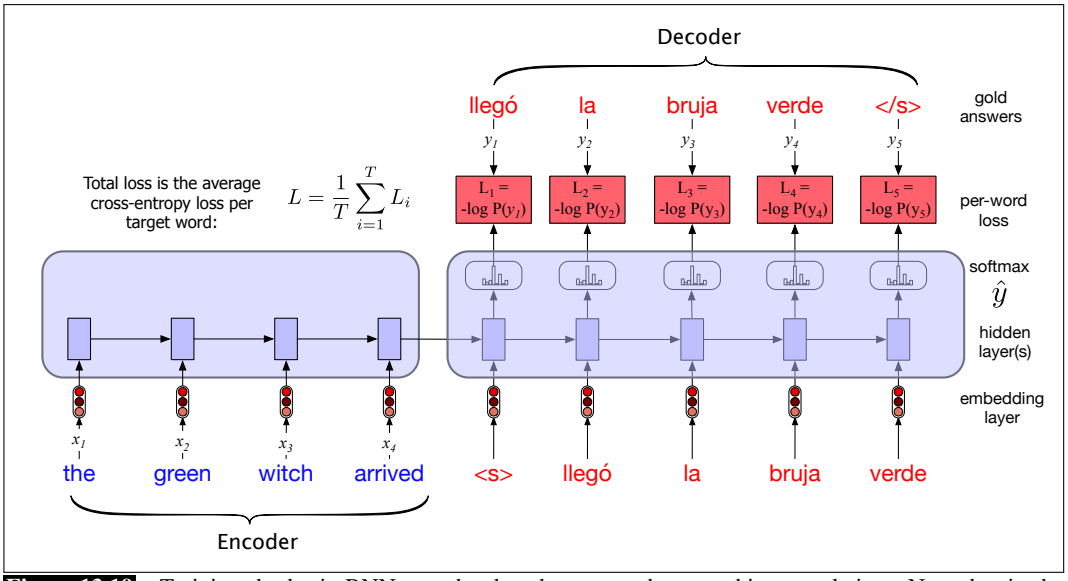


Figure 13.19 Training the basic RNN encoder-decoder approach to machine translation. Note that in the decoder we usually don't propagate the model's softmax outputs $\hat{y}_t$, but use **teacher forcing** to force each input to the correct gold value for training. We compute the softmax output distribution over $\hat{y}$ in the decoder in order to compute the loss at each token, which can then be averaged to compute a loss for the sentence. This loss is then propagated through the decoder parameters and the encoder parameters.

13.8 Attention

The simplicity of the encoder-decoder model is its clean separation of the encoder—which builds a representation of the source text—from the decoder, which uses this context to generate a target text. In the model as we've described it so far, this context vector is h_n, the hidden state of the last (n^{th}) time step of the source text. This final hidden state is thus acting as a **bottleneck**: it must represent absolutely everything about the meaning of the source text, since the only thing the decoder knows about the source text is what's in this context vector (Fig. 13.20). Information at the beginning of the sentence, especially for long sentences, may not be equally well represented in the context vector.

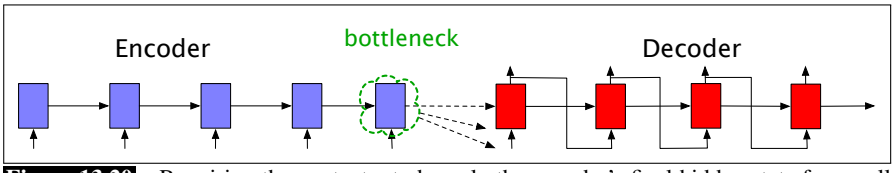


Figure 13.20 Requiring the context c to be only the encoder's final hidden state forces all the information from the entire source sentence to pass through this representational bottleneck.

attention mechanism

The **attention mechanism** is a solution to the bottleneck problem, a way of allowing the decoder to get information from *all* the hidden states of the encoder, not just the last hidden state.

In the attention mechanism, as in the vanilla encoder-decoder model, the context vector $\mathbf{c}$ is a single vector that is a function of the hidden states of the encoder. But instead of being taken from the last hidden state, it's a weighted average of **all** the

hidden states of the encoder. And this weighted average is also informed by part of the decoder state as well, the state of the decoder right before the current token i. That is, $\mathbf{c}_i = f(\mathbf{h}_1^e \ldots \mathbf{h}_n^e, \mathbf{h}_{i-1}^d)$. The weights focus on ('attend to') a particular part of the source text that is relevant for the token i that the decoder is currently producing. Attention thus replaces the static context vector with one that is dynamically derived from the encoder hidden states, but also informed by and hence different for each token in decoding.

This context vector, $\mathbf{c}_i$, is generated anew with each decoding step i and takes all of the encoder hidden states into account in its derivation. We then make this context available during decoding by conditioning the computation of the current decoder hidden state on it (along with the prior hidden state and the previous output generated by the decoder), as we see in this equation (and Fig. 13.21):

$$\mathbf{h}_i^d = g(\hat{y}_{i-1}, \mathbf{h}_{i-1}^d, \mathbf{c}_i) \tag{13.34}$$

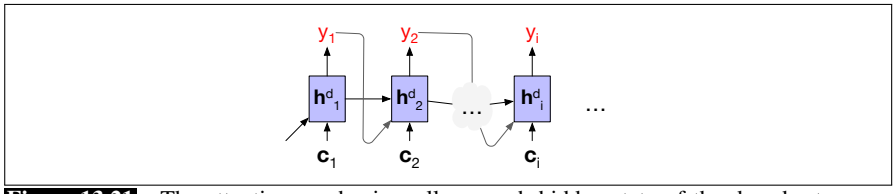


Figure 13.21 The attention mechanism allows each hidden state of the decoder to see a different, dynamic, context, which is a function of all the encoder hidden states.

The first step in computing $\mathbf{c}_i$ is to compute how much to focus on each encoder state, how *relevant* each encoder state is to the decoder state captured in $\mathbf{h}_{i-1}^d$. We capture relevance by computing— at each state i during decoding—a $score(\mathbf{h}_{i-1}^d, \mathbf{h}_j^e)$ for each encoder state j.

dot-product attention

The simplest such score, called **dot-product attention**, implements relevance as similarity: measuring how similar the decoder hidden state is to an encoder hidden state, by computing the dot product between them:

$$\text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_j^e) = \mathbf{h}_{i-1}^d \cdot \mathbf{h}_j^e \tag{13.35}$$

The score that results from this dot product is a scalar that reflects the degree of similarity between the two vectors. The vector of these scores across all the encoder hidden states gives us the relevance of each encoder state to the current step of the decoder.

To make use of these scores, we'll normalize them with a softmax to create a vector of weights, α_{ij}, that tells us the proportional relevance of each encoder hidden state j to the prior hidden decoder state, h_{i-1}^d.

$$\begin{aligned} \alpha_{ij} &= \text{softmax}(\text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_j^e)) \\ &= \frac{\exp(\text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_j^e)}{\sum_k \exp(\text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_k^e))} \end{aligned} \tag{13.36}$$

Finally, given the distribution in α, we can compute a fixed-length context vector for the current decoder state by taking a weighted average over all the encoder hidden states.

$$\mathbf{c}_i = \sum_j \alpha_{ij} \mathbf{h}_j^e \tag{13.37}$$

With this, we finally have a fixed-length context vector that takes into account information from the entire encoder state that is dynamically updated to reflect the needs of the decoder at each step of decoding. Fig. 13.22 illustrates an encoder-decoder network with attention, focusing on the computation of one context vector $\mathbf{c}_i$.

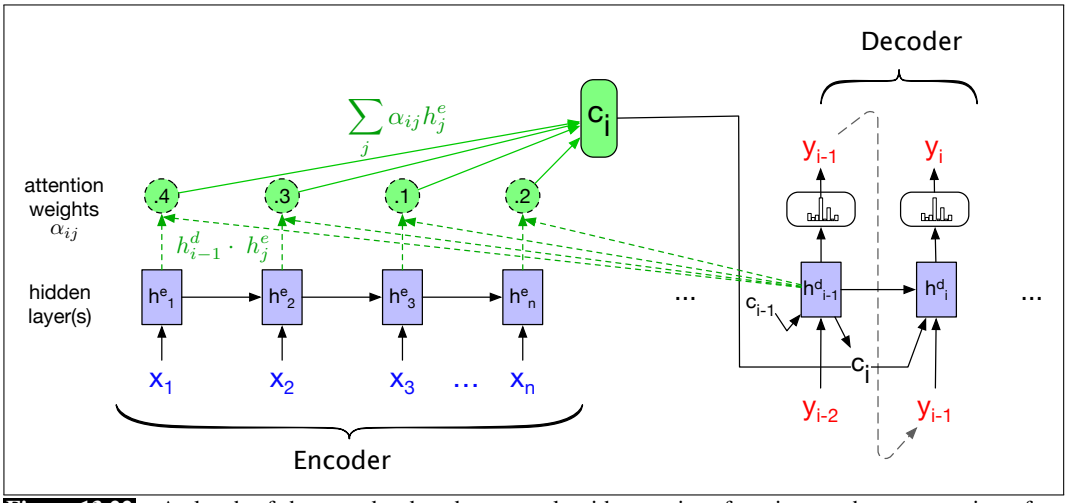


Figure 13.22 A sketch of the encoder-decoder network with attention, focusing on the computation of $\mathbf{c}_i$. The context value $\mathbf{c}_i$ is one of the inputs to the computation of $\mathbf{h}^d_i$. It is computed by taking the weighted sum of all the encoder hidden states, each weighted by their dot product with the prior decoder hidden state $\mathbf{h}^d_{i-1}$.

It's also possible to create more sophisticated scoring functions for attention models. Instead of simple dot product attention, we can get a more powerful function that computes the relevance of each encoder hidden state to the decoder hidden state by parameterizing the score with its own set of weights, $\mathbf{W}_s$.

$$\text{score}(\mathbf{h}^d_{i-1}, \mathbf{h}^e_j) = \mathbf{h}^d_{i-1} \mathbf{W}_s \mathbf{h}^e_j \tag{13.38}$$

The weights W_s, which are then trained during normal end-to-end training, give the network the ability to learn which aspects of similarity between the decoder and encoder states are important to the current application. This bilinear model also allows the encoder and decoder to use different dimensional vectors, whereas the simple dot-product attention requires that the encoder and decoder hidden states have the same dimensionality.

We'll return to the concept of attention when we define the transformer architecture in Chapter 8, which is based on a slight modification of attention called **self-attention**.

13.9 Summary

This chapter has introduced the concepts of recurrent neural networks and how they can be applied to language problems. Here's a summary of the main points that we covered:

- In simple Recurrent Neural Networks sequences are processed one element at a time, with the output of each neural unit at time t based both on the current input at t and the hidden layer from time $t-1$.

- RNNs can be trained with a straightforward extension of the backpropagation algorithm, known as **backpropagation through time** (BPTT).
- Simple recurrent networks fail on long inputs because of problems like **vanishing gradients**; instead modern systems use more complex gated architectures such as **LSTMs** that explicitly decide what to remember and forget in their hidden and context layers.
- Common language-based applications for RNNs include:
  - Probabilistic language modeling: assigning a probability to a sequence, or to the next element of a sequence given the preceding words.
  - Auto-regressive generation using a trained language model.
  - Sequence labeling like part-of-speech tagging, where each element of a sequence is assigned a label.
  - Sequence classification, where an entire text is assigned to a category, as in spam detection, sentiment analysis or topic classification.
  - Encoder-decoder architectures, where an input is mapped to an output of different length and alignment.

Historical Notes

Influential investigations of RNNs were conducted in the context of the Parallel Distributed Processing (PDP) group at UC San Diego in the 1980's. Much of this work was directed at human cognitive modeling rather than practical NLP applications (Rumelhart and McClelland 1986c, McClelland and Rumelhart 1986). Models using recurrence at the hidden layer in a feedforward network (Elman networks) were introduced by Elman (1990). Similar architectures were investigated by Jordan (1986) with a recurrence from the output layer, and Mathis and Mozer (1995) with the addition of a recurrent context layer prior to the hidden layer. The possibility of unrolling a recurrent network into an equivalent feedforward network is discussed in (Rumelhart and McClelland, 1986c).

In parallel with work in cognitive modeling, RNNs were investigated extensively in the continuous domain in the signal processing and speech communities (Giles et al. 1994, Robinson et al. 1996). Schuster and Paliwal (1997) introduced bidirectional RNNs and described results on the TIMIT phoneme transcription task.

While theoretically interesting, the difficulty with training RNNs and managing context over long sequences impeded progress on practical applications. This situation changed with the introduction of LSTMs in Hochreiter and Schmidhuber (1997) and Gers et al. (2000). Impressive performance gains were demonstrated on tasks at the boundary of signal processing and language processing including phoneme recognition (Graves and Schmidhuber, 2005), handwriting recognition (Graves et al., 2007) and most significantly speech recognition (Graves et al., 2013).

Interest in applying neural networks to practical NLP problems surged with the work of Collobert and Weston (2008) and Collobert et al. (2011). These efforts made use of learned word embeddings, convolutional networks, and end-to-end training. They demonstrated near state-of-the-art performance on a number of standard shared tasks including part-of-speech tagging, chunking, named entity recognition and semantic role labeling without the use of hand-engineered features.

Approaches that married LSTMs with pretrained collections of word-embeddings based on word2vec (Mikolov et al., 2013a) and GloVe (Pennington et al., 2014)

quickly came to dominate many common tasks: part-of-speech tagging (Ling et al., 2015), syntactic chunking (Søgaard and Goldberg, 2016), named entity recognition (Chiu and Nichols, 2016; Ma and Hovy, 2016), opinion mining (Irsoy and Cardie, 2014), semantic role labeling (Zhou and Xu, 2015a) and AMR parsing (Foland and Martin, 2016). As with the earlier surge of progress involving statistical machine learning, these advances were made possible by the availability of training data provided by CONLL, SemEval, and other shared tasks, as well as shared resources such as Ontonotes (Pradhan et al., 2007b), and PropBank (Palmer et al., 2005).

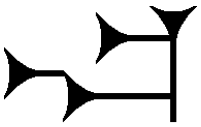
The modern neural encoder-decoder approach was pioneered by Kalchbrenner and Blunsom (2013), who used a CNN encoder and an RNN decoder. Cho et al. (2014) (who coined the name "encoder-decoder") and Sutskever et al. (2014) then showed how to use extended RNNs for both encoder and decoder. The idea that a generative decoder should take as input a soft weighting of the inputs, the central idea of attention, was first developed by Graves (2013) in the context of handwriting recognition. Bahdanau et al. (2015) extended the idea, named it "attention" and applied it to MT.

CHAPTER

14 Phonetics and Speech Feature Extraction

The characters that make up the texts we've been discussing in this book aren't just random symbols. They are also an amazing scientific invention: a theoretical model of the elements that make up human speech.

The earliest writing systems we know of (Sumerian, Chinese, Mayan) were mainly *logographic*: one symbol representing a whole word. But from the earliest stages we can find, some symbols were also used to represent the sounds that made up words. The cuneiform sign to the right pronounced *ba* and meaning "ration" in Sumerian could also function purely as the sound /ba/. The earliest Chinese characters we have, carved into bones for divination, similarly contain phonetic elements. Purely sound-based writing systems, whether syllabic (like Japanese *hiragana*), alphabetic (like the Roman alphabet), or consonantal (like Semitic writing systems), trace back to these early logo-syllabic systems, often as two cultures came together. Thus, the Arabic, Aramaic, Hebrew, Greek, and Roman systems all derive from a West Semitic script that is presumed to have been modified by Western Semitic mercenaries from a cursive form of Egyptian hieroglyphs. The Japanese syllabaries were modified from a cursive form of Chinese phonetic characters, which themselves were used in Chinese to phonetically represent the Sanskrit in the Buddhist scriptures that came to China in the Tang dynasty.



This implicit idea that the spoken word is composed of smaller units of speech underlies algorithms for both **speech recognition** (transcribing waveforms into text) and **text-to-speech** (converting text into waveforms). In this chapter we give a computational perspective on **phonetics**, the study of the speech sounds used in the languages of the world, how they are produced in the human vocal tract, how they are realized acoustically, and how they can be digitized and processed.

phonetics

14.1 Speech Sounds and Phonetic Transcription

A letter like 'p' or 'a' is already a useful model of the sounds of human speech, and indeed we'll see in Chapter 15 how to map between letters and waveforms. Nonetheless, it is helpful to represent sounds slightly more abstractly. We'll represent the pronunciation of a word as a string of **phones**, which are speech sounds, each represented with symbols adapted from the Roman alphabet.

phone

The standard phonetic representation for transcribing the world's languages is the **International Phonetic Alphabet** (**IPA**), an evolving standard first developed in 1888. But in this chapter we'll instead represent phones with the ARPAbet (Shoup, 1980), a simple phonetic alphabet (Fig. 14.1) that conveniently uses ASCII symbols to represent an American-English subset of the IPA.

IPA

Many of the IPA and ARPAbet symbols are equivalent to familiar Roman letters. So, for example, the ARPAbet phone [p] represents the consonant sound at the

ARPAbet Symbol	IPA Symbol	Word	ARPAbet Transcription	ARPAbet Symbol	IPA Symbol	Word	ARPAbet Transcription
[p]	[p]	parsley	[p aa r s l iy]	[iy]	[i]	lily	[l ih l iy]
[t]	[t]	tea	[t iy]	[ih]	[ɪ]	lily	[l ih l iy]
[k]	[k]	cook	[k uh k]	[ey]	[eɪ]	daisy	[d ey z iy]
[b]	[b]	bay	[b ey]	[eh]	[ɛ]	pen	[p eh n]
[d]	[d]	dill	[d ih l]	[ae]	[æ]	aster	[ae s t axr]
[g]	[g]	garlic	[g aa r l ix k]	[aa]	[ɑ]	poppy	[p aa p iy]
[m]	[m]	mint	[m ih n t]	[ao]	[ɔ]	orchid	[ao r k ix d]
[n]	[n]	nutmeg	[n ah t m eh g]	[uh]	[ʊ]	wood	[w uh d]
[ng]	[ŋ]	baking	[b ey k ix ng]	[ow]	[oʊ]	lotus	[l ow dx ax s]
[f]	[f]	flour	[f l aw axr]	[uw]	[u]	tulip	[t uw l ix p]
[v]	[v]	clove	[k l ow v]	[ah]	[ʌ]	butter	[b ah dx axr]
[th]	[θ]	thick	[th ih k]	[er]	[ɝ]	bird	[b er d]
[dh]	[ð]	those	[dh ow z]	[ay]	[aɪ]	iris	[ay r ix s]
[s]	[s]	soup	[s uw p]	[aw]	[aʊ]	flower	[f l aw axr]
[z]	[z]	eggs	[eh g z]	[oy]	[oɪ]	soil	[s oy l]
[sh]	[ʃ]	squash	[s k w aa sh]	[ax]	[ə]	pita	[p iy t ax]
[zh]	[ʒ]	ambrosia	[ae m b r ow zh ax]				
[ch]	[tʃ]	cherry	[ch eh r iy]				
[jh]	[dʒ]	jar	[jh aa r]				
[l]	[l]	licorice	[l ih k axr ix sh]				
[w]	[w]	kiwi	[k iy w iy]				
[r]	[r]	rice	[r ay s]				
[y]	[j]	yellow	[y eh l ow]				
[h]	[h]	honey	[h ah n iy]				

Figure 14.1 ARPAbet and IPA symbols for English consonants (left) and vowels (right).

beginning of *platypus*, *puma*, and *plantain*, the middle of *leopard*, or the end of *antelope*. In general, however, the mapping between the letters of English orthography and phones is relatively **opaque**; a single letter can represent very different sounds in different contexts. The English letter *c* corresponds to phone [k] in *cougar* [k uw g axr], but phone [s] in *cell* [s eh l]. Besides appearing as *c* and *k*, the phone [k] can appear as part of *x* (*fox* [f aa k s]), as *ck* (*jackal* [jh ae k el]) and as *cc* (*raccoon* [r ae k uw n]). Many other languages, for example, Spanish, are much more **transparent** in their sound-orthography mapping than English.

pronunciation dictionary

There are a wide variety of phonetic resources for phonetic transcription. Online **pronunciation dictionaries** give phonetic transcriptions for words. The LDC distributes pronunciation lexicons for Egyptian Arabic, Dutch, English, German, Japanese, Korean, Mandarin, and Spanish. For English, the CELEX dictionary (Baayen et al., 1995) has pronunciations for 160,595 wordforms, with syllabification, stress, and morphological and part-of-speech information. The open-source CMU Pronouncing Dictionary (CMU, 1993) has pronunciations for about 134,000 wordforms, while the fine-grained 110,000 word UNISYN dictionary (Fitt, 2002), freely available for research purposes, gives syllabifications, stress, and also pronunciations for dozens of dialects of English.

Another useful resource is a **phonetically annotated corpus**, in which a collection of waveforms is hand-labeled with the corresponding string of phones. The **TIMIT corpus** (NIST, 1990), originally a joint project between Texas Instruments (TI), MIT, and SRI, is a corpus of 6300 read sentences, with 10 sentences each from

630 speakers. The 6300 sentences were drawn from a set of 2342 sentences, some selected to have particular dialect shibboleths, others to maximize phonetic diphone coverage. Each sentence in the corpus was phonetically hand-labeled, the sequence of phones was automatically aligned with the sentence wavefile, and then the automatic phone boundaries were manually hand-corrected (Seneff and Zue, 1988).

time-aligned transcription

The result is a **time-aligned transcription**: a transcription in which each phone is associated with a start and end time in the waveform, like the example in Fig. 14.2.

she	had	your	dark	suit	in	greasy	wash	water	all	year
sh iy	hv ae dcl	jh axr	dcl d aa r kcl	s ux q	en	gcl g r iy s ix	w aa sh	q w aa dx axr q	aa l	y ix axr

Figure 14.2 Phonetic transcription from the TIMIT corpus, using special ARPAbet features for narrow transcription, such as the palatalization of [d] in *had*, unreleased final stop in *dark*, glottalization of final [t] in *suit* to [q], and flap of [t] in *water*. The TIMIT corpus also includes time-alignments (not shown).

The Switchboard Transcription Project phonetically annotated corpus consists of 3.5 hours of sentences extracted from the Switchboard corpus (Greenberg et al., 1996), together with transcriptions time-aligned at the syllable level. Figure 14.3 shows an example .

0.470	0.640	0.720	0.900	0.953	1.279	1.410	1.630
dh er	k aa	n ax	v ih m	b ix	t w iy n	r ay	n aw

Figure 14.3 Phonetic transcription of the Switchboard phrase *they're kind of in between right now*. Note vowel reduction in *they're* and *of*, coda deletion in *kind* and *right*, and resyllabification (the [v] of *of* attaches as the onset of *in*). Time is given in number of seconds from the beginning of sentence to the start of each syllable.

The Buckeye corpus (Pitt et al. 2007, Pitt et al. 2005) is a phonetically transcribed corpus of spontaneous American speech, containing about 300,000 words from 40 talkers. Phonetically transcribed corpora are also available for other languages, including the Kiel corpus of German and Mandarin corpora transcribed by the Chinese Academy of Social Sciences (Li et al., 2000).

14.2 Articulatory Phonetics

articulatory phonetics

Articulatory phonetics is the study of how these phones are produced as the various organs in the mouth, throat, and nose modify the airflow from the lungs.

The Vocal Organs

Figure 14.4 shows the organs of speech. Sound is produced by the rapid movement of air. Humans produce most sounds in spoken languages by expelling air from the lungs through the windpipe (technically, the **trachea**) and then out the mouth or nose. As it passes through the trachea, the air passes through the **larynx**, commonly known as the Adam's apple or voice box. The larynx contains two small folds of muscle, the **vocal folds** (often referred to non-technically as the **vocal cords**), which can be moved together or apart. The space between these two folds is called the **glottis**. If the folds are close together (but not tightly closed), they will vibrate as air passes through them; if they are far apart, they won't vibrate. Sounds made with the vocal folds together and vibrating are called **voiced**; sounds made without this vocal cord vibration are called **unvoiced** or **voiceless**. Voiced sounds include [b], [d], [g],

glottis

voiced sound

unvoiced sound

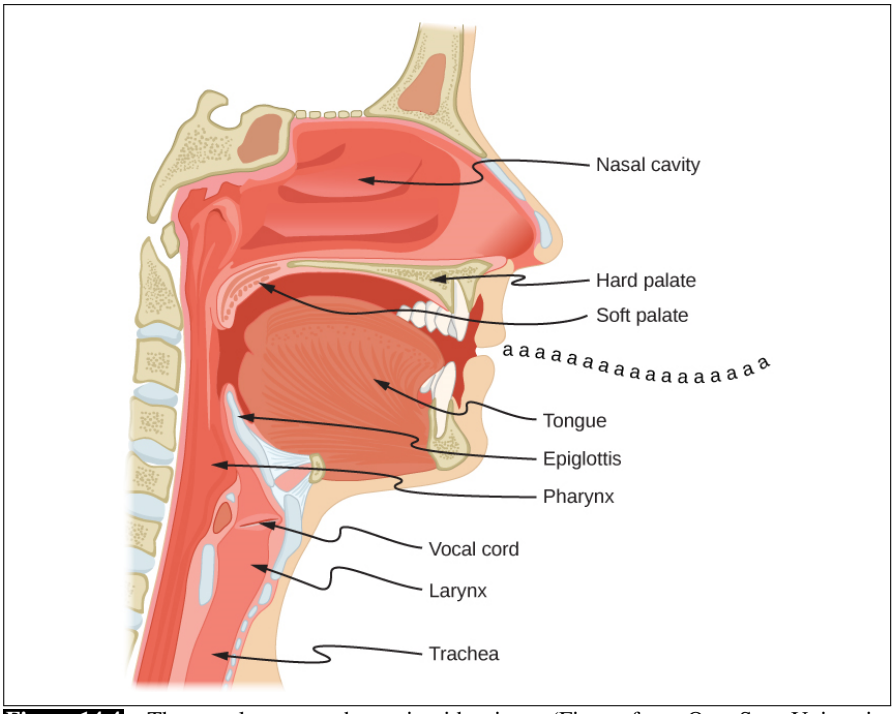


Figure 14.4 The vocal organs, shown in side view. (Figure from OpenStax University Physics, CC BY 4.0)

[v], [z], and all the English vowels, among others. Unvoiced sounds include [p], [t], [k], [f], [s], and others.

The area above the trachea is called the **vocal tract**; it consists of the **oral tract** and the **nasal tract**. After the air leaves the trachea, it can exit the body through the mouth or the nose. Most sounds are made by air passing through the mouth. Sounds made by air passing through the nose are called **nasal sounds**; nasal sounds (like English [m], [n], and [ng]) use both the oral and nasal tracts as resonating cavities.

nasal

consonant
vowel

Phones are divided into two main classes: **consonants** and **vowels**. Both kinds of sounds are formed by the motion of air through the mouth, throat or nose. Consonants are made by restriction or blocking of the airflow in some way, and can be voiced or unvoiced. Vowels have less obstruction, are usually voiced, and are generally louder and longer-lasting than consonants. The technical use of these terms is much like the common usage; [p], [b], [t], [d], [k], [g], [f], [v], [s], [z], [r], [l], etc., are consonants; [aa], [ae], [ao], [ih], [aw], [ow], [uw], etc., are vowels. **Semivowels** (such as [y] and [w]) have some of the properties of both; they are voiced like vowels, but they are short and less syllabic like consonants.

Consonants: Place of Articulation

place of articulation

Because consonants are made by restricting airflow, we can group them into classes by their point of maximum restriction, their **place of articulation** (Fig. 14.5).

labial

Labial: Consonants whose main restriction is formed by the two lips coming together have a **bilabial** place of articulation. In English these include [p] as in *possum*, [b] as in *bear*, and [m] as in *marmot*. The English **labiodental** consonants [v] and [f] are made by pressing the bottom lip against the upper row of teeth and letting the air flow through the space in the upper teeth.

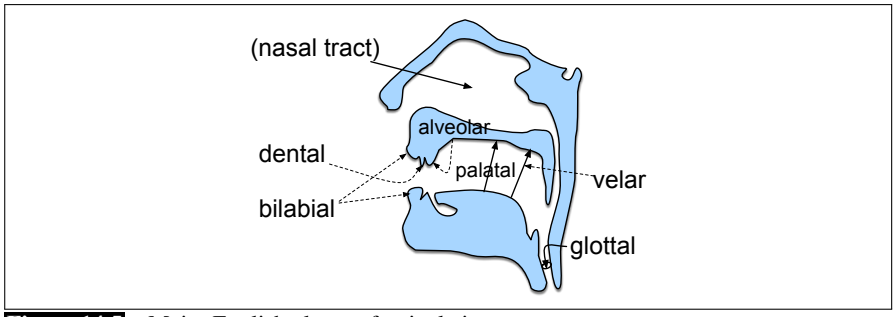


Figure 14.5 Major English places of articulation.

dental

Dental: Sounds that are made by placing the tongue against the teeth are dentals. The main dentals in English are the [th] of *thing* and the [dh] of *though*, which are made by placing the tongue behind the teeth with the tip slightly between the teeth.

alveolar

Alveolar: The alveolar ridge is the portion of the roof of the mouth just behind the upper teeth. Most speakers of American English make the phones [s], [z], [t], and [d] by placing the tip of the tongue against the alveolar ridge. The word **coronal** is often used to refer to both dental and alveolar.

palatal
palate

Palatal: The roof of the mouth (the **palate**) rises sharply from the back of the alveolar ridge. The **palato-alveolar** sounds [sh] (*shrimp*), [ch] (*china*), [zh] (*Asian*), and [jh] (*jar*) are made with the blade of the tongue against the rising back of the alveolar ridge. The palatal sound [y] of *yak* is made by placing the front of the tongue up close to the palate.

velar

Velar: The **velum**, or soft palate, is a movable muscular flap at the very back of the roof of the mouth. The sounds [k] (*cuckoo*), [g] (*goose*), and [ŋ] (*kingfisher*) are made by pressing the back of the tongue up against the velum.

glottal

Glottal: The glottal stop [q] is made by closing the glottis (by bringing the vocal folds together).

Consonants: Manner of Articulation

manner of articulation

Consonants are also distinguished by *how* the restriction in airflow is made, for example, by a complete stoppage of air or by a partial blockage. This feature is called the **manner of articulation** of a consonant. The combination of place and manner of articulation is usually sufficient to uniquely identify a consonant. Following are the major manners of articulation for English consonants:

stop

A **stop** is a consonant in which airflow is completely blocked for a short time. This blockage is followed by an explosive sound as the air is released. The period of blockage is called the **closure**, and the explosion is called the **release**. English has voiced stops like [b], [d], and [g] as well as unvoiced stops like [p], [t], and [k]. Stops are also called **plosives**.

nasal

The **nasal** sounds [n], [m], and [ng] are made by lowering the velum and allowing air to pass into the nasal cavity.

fricatives

In **fricatives**, airflow is constricted but not cut off completely. The turbulent airflow that results from the constriction produces a characteristic "hissing" sound. The English labiodental fricatives [f] and [v] are produced by pressing the lower lip against the upper teeth, allowing a restricted airflow between the upper teeth. The dental fricatives [th] and [dh] allow air to flow around the tongue between the teeth. The alveolar fricatives [s] and [z] are produced with the tongue against the

alveolar ridge, forcing air over the edge of the teeth. In the palato-alveolar fricatives [sh] and [zh], the tongue is at the back of the alveolar ridge, forcing air through a groove formed in the tongue. The higher-pitched fricatives (in English [s], [z], [sh] and [zh]) are called **sibilants**. Stops that are followed immediately by fricatives are called **affricates**; these include English [ch] (*chicken*) and [jh] (*giraffe*).

sibilants

approximant

In **approximants**, the two articulators are close together but not close enough to cause turbulent airflow. In English [y] (*yellow*), the tongue moves close to the roof of the mouth but not close enough to cause the turbulence that would characterize a fricative. In English [w] (*wood*), the back of the tongue comes close to the velum. American [r] can be formed in at least two ways; with just the tip of the tongue extended and close to the palate or with the whole tongue bunched up near the palate. [l] is formed with the tip of the tongue up against the alveolar ridge or the teeth, with one or both sides of the tongue lowered to allow air to flow over it. [l] is called a **lateral** sound because of the drop in the sides of the tongue.

tap

A **tap** or **flap** [dx] is a quick motion of the tongue against the alveolar ridge. The consonant in the middle of the word *lotus* ([l ow dx ax s]) is a tap in most dialects of American English; speakers of many U.K. dialects would use a [t] instead.

Vowels

Like consonants, vowels can be characterized by the position of the articulators as they are made. The three most relevant parameters for vowels are what is called vowel **height**, which correlates roughly with the height of the highest part of the tongue, vowel **frontness** or **backness**, indicating whether this high point is toward the front or back of the oral tract and whether the shape of the lips is **rounded** or not. Figure 14.6 shows the position of the tongue for different vowels.

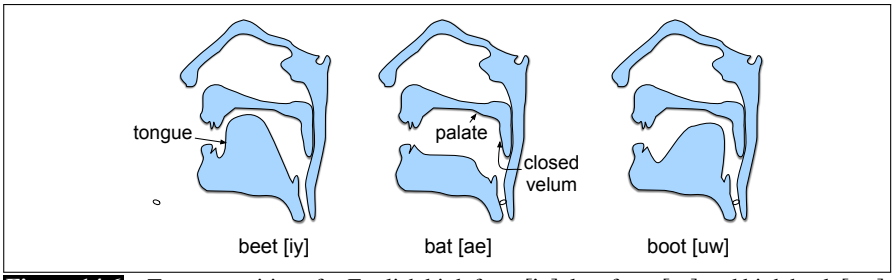


Figure 14.6 Tongue positions for English high front [iy], low front [ae] and high back [uw].

In the vowel [iy], for example, the highest point of the tongue is toward the front of the mouth. In the vowel [uw], by contrast, the high-point of the tongue is located toward the back of the mouth. Vowels in which the tongue is raised toward the front are called **front vowels**; those in which the tongue is raised toward the back are called **back vowels**. Note that while both [ih] and [eh] are front vowels, the tongue is higher for [ih] than for [eh]. Vowels in which the highest point of the tongue is comparatively high are called **high vowels**; vowels with mid or low values of maximum tongue height are called **mid vowels** or **low vowels**, respectively.

Front vowel

back vowel

high vowel

Figure 14.7 shows a schematic characterization of the height of different vowels. It is schematic because the abstract property **height** correlates only roughly with actual tongue positions; it is, in fact, a more accurate reflection of acoustic facts. Note that the chart has two kinds of vowels: those in which tongue height is represented as a point and those in which it is represented as a path. A vowel in which the tongue position changes markedly during the production of the vowel is a **diphthong**. En-

diphthong

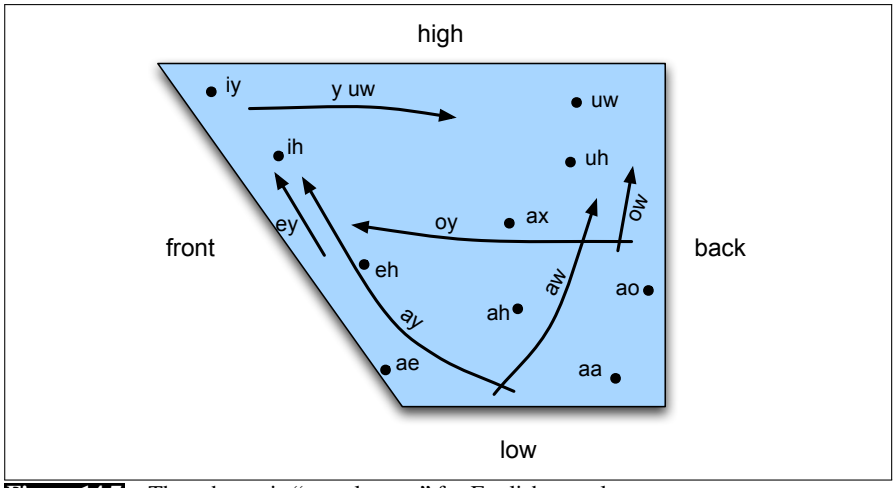


Figure 14.7 The schematic "vowel space" for English vowels.

glish is particularly rich in diphthongs.

The second important articulatory dimension for vowels is the shape of the lips. Certain vowels are pronounced with the lips rounded (the same lip shape used for whistling). These **rounded** vowels include [uw], [ao], and [ow].

rounded vowel

Syllables

syllable

Consonants and vowels combine to make a **syllable**. A syllable is a vowel-like (or **sonorant**) sound together with some of the surrounding consonants that are most closely associated with it. The word *dog* has one syllable, [d aa g] (in our dialect); the word *catnip* has two syllables, [k ae t] and [n ih p]. We call the vowel at the core of a syllable the **nucleus**. Initial consonants, if any, are called the **onset**. Onsets with more than one consonant (as in *strike* [s t r ay k]), are called **complex onsets**. The **coda** is the optional consonant or sequence of consonants following the nucleus. Thus [d] is the onset of *dog*, and [g] is the coda. The **rime**, or **rhyme**, is the nucleus plus coda. Figure 14.8 shows some sample syllable structures.

nucleus
onset
coda
rime

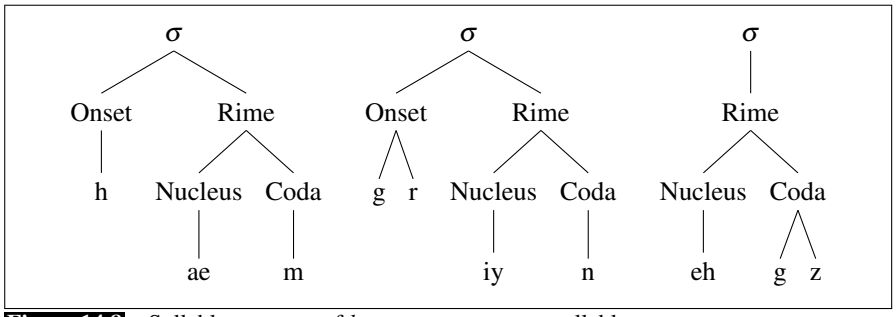


Figure 14.8 Syllable structure of *ham*, *green*, *eggs*. σ=syllable.

The task of automatically breaking up a word into syllables is called **syllabification**. Syllable structure is also closely related to the **phonotactics** of a language. The term **phonotactics** means the constraints on which phones can follow each other in a language. For example, English has strong constraints on what kinds of consonants can appear together in an onset; the sequence [zdr], for example, cannot be a

syllabification
phonotactics

legal English syllable onset. Phonotactics can be represented by a language model or finite-state model of phone sequences.

14.3 Prosody

prosody

Prosody is the study of the intonational and rhythmic aspects of language, and in particular the use of **F0**, **energy**, and **duration** to convey pragmatic, affective, or conversation-interactional meanings.[1] We'll introduce these acoustic quantities in detail in the next section when we turn to acoustic phonetics, but briefly we can think of energy as the acoustic quality that we perceive as loudness, and F0 as the frequency of the sound that is produced, the acoustic quality that we hear as the pitch of an utterance. Prosody can be used to mark **discourse structure**, like the difference between statements and questions, or the way that a conversation is structured. Prosody is used to mark the **saliency** of a particular word or phrase. Prosody is heavily used for paralinguistic functions like conveying affective meanings like happiness, surprise, or anger. And prosody plays an important role in managing turn-taking in conversation.

14.3.1 Prosodic Prominence: Accent, Stress and Schwa

prominence

In a natural utterance of American English, some words sound more **prominent** than others, and certain syllables in these words are also more **prominent** than others. What we mean by prominence is that these words or syllables are perceptually more salient to the listener. Speakers make a word or syllable more salient in English by saying it louder, saying it slower (so it has a longer duration), or by varying F0 during the word, making it higher or more variable.

pitch accent

Accent We represent prominence via a linguistic marker called **pitch accent**. Words or syllables that are prominent are said to **bear** (be associated with) a pitch accent. Thus this utterance might be pronounced by **accenting** the underlined words:

(14.1) I'm a little surprised to hear it characterized as happy.

Lexical Stress The syllables that bear pitch accent are called **accented** syllables. Not every syllable of a word can be accented: pitch accent has to be realized on the syllable that has **lexical stress**. Lexical stress is a property of the word's pronunciation in dictionaries; the syllable that has lexical stress is the one that will be louder or longer if the word is accented. For example, the word *surprised* is stressed on its second syllable, not its first. (Try stressing the other syllable by saying SURprised; hopefully that sounds wrong to you). Thus, if the word *surprised* receives a pitch accent in a sentence, it is the second syllable that will be stronger. The following example shows underlined accented words with the stressed syllable bearing the accent (the louder, longer syllable) in boldface:

lexical stress

(14.2) I'm a little sur**prised** to hear it **char**acterized as **hap**py.

Stress is marked in dictionaries. The CMU dictionary (CMU, 1993), for example, marks vowels with 0 (unstressed) or 1 (stressed) as in entries for *counter*: [K AW1 N T ER0], or *table*: [T EY1 B AH0 L]. Difference in lexical stress can affect word meaning; the noun *content* is pronounced [K AA1 N T EH0 N T], while the adjective is pronounced [K AA0 N T EH1 N T].

[1] The word is used in a different but related way in poetry, to mean the study of verse metrical structure.

Reduced Vowels and Schwa Unstressed vowels can be weakened even further to **reduced vowels**, the most common of which is **schwa** ([ax]), as in the second vowel of *parakeet*: [p ae r ax k iy t]. In a reduced vowel the articulatory gesture isn't as complete as for a full vowel. Not all unstressed vowels are reduced; any vowel, and diphthongs in particular, can retain its full quality even in unstressed position. For example, the vowel [iy] can appear in stressed position as in the word *eat* [iy t] or in unstressed position as in the word *carry* [k ae r iy].

reduced vowel
schwa

prominence

In summary, there is a continuum of prosodic **prominence**, for which it is often useful to represent levels like accented, stressed, full vowel, and reduced vowel.

14.3.2 Prosodic Structure

Spoken sentences have prosodic structure: some words seem to group naturally together, while some words seem to have a noticeable break or disjuncture between them. Prosodic structure is often described in terms of **prosodic phrasing**, meaning that an utterance has a prosodic phrase structure in a similar way to it having a syntactic phrase structure. For example, the sentence *I wanted to go to London, but could only get tickets for France* seems to have two main **intonation phrases**, their boundary occurring at the comma. Furthermore, in the first phrase, there seems to be another set of lesser prosodic phrase boundaries (often called **intermediate phrases**) that split up the words as *I wanted* | *to go* | *to London*. These kinds of intonation phrases are often correlated with syntactic structure constituents (Price et al. 1991, Bennett and Elfner 2019).

prosodic phrasing

intonation phrase

intermediate phrase

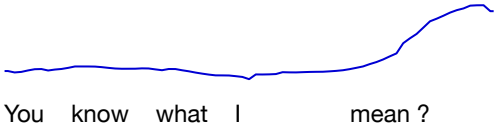
Automatically predicting prosodic boundaries can be important for tasks like TTS. Modern approaches use sequence models that take either raw text or text annotated with features like parse trees as input, and make a break/no-break decision at each word boundary. They can be trained on data labeled for prosodic structure like the Boston University Radio News Corpus (Ostendorf et al., 1995).

14.3.3 Tune

Two utterances with the same prominence and phrasing patterns can still differ prosodically by having different **tunes**. The **tune** of an utterance is the rise and fall of its F0 over time. A very obvious example of tune is the difference between statements and yes-no questions in English. The same words can be said with a final F0 rise to indicate a yes-no question (called a **question rise**):

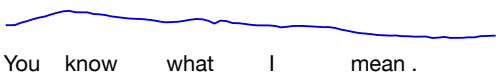
tune

question rise



final fall

or a final drop in F0 (called a **final fall**) to indicate a declarative intonation:



Languages make wide use of tune to express meaning (Xu, 2005). In English, for example, besides this well-known rise for yes-no questions, a phrase containing a list of nouns separated by commas often has a short rise called a **continuation rise** after each noun. Other examples include the characteristic English contours for expressing **contradiction** and expressing **surprise**.

continuation rise

Linking Prominence and Tune

Pitch accents come in different varieties that are related to tune; high pitched accents, for example, have different functions than low pitched accents. There are many typologies of accent classes in different languages. One such typology is part of the **ToBI** (Tone and Break Indices) theory of intonation (Silverman et al. 1992). Each word in ToBI can be associated with one of five types of **pitch accents** shown in Fig. 14.9. Each utterance in ToBI consists of a sequence of intonational phrases, each of which ends in one of four **boundary tones** shown in Fig. 14.9, representing the utterance final aspects of tune. There are version of ToBI for many languages.

ToBI

boundary tone

Pitch Accents		Boundary Tones	
H*	peak accent	**L-L%**	"final fall": "declarative contour" of American English
L*	low accent	**L-H%**	continuation rise
L*+H	scooped accent	**H-H%**	"question rise": cantonical yes-no question contour
L+H*	rising peak accent	**H-L%**	final level plateau
H+!H*	step down		

Figure 14.9 The accent and boundary tones labels from the ToBI transcription system for American English intonation (Beckman and Ayers 1997, Beckman and Hirschberg 1994).

14.4 Acoustic Phonetics and Signals

We begin with a very brief introduction to the acoustic waveform and its digitization and frequency analysis; the interested reader is encouraged to consult the references at the end of the chapter.

14.4.1 Waves

Acoustic analysis is based on the sine and cosine functions. Figure 14.10 shows a plot of a sine wave, in particular the function

$$y = A * sin(2\pi ft) \tag{14.3}$$

where we have set the amplitude A to 1 and the frequency f to 10 cycles per second.

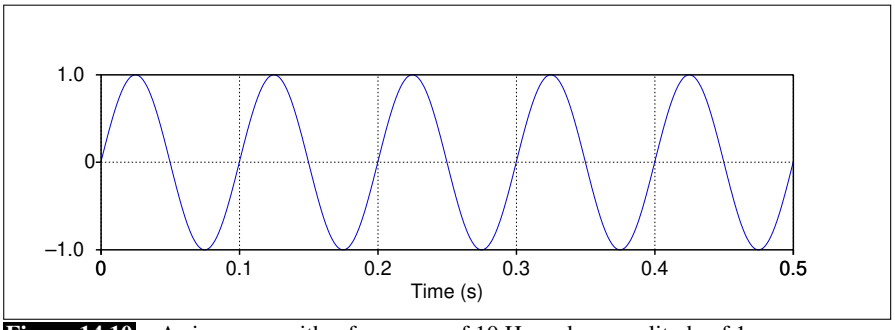


Figure 14.10 A sine wave with a frequency of 10 Hz and an amplitude of 1.

Recall from basic mathematics that two important characteristics of a wave are its **frequency** and **amplitude**. The frequency is the number of times a second that

frequency

amplitude

a wave repeats itself, that is, the number of **cycles**. We usually measure frequency in **cycles per second**. The signal in Fig. 14.10 repeats itself 5 times in .5 seconds, hence 10 cycles per second. Cycles per second are usually called **hertz** (shortened to **Hz**), so the frequency in Fig. 14.10 would be described as 10 Hz. The **amplitude** *A* of a sine wave is the maximum value on the Y axis. The **period** *T* of the wave is the time it takes for one cycle to complete, defined as

Hertz

period

$$T = \frac{1}{f} \tag{14.4}$$

Each cycle in Fig. 14.10 lasts a tenth of a second; hence $T = .1$ seconds.

14.4.2 Speech Sound Waves

Let's turn from hypothetical waves to sound waves. The input to a speech recognizer, like the input to the human ear, is a complex series of changes in air pressure. These changes in air pressure obviously originate with the speaker and are caused by the specific way that air passes through the glottis and out the oral or nasal cavities. We represent sound waves by plotting the change in air pressure over time. One metaphor which sometimes helps in understanding these graphs is that of a vertical plate blocking the air pressure waves (perhaps in a microphone in front of a speaker's mouth, or the eardrum in a hearer's ear). The graph measures the amount of **compression** or **rarefaction** (uncompression) of the air molecules at this plate. Figure 14.11 shows a short segment of a waveform taken from the Switchboard corpus of telephone speech of the vowel [iy] from someone saying "she just had a baby".

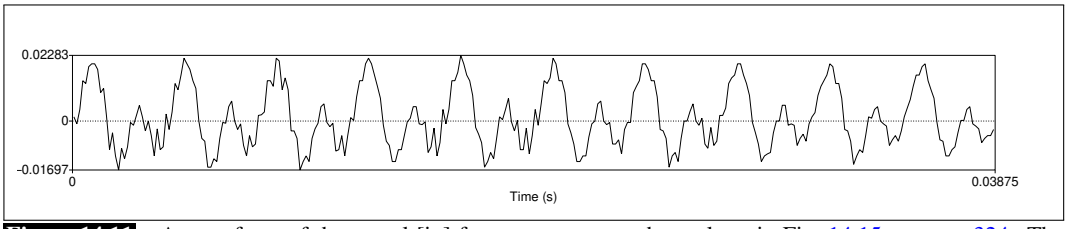


Figure 14.11 A waveform of the vowel [iy] from an utterance shown later in Fig. 14.15 on page 324. The *y*-axis shows the level of air pressure above and below normal atmospheric pressure. The *x*-axis shows time. Notice that the wave repeats regularly.

The first step in digitizing a sound wave like Fig. 14.11 is to convert the analog representations (first air pressure and then analog electric signals in a microphone) into a digital signal. This **analog-to-digital conversion** has two steps: **sampling** and **quantization**. To sample a signal, we measure its amplitude at a particular time; the **sampling rate** is the number of samples taken per second. To accurately measure a wave, we must have at least two samples in each cycle: one measuring the positive part of the wave and one measuring the negative part. More than two samples per cycle increases the amplitude accuracy, but fewer than two samples causes the frequency of the wave to be completely missed. Thus, the maximum frequency wave that can be measured is one whose frequency is half the sample rate (since every cycle needs two samples). This maximum frequency for a given sampling rate is called the **Nyquist frequency**. Most information in human speech is in frequencies below 10,000 Hz; thus, a 20,000 Hz sampling rate would be necessary for complete accuracy. But telephone speech is filtered by the switching network, and only

sampling

Nyquist frequency

frequencies less than 4,000 Hz are transmitted by telephones. Thus, an 8,000 Hz sampling rate is sufficient for **telephone-bandwidth** speech like the Switchboard corpus, while 16,000 Hz sampling is often used for microphone speech.

Even an 8,000 Hz sampling rate requires 8000 amplitude measurements for each second of speech, so it is important to store amplitude measurements efficiently. They are usually stored as integers, either 8 bit (values from -128–127) or 16 bit (values from -32768–32767). This process of representing real-valued numbers as integers is called **quantization** because the difference between two integers acts as a minimum granularity (a quantum size) and all values that are closer together than this quantum size are represented identically.

quantization

Once data is quantized, it is stored in various formats. One parameter of these formats is the sample rate and sample size discussed above; telephone speech is often sampled at 8 kHz and stored as 8-bit samples, and microphone data is often sampled at 16 kHz and stored as 16-bit samples. Another parameter is the number of **channels**. For stereo data or for two-party conversations, we can store both channels in the same file or we can store them in separate files. A final parameter is individual sample storage—linearly or compressed. One common compression format used for telephone speech is μ-law (often written u-law but still pronounced mu-law). The intuition of log compression algorithms like μ-law is that human hearing is more sensitive at small intensities than large ones; the log represents small values with more faithfulness at the expense of more error on large values. The linear (unlogged) values are generally referred to as **linear PCM** values (PCM stands for pulse code modulation, but never mind that). Here's the equation for compressing a linear PCM sample value x to 8-bit μ-law, (where μ=255 for 8 bits):

channel

PCM

$$F(x) = \frac{\text{sgn}(x)\log(1+\mu|x|)}{\log(1+\mu)} \quad -1 \le x \le 1 \tag{14.5}$$

There are a number of standard file formats for storing the resulting digitized wavefile, such as Microsoft's .wav and Apple's AIFF all of which have special headers; simple headerless "raw" files are also used. For example, the .wav format is a subset of Microsoft's RIFF format for multimedia files; RIFF is a general format that can represent a series of nested chunks of data and control information. Figure 14.12 shows a simple .wav file with a single data chunk together with its format chunk.

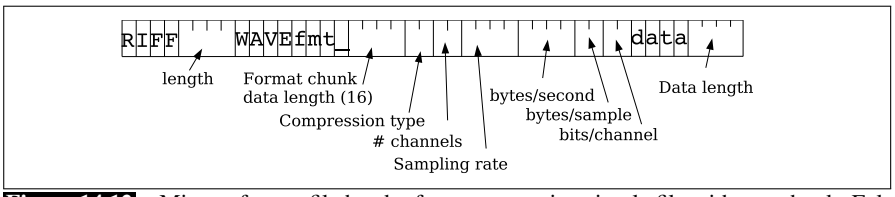


Figure 14.12 Microsoft wavefile header format, assuming simple file with one chunk. Following this 44-byte header would be the data chunk.

14.4.3 Frequency and Amplitude; Pitch and Loudness

Sound waves, like all waves, can be described in terms of frequency, amplitude, and the other characteristics that we introduced earlier for pure sine waves. In sound waves, these are not quite as simple to measure as they were for sine waves. Let's consider frequency. Note in Fig. 14.11 that although not exactly a sine, the wave is nonetheless periodic, repeating 10 times in the 38.75 milliseconds (.03875 seconds)

captured in the figure. Thus, the frequency of this segment of the wave is 10/.03875 or 258 Hz.

Where does this periodic 258 Hz wave come from? It comes from the speed of vibration of the vocal folds; since the waveform in Fig. 14.11 is from the vowel [iy], it is voiced. Recall that voicing is caused by regular openings and closing of the vocal folds. When the vocal folds are open, air is pushing up through the lungs, creating a region of high pressure. When the folds are closed, there is no pressure from the lungs. Thus, when the vocal folds are vibrating, we expect to see regular peaks in amplitude of the kind we see in Fig. 14.11, each major peak corresponding to an opening of the vocal folds. The frequency of the vocal fold vibration, or the frequency of the complex wave, is called the **fundamental frequency** of the waveform, often abbreviated **F0**. We can plot F0 over time in a **pitch track**. Figure 14.13 shows the pitch track of a short question, "Three o'clock?" represented below the waveform. Note the rise in F0 at the end of the question.

fundamental frequency
F0
pitch track

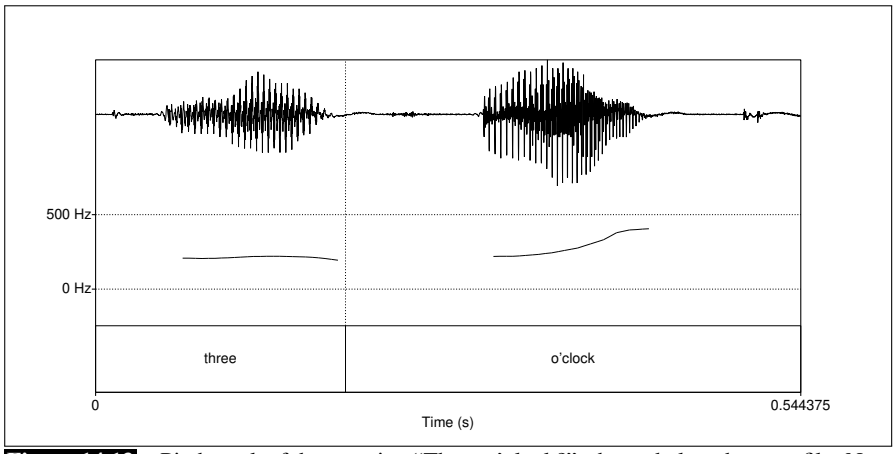


Figure 14.13 Pitch track of the question "Three o'clock?", shown below the wavefile. Note the rise in F0 at the end of the question. Note the lack of pitch trace during the very quiet part (the "o'" of "o'clock"; automatic pitch tracking is based on counting the pulses in the voiced regions, and doesn't work if there is no voicing (or insufficient sound).

The vertical axis in Fig. 14.11 measures the amount of air pressure variation; pressure is force per unit area, measured in Pascals (Pa). A high value on the vertical axis (a high amplitude) indicates that there is more air pressure at that point in time, a zero value means there is normal (atmospheric) air pressure, and a negative value means there is lower than normal air pressure (rarefaction).

In addition to this value of the amplitude at any point in time, we also often need to know the average amplitude over some time range, to give us some idea of how great the average displacement of air pressure is. But we can't just take the average of the amplitude values over a range; the positive and negative values would (mostly) cancel out, leaving us with a number close to zero. Instead, we generally use the RMS (root-mean-square) amplitude, which squares each number before averaging (making it positive), and then takes the square root at the end.

$$\text{RMS amplitude}_{i=1}^{N} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2} \tag{14.6}$$

power

The **power** of the signal is related to the square of the amplitude. If the number

of samples of a sound is N, the power is

$$\text{Power} = \frac{1}{N}\sum_{i=1}^{N} x_i^2 \tag{14.7}$$

intensity

Rather than power, we more often refer to the **intensity** of the sound, which normalizes the power to the human auditory threshold and is measured in dB. If P_0 is the auditory threshold pressure (which is 2×10^{-5} Pa), then intensity is defined as follows:

$$\text{Intensity} = 10\log_{10}\frac{1}{NP_0}\sum_{i=1}^{N} x_i^2 \tag{14.8}$$

Figure 14.14 shows an intensity plot for the sentence "Is it a long movie?" from the CallHome corpus, again shown below the waveform plot.

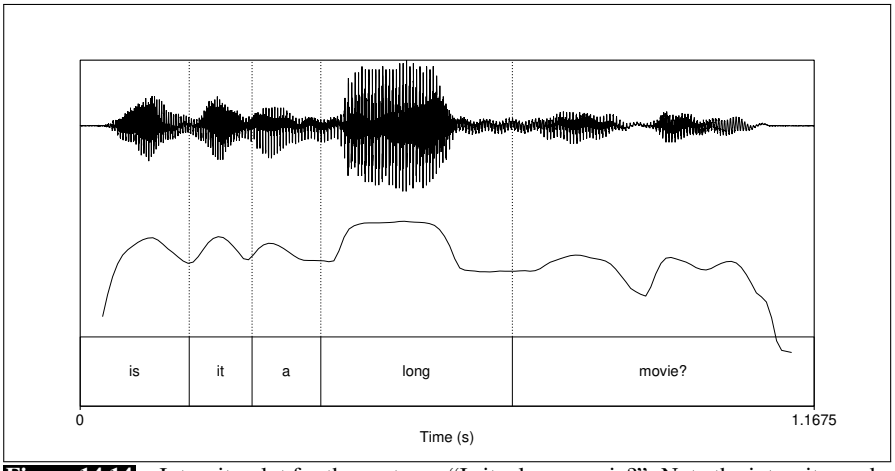


Figure 14.14 Intensity plot for the sentence "Is it a long movie?". Note the intensity peaks at each vowel and the especially high peak for the word *long*.

pitch

Two important perceptual properties, **pitch** and **loudness**, are related to frequency and intensity. The **pitch** of a sound is the mental sensation, or perceptual correlate, of fundamental frequency; in general, if a sound has a higher fundamental frequency we perceive it as having a higher pitch. We say "in general" because the relationship is not linear, since human hearing has different acuities for different frequencies. Roughly speaking, human pitch perception is most accurate between 100 Hz and 1000 Hz and in this range pitch correlates linearly with frequency. Human hearing represents frequencies above 1000 Hz less accurately, and above this range, pitch correlates logarithmically with frequency. Logarithmic representation means that the differences between high frequencies are compressed and hence not as accurately perceived. There are various psychoacoustic models of pitch perception scales. One common model is the **mel** scale (Stevens et al. 1937, Stevens and Volkmann 1940). A mel is a unit of pitch defined such that pairs of sounds which are perceptually equidistant in pitch are separated by an equal number of mels. The mel frequency m can be computed from the raw acoustic frequency as follows:

Mel

$$m = 1127\ln(1 + \frac{f}{700}) \tag{14.9}$$

As we'll see in Chapter 15, the mel scale plays an important role in speech recognition.

The **loudness** of a sound is the perceptual correlate of the **power**. So sounds with higher amplitudes are perceived as louder, but again the relationship is not linear. First of all, as we mentioned above when we defined μ-law compression, humans have greater resolution in the low-power range; the ear is more sensitive to small power differences. Second, it turns out that there is a complex relationship between power, frequency, and perceived loudness; sounds in certain frequency ranges are perceived as being louder than those in other frequency ranges.

pitch extraction

Various algorithms exist for automatically extracting F0. In a slight abuse of terminology, these are called **pitch extraction** algorithms. The autocorrelation method of pitch extraction, for example, correlates the signal with itself at various offsets. The offset that gives the highest correlation gives the period of the signal. There are various publicly available pitch extraction toolkits; for example, an augmented autocorrelation pitch tracker is provided with Praat (Boersma and Weenink, 2005).

14.4.4 Interpretation of Phones from a Waveform

Much can be learned from a visual inspection of a waveform. For example, vowels are pretty easy to spot. Recall that vowels are voiced; another property of vowels is that they tend to be long and are relatively loud (as we can see in the intensity plot in Fig. 14.14). Length in time manifests itself directly on the x-axis, and loudness is related to (the square of) amplitude on the y-axis. We saw in the previous section that voicing is realized by regular peaks in amplitude of the kind we saw in Fig. 14.11, each major peak corresponding to an opening of the vocal folds. Figure 14.15 shows the waveform of the short sentence "she just had a baby". We have labeled this waveform with word and phone labels. Notice that each of the six vowels in Fig. 14.15, [iy], [ax], [ae], [ax], [ey], [iy], all have regular amplitude peaks indicating voicing.

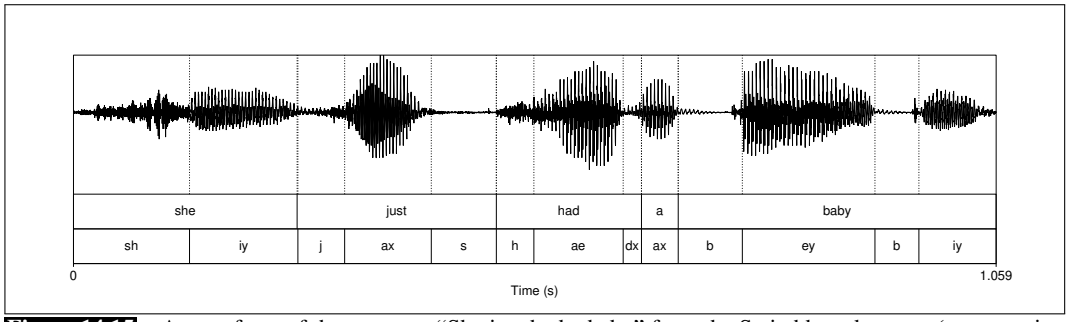


Figure 14.15 A waveform of the sentence "She just had a baby" from the Switchboard corpus (conversation 4325). The speaker is female, was 20 years old in 1991, which is approximately when the recording was made, and speaks the South Midlands dialect of American English.

For a stop consonant, which consists of a closure followed by a release, we can often see a period of silence or near silence followed by a slight burst of amplitude. We can see this for both of the [b]'s in *baby* in Fig. 14.15.

Another phone that is often quite recognizable in a waveform is a fricative. Recall that fricatives, especially very strident fricatives like [sh], are made when a narrow channel for airflow causes noisy, turbulent air. The resulting hissy sounds have a noisy, irregular waveform. This can be seen somewhat in Fig. 14.15; it's even clearer in Fig. 14.16, where we've magnified just the first word *she*.

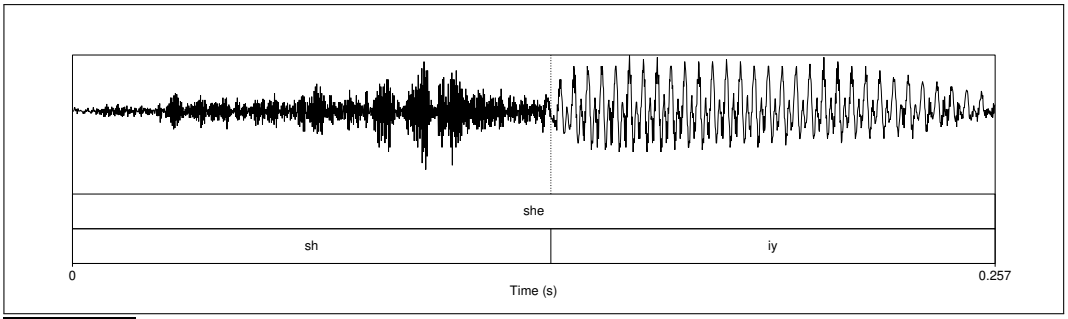


Figure 14.16 A more detailed view of the first word "she" extracted from the wavefile in Fig. 14.15. Notice the difference between the random noise of the fricative [sh] and the regular voicing of the vowel [iy].

14.4.5 Spectra and the Frequency Domain

While some broad phonetic features (such as energy, pitch, and the presence of voicing, stop closures, or fricatives) can be interpreted directly from the waveform, most computational applications such as speech recognition (as well as human auditory processing) are based on a different representation of the sound in terms of its component frequencies. The insight of **Fourier analysis** is that every complex wave can be represented as a sum of many sine waves of different frequencies. Consider the waveform in Fig. 14.17. This waveform was created (in Praat) by summing two sine waveforms, one of frequency 10 Hz and one of frequency 100 Hz.

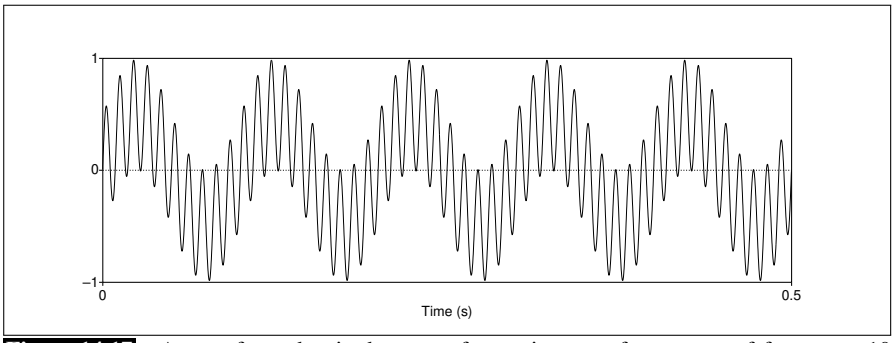


Figure 14.17 A waveform that is the sum of two sine waveforms, one of frequency 10 Hz (note five repetitions in the half-second window) and one of frequency 100 Hz, both of amplitude 1.

spectrum

We can represent these two component frequencies with a **spectrum**. The spectrum of a signal is a representation of each of its frequency components and their amplitudes. Figure 14.18 shows the spectrum of Fig. 14.17. Frequency in Hz is on the x-axis and amplitude on the y-axis. Note the two spikes in the figure, one at 10 Hz and one at 100 Hz. Thus, the spectrum is an alternative representation of the original waveform, and we use the spectrum as a tool to study the component frequencies of a sound wave at a particular time point.

Let's look now at the frequency components of a speech waveform. Figure 14.19 shows part of the waveform for the vowel [ae] of the word *had*, cut out from the sentence shown in Fig. 14.15.

Note that there is a complex wave that repeats about ten times in the figure; but there is also a smaller repeated wave that repeats four times for every larger pattern (notice the four small peaks inside each repeated wave). The complex wave has a frequency of about 234 Hz (we can figure this out since it repeats roughly 10 times

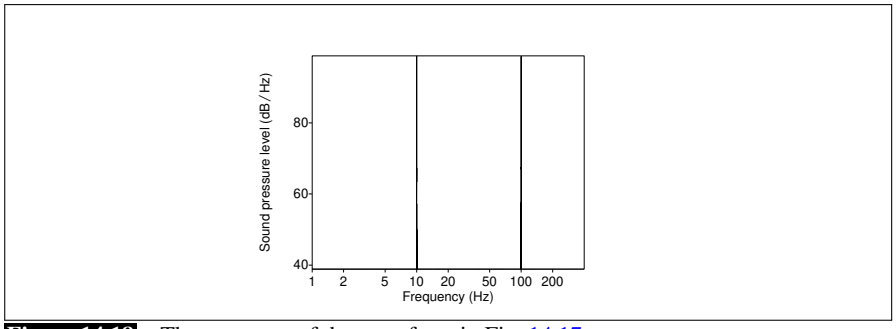


Figure 14.18 The spectrum of the waveform in Fig. 14.17.

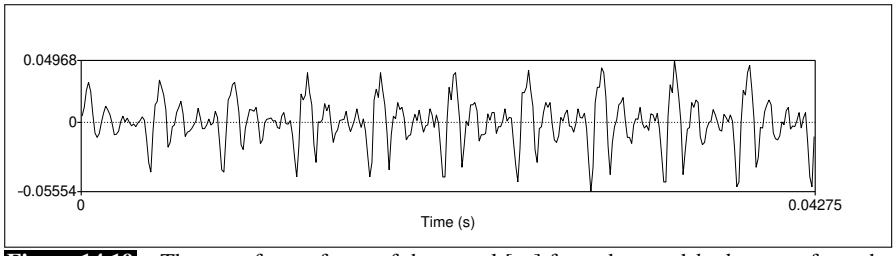


Figure 14.19 The waveform of part of the vowel [ae] from the word *had* cut out from the waveform shown in Fig. 14.15.

in .0427 seconds, and 10 cycles/.0427 seconds = 234 Hz).

The smaller wave then should have a frequency of roughly four times the frequency of the larger wave, or roughly 936 Hz. Then, if you look carefully, you can see two little waves on the peak of many of the 936 Hz waves. The frequency of this tiniest wave must be roughly twice that of the 936 Hz wave, hence 1872 Hz.

Figure 14.20 shows a smoothed spectrum for the waveform in Fig. 14.19, computed with a discrete Fourier transform (DFT).

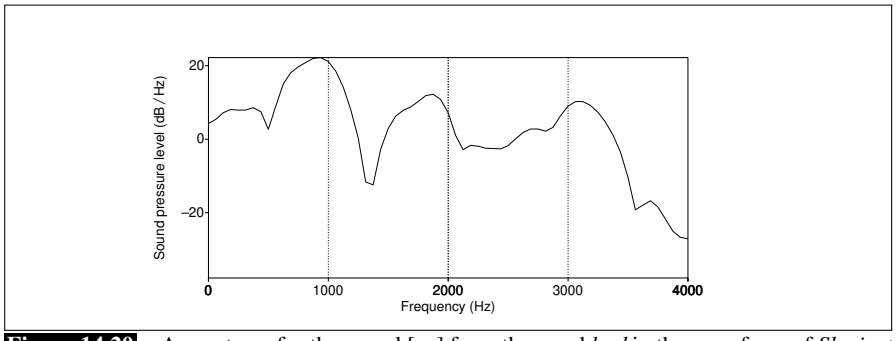


Figure 14.20 A spectrum for the vowel [ae] from the word *had* in the waveform of *She just had a baby* in Fig. 14.15.

The *x*-axis of a spectrum shows frequency, and the *y*-axis shows some measure of the magnitude of each frequency component (in decibels (dB), a logarithmic measure of amplitude that we saw earlier). Thus, Fig. 14.20 shows significant frequency components at around 930 Hz, 1860 Hz, and 3020 Hz, along with many other lower-magnitude frequency components. These first two components are just what we noticed in the time domain by looking at the wave in Fig. 14.19!

Why is a spectrum useful? It turns out that these spectral peaks that are easily visible in a spectrum are characteristic of different phones; phones have characteris-

tic spectral "signatures". Just as chemical elements give off different wavelengths of light when they burn, allowing us to detect elements in stars by looking at the spectrum of the light, we can detect the characteristic signature of the different phones by looking at the spectrum of a waveform. This use of spectral information is essential to both human and machine speech recognition. In human audition, the function of the **cochlea**, or **inner ear**, is to compute a spectrum of the incoming waveform. Similarly, the acoustic features used in speech recognition are spectral representations.

cochlea

Let's look at the spectrum of different vowels. Since some vowels change over time, we'll use a different kind of plot called a **spectrogram**. While a spectrum shows the frequency components of a wave at one point in time, a **spectrogram** is a way of envisioning how the different frequencies that make up a waveform change over time. The *x*-axis shows time, as it did for the waveform, but the *y*-axis now shows frequencies in hertz. The darkness of a point on a spectrogram corresponds to the amplitude of the frequency component. Very dark points have high amplitude, light points have low amplitude. Thus, the spectrogram is a useful way of visualizing the three dimensions (time x frequency x amplitude).

spectrogram

Figure 14.21 shows spectrograms of three American English vowels, [ih], [ae], and [uh]. Note that each vowel has a set of dark bars at various frequency bands, slightly different bands for each vowel. Each of these represents the same kind of spectral peak that we saw in Fig. 14.19.

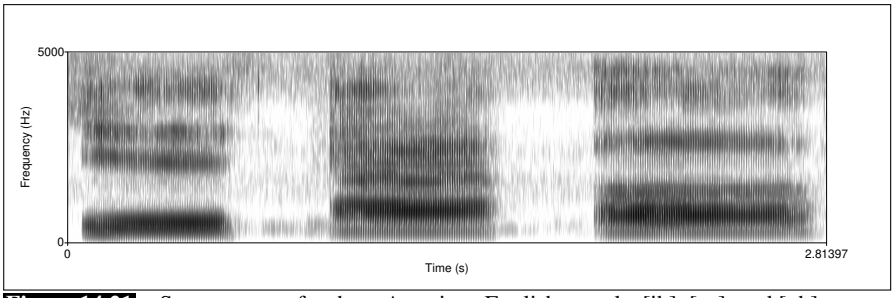


Figure 14.21 Spectrograms for three American English vowels, [ih], [ae], and [uh]

formant

Each dark bar (or spectral peak) is called a **formant**. As we discuss below, a formant is a frequency band that is particularly amplified by the vocal tract. Since different vowels are produced with the vocal tract in different positions, they will produce different kinds of amplifications or resonances. Let's look at the first two formants, called F1 and F2. Note that F1, the dark bar closest to the bottom, is in a different position for the three vowels; it's low for [ih] (centered at about 470 Hz) and somewhat higher for [ae] and [uh] (somewhere around 800 Hz). By contrast, F2, the second dark bar from the bottom, is highest for [ih], in the middle for [ae], and lowest for [uh].

We can see the same formants in running speech, although the reduction and coarticulation processes make them somewhat harder to see. Figure 14.22 shows the spectrogram of "she just had a baby", whose waveform was shown in Fig. 14.15. F1 and F2 (and also F3) are pretty clear for the [ax] of *just*, the [ae] of *had*, and the [ey] of *baby*.

What specific clues can spectral representations give for phone identification? First, since different vowels have their formants at characteristic places, the spectrum can distinguish vowels from each other. We've seen that [ae] in the sample waveform had formants at 930 Hz, 1860 Hz, and 3020 Hz. Consider the vowel [iy] at the

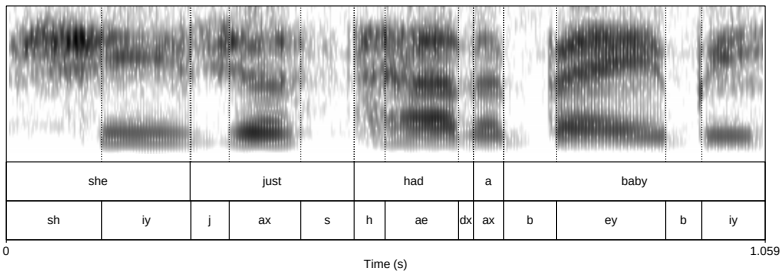


Figure 14.22 A spectrogram of the sentence "she just had a baby" whose waveform was shown in Fig. 14.15. We can think of a spectrogram as a collection of spectra (time slices), like Fig. 14.20 placed end to end.

beginning of the utterance in Fig. 14.15. The spectrum for this vowel is shown in Fig. 14.23. The first formant of [iy] is 540 Hz, much lower than the first formant for [ae], and the second formant (2581 Hz) is much higher than the second formant for [ae]. If you look carefully, you can see these formants as dark bars in Fig. 14.22 just around 0.5 seconds.

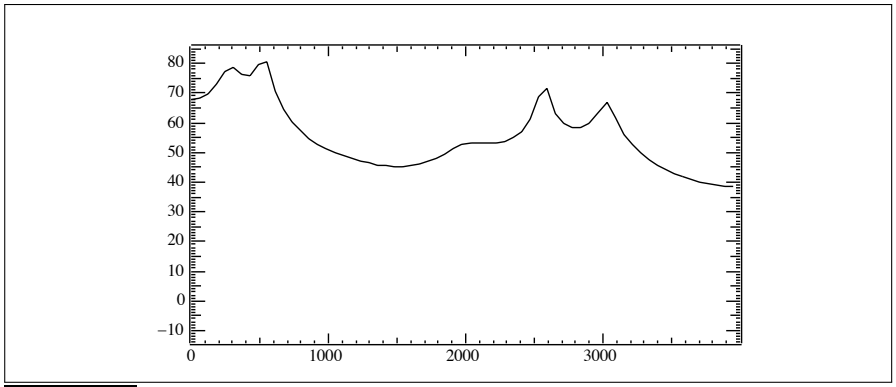


Figure 14.23 A smoothed (LPC) spectrum for the vowel [iy] at the start of *She just had a baby*. Note that the first formant (540 Hz) is much lower than the first formant for [ae] shown in Fig. 14.20, and the second formant (2581 Hz) is much higher than the second formant for [ae].

The location of the first two formants (called F1 and F2) plays a large role in determining vowel identity, although the formants still differ from speaker to speaker. Higher formants tend to be caused more by general characteristics of a speaker's vocal tract rather than by individual vowels. Formants also can be used to identify the nasal phones [n], [m], and [ng] and the liquids [l] and [r].

14.4.6 The Source-Filter Model

Why do different vowels have different spectral signatures? As we briefly mentioned above, the formants are caused by the resonant cavities of the mouth. The **source-filter model** is a way of explaining the acoustics of a sound by modeling how the pulses produced by the glottis (the **source**) are shaped by the vocal tract (the **filter**).

source-filter model

Let's see how this works. Whenever we have a wave such as the vibration in air caused by the glottal pulse, the wave also has **harmonics**. A harmonic is another wave whose frequency is a multiple of the fundamental wave. Thus, for example, a

harmonic

115 Hz glottal fold vibration leads to harmonics (other waves) of 230 Hz, 345 Hz, 460 Hz, and so on on. In general, each of these waves will be weaker, that is, will have much less amplitude than the wave at the fundamental frequency.

It turns out, however, that the vocal tract acts as a kind of filter or amplifier; indeed any cavity, such as a tube, causes waves of certain frequencies to be amplified and others to be damped. This amplification process is caused by the shape of the cavity; a given shape will cause sounds of a certain frequency to resonate and hence be amplified. Thus, by changing the shape of the cavity, we can cause different frequencies to be amplified.

When we produce particular vowels, we are essentially changing the shape of the vocal tract cavity by placing the tongue and the other articulators in particular positions. The result is that different vowels cause different harmonics to be amplified. So a wave of the same fundamental frequency passed through different vocal tract positions will result in different harmonics being amplified.

We can see the result of this amplification by looking at the relationship between the shape of the vocal tract and the corresponding spectrum. Figure 14.24 shows the vocal tract position for three vowels and a typical resulting spectrum. The formants are places in the spectrum where the vocal tract happens to amplify particular harmonic frequencies.

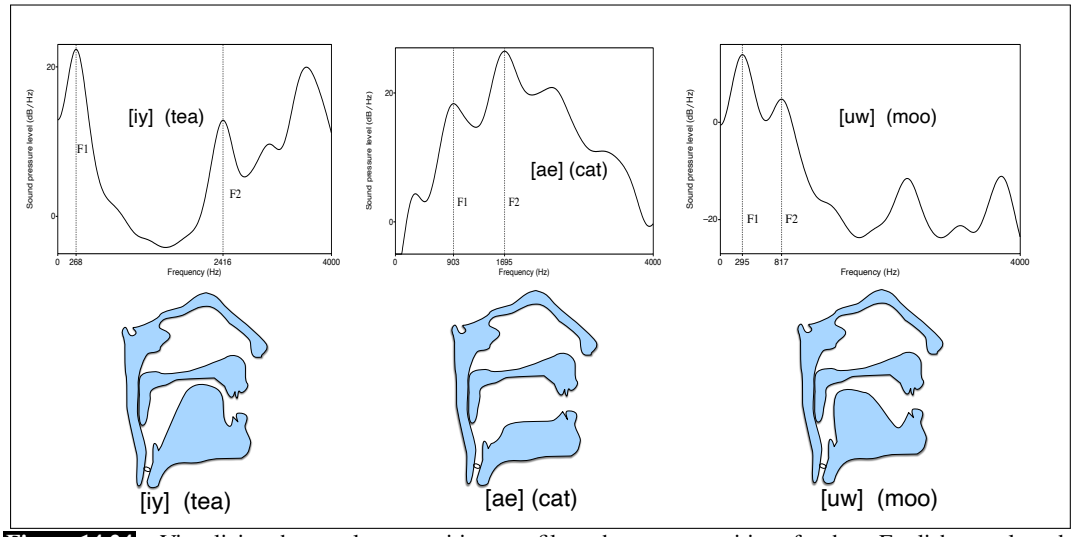


Figure 14.24 Visualizing the vocal tract position as a filter: the tongue positions for three English vowels and the resulting smoothed spectra showing F1 and F2.

14.5 Feature Extraction for Speech Recognition: Log Mel Spectrum

The same tools that we used to analyze the acoustic phonetics of the waveforms are also often used as inputs to speech processing algorithms. In this section we introduce a signal processing pipeline that is often used as part of tasks like automatic speech recognition (ASR), as we will see in Chapter 15. The first step in speech processing is often to transform the input waveform into a sequence of acoustic **fea-**

feature vector

ture vectors, each vector representing the information in a small time window of the signal. Sometimes speech recognition or processing algorithms will start with the waveform, in which case that processing is done by the convolutional networks (convnets) that we will introduce in Chapter 15.

Other systems begin instead at a higher level, with the log mel spectrum. So in this section we introduce this commonly used feature vector: sequences of **log mel spectrum** vectors. In the following section we'll introduce an alternative vector, the **MFCC** representation. We'll introduce these concepts at a relatively high level; a speech signal processing course is recommended for more details.

We begin by repeating from Section 14.4.2 the process of digitizing and quantizing an analog speech waveform.

14.5.1 Sampling and Quantization

The input to a speech recognizer is a complex series of changes in air pressure. These changes in air pressure obviously originate with the speaker and are caused by the specific way that air passes through the glottis and out the oral or nasal cavities. We represent sound waves by plotting the change in air pressure over time. One metaphor which sometimes helps in understanding these graphs is that of a vertical plate blocking the air pressure waves (perhaps in a microphone in front of a speaker's mouth, or the eardrum in a hearer's ear). The graph measures the amount of **compression** or **rarefaction** (uncompression) of the air molecules at this plate. Figure 14.25 (repeated from Fig. 14.11) shows a short segment of a waveform taken from the Switchboard corpus of telephone speech of the vowel [iy] from someone saying "she just had a baby".

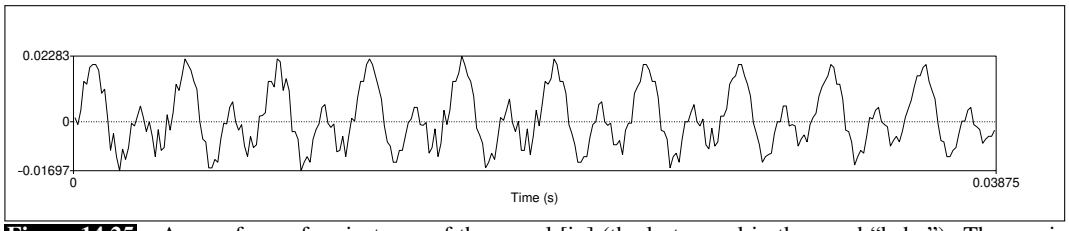


Figure 14.25 A waveform of an instance of the vowel [iy] (the last vowel in the word "baby"). The *y*-axis shows the level of air pressure above and below normal atmospheric pressure. The *x*-axis shows time. Notice that the wave repeats regularly. Repeated from Fig. 14.11.

sampling

The first step in digitizing a sound wave like Fig. 14.11 is to convert the analog representations (first air pressure and then analog electric signals in a microphone) into a digital signal. This **analog-to-digital conversion** has two steps: **sampling** and **quantization**. To sample a signal, we measure its amplitude at a particular time; the **sampling rate** is the number of samples taken per second. To accurately measure a wave, we must have at least two samples in each cycle: one measuring the positive part of the wave and one measuring the negative part. More than two samples per cycle increases the amplitude accuracy, but fewer than two samples causes the frequency of the wave to be completely missed. Thus, the maximum frequency wave that can be measured is one whose frequency is half the sample rate (since every cycle needs two samples). This maximum frequency for a given sampling rate is called the **Nyquist frequency**. Most information in human speech is in frequencies below 10,000 Hz; thus, a 20,000 Hz sampling rate would be necessary for complete accuracy. But telephone speech is filtered by the switching network, and only frequencies less than 4,000 Hz are transmitted by telephones. Thus, an 8,000 Hz

Nyquist frequency

sampling rate is sufficient for **telephone-bandwidth** speech like the Switchboard corpus, while 16,000 Hz sampling is often used for microphone speech.

Although using higher sampling rates produces higher ASR accuracy, we can't combine different sampling rates for training and testing ASR systems. Thus if we are testing on a telephone corpus like Switchboard (8 KHz sampling), we must downsample our training corpus to 8 KHz. Similarly, if we are training on multiple corpora and one of them includes telephone speech, we downsample all the wideband corpora to 8KHz.

Amplitude measurements are stored as integers, either 8 bit (values from -128–127) or 16 bit (values from -32768–32767). This process of representing real-valued numbers as integers is called **quantization**; all values that are closer together than the minimum granularity (the quantum size) are represented identically. We refer to each sample at time index n in the digitized, quantized waveform as $x[n]$.

quantization

Once data is quantized, it is stored in various formats. One parameter of these formats is the sample rate and sample size discussed above; telephone speech is often sampled at 8 kHz and stored as 8-bit samples, and microphone data is often sampled at 16 kHz and stored as 16-bit samples. Another parameter is the number of **channels**. For stereo data or for two-party conversations, we can store both channels in the same file or we can store them in separate files. A final parameter is individual sample storage—linearly or compressed. One common compression format used for telephone speech is μ-law (often written u-law but still pronounced mu-law). The intuition of log compression algorithms like μ-law is that human hearing is more sensitive at small intensities than large ones; the log represents small values with more faithfulness at the expense of more error on large values. The linear (unlogged) values are generally referred to as **linear PCM** values (PCM stands for pulse code modulation, but never mind that). Here's the equation for compressing a linear PCM sample value x to 8-bit μ-law, (where μ=255 for 8 bits):

channel

PCM

$$F(x) = \frac{\text{sgn}(x)\log(1+\mu|x|)}{\log(1+\mu)} \quad -1 \le x \le 1 \tag{14.10}$$

14.5.2 Windowing

From the digitized, quantized representation of the waveform, we need to extract spectral features from a small **window** of speech that characterizes part of a particular phoneme. Inside this small window, we can roughly think of the signal as **stationary** (that is, its statistical properties are constant within this region). (By contrast, in general, speech is a **non-stationary** signal, meaning that its statistical properties are not constant over time). We extract this roughly stationary portion of speech by using a window which is non-zero inside a region and zero elsewhere, running this window across the speech signal and multiplying it by the input waveform to produce a windowed waveform.

stationary

non-stationary

The speech extracted from each window is called a **frame**. The windowing is characterized by three parameters: the **window size** or **frame size** of the window (its width in milliseconds), the **frame stride**, (also called **shift** or **offset**) between successive windows, and the **shape** of the window.

frame

stride

To extract the signal we multiply the value of the signal at time n, $s[n]$ by the value of the windowing function at time n, $w[n]$:

$$y[n] = w[n]s[n] \tag{14.11}$$

The window shape sketched in Fig. 14.26 is **rectangular**; you can see the ex-

rectangular

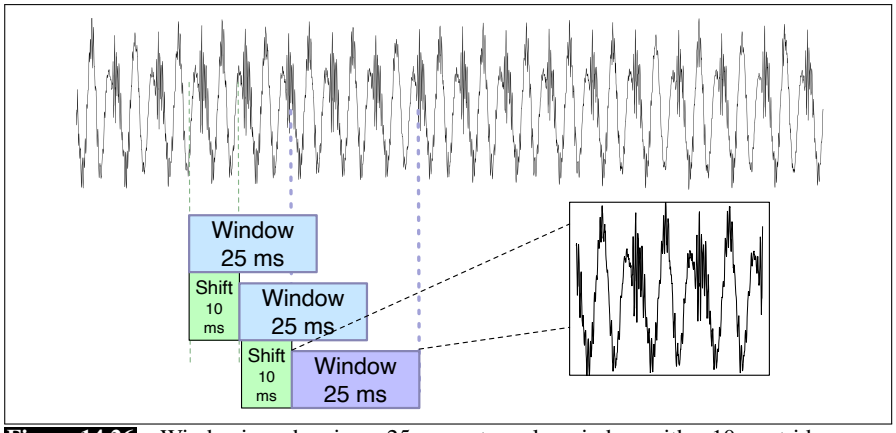


Figure 14.26 Windowing, showing a 25 ms rectangular window with a 10ms stride.

tracted windowed signal looks just like the original signal. The rectangular window, however, abruptly cuts off the signal at its boundaries, which creates problems when we do Fourier analysis. For this reason, for acoustic feature creation we more commonly use the **Hamming** window, which shrinks the values of the signal toward zero at the window boundaries, avoiding discontinuities. Figure 14.27 shows both; the equations are as follows (assuming a window that is L frames long):

Hamming

$$rectangular \quad w[n] = \begin{cases} 1 & 0 \le n \le L-1 \\ 0 & \text{otherwise} \end{cases} \tag{14.12}$$

$$Hamming \quad w[n] = \begin{cases} 0.54 - 0.46\cos(\frac{2\pi n}{L}) & 0 \le n \le L-1 \\ 0 & \text{otherwise} \end{cases} \tag{14.13}$$

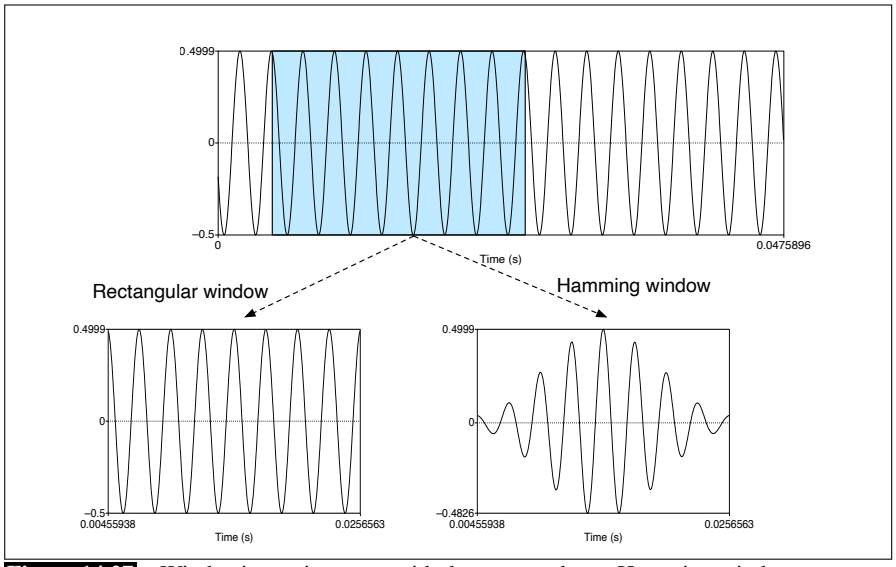


Figure 14.27 Windowing a sine wave with the rectangular or Hamming windows.

14.5.3 Discrete Fourier Transform

Discrete Fourier transform DFT

The next step is to extract spectral information for our windowed signal; we need to know how much energy the signal contains at different frequency bands. The tool for extracting spectral information for discrete frequency bands for a discrete-time (sampled) signal is the **discrete Fourier transform** or **DFT**.

The input to the DFT is a windowed signal $x[n]...x[m]$, and the output, for each of N discrete frequency bands, is a complex number $X[k]$ representing the magnitude and phase of that frequency component in the original signal. If we plot the magnitude against the frequency, we can visualize the **spectrum** (see Chapter 14 for more on spectra). For example, Fig. 14.28 shows a 25 ms Hamming-windowed portion of a signal and its spectrum as computed by a DFT (with some additional smoothing).

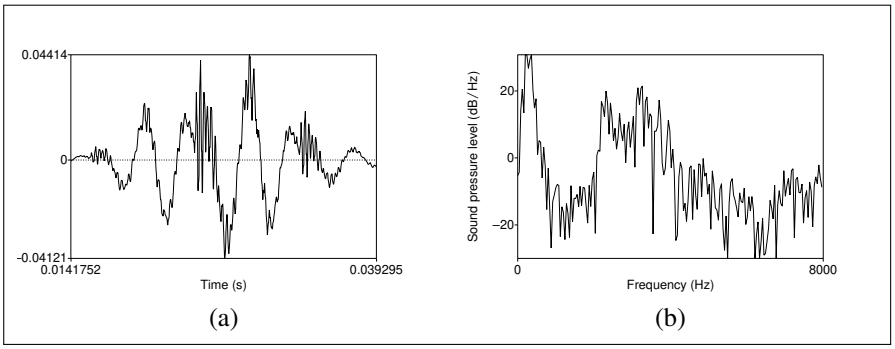


Figure 14.28 (a) A 25 ms Hamming-windowed portion of a signal from the vowel [iy] and (b) its spectrum computed by a DFT.

Euler's formula

We do not introduce the mathematical details of the DFT here, except to note that Fourier analysis relies on **Euler's formula**, with j as the imaginary unit:

$$e^{j\theta} = \cos\theta + j\sin\theta \tag{14.14}$$

As a brief reminder for those students who have already studied signal processing, the DFT is defined as follows:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi}{N}kn} \tag{14.15}$$

fast Fourier transform FFT

A commonly used algorithm for computing the DFT is the **fast Fourier transform** or **FFT**. This implementation of the DFT is very efficient but only works for values of N that are powers of 2.

14.5.4 Mel Filter Bank and Log

The results of the FFT tell us the energy at each frequency band. Human hearing, however, is not equally sensitive at all frequency bands; it is less sensitive at higher frequencies. This bias toward low frequencies helps human recognition, since information in low frequencies (like formants) is crucial for distinguishing vowels or nasals, while information in high frequencies (like stop bursts or fricative noise) is less crucial for successful recognition. Modeling this human perceptual property improves speech recognition performance in the same way.

mel

We implement this intuition by collecting energies, not equally at each frequency band, but according to the **mel** scale, an auditory frequency scale. A **mel** (Stevens

et al. 1937, Stevens and Volkmann 1940) is a unit of pitch. Pairs of sounds that are perceptually equidistant in pitch are separated by an equal number of mels. The mel frequency m can be computed from the raw acoustic frequency by a log transformation:

$$mel(f) = 1127 \ln(1 + \frac{f}{700}) \tag{14.16}$$

We implement this intuition by creating a bank of filters that collect energy from each frequency band, spread logarithmically so that we have very fine resolution at low frequencies, and less resolution at high frequencies. Figure 14.29 shows a sample bank of triangular filters that implement this idea, that can be multiplied by the spectrum to get a mel spectrum.

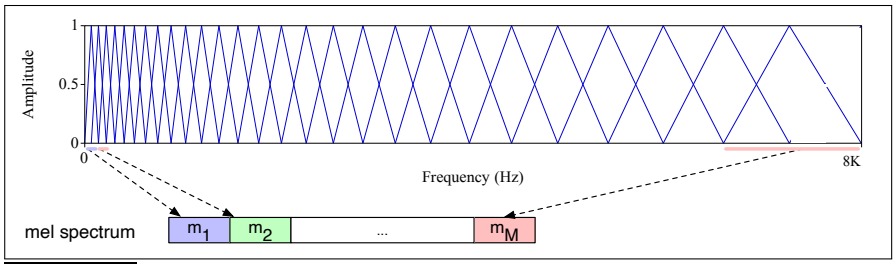


Figure 14.29 The mel filter bank (Davis and Mermelstein, 1980). Each triangular filter, spaced logarithmically along the mel scale, collects energy from a given frequency range.

Finally, we take the log of each of the mel spectrum values. The human response to signal level is logarithmic (like the human response to frequency). Humans are less sensitive to slight differences in amplitude at high amplitudes than at low amplitudes. In addition, using a log makes the feature estimates less sensitive to variations in input such as power variations due to the speaker's mouth moving closer or further from the microphone.

channel

We call each scalar output from a particular filter a **channel**, and so the output for each input frame from the filterbank is a vector of, say 80 or 128 channels, each of which represents the log energy of a particular (mel-spaced) frequency band.

Before we send this log mel channel vector to the downstream neural network layers, it's common for speech systems to rescale them so they have comparable ranges. A common type of normalization for speech is to scale the input to be between -1 and 1 with **zero mean** across the entire pretraining dataset (see Section 4.3.2 in Chapter 4).

14.6 MFCC: Mel Frequency Cepstral Coefficients

MFCC

The **MFCC**, **mel frequency cepstral coefficients**, is a useful representation of the waveform that emphasizes aspects of the signal that are relevant for detection of phonetic units. The MFCC is a 39-dimensional feature vector consisting of:

12 cepstral coefficients	1 energy coefficient
12 delta cepstral coefficients	1 delta energy coefficient
12 double delta cepstral coefficients	1 double delta energy coefficient

Below we sketch how these features are computed; students interested in more detail are encouraged to follow up with a signal processing course.

The Cepstrum: Inverse Discrete Fourier Transform

cepstrum

MFCC coefficients are based on the **cepstrum**. One way to think about the cepstrum is as a useful way of separating the **source** and **filter**. Recall from Section 14.4.6 that the speech waveform is created when a glottal source waveform of a particular fundamental frequency is passed through the vocal tract, which because of its shape has a particular filtering characteristic. But many characteristics of the glottal **source** (its fundamental frequency, the details of the glottal pulse, etc.) are not important for distinguishing different phones. Instead, the most useful information for phone detection is the **filter**, that is, the exact position of the vocal tract. If we knew the shape of the vocal tract, we would know which phone was being produced. This suggests that useful features for phone detection would find a way to deconvolve (separate) the source and filter and show us only the vocal tract filter. It turns out that the cepstrum is one way to do this.

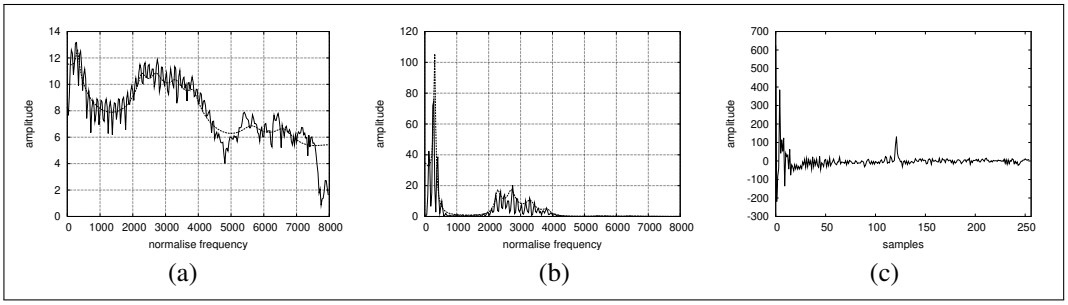


Figure 14.30 The magnitude spectrum (a), log magnitude spectrum (b), and cepstrum (c), from Taylor (2009), by permission. The two spectra have a smoothed spectral envelope laid on top to help visualize the spectrum.

For simplicity, let's consider as input the log magnitude spectrum and ignore the mel scaling. The cepstrum can be thought of as the *spectrum of the log of the spectrum*. This may sound confusing. But let's begin with the easy part: the *log of the spectrum*. That is, the cepstrum begins with a standard magnitude spectrum, such as the one for a vowel shown in Fig. 14.30(a) from Taylor (2009). We then take the log, that is, replace each amplitude value in the magnitude spectrum with its log, as shown in Fig. 14.30(b).

The next step is to visualize the log spectrum *as if itself were a waveform*. In other words, consider the log spectrum in Fig. 14.30(b). Let's imagine removing the axis labels that tell us that this is a spectrum (frequency on the x-axis) and imagine that we are dealing with just a normal speech signal with time on the x-axis. What can we now say about the spectrum of this "pseudo-signal"? Notice that there is a high frequency repetitive component in this wave: small waves that repeat about 8 times in each 1000 along the x-axis, for a frequency of about 120 Hz. This high frequency component is caused by the fundamental frequency of the signal and represents the little peaks in the spectrum at each harmonic of the signal. In addition, there are some lower frequency components in this "pseudo-signal"; for example, the envelope or formant structure has about four large peaks in the window, for a much lower frequency.

Figure 14.30(c) shows the **cepstrum**: the spectrum that we have been describing of the log spectrum. This cepstrum (the word **cepstrum** is formed by reversing the first four letters of **spectrum**) is shown with **samples** along the x-axis. This is because by taking the spectrum of the log spectrum, we have left the frequency domain of the spectrum, and gone back to the time domain. It turns out that the

correct unit of a cepstrum is the sample.

Examining this cepstrum, we see that there is indeed a large peak around 120, corresponding to the F0 and representing the glottal pulse. There are other various components at lower values on the x-axis. These represent the vocal tract filter (the position of the tongue and the other articulators). Thus, if we are interested in detecting phones, we can make use of just the lower cepstral values. If we are interested in detecting pitch, we can use the higher cepstral values.

For the purposes of MFCC extraction, we generally just take the first 12 cepstral values. These 12 coefficients will represent information solely about the vocal tract filter, cleanly separated from information about the glottal source.

It turns out that cepstral coefficients have the extremely useful property that the variance of the different coefficients tends to be uncorrelated. This is not true for the spectrum, where spectral coefficients at different frequency bands are correlated.

For those who have had signal processing, the cepstrum is more formally defined as the **inverse DFT of the log magnitude of the DFT of a signal**; hence, for a windowed frame of speech $x[n]$,

$$c[n] = \sum_{n=0}^{N-1} log\left(\left|\sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi}{N}kn}\right|\right) e^{j\frac{2\pi}{N}kn} \tag{14.17}$$

Energy To the 12 cepstral coefficients from the prior section we add a 13th feature: the energy from the frame. Energy is a useful cue for phone detection (for example vowels and sibilants have more energy than stops). The **energy** in a frame is the sum over time of the power of the samples in the frame; thus, for a signal x in a window from time sample t_1 to time sample t_2, the energy is

energy

$$Energy = \sum_{t=t_1}^{t_2} x^2[t] \tag{14.18}$$

Delta features We also add features related to the change in cepstral features over time. Changes in the speech signal, like the slope of a formant at its transitions, or the change from a stop closure to stop burst, can again provide a useful cue for phone identity. To each of the 13 features (12 cepstral features plus energy) a **delta** or **velocity** feature and a **double delta** or **acceleration** feature. Each of the 13 delta features represents the change between frames in the corresponding cepstral/energy feature, and each of the 13 double delta features represents the change between frames in the corresponding delta features. These deltas can be simply computed by just subtracting the value at a frame from the prior value, but in practice it's common to fit a polynomial and take its first and second derivative.

delta feature

double delta

14.7 Summary

This chapter has introduced many of the important concepts of phonetics and computational phonetics.

- We can represent the pronunciation of words in terms of units called **phones**. The standard system for representing phones is the **International Phonetic Alphabet** or **IPA**. The most common computational system for transcription of English is the **ARPAbet**, which conveniently uses ASCII symbols.

- Phones can be described by how they are produced **articulatorily** by the vocal organs; consonants are defined in terms of their **place** and **manner** of articulation and **voicing**; vowels by their **height**, **backness**, and **roundness**.
- Speech sounds can also be described **acoustically**. Sound waves can be described in terms of **frequency**, **amplitude**, or their perceptual correlates, **pitch** and **loudness**.
- The **spectrum** of a sound describes its different frequency components. While some phonetic properties are recognizable from the waveform, both humans and machines rely on spectral analysis for phone detection.
- A **spectrogram** is a plot of a spectrum over time. Vowels are described by characteristic harmonics called **formants**.

Historical Notes

The major insights of articulatory phonetics date to the linguists of 800–150 B.C. India. They invented the concepts of place and manner of articulation, worked out the glottal mechanism of voicing, and understood the concept of assimilation. European science did not catch up with the Indian phoneticians until over 2000 years later, in the late 19th century. The Greeks did have some rudimentary phonetic knowledge; by the time of Plato's *Theaetetus* and *Cratylus*, for example, they distinguished vowels from consonants, and stop consonants from continuants. The Stoics developed the idea of the syllable and were aware of phonotactic constraints on possible words. An unknown Icelandic scholar of the 12th century exploited the concept of the phoneme and proposed a phonemic writing system for Icelandic, including diacritics for length and nasality. But his text remained unpublished until 1818 and even then was largely unknown outside Scandinavia (Robins, 1967). The modern era of phonetics is usually said to have begun with Sweet, who proposed what is essentially the phoneme in his *Handbook of Phonetics* 1877. He also devised an alphabet for transcription and distinguished between *broad* and *narrow* transcription, proposing many ideas that were eventually incorporated into the IPA. Sweet was considered the best practicing phonetician of his time; he made the first scientific recordings of languages for phonetic purposes and advanced the state of the art of articulatory description. He was also infamously difficult to get along with, a trait that is well captured in Henry Higgins, the stage character that George Bernard Shaw modeled after him. The phoneme was first named by the Polish scholar Baudouin de Courtenay, who published his theories in 1894.

Introductory phonetics textbooks include Ladefoged (1993) and Clark and Yallop (1995). Wells (1982) is the definitive three-volume source on dialects of English.

Many of the classic insights in acoustic phonetics had been developed by the late 1950s or early 1960s; just a few highlights include techniques like the sound spectrograph (Koenig et al., 1946), theoretical insights like the working out of the source-filter theory and other issues in the mapping between articulation and acoustics ((Fant, 1960), Stevens et al. 1953, Stevens and House 1955, Heinz and Stevens 1961, Stevens and House 1961) the F1xF2 space of vowel formants (Peterson and Barney, 1952), the understanding of the phonetic nature of stress and the use of duration and intensity as cues (Fry, 1955), and a basic understanding of issues in phone perception (Miller and Nicely 1955,Liberman et al. 1952). Lehiste (1967) is a collection of classic papers on acoustic phonetics. Many of the seminal papers of Gunnar Fant have been collected in Fant (2004).

Speech feature-extraction algorithms were developed in the 1960s and early 1970s, including the efficient fast Fourier transform (FFT) (Cooley and Tukey, 1965), the application of cepstral processing to speech (Oppenheim et al., 1968), and the development of LPC for speech coding (Atal and Hanauer, 1971).

Excellent textbooks on acoustic phonetics include Johnson (2003) and Ladefoged (1996). Coleman (2005) includes an introduction to computational processing of acoustics and speech from a linguistic perspective. Stevens (1998) lays out an influential theory of speech sound production. There are a number of software packages for acoustic phonetic analysis. Many of the figures in this book were generated by the **Praat** package (Boersma and Weenink, 2005), which includes pitch, spectral, and formant analysis, as well as a scripting language.

Praat

Exercises

14.1 Find the mistakes in the ARPAbet transcriptions of the following words:

a. "three" [dh r i]
b. "sing" [s ih n g]
c. "eyes" [ay s]
d. "study" [s t uh d i]
e. "though" [th ow]
f. "planning" [p pl aa n ih ng]
g. "slight" [s l iy t]

14.2 Ira Gershwin's lyric for *Let's Call the Whole Thing Off* talks about two pronunciations (each) of the words "tomato", "potato", and "either". Transcribe into the ARPAbet both pronunciations of each of these three words.

14.3 Transcribe the following words in the ARPAbet:

1. dark
2. suit
3. greasy
4. wash
5. water

14.4 Take a wavefile of your choice. Some examples are on the textbook website. Download the Praat software, and use it to transcribe the wavefiles at the word level and into ARPAbet phones, using Praat to help you play pieces of each wavefile and to look at the wavefile and the spectrogram.

14.5 Record yourself saying five of the English vowels: [aa], [eh], [ae], [iy], [uw]. Find F1 and F2 for each of your vowels.

CHAPTER

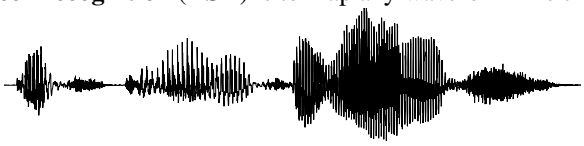
15 Automatic Speech Recognition

I KNOW not whether
I see your meaning: if I do, it lies
Upon the wordy wavelets of your voice,
Dim as an evening shadow in a brook,
Thomas Lovell Beddoes, 1851

Understanding spoken language, or at least transcribing the words into writing, is one of the earliest goals of computer language processing. In fact, speech processing predates the computer by many decades! The first machine that recognized speech was a toy from the 1920s. "Radio Rex", shown to the right, was a celluloid dog that moved (by means of a spring) when the spring was released by 500 Hz acoustic energy. Since 500 Hz is roughly the first formant of the vowel [eh] in "Rex", Rex seemed to come when he was called (David, Jr. and Selfridge, 1962).

In modern times, we expect more of our automatic systems. The task of **automatic speech recognition** (**ASR**) is to map any waveform like this:

ASR



to the appropriate string of words:

```
It's time for lunch!
```

Automatic transcription of speech by any speaker in any environment is still far from solved, but ASR technology has matured to the point where it is now viable for many practical tasks. Speech is a natural interface for communicating with appliances, or with digital assistants or chatbots, especially on cellphones, where keyboards are less convenient. ASR is also useful for general transcription, for example for automatically generating captions for audio or video text (transcribing movies or videos or live discussions). Transcription is important in fields like law where dictation plays an important role. Finally, ASR is important as part of augmentative communication (interaction between computers and humans with some disability resulting in difficulties or inabilities in typing or audition). The blind Milton famously dictated *Paradise Lost* to his daughters, and Henry James dictated his later novels after a repetitive stress injury.

In the next sections we'll introduce the various goals of the ASR task, describe how acoustic features are extracted, and introduce the **convolutional neural net** architecture which is commonly used as an initial layer in speech recognition tasks.

We'll then introduce two families of methods for ASR. The first is the **encoder-decoder** paradigm, and we'll introduce the baseline attention-based encoder decoder algorithm, sometimes called Listen Attend and Spell after an early implementation. We'll also introduce a more advanced encoder-decoder system, OpenAI's Whisper system (Radford et al., 2023) as well an open system based on the same architecture, OWSM (the Open Whisper-style Speech Model) (Peng et al., 2023). (These models have additional capabilities including translation, as we'll discuss later). The second is the use of self-supervised speech models (sometimes called SSL for self-supervised learning) like Wav2Vec2.0 or HuBERT, which are encoders that learn abstract representations of speech that can be used for ASR by pairing them with the **CTC** loss function for decoding.

We'll conclude with the standard **word error rate** metric used to evaluate ASR.

15.1 The Automatic Speech Recognition Task

Before describing algorithms for ASR, let's talk about how the ASR task itself varies. One dimension of variation is vocabulary size. Some ASR tasks have long been solved with extremely high accuracy, like those with a 2-word vocabulary (*yes* versus *no*) or an 11 word vocabulary like **digit recognition** (recognizing sequences of digits including *zero* to *nine* plus *oh*). Open-ended tasks like accurately transcribing videos or human conversations, with large vocabularies of 60,000 or more words, are much harder.

digit recognition

A second dimension of variation is who the speaker is talking to. Humans speaking to machines (either dictating or talking to a dialogue system) are easier to recognize than humans speaking to humans. **Read speech**, in which humans are reading out loud, for example in audio books, is also relatively easy to recognize. Recognizing the speech of two humans talking to each other in **conversational speech**, for example, for transcribing a business meeting, is the hardest. It seems that when humans talk to machines, or read without an audience present, they simplify their speech quite a bit, talking more slowly and more clearly.

read speech

conversational speech

A third dimension of variation is channel and noise. Speech is easier to recognize if it's recorded in a quiet room with head-mounted microphones than if it's recorded by a distant microphone on a noisy city street, or in a car with the window open.

A final dimension of variation is accent or speaker-class characteristics. Speech is easier to recognize if the speaker is speaking the same dialect or variety that the system was trained on. Speech by speakers of regional or ethnic dialects, or speech by children can be quite difficult to recognize if the system is only trained on speakers of standard dialects, or only adult speakers.

A number of publicly available corpora with human-created transcripts are used to create ASR test and training sets to explore this variation; we mention a few of them here since you will encounter them in the literature. **LibriSpeech** is a large open-source read-speech 16 kHz dataset with over 1000 hours of audio books from the LibriVox project, which has volunteers read and record copyright-free books (Panayotov et al., 2015). It has transcripts aligned at the sentence level. It is divided into an easier ("clean") and a more difficult portion ("other") with the clean portion of higher recording quality and with accents closer to US English. The division was done when the corpus was first released by running a speech recognizer (trained on read speech from the Wall Street Journal) on all the audio, computing the word error rate (WER, formally defined below) for each speaker based on the gold transcripts,

LibriSpeech

and dividing the speakers roughly in half, with recordings from lower-WER speakers called "clean" and recordings from higher-WER speakers "other".

Switchboard

The **Switchboard** corpus of prompted telephone conversations between strangers was collected in the early 1990s; it contains 2430 conversations averaging 6 minutes each, totaling 240 hours of 8 kHz speech and about 3 million words (Godfrey et al., 1992). Switchboard has the singular advantage of an enormous amount of auxiliary hand-done linguistic labeling, including parses, dialogue act tags, phonetic and prosodic labeling, and discourse and information structure. The **CALLHOME** corpus was collected in the late 1990s and consists of 120 unscripted 30-minute telephone conversations between native speakers of English who were usually close friends or family (Canavan et al., 1997).

CALLHOME

CHiME

A variety of corpora try to include input that is more natural. The **CHiME** Challenge is a series of difficult shared tasks with corpora that deal with robustness in ASR. The CHiME 6 task, for example, is ASR of conversational speech in real home environments (specifically dinner parties). The corpus contains recordings of twenty different dinner parties in real homes, each with four participants, and in three locations (kitchen, dining area, living room), recorded with distant microphones.

AMI

The **AMI** Meeting Corpus contains 100 hours of recorded group meetings (some natural meetings, some specially organized), with manual transcriptions and some additional hand-labels (Renals et al., 2007).

CORAAL

CORAAL is a collection of over 150 sociolinguistic interviews with African American speakers, with the goal of studying African American English (**AAE**), the many variations of language used in African American communities and others (Kendall and Farrington, 2020). The interviews are anonymized with transcripts aligned at the utterance level.

There are a wide variety of corpora available in other languages. In Chinese, for example, the **HKUST** Mandarin Telephone Speech corpus has 1206 transcribed ten-minute telephone conversations between speakers of Mandarin across China including conversations between friends and between strangers (Liu et al., 2006). The **AISHELL-1** corpus contains 170 hours of Mandarin read speech of sentences taken from various domains, read by different speakers mainly from northern China (Bu et al., 2017).

HKUST

AISHELL-1

Finally, there are many multilingual corpora. Common Voice (Ardila et al., 2020) is a freely available crowd-sourced corpus of transcribed read speech, stored in MPEG-3 format and designed for ASR. Crowd-working volunteers record themselves reading scripted speech, with scripts often extracted from Wikipedia articles. The recordings are then verified by other contributors. As of the writing of this chapter, Common Voice includes 33,150 hours of speech from 133 languages. FLEURS (Conneau et al., 2023) is a parallel speech dataset, built on the MT benchmark FLoRes-101 (Goyal et al., 2022), which has 3001 sentences extracted from English Wikipedia and translated into 101 other languages by human translators. For a subset of 2009 of the sentences in each of the 102 languages, FLEURS has recordings of 3 different native speakers reading the sentence, in total about 12 hours of speech per language.

Figure 15.1 shows the rough percentage of incorrect words (the **word error rate**, or WER, defined on page 360) from roughly state-of-the-art systems as of the time of this writing on some of these tasks. Note that the error rate on English read speech (like the LibriSpeech clean audiobook corpus) is around 2% ; transcription of speech read in English is highly accurate. By contrast, the error rate for transcribing conversations between humans is higher; 5.8 to 11% for the Switchboard and CALLHOME corpora or AMI meetings. The error rate is higher yet again for speakers of varieties

like African American English, and yet again for difficult conversational tasks like transcription of 4-speaker dinner party speech, which can have error rates as high as 25.5%. Character error rates (CER) are also higher for Mandarin natural conversation than for Mandarin read speech. Error rates are even higher for lower resource languages; we've shown a handful of examples.

English Tasks	**WER%**
LibriSpeech audiobooks 960hour clean	1.4
LibriSpeech audiobooks 960hour other	2.6
Switchboard telephone conversations between strangers	5.8
CALLHOME telephone conversations between family	11
AMI meetings	11
Sociolinguistic interviews, CORAAL (AAE)	16.2
CHiME6 dinner parties with distant microphones	25.5
Sample tasks in other languages	**WER%**
Common Voice 15 Vietnamese	39.8
Common Voice 15 Swahili	51.2
FLEURS Bengali	50
Chinese (Mandarin) Tasks	**CER%**
AISHELL-1 Mandarin read speech corpus	3.9
HKUST Mandarin Chinese telephone conversations	18.5

Figure 15.1 Rough Word Error Rates (WER = % of words misrecognized) reported around 2023-4 for ASR on various American English and other language recognition tasks, and character error rates (CER) for two Chinese recognition tasks.

15.2 Convolutional Neural Networks

CNN

The convolutional neural network, or **CNN** (and sometimes shortened as **convnet**), is a network architecture that is particularly useful for extracting features in speech and vision applications. A convolutional layer for speech takes as input a representation of the audio input (either as the raw audio or as Mel spectra) and produces as output a sequence of latent representations of the input speech. In ASR systems like Whisper, wav2vec2.0, or HuBERT, convolutional layers are stacked as an initial set of layers producing speech representations that are then passed to transformer layers.

A standard feedforward layer is fully connected; every input is connected to every output. By contrast, a convolutional network makes use of the idea of a kernel, a kind of smaller network that we pass over the input. For example in image classification tasks, we pass the kernel horizontally and vertically over the image to recognize visual features, and so we describe a visual as a 2d (for 2 dimensional) convolutional network. For speech, we will slide our kernel over the signal in the time dimension to extract speech features, so CNNs for speech are 1d convolutional networks.

Let's flesh out this intuition a bit more. We'll start with a very schematic version of a convolutional layer that takes as input a single sequence of vectors $\mathbf{x}_1 \dots \mathbf{x}_t$ and produces as output a single sequence of vectors $\mathbf{z}_1 \dots \mathbf{z}_t$, of the same length t. Afterwards we'll see how to deal with more complex inputs and outputs.

kernel
convolving

A CNN uses a **kernel**, a small vector of weights $\mathbf{w}_1 \dots \mathbf{w}_k$, to extract features. It does this by **convolving** this kernel with the input. The **convolution** of a kernel with

a signal has 3 steps:

1. Flip the kernel left-to-right
2. Pass the kernel frame by frame (temporally) across the input
   - At each frame computing the dot product of the kernel with the local input values
3. The output is the resulting sequence of dot products

We can think of the convolution process as finding regions in the signal that are similar to the kernel, since the dot product is high when two vectors are similar. The convolution operation is represented by the * operator (an unfortunate overloading of this symbol that also refers to simple multiplication). Let's see how to compute $\mathbf{x} * \mathbf{w}$, the convolution of a single vector $\mathbf{x}$ with a kernel vector $\mathbf{w}$. Let's first think about the simple case of a kernel width of 1. We compute each output element $\mathbf{z}_j$ as the product of the kernel with $\mathbf{x}_j$:

$$\textit{convolution with width-1 kernel:} \quad \mathbf{z}_j = \mathbf{x}_j \mathbf{w}_0 \quad \forall j : 1 \leq j \leq t \tag{15.1}$$

Fig. 15.2 shows an intuition of this computation.

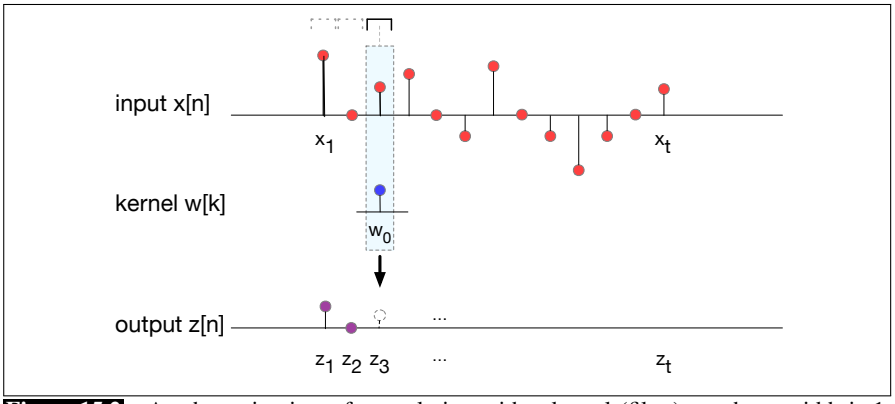


Figure 15.2 A schematic view of convolution with a kernel (filter) $\mathbf{w}$ whose width is 1. The kernel is walked across the input, and the output at each frame $\mathbf{z}_i$ is the dot product of the kernel with the input frame. With a kernel of length 1 we don't have to worry about flipping the kernel, and the dot product is just the scalar product. The figure shows the computation of $\mathbf{z}_3$ as $\mathbf{x}_3 \times \mathbf{w}_1$.

Let's now turn to longer kernels. Although we've described the first step of the convolution as flipping the kernel, in fact in ASR systems (or in component libraries like pytorch) we skip this step. Technically this means that the algorithm we are using is not in fact convolution, it's instead **cross-correlation**, which is the name for an algorithm of walking a kernel across a signal, computing its dot product frame by frame, without flipping it first. The difference doesn't matter, since the parameters of the kernel will be learned during training, and so the model could easily learn a kernel with the parameters in either order. Still, for historical reasons we still call this process a 1d convolution rather than cross-correlation.

cross-correlation

Let's see a more general equation for these longer kernels. To avoid the convolution being undefined at the left and right edges of the signal, we can **pad** the input by adding a small number p of zeros at the beginning and end of the signal, so that we can start the center of the kernel at the first element $\mathbf{x}_1$, and there will be a defined value to the left of $\mathbf{x}_1$. This also turns out to make it simple to have the

pad

output length as the same as the input length. To do this, it's convenient to define the kernel vector as having an odd number of elements of length $k = 2p + 1$, thus with the center element having p elements on either side. Each element z_j of the output vector $\mathbf{z}$ is then computed as the following dot product:

$$\mathbf{z}_j = \sum_{i=-p}^{p} \mathbf{x}_{j+i}\mathbf{w}_{i+p} \tag{15.2}$$

Fig. 15.3 shows the computation of the convolution $\mathbf{x} * \mathbf{w}$ with a kernel whose width is 3, and with padding of 1 frame at the beginning and end of $\mathbf{x}$ with a value of zero.

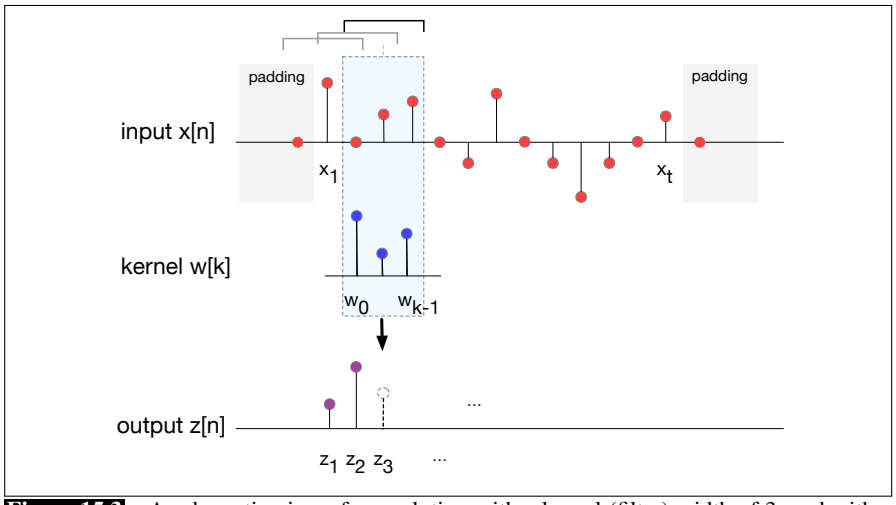


Figure 15.3 A schematic view of convolution with a kernel (filter) width of 3, and with a padding of 1, showing a zero value added at the start and end of the signal. The (already flipped) kernel is walked across the input, and the output at each frame $\mathbf{z}_i$ is the dot product of the kernel with the input in the window. The figure shows the computation of $\mathbf{z}_3$.

Note that the size k (the **receptive field**) of the kernel is designed to be small compared to the signal. For example for the convolutional layers in Whisper, the kernel width is 3 frames, meaning the kernel is a vector of length 3 (we say that the kernel has a **receptive field** of 3). That means that the kernel is being compared to 3 frames of speech. In Whisper there is a frame every 10 ms and each frame represent a window of 25ms of speech information. That means each kernel is extracting information from about 40 ms of speech (10 + 10 + 12.5 + 12.5). That's long enough to extract various phonetic features like formant transitions or stop closures or aspiration.

receptive field

We've now described a simplified view of convolution in which the input is a single vector $\mathbf{x}$ and the output is a single vector $\mathbf{z}$, both corresponding to a signal over time. In practice, the input to a convolutional layer is commonly the output from a log mel spectrum, which means it has many (say 128) channels, one for each log mel filters output. The kernel will have separate vectors for each of these input channels. We say that the kernel has a **depth** of 128, meaning that the kernel is of shape [128,3].

depth

To get the output of the kernel, we sum over all the input channels. That is, we get a single output $\mathbf{z}^c$ for each of the input channels $\mathbf{x}^c$ by convolving the kernel $\mathbf{w}$

with it, and then we sum up all the resulting outputs:

$$\mathbf{z} = \sum_{c=1}^{C_i} \mathbf{x}^c * \mathbf{w} \tag{15.3}$$

The output at frame j, $\mathbf{z}_j$, thus integrates information from all of the input channels.

Finally, the output from a convolution layer is also more complex than just a vector consisting of a single scalar value to represent each frame. Instead, the output of the convolution layer for a given input frame needs to be an embedding, a latent representation of that frame. As with all neural models, latent representations should have the model dimensionality, whatever that is. For example the model dimensionality of Whisper is 1280, and so the convolutional layer needs to have one output channel for each of these 1280 dimensions of the model. In order to do this, we'll actually learn one separate kernel for each of the model dimensions. That is, we'll learn 1280 separate kernels, each kernel having the depth of the number of input channels (for example 128), and a filter-width (say of 3). That way, the embedding representation of each frame will have 1280 independently computed features of the input signal. We show a schematic in Fig. 15.4

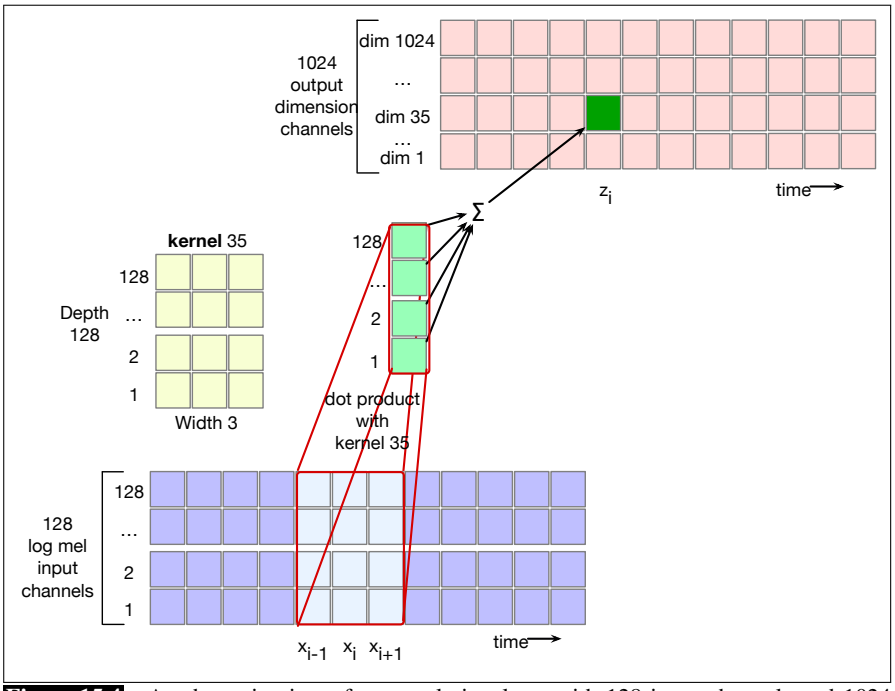


Figure 15.4 A schematic view of a convolutional net with 128 input channels and 1024 output channels. We see how at time point i one of the 1024 kernels ("kernel 35", each of depth 128 and width 3) is dot-product-ed with (each of) the 128 log mel spectrum input vectors, and then summed to produce a single value for one dimension of the output embedding at time i.

stride

A 1d convolution layer can also have a **stride**. Stride is the amount that we move the kernel over the input between each step. The figures above show a stride of 1, meaning that we first position the kernel over $\mathbf{x}_1$, then $\mathbf{x}_2$, then $\mathbf{x}_3$, and so on. For a stride of 2, we would first position the kernel over $\mathbf{x}_1$, then $\mathbf{x}_3$, then $\mathbf{x}_5$, and so on. A longer stride means a shorter output sequence; a stride of two means the output

sequence **z** will be half the length of the input sequence **x**. Convolutional layers with strides greater than 1 are commonly used to shorten an input sequence. This is useful partly because a shorter signal takes less memory and computational bandwidth, but also, as we'll see in the next section, because it helps address the mismatch between the length of acoustic frame embeddings (10 ms) and letters or words, which cover much more of the signal.

Finally, in practice a convolutional layer can be followed by an output nonlinearity, like a ReLU layer.

15.3 The Encoder-Decoder Architecture for ASR

AED
listen attend and spell

The first ASR architecture we introduce is the **encoder-decoder** architecture, the same architecture introduced for MT in Chapter 12. Fig. 15.5 sketches this architecture, called **attention-based encoder decoder** or **AED**, or **listen attend and spell** (**LAS**) after the two papers which first applied it to speech (Chorowski et al. 2014, Chan et al. 2016).

The input to the architecture **x** is a sequence of t acoustic feature vectors $\mathbf{X} = \mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_t$, one vector per 10 ms frame. We often start from the log mel spectral features described in the previous section, although it's also possible to start from a raw wavefile. The output sequence Y can be either letters or tokens (BPE or sentencepiece); we'll assume letters just to simplify the explanation here. Thus the output sequence $Y = (\langle \text{SOS} \rangle, y_1, ..., y_m \langle \text{EOS} \rangle)$, assuming special start of sequence and end of sequence tokens $\langle \text{sos} \rangle$ and $\langle \text{eos} \rangle$ and each y_i is a character; for English we might choose the set:

$$y_i \in \{a, b, c, ..., z, 0, ..., 9, \langle \text{space} \rangle, \langle \text{comma} \rangle, \langle \text{period} \rangle, \langle \text{apostrophe} \rangle, \langle \text{unk} \rangle\}$$

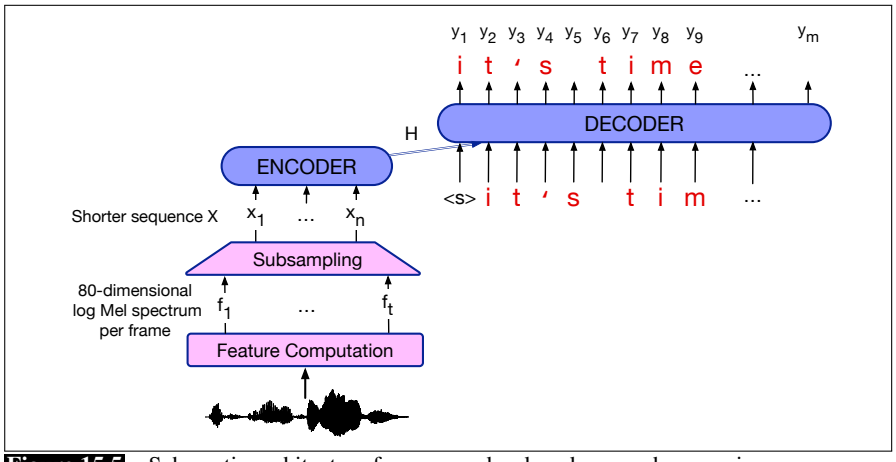


Figure 15.5 Schematic architecture for an encoder-decoder speech recognizer.

This architecture is also used in the Whisper model from OpenAI (Radford et al., 2023). Fig. 15.6 shows a subpart of the Whisper architecture (Whisper also does other speech tasks like speech translation and voice activity detection, which we'll discuss in the next chapter). Whisper models and inference code are publicly released, but the training code and training data are not. However, there are open-source projects that use a Whisper-style architecture, like the Open Whisper-style

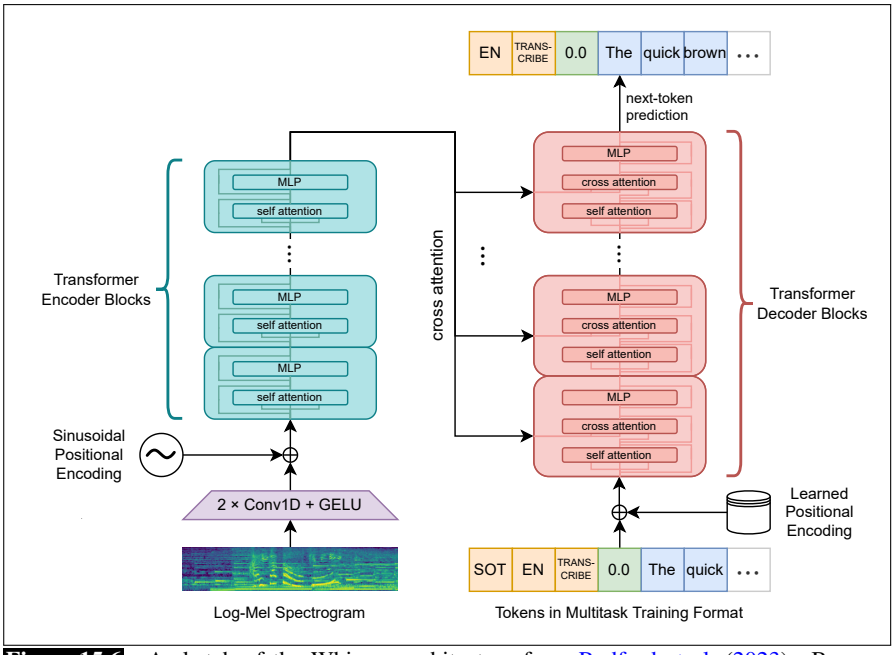


Figure 15.6 A sketch of the Whisper architecture from Radford et al. (2023). Because Whisper is a multitask system that also does translation, Whisper's transcription format has a Start of Transcript (SOT) token, a language tag, and then an instruction token for whether to transcribe or translate.

Speech Model (OWSM), which reproduces Whisper-style training but offers a fully open-source toolkit and publicly available data (Peng et al., 2023).

15.3.1 Input and Convolutional Layers

The encoder-decoder architecture is particularly appropriate when input and output sequences have stark length differences, as they do for speech, with long acoustic feature sequences mapping to much shorter sequences of letters or words. For example English, words are on average 5 letters or 1.3 BPE tokens long (Bostrom and Durrett, 2020) and, in natural conversation, the average word lasts about 250 milliseconds (Yuan et al., 2006), or 25 frames of 10ms. So the speech signal in 10ms frames is about 5 (25/5) to 19 (25/1.3) times longer than the text signal in words or tokens.

Because this length difference is so extreme for speech, encoder-decoder architectures for speech usually have a compression stage that shortens the acoustic feature sequence before the encoder stage. (We can additionally make use of a loss function that is designed to deal well with compression, like the CTC loss function we'll introduce later.)

The goal of the subsampling is to produce a shorter sequence $\mathbf{X} = \mathbf{x}_1, ..., \mathbf{x}_n$ that will be the input to the transformer encoder. A very simple baseline algorithm is a method sometimes called **low frame rate** (Pundak and Sainath, 2016): for time i we stack (concatenate) the acoustic feature vector f_i with the prior two vectors f_{i-1} and f_{i-2} to make a new vector three times longer. Then we simply delete f_{i-1} and f_{i-2}. Thus instead of (say) a 40-dimensional acoustic feature vector every 10 ms, we have a longer vector (say 120-dimensional) every 30 ms, with a shorter sequence length $n = \frac{t}{3}$.

low frame rate

But the most common way of creating a shorter input sequence is to use the convolutional layers we introduced in the previous section. When a convolutional layer has a stride greater than 1, the output sequence becomes shorter than the input sequence. Let's see this in two commonly used ASR systems.

The Whisper system (Radford et al., 2023) has an audio context window of 30 seconds. It extracts 128 channel log mel features for each frame, with a 25ms window and a stride of 10ms. These are then normalized to 0 mean and a range of -1 to 1. A stride of 10 ms (100 frames per second) means there are 3000 input frames in a 30 second context window. These 3000 frames are passed to two convolutional layers, each one followed by a nonlinearity (Whisper uses GELU (Gaussian Error Linear Unit), which is a smoother version of ReLU). The first convolutional layer has 128 input channels and uses a stride of 1, with number of output channels being the model dimensionality, and the window length is 3000. For the second convolutional layer the number of input and output channels is the model dimensionality, and there is a stride of 2. The stride of 2 in the second convolutional layer makes the output sequence half the length of the input sequence, bringing the output window length down to 1500 and producing an audio token every 20 ms. Sinusoidal position embeddings are added to these audio encodings before the output of this front end is passed to the transformer encoder.

HuBERT (Hsu et al., 2021) uses an alternative front end architecture, in which convolutional layers are used to completely replace the computation of the spectrum. So the input is raw 16kHz sampled audio, and it is passed through seven 512-channel layers with strides [5,2,2,2,2,2,2] and kernel widths [10,3,3,3,3,2,2] which learn both to extract spectral information, and to shorten the input sequence by 320x, from 16kHz (= one representation per .0625 ms) down to a 20 ms framerate. Positional encodings are added to the input, and then a GELU and layer norm are applied before the output is passed to the transformer encoder.

15.3.2 Inference

After the convolutional stage, encoder-decoders for speech use the same architecture (transformer with cross-attention) as for MT.

Let's remind ourselves of the encoder-decoder architecture that we introduced in Chapter 12. It uses two transformers: an **encoder**, which is the same as the basic transformer from Chapter 8, and a **decoder**, which has one addition: a new layer called the **cross-attention** layer. The encoder takes the acoustic input $\mathbf{X} = \mathbf{x}_1, ..., \mathbf{x}_n$ and maps them to an output representation $\mathbf{H}^{enc} = \mathbf{h}_1, ..., \mathbf{h}_n$; via a stack of encoder blocks.

The decoder is essentially a conditional language model that attends to the encoder representation and generates the target text (letters or tokens) one by one, at each timestep conditioning on the audio representations from the encoder and the previously generated text to generate a new letter or token.

cross-attention

The transformer blocks in the decoder have an extra layer with a special kind of attention, **cross-attention**. Cross-attention has the same form as the multi-head attention in a normal transformer block, except that while the queries as usual come from the previous layer of the decoder, the keys and values come from the output of the *encoder*.

That is, where in standard multi-head attention the input to each attention layer is $\mathbf{X}$, in cross attention the input is the final output of the encoder $\mathbf{H}^{enc} = \mathbf{h}_1, ..., \mathbf{h}_n$. $\mathbf{H}^{enc}$ is of shape $[n \times d]$, each row representing one acoustic input token. To link the keys and values from the encoder with the query from the prior layer of the decoder,

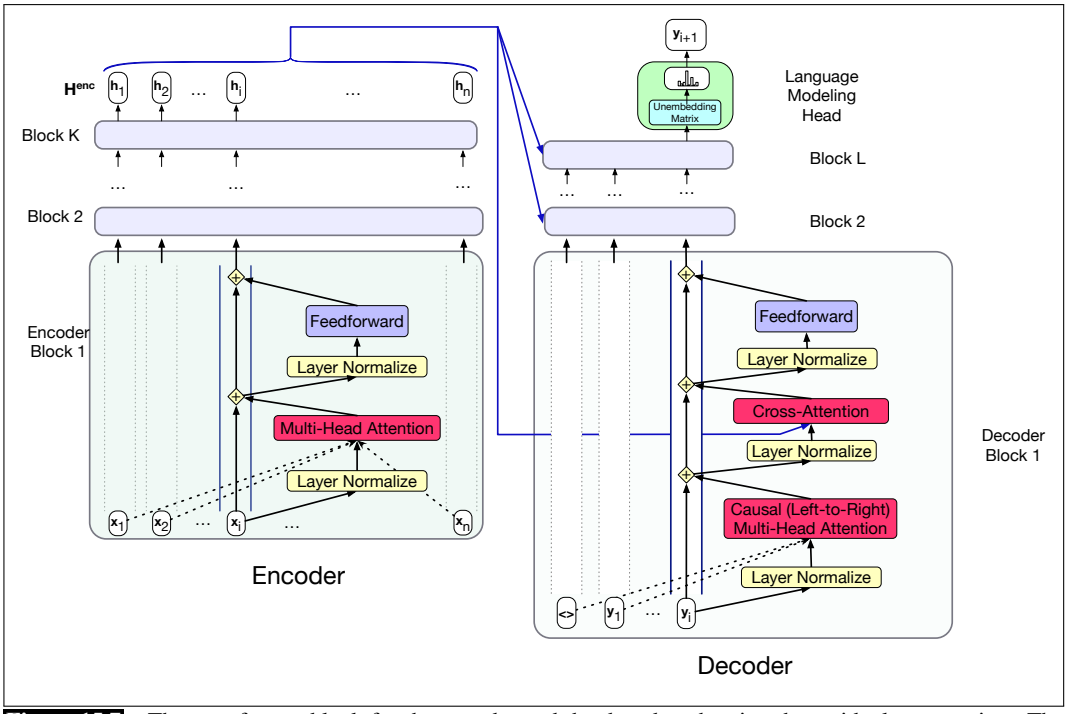


Figure 15.7 The transformer block for the encoder and the decoder, showing the residual stream view. The final output of the encoder $\mathbf{H}^{enc} = \mathbf{h}_1, ..., \mathbf{h}_n$ is the context used in the decoder. The decoder is a standard transformer except with one extra layer, the **cross-attention** layer, which takes that encoder output $\mathbf{H}^{enc}$ and uses it to form its $\mathbf{K}$ and $\mathbf{V}$ inputs.

we multiply the encoder output $\mathbf{H}^{enc}$ by the cross-attention layer's key weights $\mathbf{W}^{\mathbf{K}}$ and value weights $\mathbf{W}^{\mathbf{V}}$. The query comes from the output from the prior decoder layer $\mathbf{H}^{dec[\ell-1]}$, which is multiplied by the cross-attention layer's query weights $\mathbf{W}^{\mathbf{Q}}$:

$$\mathbf{Q} = \mathbf{H}^{dec[\ell-1]}\mathbf{W}^{\mathbf{Q}}; \;\; \mathbf{K} = \mathbf{H}^{enc}\mathbf{W}^{\mathbf{K}}; \;\; \mathbf{V} = \mathbf{H}^{enc}\mathbf{W}^{\mathbf{V}} \tag{15.4}$$

$$\text{CrossAttention}(\mathbf{Q},\mathbf{K},\mathbf{V}) \;=\; \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\intercal}}{\sqrt{d_k}}\right)\mathbf{V} \tag{15.5}$$

The cross attention thus allows the decoder to attend to the acoustic input as projected into the entire encoder final output representations. The other attention layer in each decoder block, the multi-head attention layer, is the same causal (left-to-right) attention that we saw in Chapter 8. But the multi-head attention in the encoder, however, is allowed to look ahead at the entire source audio, so it is not masked.

For inference, the probability of the output string y is decomposed as:

$$p(y_1,\ldots,y_n) = \prod_{i=1}^{n} p(y_i|y_1,\ldots,y_{i-1},\mathbf{X}) \tag{15.6}$$

We can produce each letter of the output via greedy decoding:

$$\hat{y}_i \;=\; \text{argmax}_{\text{char}\in\text{ Alphabet}} P(\text{char}|y_1 \ldots y_{i-1},\mathbf{X}) \tag{15.7}$$

Alternatively encoder-decoders like Whisper or OWSM also use beam search as described in the next section. This is particularly relevant when we are adding a language model.

Adding a language model Since an encoder-decoder model is essentially a conditional language model, encoder-decoders implicitly learn a language model for the output domain of letters from their training data. However, the training data (speech paired with text transcriptions) may not include sufficient text to train a good language model. After all, it's easier to find enormous amounts of pure text training data than it is to find text paired with speech. Thus we can can usually improve a model at least slightly by incorporating a very large language model.

n-best list
rescore

The simplest way to do this is to use beam search to get a final beam of hypothesized sentences; this beam is sometimes called an **n-best list**. We then use a language model to **rescore** each hypothesis on the beam. The scoring is done by interpolating the score assigned by the language model with the encoder-decoder score used to create the beam, with a weight λ tuned on a held-out set. Also, since most models prefer shorter sentences, ASR systems normally have some way of adding a length factor. One way to do this is to normalize the probability by the number of characters in the hypothesis $|Y|_c$. The following is the scoring function for Listen, Attend, and Spell (Chan et al., 2016):

$$\text{score}(Y|\mathbf{X}) = \frac{1}{|Y|_c}\log P(Y|\mathbf{X}) + \lambda \log P_{LM}(Y) \tag{15.8}$$

15.3.3 Learning

Encoder-decoders for speech are trained with the normal cross-entropy loss generally used for conditional language models. At timestep i of decoding, the loss is the log probability of the correct token (letter) y_i:

$$L_{CE} = -\log p(\mathbf{y}_i|y_1,\ldots,\mathbf{y}_{i-1},\mathbf{X}) \tag{15.9}$$

The loss for the entire sentence is the sum of these losses:

$$L_{CE} = -\sum_{i=1}^{m}\log p(\mathbf{y}_i|\mathbf{y}_1,\ldots,\mathbf{y}_{i-1},\mathbf{X}) \tag{15.10}$$

This loss is then backpropagated through the entire end-to-end model to train the entire encoder-decoder.

As we described in Chapter 12, we normally use teacher forcing, in which the decoder history is forced to be the correct gold y_i rather than the predicted $\hat{y}_i$. It's also possible to use a mixture of the gold and decoder output, for example using the gold output 90% of the time, but with probability .1 taking the decoder output instead:

$$L_{CE} = -\log p(\mathbf{y}_i|\mathbf{y}_1,\ldots,\hat{\mathbf{y}}_{i-1},\mathbf{X}) \tag{15.11}$$

Modern data sizes are quite large. For example Whisper-v2 is trained on a corpus of 680,000 hours of speech, mostly from English, but also including 118,000 hours from 96 other languages. Data quality is important, so systems that scrape web data for training implement methods to remove ASR-generated transcripts from their training corpora, such as filtering data that is all uppercase or all lowercase. The open OWSM system is trained on 180k hours, mainly hand-transcribed publicly available data, including such datasets as LibriSpeech and Multilingual LibriSpeech, Common Voice, FLEURS, Switchboard, AMI, and others; see (Peng et al., 2023) for details.

15.4 Self-supervised models: HuBERT

self-supervised

An alternative to the encoder-decoder architecture are the class of **self-supervised** speech models. These models don't directly learn to map an acoustic input to a string of letters and tokens. Instead, they first bootstrap a set of discrete phonetic units from the acoustic input, learning to map from waveforms to these induced units. This pretraining phase doesn't require transcripts; just unlabeled speech files. After they are pretrained, these models can then be finetuned to do ASR on a smaller set of labeled data, audio paired with transcripts. These models have the advantage that they can take advantage of large amounts of untranscribed audio for most of their training.

HuBERT
wav2vec 2.0

Here we'll introduce one self-supervised model called **HuBERT** (Hsu et al., 2021). HuBERT and similar models like **wav2vec 2.0** (Baevski et al., 2020) use the same intuition as the masked language models like BERT introduced in Chapter 9: we mask out some part of the input and train the model to guess what was hidden by the mask.

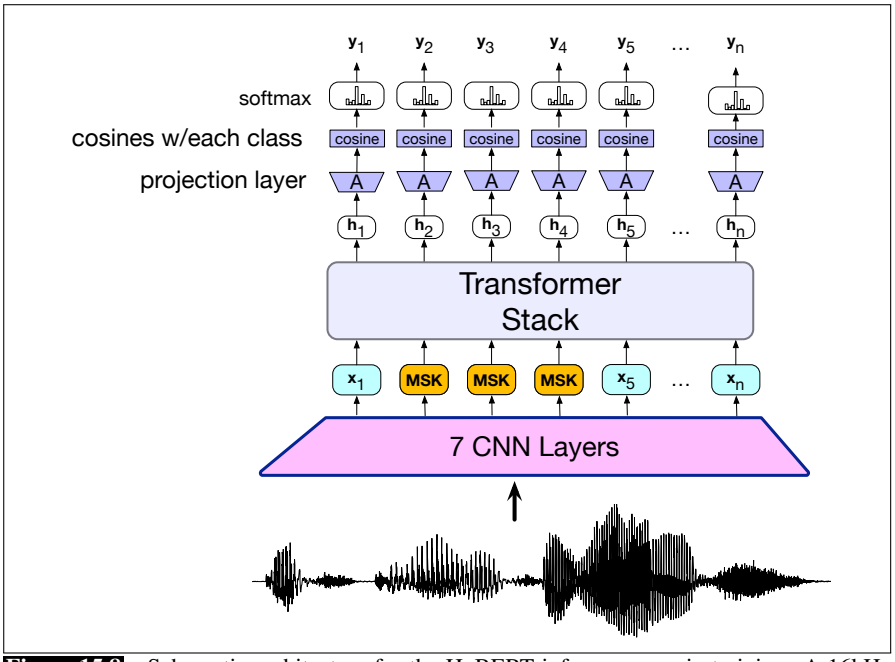


Figure 15.8 Schematic architecture for the HuBERT inference pass in training. A 16kHz wavfile is passed through a series of convolutional layers, some frames are replaced with a MASK token, and then the sequence is passed though a transformer stack, and then a linear layer that projects the transformer output to an output embedding. This embedding is compared via cosine with the embeddings for each of the 100/500 phonetic classes to produce a logit which is passed through softmax to get a probability distribution over the classes at each frame.

15.4.1 HuBERT forward pass

Let's first show just the forward pass for HuBERT used during training, and then we'll see this in its full training context with the backwards pass. As discussed earlier, the input to the HuBERT forward pass is raw 16kHz wavefiles as input,

and the output at each 20ms time frame will be a probability distribution over a set of induced phonetic classes C (100 classes or 500 classes, depending on the stage). Fig. 15.8 shows a sketch of the components. The wavefile is passed through 7 512-channel convolutional layers which learn both to extract spectral information, and to shorten the input sequence down to a 20 ms frame, after which positional encodings are added, and then GELU and layer norm. Selected tokens are then replaced with a mask token, a trained embedding that is shared by all masked frames. The whole sequence is passed through a transformer stack, and the output is passed through a linear projection layer $\mathbf{A}$. The output embedding at each 20ms frame is then compared via cosine with each of the embeddings for the 100 (/500) phonetic classes, resulting in a set of 100 logits representing the similarity of the current 20ms audio timestep to each class. These are then passed through a softmax to get a probability distribution over the classes.

15.4.2 Learning for HuBERT

Let's first discuss how we induce the 100 or 500 phonetic classes that are the target of training. To bootstrap these units, HuBERT starts with **mel frequency cepstral coefficients**, or **MFCC** vectors, a 39-dimensional feature vector that emphasizes aspects of the signal that are relevant for detection of phonetic units. These vectors can be extracted from the acoustic signal as summarized in Section 14.6. We extract MFCC vectors for the entire acoustic training dataset (the original HuBERT implementation used 960 hours of LibriSpeech data resulting in 172 million vectors). Next we **cluster** the MFCC vectors using the **k-means** clustering algorithm described below in Section 15.4.3. Clustering means to group the vectors into k classes. The output of clustering is a **codebook** of k vectors, called **codewords** or **templates** or **prototypes**, each representing a cluster. Each of these k clusters is an acoustic unit that we can use as the gold targets for training.

Now let's consider the entire training process. After the acoustic input is run through the CNN layers, a span of tokens in the context window is chosen to be masked, and for those tokens the CNN output is replaced by a MASK embedding. The entire context window is passed through the transformer layers, and the transformer output h_t^L at each timestep t is multiplied by the projection layer matrix $\mathbf{A}$ to project it into the class embedding space. The resulting representation is then compared to the embedding for each of the classes in C (using cosine), and a softmax (with temperature parameter τ=0.1) is used to turn the similarity into a probability:

$$p(c|\mathbf{X},t) = \frac{\exp(\text{sim}(\mathbf{A}\mathbf{h}_t, \mathbf{e_c})/\tau)}{\sum_{c'=1}^{C} \exp(\text{sim}(\mathbf{A}\mathbf{h_t}, \mathbf{e}_{c'})/\tau)} \tag{15.12}$$

As Fig. 15.9 shows, in parallel with this forward pass, the input waveform is passed through an MFCC to create a 39-dimensional vector which is then mapped to one of the 100 classes by choosing the most similar centroid in the codebook. The loss function is then the sum, over the set of masked tokens M, of the probability that the model assigns to these correct units:

$$L = \sum_{t \in M} \log p(z_t|\mathbf{X}) \tag{15.13}$$

Thus, as in masked language modeling, the model is being trained to predict the

units associated with the masked frames. This loss is then backpropagated through the model

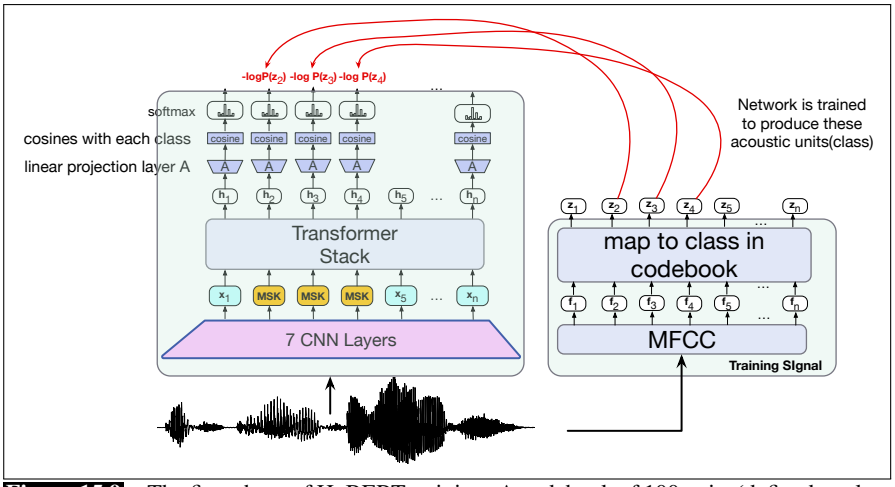


Figure 15.9 The first phase of HuBERT training. A codebook of 100 units (defined as clusters of 39-dimensional MFCC vectors) is used as the targets for training. For each timestep t, computes the probability of that class, and uses the logprob as the loss.

Once the model has been initially trained to map to MFCC vector centroids, a second stage of training occurs, where we take the representations produced by the model, cluster them into 500 clusters, and use those instead as the target for training. The intuition is that the initial MFCC clusters will bias the model toward phonetic representations, but after enough training the model will learn more accurate and fine-grained representations. Fig. 15.10 shows the intuition.

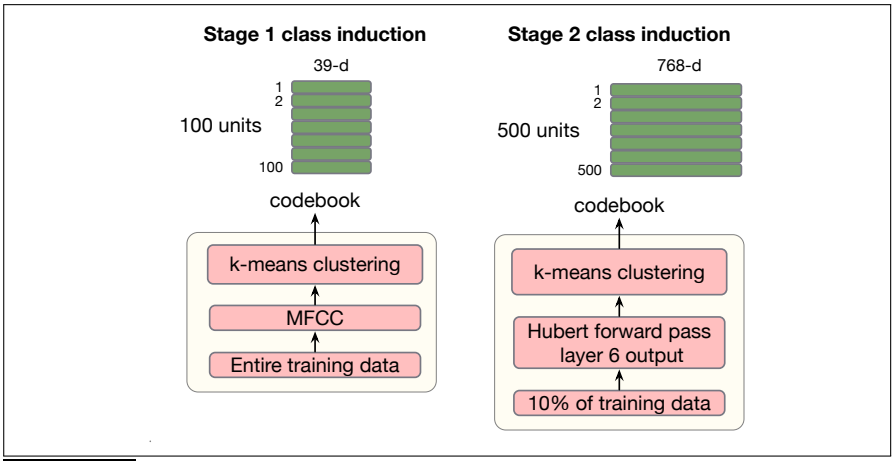


Figure 15.10 Creating the targets for the two stages of HuBERT training. In the first stage, 100 acoustic units are created by computing 39-dimensional MFCC vectors for the entire training data and then clustering them with k-means. In the second stage, 500 units are created by passing a subsample of the training data through the HuBERT model after the first stage training, taking the output of an intermediate transformer layer (layer 6) and clustering them with k-means.

After HuBERT has been pretrained, the projection and cosine layers are removed and a randomly initialized linear + softmax layer is added, mapping into 29 classes

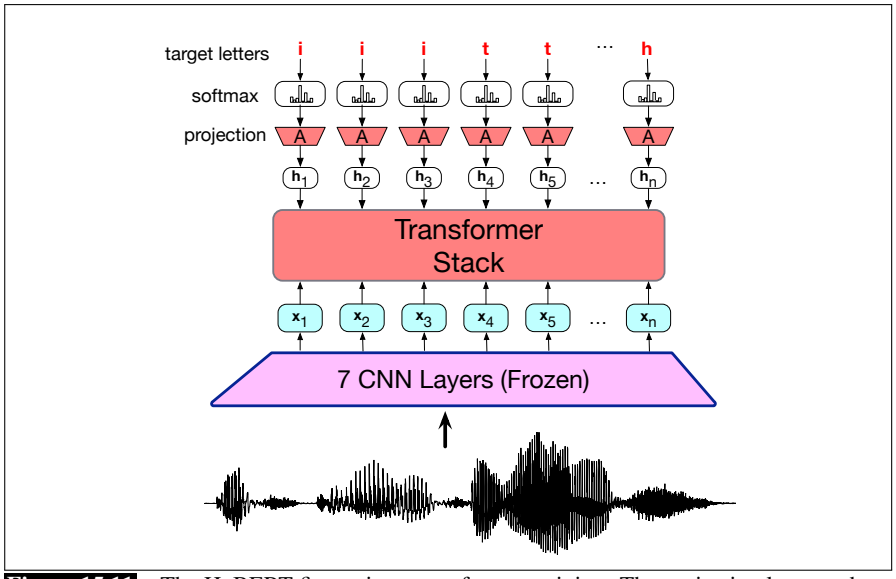


Figure 15.11 The HuBERT finetuning pass after pretraining. The projection layer and cosine steps are removed, leaving only a randomly initialized projection/softmax layer. The CNN layers are frozen, and the rest of the model is finetuned on a dataset of audio with transcripts, trained with the CTC loss (Section 15.5) to produce letters as output. The parameters that are updated in finetuning are shown in red (the projection layer and the transformer stack).

(corresponding to the 26 English letters and a few extra characters) for the ASR task. The CNNs are frozen and the rest of the model is finetuned for ASR using the CTC loss function to be described in Section 15.5.

15.4.3 K-means clustering

k-means

In this section we give the **k-means** clustering algorithm more formally. K-means is a family of algorithms for grouping a set of vector data into k clusters. Clustering is useful whenever we want to treat a group of elements in the same way. In speech processing it is very commonly used whenever we need to convert a set of vectors over real values into a set of discrete symbols. Besides its use here in HuBERT, we'll return to it in Chapter 16 as an algorithm for creating discrete acoustic tokens for TTS.

We generally use the name k-means to mean a simple version of the family: a two-step iterative algorithm that is given a set of N vectors $\mathbf{v}^{(1)}..\mathbf{v}^{(N)}$ each of d dimensions, i.e. $\forall i, \mathbf{v}^{(i)} \in \mathbb{R}^d$, and a constant k, where usually $N >> k$.

centroid

The two-step algorithm is based on iteratively updating a set of k **centroid** vectors. A **centroid** is the geometric center of a set of a points in n-dimensional space.

The algorithm has two steps. In the assignment step, given a set of k current centroids and a dataset of vectors, it assigns each vector to the cluster whose codeword is the closest (by squared Euclidean distance). In the re-estimation step, it recomputes the codeword for each cluster by recomputing the mean vector. Note that the resulting mean vector need not be an actual point from the dataset. We iterate back and forth between these two steps.

Here's the algorithm:

codeword
template
prototype

Initialization: For each cluster k choose a random vector $\mu_k \in \mathbb{R}^d$ to be the **codeword** (also called **template** or **prototype**) for the cluster. The result is a

codebook

codebook that has one codeword for each of the k clusters.

Then repeat iteratively until convergence:

1. **Assignment**: For each vector $\mathbf{v}^{(\mathbf{i})}$ in the dataset assign it to one of the k clusters by choosing the one with the nearest codeword μ. Most simply we can define 'nearest' as the cluster whose codeword has the smallest squared Euclidean distance to $\mathbf{v}^{(\mathbf{i})}$.

$$\text{cluster}^{(i)} = \underset{1<j<k}{\text{argmin}} ||\mathbf{v}^{(i)} - \mu_j||^2 \tag{15.14}$$

   where $||\mathbf{v}||$ is the L2 norm of the vector $\sum_{j=1}^{d} \mathbf{v}_j^2$

2. **Re-estimation**: Re-estimate the codeword for each cluster by recomputing the mean (centroid) of all the vectors in the cluster. If S_i is the set of vectors in cluster i, then

$$\forall i: \quad \mu_i = \frac{1}{|S_i|} \sum_{\mathbf{v} \in S_i} \mathbf{v} \tag{15.15}$$

15.5 CTC

We pointed out in the previous section that speech recognition has two particular properties that make it very appropriate for the encoder-decoder architecture, where the encoder produces an encoding of the input that the decoder uses attention to explore. First, in speech we have a very long acoustic input sequence X mapping to a much shorter sequence of letters Y, and second, it's hard to know exactly which part of X maps to which part of Y.

CTC

In this section we briefly introduce an alternative to encoder-decoder: an algorithm and loss function called **CTC**, short for **Connectionist Temporal Classification** (Graves et al., 2006), that deals with these problems in a very different way. The intuition of CTC is to output a single character for every frame of the input, so that the output is the same length as the input, and then to apply a collapsing function that combines sequences of identical letters, resulting in a shorter sequence.

Let's imagine inference on someone saying the word *dinner*, and let's suppose we had a function that chooses the most probable letter for each input spectral frame representation x_i. We'll call the sequence of letters corresponding to each input

alignment

frame an **alignment**, because it tells us where in the acoustic signal each letter aligns to. Fig. 15.12 shows one such alignment, and what happens if we use a collapsing function that just removes consecutive duplicate letters.

Well, that doesn't work; our naive algorithm has transcribed the speech as *diner*, not *dinner*! Collapsing doesn't handle double letters. There's also another problem with our naive function; it doesn't tell us what symbol to align with silence in the input. We don't want to be transcribing silence as random letters!

The CTC algorithm solves both problems by adding to the transcription alphabet

blank

a special symbol for a **blank**, which we'll represent as ␣. The blank can be used in the alignment whenever we don't want to transcribe a letter. Blank can also be used between letters; since our collapsing function collapses only consecutive duplicate letters, it won't collapse across ␣. More formally, let's define the mapping $B : a \rightarrow y$ between an alignment a and an output y, which collapses all repeated letters and then removes all blanks. Fig. 15.13 sketches this collapsing function B.

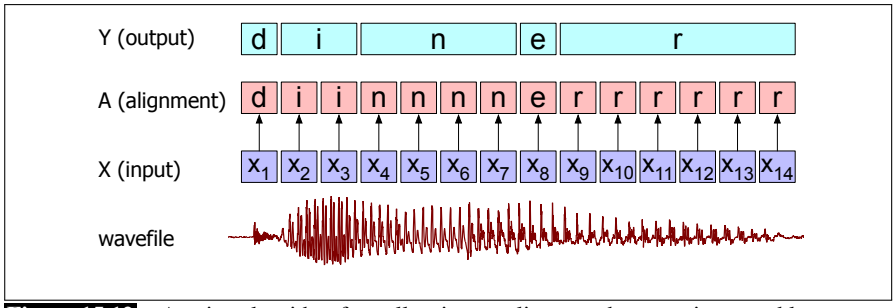


Figure 15.12 A naive algorithm for collapsing an alignment between input and letters.

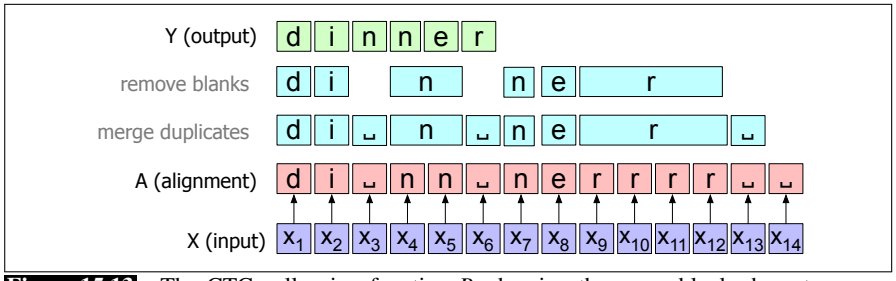


Figure 15.13 The CTC collapsing function B, showing the space blank character ␣; repeated (consecutive) characters in an alignment A are removed to form the output Y.

The CTC collapsing function is many-to-one; lots of different alignments map to the same output string. For example, the alignment shown in Fig. 15.13 is not the only alignment that results in the string *dinner*. Fig. 15.14 shows some other alignments that would produce the same output.

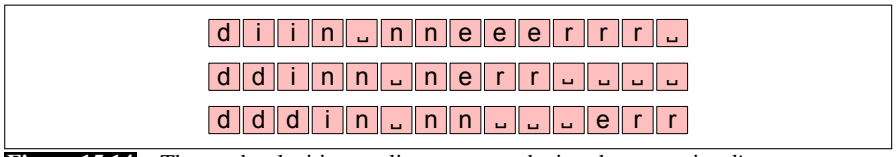


Figure 15.14 Three other legitimate alignments producing the transcript *dinner*.

It's useful to think of the set of all alignments that might produce the same output Y. We'll use the inverse of our B function, called B^{-1}, and represent that set as $B^{-1}(Y)$.

15.5.1 CTC Inference

Before we see how to compute $P_{\text{CTC}}(Y|\mathbf{X})$ let's first see how CTC assigns a probability to one particular alignment $\hat{A} = \{\hat{a}_1, \ldots, \hat{a}_n\}$. CTC makes a strong conditional independence assumption: it assumes that, given the input $\mathbf{X}$, the CTC model output a_t at time t is independent of the output labels at any other time a_i. Thus:

$$P_{\text{CTC}}(\mathbf{A}|\mathbf{X}) = \prod_{t=1}^{T} p(a_t|\mathbf{X}) \tag{15.16}$$

Thus to find the best alignment $\hat{A} = \{\hat{a}_1, \ldots, \hat{a}_T\}$ we can greedily choose the character with the max probability at each time step t:

$$\hat{a}_t = \underset{c \in C}{\text{argmax}}\, p_t(c|\mathbf{X}) \tag{15.17}$$

We then pass the resulting sequence A to the CTC collapsing function B to get the output sequence Y.

Let's talk about how this simple inference algorithm for finding the best alignment A would be implemented. Because we are making a decision at each time point, we can treat CTC as a sequence-modeling task, where we output one letter $\hat{\mathbf{y}}_t$ at time t corresponding to each input token $\mathbf{x}_t$, eliminating the need for a full decoder. Fig. 15.15 sketches this architecture, where we take an encoder, produce a hidden state $\mathbf{h}_t$ at each timestep, and decode by taking a softmax over the character vocabulary at each time step.

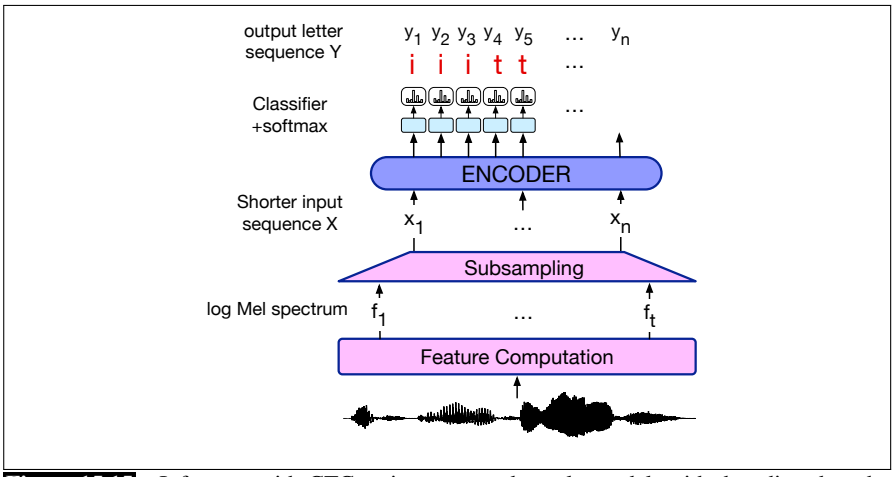


Figure 15.15 Inference with CTC: using an encoder-only model, with decoding done by simple softmaxes over the hidden state $\mathbf{h}_t$ at each output step.

Alas, there is a potential flaw with the inference algorithm sketched in (Eq. 15.17) and Fig. 15.14. The problem is that we chose the most likely alignment A, but the most likely alignment may not correspond to the most likely final collapsed output string Y. That's because there are many possible alignments that lead to the same output string, and hence the most likely output string might not correspond to the most probable alignment. For example, imagine the most probable alignment A for an input $\mathbf{X} = [x_1 x_2 x_3]$ is the string [a b ϵ] but the next two most probable alignments are [b ϵ b] and [ϵ b b]. The output Y =[b b], summing over those two alignments, might be more probable than Y =[a b].

For this reason, the most probable output sequence Y is the one that has, not the single best CTC alignment, but the highest sum over the probability of all its possible alignments:

$$\begin{aligned} P_{CTC}(Y|\mathbf{X}) &= \sum_{A \in B^{-1}(Y)} P(A|\mathbf{X}) \\ &= \sum_{A \in B^{-1}(Y)} \prod_{t=1}^{T} p(a_t|h_t) \\ \hat{Y} &= \operatorname*{argmax}_{Y} P_{CTC}(Y|\mathbf{X}) \end{aligned} \tag{15.18}$$

Alas, summing over all alignments is very expensive (there are a lot of alignments), so we approximate this sum by using a version of Viterbi beam search that cleverly keeps in the beam the high-probability alignments that map to the same output string,

and sums those as an approximation of (Eq. 15.18). See Hannun (2017) for a clear explanation of this extension of beam search for CTC.

Because of the strong conditional independence assumption mentioned earlier (that the output at time t is independent of the output at time $t-1$, given the input), CTC does not implicitly learn a language model over the data (unlike the attention-based encoder-decoder architectures). It is therefore essential when using CTC to interpolate a language model (and some sort of length factor $L(Y)$) using interpolation weights that are trained on a devset:

$$score_{\text{CTC}}(Y|\mathbf{X}) = \log P_{\text{CTC}}(Y|\mathbf{X}) + \lambda_1 \log P_{LM}(Y)\lambda_2 L(Y) \tag{15.19}$$

15.5.2 CTC Training

To train a CTC-based ASR system, we use negative log-likelihood loss with a special CTC loss function. Thus the loss for an entire dataset D is the sum of the negative log-likelihoods of the correct output Y for each input $\mathbf{X}$:

$$L_{CTC} = \sum_{(X,Y)\in D} -\log P_{\text{CTC}}(Y|\mathbf{X}) \tag{15.20}$$

To compute CTC loss function for a single input pair $(\mathbf{X}, Y)$, we need the probability of the output Y given the input $\mathbf{X}$. As we saw in Eq. 15.18, to compute the probability of a given output Y we need to sum over all the possible alignments that would collapse to Y. In other words:

$$P_{\text{CTC}}(Y|\mathbf{X}) = \sum_{A\in B^{-1}(Y)} \prod_{t=1}^{T} p(a_t|h_t) \tag{15.21}$$

Naively summing over all possible alignments is not feasible (there are too many alignments). However, we can efficiently compute the sum by using dynamic programming to merge alignments, with a version of the **forward-backward algorithm** also used to train HMMs (Appendix A) and CRFs. The original dynamic programming algorithms for both training and inference are laid out in (Graves et al., 2006); see (Hannun, 2017) for a detailed explanation of both.

15.5.3 Combining CTC and Encoder-Decoder

It's also possible to combine the two architectures/loss functions we've described, the cross-entropy loss from the encoder-decoder architecture, and the CTC loss. Fig. 15.16 shows a sketch. For training, we can simply weight the two losses with a λ tuned on a devset:

$$L = -\lambda \log P_{encdec}(Y|\mathbf{X}) - (1-\lambda)\log P_{ctc}(Y|\mathbf{X}) \tag{15.22}$$

For inference, we can combine the two with the language model (or the length penalty), again with learned weights:

$$\hat{Y} = \underset{Y}{\text{argmax}} \left[\lambda \log P_{encdec}(Y|\mathbf{X}) - (1-\lambda)\log P_{CTC}(Y|\mathbf{X}) + \gamma \log P_{LM}(Y)\right] \tag{15.23}$$

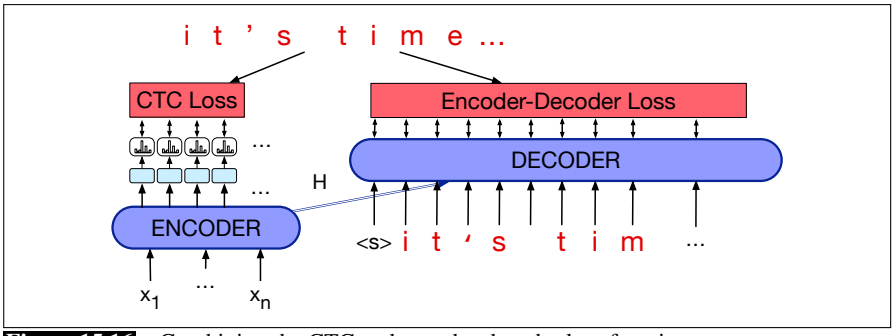


Figure 15.16 Combining the CTC and encoder-decoder loss functions.

15.5.4 Streaming Models: RNN-T for improving CTC

Because of the strong independence assumption in CTC (assuming that the output at time t is independent of the output at time $t-1$), recognizers based on CTC don't achieve as high an accuracy as the attention-based encoder-decoder recognizers. CTC recognizers have the advantage, however, that they can be used for **streaming**. Streaming means recognizing words on-line rather than waiting until the end of the sentence to recognize them. Streaming is crucial for many applications, from commands to dictation, where we want to start recognition while the user is still talking. Algorithms that use attention need to compute the hidden state sequence over the entire input first in order to provide the attention distribution context, before the decoder can start decoding. By contrast, a CTC algorithm can input letters from left to right immediately.

streaming

If we want to do streaming, we need a way to improve CTC recognition to remove the conditional independent assumption, enabling it to know about output history. The RNN-Transducer (**RNN-T**), shown in Fig. 15.17, is just such a model (Graves 2012, Graves et al. 2013). The RNN-T has two main components: a CTC acoustic model, and a separate language model component called the **predictor** that conditions on the output token history. At each time step t, the CTC encoder outputs a hidden state h_t^{enc} given the input $x_1 ... x_t$. The language model predictor takes as input the previous output token (not counting blanks), outputting a hidden state h_u^{pred}. The two are passed through another network whose output is then passed through a softmax to predict the next character.

RNN-T

$$P_{RNN-T}(Y|\mathbf{X}) = \sum_{A \in B^{-1}(Y)} P(A|\mathbf{X})$$

$$= \sum_{A \in B^{-1}(Y)} \prod_{t=1}^{T} p(a_t|h_t, y_{<u_t})$$

15.6 ASR Evaluation: Word Error Rate

word error

The standard evaluation metric for speech recognition systems is the **word error** rate. The word error rate is based on how much the word string returned by the recognizer (the **hypothesized** word string) differs from a **reference** transcription. The first step in computing word error is to compute the **minimum edit distance** in

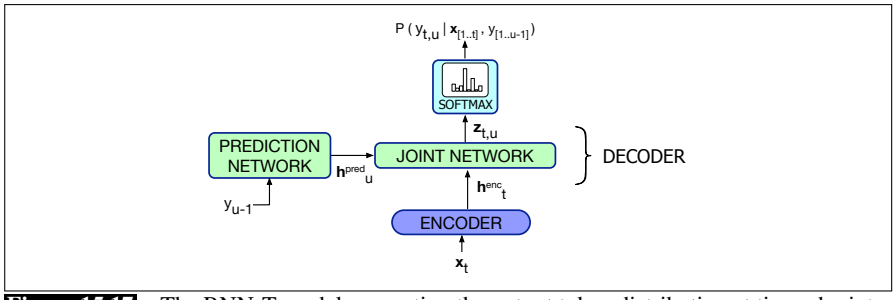


Figure 15.17 The RNN-T model computing the output token distribution at time t by integrating the output of a CTC acoustic encoder and a separate 'predictor' language model.

words between the hypothesized and correct strings, giving us the minimum number of word **substitutions**, word **insertions**, and word **deletions** necessary to map between the correct and hypothesized strings. The word error rate (WER) is then defined as follows (note that because the equation includes insertions, the error rate can be greater than 100%):

$$\text{Word Error Rate} = 100 \times \frac{\text{Insertions} + \text{Substitutions} + \text{Deletions}}{\text{Total Words in Correct Transcript}}$$

alignment

Here is a sample **alignment** between a reference and a hypothesis utterance from the CallHome corpus, showing the counts used to compute the error rate:

REF:	i	***	**	UM	the	PHONE	IS		i	LEFT	THE	portable	****	PHONE	UPSTAIRS	last night
HYP:	i	GOT	IT	TO	the	*****	FULLEST		i	LOVE	TO	portable	FORM	OF	STORES	last night
Eval:		I	I	S		D	S			S	S		I	S	S	

This utterance has six substitutions, three insertions, and one deletion:

$$\text{Word Error Rate} = 100\frac{6+3+1}{13} = 76.9\%$$

The standard method for computing word error rates is a free script called **sclite**, available from the National Institute of Standards and Technologies (NIST) (NIST, 2005). Sclite is given a series of reference (hand-transcribed, gold-standard) sentences and a matching set of hypothesis sentences. Besides performing alignments, and computing word error rate, sclite performs a number of other useful tasks. For example, for **error analysis** it gives useful information such as confusion matrices showing which words are often misrecognized for others, and summarizes statistics of words that are often inserted or deleted. `sclite` also gives error rates by speaker (if sentences are labeled for speaker ID), as well as useful statistics like the **sentence error rate**, the percentage of sentences with at least one word error.

Sentence error rate

Text normalization before evaluation

It's normal for systems to normalize text before computing word error rate. There are a variety of packages for implementing normalization rules. For example some standard English normalization rules include:

1. Removing metalanguage [non-language, notes, transcription comments] that occur between matching brackets ([,])
2. Remove or standardize interjections or filled pauses ("uh", "um", "err")
3. Standardize contracted and non-contracted forms of English ("I'm"/"I am")

4. Normalize non-standard-words (number, quantities, dates, times) [e.g., "$100 → "One hundred dollars"]
5. Unify US and UK spelling conventions

Statistical significance for ASR: MAPSSWE or MacNemar

As with other language processing algorithms, we need to know whether a particular improvement in word error rate is significant or not.

The standard statistical tests for determining if two word error rates are different is the Matched-Pair Sentence Segment Word Error (MAPSSWE) test, introduced in Gillick and Cox (1989).

The MAPSSWE test is a parametric test that looks at the difference between the number of word errors the two systems produce, averaged across a number of segments. The segments may be quite short or as long as an entire utterance; in general, we want to have the largest number of (short) segments in order to justify the normality assumption and to maximize power. The test requires that the errors in one segment be statistically independent of the errors in another segment. Since ASR systems tend to use trigram LMs, we can approximate this requirement by defining a segment as a region bounded on both sides by words that both recognizers get correct (or by turn/utterance boundaries). Here's an example from NIST (2007) with four regions:

```
         I            II             III               IV
REF:  |it was|the best|of|times it|was the worst|of times|  |it was
      |      |        |  |        |            |        |  |
SYS A:|ITS   |the best|of|times it|IS the worst |of times|OR|it was
      |      |        |  |        |            |        |  |
SYS B:|it was|the best|  |times it|WON the TEST |of times|  |it was
```

In region I, system A has two errors (a deletion and an insertion) and system B has zero; in region III, system A has one error (a substitution) and system B has two. Let's define a sequence of variables Z representing the difference between the errors in the two systems as follows:

N_A^i	the number of errors made on segment i by system A
N_B^i	the number of errors made on segment i by system B
Z	$N_A^i - N_B^i, i = 1, 2, \cdots, n$ where n is the number of segments

In the example above, the sequence of Z values is $\{2, -1, -1, 1\}$. Intuitively, if the two systems are identical, we would expect the average difference, that is, the average of the Z values, to be zero. If we call the true average of the differences mu_z, we would thus like to know whether $mu_z = 0$. Following closely the original proposal and notation of Gillick and Cox (1989), we can estimate the true average from our limited sample as $\hat{\mu}_z = \sum_{i=1}^{n} Z_i/n$. The estimate of the variance of the Z_i's is

$$\sigma_z^2 = \frac{1}{n-1}\sum_{i=1}^{n}(Z_i - \mu_z)^2 \tag{15.24}$$

Let

$$W = \frac{\hat{\mu}_z}{\sigma_z/\sqrt{n}} \tag{15.25}$$

For a large enough n (> 50), W will approximately have a normal distribution with unit variance. The null hypothesis is $H_0 : \mu_z = 0$, and it can thus be rejected if

$2 * P(Z \geq |w|) \leq 0.05$ (two-tailed) or $P(Z \geq |w|) \leq 0.05$ (one-tailed), where Z is standard normal and w is the realized value W; these probabilities can be looked up in the standard tables of the normal distribution.

McNemar's test

Earlier work sometimes used **McNemar's test** for significance, but McNemar's is only applicable when the errors made by the system are independent, which is not true in continuous speech recognition, where errors made on a word are extremely dependent on errors made on neighboring words.

Could we improve on word error rate as a metric? It would be nice, for example, to have something that didn't give equal weight to every word, perhaps valuing content words like *Tuesday* more than function words like *a* or *of*. While researchers generally agree that this would be a good idea, it has proved difficult to agree on a metric that works in every application of ASR.

15.7 Summary

This chapter introduced the fundamental algorithms of automatic speech recognition (ASR).

- The task of **speech recognition** (or speech-to-text) is to map acoustic waveforms to sequences of graphemes.
- The input to a speech recognizer is a series of acoustic waves. that are **sampled**, **quantized**, and converted to a **spectral representation** like the **log mel spectrum**.
- Two common paradigms for speech recognition are the **encoder-decoder with attention** model, and models based on the **CTC loss function**. Attention-based models have higher accuracies, but models based on CTC more easily adapt to **streaming**: outputting graphemes online instead of waiting until the acoustic input is complete.
- ASR is evaluated using the Word Error Rate; the edit distance between the hypothesis and the gold transcription.

Historical Notes

A number of speech recognition systems were developed by the late 1940s and early 1950s. An early Bell Labs system could recognize any of the 10 digits from a single speaker (Davis et al., 1952). This system had 10 speaker-dependent stored patterns, one for each digit, each of which roughly represented the first two vowel formants in the digit. They achieved 97%–99% accuracy by choosing the pattern that had the highest relative correlation coefficient with the input. Fry (1959) and Denes (1959) built a phoneme recognizer at University College, London, that recognized four vowels and nine consonants based on a similar pattern-recognition principle. Fry and Denes's system was the first to use phoneme transition probabilities to constrain the recognizer.

The late 1960s and early 1970s produced a number of important paradigm shifts. First were a number of feature-extraction algorithms, including the efficient fast Fourier transform (FFT) (Cooley and Tukey, 1965), the application of cepstral processing to speech (Oppenheim et al., 1968), and the development of LPC for speech coding (Atal and Hanauer, 1971). Second were a number of ways of handling **warp-**

warping

ing; stretching or shrinking the input signal to handle differences in speaking rate and segment length when matching against stored patterns. The natural algorithm for solving this problem was dynamic programming, and, as we saw in Appendix A, the algorithm was reinvented multiple times to address this problem. The first application to speech processing was by Vintsyuk (1968), although his result was not picked up by other researchers, and was reinvented by Velichko and Zagoruyko (1970) and Sakoe and Chiba (1971) (and 1984). Soon afterward, Itakura (1975) combined this dynamic programming idea with the LPC coefficients that had previously been used only for speech coding. The resulting system extracted LPC features from incoming words and used dynamic programming to match them against stored LPC templates. The non-probabilistic use of dynamic programming to match a template against incoming speech is called **dynamic time warping**.

dynamic time warping

The third innovation of this period was the rise of the HMM. Hidden Markov models seem to have been applied to speech independently at two laboratories around 1972. One application arose from the work of statisticians, in particular Baum and colleagues at the Institute for Defense Analyses in Princeton who applied HMMs to various prediction problems (Baum and Petrie 1966, Baum and Eagon 1967). James Baker learned of this work and applied the algorithm to speech processing (Baker, 1975a) during his graduate work at CMU. Independently, Frederick Jelinek and collaborators (drawing from their research in information-theoretical models influenced by the work of Shannon (1948)) applied HMMs to speech at the IBM Thomas J. Watson Research Center (Jelinek et al., 1975). One early difference was the decoding algorithm; Baker's DRAGON system used Viterbi (dynamic programming) decoding, while the IBM system applied Jelinek's stack decoding algorithm (Jelinek, 1969). Baker then joined the IBM group for a brief time before founding the speech-recognition company Dragon Systems.

The use of the HMM, with Gaussian Mixture Models (GMMs) as the phonetic component, slowly spread through the speech community, becoming the dominant paradigm by the 1990s. One cause was encouragement by ARPA, the Advanced Research Projects Agency of the U.S. Department of Defense. ARPA started a five-year program in 1971 to build 1000-word, constrained grammar, few speaker speech understanding (Klatt, 1977), and funded four competing systems of which Carnegie-Mellon University's Harpy system (Lowerre, 1976), which used a simplified version of Baker's HMM-based DRAGON system was the best of the tested systems. ARPA (and then DARPA) funded a number of new speech research programs, beginning with 1000-word speaker-independent read-speech tasks like "Resource Management" (Price et al., 1988), recognition of sentences read from the *Wall Street Journal* (WSJ), Broadcast News domain (LDC 1998, Graff 1997) (transcription of actual news broadcasts, including quite difficult passages such as on-the-street interviews) and the Switchboard, CallHome, CallFriend, and Fisher domains (Godfrey et al. 1992, Cieri et al. 2004) (natural telephone conversations between friends or strangers). Each of the ARPA tasks involved an approximately annual **bakeoff** at which systems were evaluated against each other. The ARPA competitions resulted in wide-scale borrowing of techniques among labs since it was easy to see which ideas reduced errors the previous year, and the competitions were probably an important factor in the eventual spread of the HMM paradigm.

bakeoff

By around 1990 neural alternatives to the HMM/GMM architecture for ASR arose, based on a number of earlier experiments with neural networks for phoneme recognition and other speech tasks. Architectures included the time-delay neural network (**TDNN**)—the first use of convolutional networks for speech— (Waibel

hybrid

et al. 1989, Lang et al. 1990), RNNs (Robinson and Fallside, 1991), and the **hybrid** HMM/MLP architecture in which a feedforward neural network is trained as a phonetic classifier whose outputs are used as probability estimates for an HMM-based architecture (Morgan and Bourlard 1990, Bourlard and Morgan 1994, Morgan and Bourlard 1995).

While the hybrid systems showed performance close to the standard HMM/GMM models, the problem was speed: large hybrid models were too slow to train on the CPUs of that era. For example, the largest hybrid system, a feedforward network, was limited to a hidden layer of 4000 units, producing probabilities over only a few dozen monophones. Yet training this model still required the research group to design special hardware boards to do vector processing (Morgan and Bourlard, 1995). A later analytic study showed the performance of such simple feedforward MLPs for ASR increases sharply with more than 1 hidden layer, even controlling for the total number of parameters (Maas et al., 2017). But the computational resources of the time were insufficient for more layers.

Over the next two decades a combination of Moore's law and the rise of GPUs allowed deep neural networks with many layers. Performance was getting close to traditional systems on smaller tasks like TIMIT phone recognition by 2009 (Mohamed et al., 2009), and by 2012, the performance of hybrid systems had surpassed traditional HMM/GMM systems (Jaitly et al. 2012, Dahl et al. 2012, inter alia). Originally it seemed that unsupervised pretraining of the networks using a technique like deep belief networks was important, but by 2013, it was clear that for hybrid HMM/GMM feedforward networks, all that mattered was to use a lot of data and enough layers, although a few other components did improve performance: using log mel features instead of MFCCs, using dropout, and using rectified linear units (Deng et al. 2013, Maas et al. 2013, Dahl et al. 2013).

Meanwhile early work had proposed the CTC loss function by 2006 (Graves et al., 2006), and by 2012 the RNN-Transducer was defined and applied to phone recognition (Graves 2012, Graves et al. 2013), and then to end-to-end speech recognition rescoring (Graves and Jaitly, 2014), and then recognition (Maas et al., 2015), with advances such as specialized beam search (Hannun et al., 2014). (Our description of CTC in the chapter draws on Hannun (2017), which we encourage the interested reader to follow).

The encoder-decoder architecture was applied to speech at about the same time by two different groups, in the Listen Attend and Spell system of Chan et al. (2016) and the attention-based encoder decoder architecture of Chorowski et al. (2014) and Bahdanau et al. (2016). By 2018 Transformers were included in this encoder-decoder architecture. Karita et al. (2019) is a nice comparison of RNNs vs Transformers in encoder-architectures for ASR, TTS, and speech-to-speech translation.

Kaldi
ESPnet

Popular toolkits for speech processing include **Kaldi** (Povey et al., 2011) and **ESPnet** (Watanabe et al. 2018, Hayashi et al. 2020).

Exercises

CHAPTER

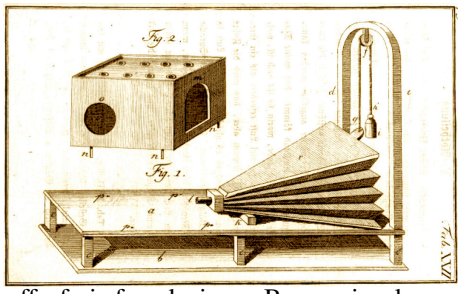
16 Text-to-Speech

"Words mean more than what is set down on paper. It takes the human voice to infuse them with shades of deeper meaning."

Maya Angelou, *I Know Why the Caged Bird Sings*

The task of mapping from text to speech is a task with an even longer history than speech to text. In Vienna in 1769, Wolfgang von Kempelen built for the Empress Maria Theresa the famous Mechanical Turk, a chess-playing automaton consisting of a wooden box filled with gears, behind which sat a robot mannequin who played chess by moving pieces with his mechanical arm. The Turk toured Europe and the Americas for decades, defeating Napoleon Bonaparte and even playing Charles Babbage. The Mechanical Turk might have been one of the early successes of artificial intelligence were it not for the fact that it was, alas, a hoax, powered by a human chess player hidden inside the box.

What is less well known is that von Kempelen, an extraordinarily prolific inventor, also built between 1769 and 1790 what was definitely not a hoax: the first full-sentence speech synthesizer, shown partially to the right. His device consisted of a bellows to simulate the lungs, a rubber mouthpiece and a nose aperture, a reed to simulate the vocal folds, various whistles for the fricatives, and a small auxiliary bellows to provide the puff of air for plosives. By moving levers with both hands to open and close apertures, and adjusting the flexible leather "vocal tract", an operator could produce different consonants and vowels.

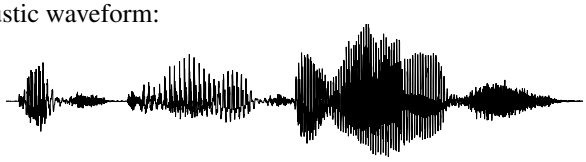


More than two centuries later, we no longer build our synthesizers out of wood and leather, nor do we need human operators. The modern task of **text-to-speech** or **TTS**, also called **speech synthesis**, is exactly the reverse of ASR; to map text:

text-to-speech
TTS
speech synthesis

```
It's time for lunch!
```

to an acoustic waveform:



TTS has a wide variety of applications. It is used in spoken language models that interact with people, for reading text out loud, for games, and to produce speech for sufferers of neurological disorders, like the late astrophysicist Steven Hawking after he lost the use of his voice because of ALS.

In this chapter we introduce an algorithm for TTS that, like the ASR algorithms of the prior chapter, are trained on enormous amounts of speech datasets. We'll also briefly touch on other speech applications.

16.1 TTS overview

The task of text-to-speech is to generate a speech waveform that corresponds to a desired text, using in a particular voice specified by the user.

Historically TTS was done by collecting hundreds of hours of speech from a single talker in a lab and training a large system on it. The resulting TTS system only worked in one voice; if you wanted a second voice, you went back and collected data from a second talker.

The modern method is instead to train a speaker-independent synthesizer on tens of thousands of hours of speech from thousands of talkers. To create speech in a new voice unseen in training, we use a very small amount of speech from the desired talker to guide the creation of the voice. So the input to a modern TTS system is a text prompt and perhaps 3 seconds of speech from the voice we'd like to generate the speech in. This TTS task is called **zero-shot TTS** because the desired voice may never have been seen in training.

zero-shot TTS

The way modern TTS systems address this task is to use language modeling, and in particular conditional generation. The intuition is to take an enormous dataset of speech, and use an **audio tokenizer** based on an **audio codec** to induce discrete audio tokens from that speech that represent the speech. Then we can train a language model whose vocabulary includes both speech tokens and text tokens.

We train this language model to take as input two sequences, a text transcript and a small sample of speech from the desired talker, to tokenize both the text and the speech into discrete tokens, and then to conditionally generate discrete samples of the speech corresponding to the text string, in the desired voice.

At inference time we prompt this language model with a tokenized text string and a sample of the desired voice (tokenized by the codec into discrete audio tokens) and conditionally generate to produce the desired audio tokens. Then these tokens can be converted into a waveform.

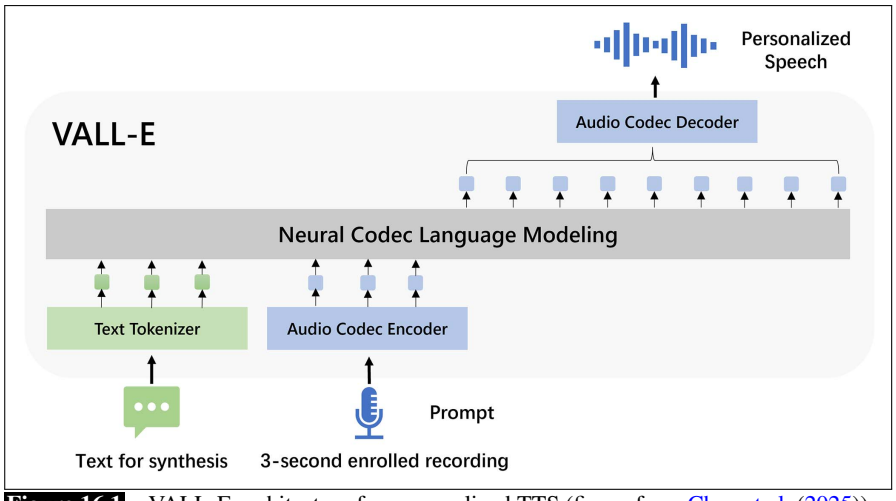


Figure 16.1 VALL-E architecture for personalized TTS (figure from Chen et al. (2025)).

Fig. 16.1 from Chen et al. (2025) shows the intuition for one such TTS system, called **VALL-E**. VALL-E is trained on 60K hours of English speech, from over 7000 unique talkers. Systems like VALL-E have 2 components:

VALL-E

1. The **audio tokenizer**, generally based on an **audio codec**, a system we'll de-

scribe in the next section. Codecs have three parts: an encoder (that turns speech into embedding vectors), a quantizer (that turns the embeddings into discrete tokens) and decoders (that turns the discrete tokens back into speech).

2. The 2-stage **conditional language model** that can generate audio tokens corresponding to the desired text. We'll sketch this in Section 16.3.

16.2 Using a codec to learn discrete audio tokens

Modern TTS systems are based around converting the waveform into a sequence of discrete audio tokens. This idea of manipulating discrete audio tokens is also useful for other speech-enabled systems like **spoken language models**, which take text or speech input and can generate text or speech output to solve tasks like speech-to-speech translation, diarization, or spoken question answering. Having discrete tokens means that we can make use of language model technology, since language models are specialized for sequences of discrete tokens. Audio tokenizers are thus an important component of the modern speech toolkit.

codec

The standard way to learn audio tokens is from a neural **audio codec** (the word is formed from **coder/decoder**). Historically a codec was a hardware device that digitized analog symbols. More generally we use the word to mean a mechanism for encoding analog speech signals into a digitized compressed representation that can be efficiently stored and sent. Codecs are still used for compression, but for TTS and also for spoken language models, we employ them for converting speech into discrete tokens.

Of course the digital representation of speech we described in Chapter 14 is already discrete. For example 16 kHz speech stored in 16-bit format could be thought of as a series of $2^{16} = 65{,}536$ symbols, with 16,000 of those symbols per second of speech. But a system that generates 16,000 symbols per second makes the speech signal too long to be feasibly processed by a language model, especially one based on transformers with their inefficient quadratic attention. Instead we want symbols that represent longer chunks of speech, perhaps something on the order of a few hundred tokens a second.

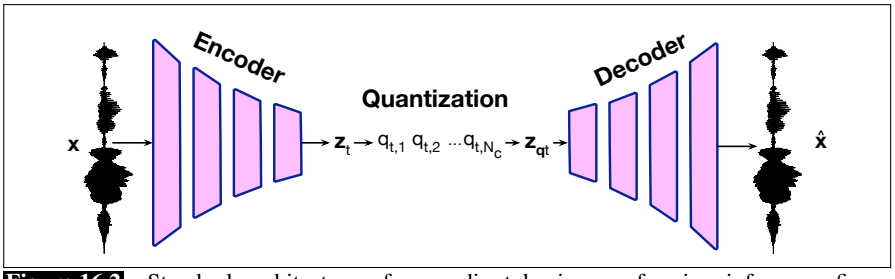


Figure 16.2 Standard architecture of an audio tokenizer performing inference, figure adapted from Mousavi et al. (2025). An input waveform $\mathbf{x}$ is encoded (generally using a series of downsampling convolution networks) into a series of embeddings $\mathbf{z_t}$. Each embedding is then passed through a quantizer to produce a series of quantized tokens q_t. To regenerate the speech signal, the quantized tokens are re-mapped back to a vector $\mathbf{z_{q_t}}$ and then encoded (usually using a series of upsampling convolution networks) back to a waveform. We'll discuss how the architecture is trained in Section 16.2.4.

Fig. 16.2 adapted from Mousavi et al. (2025). shows the standard architecture of an audio tokenizer. Audio tokenizers take as input an audio waveform, and are

trained to recreate the same audio waveform out, via an intermediate representation consisting of discrete tokens created by vector quantization.

Audio tokenizers have three stages:

1. an **encoder** maps the acoustic waveform, a series of T values $\mathbf{x} = x_1, x_2, ..., x_T$, to a sequence of τ embeddings $\mathbf{z} = \mathbf{z_1}, \mathbf{z_2}, ..., \mathbf{z_\tau}$. τ is typically 100-1000 times smaller than T.
2. a **vector quantizer** that takes each embedding $\mathbf{z_t}$ corresponding to part of the waveform, and represents it by a sequence of discrete tokens each taken from one of the N_c codebooks, $q_t = q_{t,1}, q_{t,2}, ..., q_{t,N_c}$. The vector quantizer also sums the vector codewords from each codebook to create a quantizer output vector $\mathbf{z_{q_t}}$.
3. a **decoder** that generates a lossy reconstructed waveform span $\hat{\mathbf{x}}$ from the quantizer output vector $\mathbf{z_{q_t}}$.

Audio tokenizers are generally learned end-to-end, using loss functions that reward a tokenization that allows the system to reconstruct the input waveform.

In the following subsections we'll go through the components of one particular tokenizer, the ENCODEC tokenizer of Défossez et al. (2023).

16.2.1 The Encoder and Decoder for the ENCODEC model

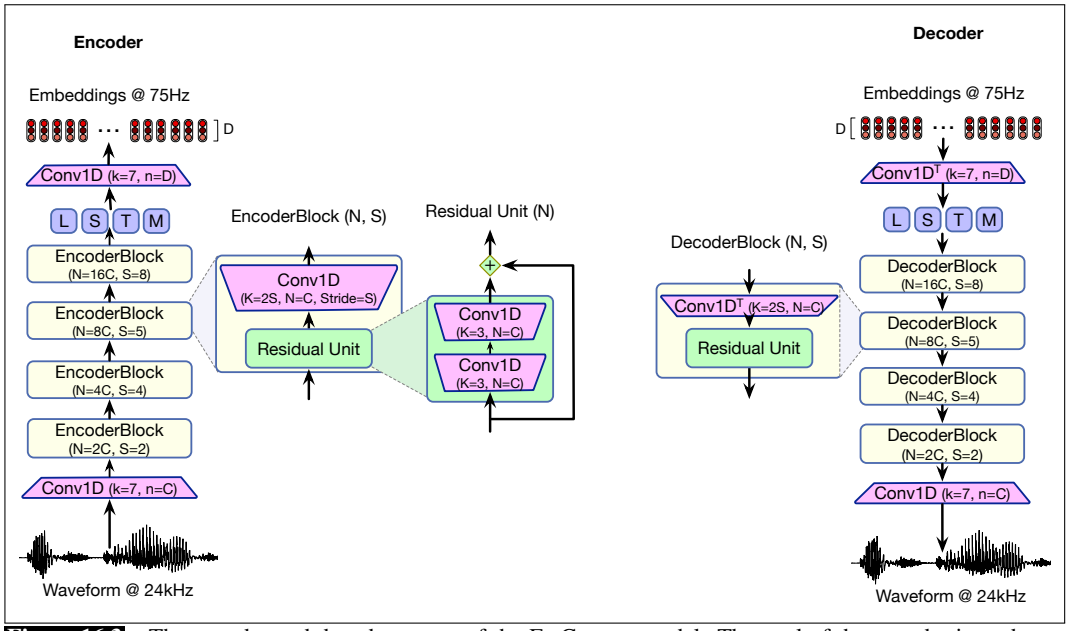


Figure 16.3 The encoder and decoder stages of the ENCODEC model. The goal of the encoder is to downsample an input waveform by encoded it as a series of embeddings $\mathbf{z}_t$ at 75Hz, i.e. 75 embeddings a second. Because the original signal was represented at 24kHz, this is a downsampling of $\frac{24000}{75} = 320$ times. Between the encoder and decoder is a quantization step producing a lossy embedding $\mathbf{z_{q_t}}$. The goal of the decoder is to take the lossy embedding $\mathbf{z}_q t$ and upsample it, converting it back to a waveform.

The encoder and decoder of the ENCODEC model (Défossez et al., 2023) are sketched in Fig. 16.3. The goal of the encoder is to downsample a span of waveform at time t, which is at 24kHz—one second of speech has 24,000 real values—to an embedding representation $\mathbf{z}_t$ at 75Hz—one second of audio is represented by 75

vectors, each of dimensionality D. For the purposes of this explanation, we'll use $D = 256$.

This downsampling is accomplished by having a series of encoder blocks that are made up of convolutional layers with strides larger than 1 that iteratively downsample the audio, as we discussed at the end of Section 15.2. The convolution blocks are sketched in Fig. 16.3, and include a long series of convolutions as well as residual units that add a convolution to the prior input.

The output of the encoder is an embedding $\mathbf{z}_t$ at time t, 75 of which are produced per second. This embedding is then quantized (as discussed in the next section), turning each embedding $\mathbf{z}_t$ into a series of N_c discrete symbols $q_t = q_{t,1}, q_{t,2}, ..., q_{t,N_c}$, and also turning the series of symbols into a new quantizer output vector $\mathbf{z}_{\mathbf{q}_t}$. Finally, the decoder takes the output embedding from the quantizer $\mathbf{z}_{\mathbf{q}_t}$ and generates a waveform via a symmetric set of convnets that upsample the audio.

In summary, a 24kHz waveform comes through, we encode/downsample it into a vector $\mathbf{z}_t$ of dimensionality $D = 256$, quantize it into discrete symbols q_t, turn it back into a vector $\mathbf{z}_{\mathbf{q}_t}$ of dimensionality $D = 256$, and then decode/upsample that vector back into a waveform at 24kHz.

16.2.2 Vector Quantization

vector quantization
VQ

The goal of the **vector quantization** or **VQ** step is to turn a series of vectors into a series of discrete symbols.

Historically vector quantization (Gray, 1984) was used to compress a speech signal, to reduce the bit rate for transmission or storage. To compress a sequence of vector representations of speech, we turn each vector into an integer, an index representing a class or cluster. Then instead of transmitting a big vector of floating point numbers, we transmit that integer index. At the other end of the transmission, we reconstitute the vector from the index.

For TTS and other modern speech applications we use vector quantization for a different reason: because VQ conveniently creates discrete tokens, and those fit well into the language modeling paradigm, since language models do well at predicting sequences of discrete tokens.

In practice for the ENCODEC model and other audio tokenizers, we use a powerful form of vector quantization called **residual vector quantization** that we'll define in the following section. But it will be helpful to first see the basic VQ algorithm before we extend it.

Vector quantization has a training phase and an inference phase. We already introduced the core of the basic VQ training algorithm when we described k-means clustering of vectors in Section 15.4.3, since k-means clustering is the most common algorithm used to implement VQ. To review, in VQ training, we run a big set of speech wavefiles through an encoder to generate N vectors, each one corresponding to some frame of speech. Then we cluster all these N vectors into k clusters; k is set by the designer as a parameter to the algorithm as the number of discrete symbols we want, generally with $k << N$. In the simplest VQ algorithm, we use the iterative k-means algorithm to learn the clusters. Recall from Section 15.4.3 that k-means is a two-step algorithm based on iteratively updating a set of k **centroid** vectors. A **centroid** is the geometric center of a set of a points in n-dimensional space.

centroid

The k-means algorithm for clustering starts by assigning a random vector to each cluster k. Then there are two iterative steps. In the **assignment** step, given a set of k current centroids and the entire dataset of vectors, each vector is assigned to the cluster whose codeword is the closest (by squared Euclidean distance). In the **re-**

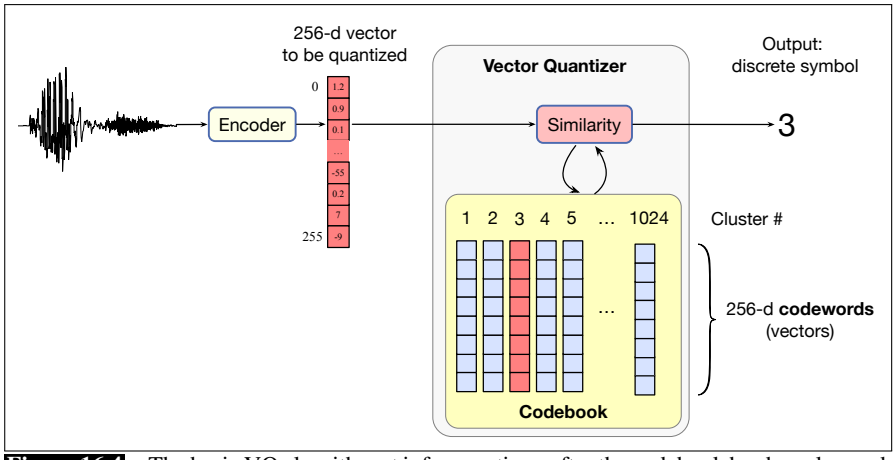


Figure 16.4 The basic VQ algorithm at inference time, after the codebook has been learned. The input is a span of speech encoded by the encoder into a vector of dimensionality $D = 256$. This vector is compared with each codeword (cluster centroid) in the codebook. The codeword for cluster 3 is most similar, so the VQ outputs 3 as the discrete representation of this vector.

estimation step, the codeword for each cluster is recomputed by recalculating a new mean vector. The result is that the clusters and their centroids slowly adjust to the training space. We iterate back and forth between these two steps until the algorithm converges.

VQ can also be used as part of end-to-end training, as we will discuss below, in which case instead of iterative k-means, we instead recompute the means during minibatch training via online algorithms like **exponential moving averages**.

codeword At the end of clustering, the cluster index can be used as a discrete symbol. Each cluster is also associated with a **codeword**, the vector which is the centroid of all the vectors in the cluster. We call the list of cluster ids (tokens) together with their **codebook** codeword the **codebook**, and we often call the cluster id the **code**.

code In inference, when a new vector comes in, we compare it to each vector in the codebook. Whichever codeword is closest, we assign it to that codeword's associated cluster. Fig. 16.4 shows an intuition of this inference step in the context of speech encoding:

1. an input speech waveform is encoded into a vector $\mathbf{v}$,
2. this input vector $\mathbf{v}$ is compared to each of the 1024 possible codewords in the codebook,
3. $\mathbf{v}$ is found to be most similar to codeword 3,
4. and so the output of VQ is the discrete symbol 3 as a representation of $\mathbf{v}$.

As we will see below, for training the ENCODEC model end-to-end we will need a way to turn this discrete symbol back into a waveform. For simple VQ we do that by directly using the codeword for that cluster, passing that codeword to the decoder for it to reconstruct the waveform. Of course the codeword vector won't exactly match the original vector encoding of the input speech span, especially with only 1024 possible codewords, but the hope is that it's at least close if our codebook is good, and the decoder will still produce reasonable speech. Nonetheless, more powerful methods are usually used, as we'll see in the next section.

16.2.3 Residual Vector Quantization

In practice, simple VQ doesn't produce good enough reconstructions, at least not with codebook sizes of 1024. 1024 codeword vectors just isn't enough to represent the wide variety of embeddings we get from encoding all possible speech waveforms. So what the ENCODEC model (and many other audio tokenization methods) use instead is a more sophisticated variant called **residual vector quantization**, or **RVQ**. In residual vector quantization, we use multiple codebooks arranged in a kind of hierarchy.

residual vector quantization
RVQ

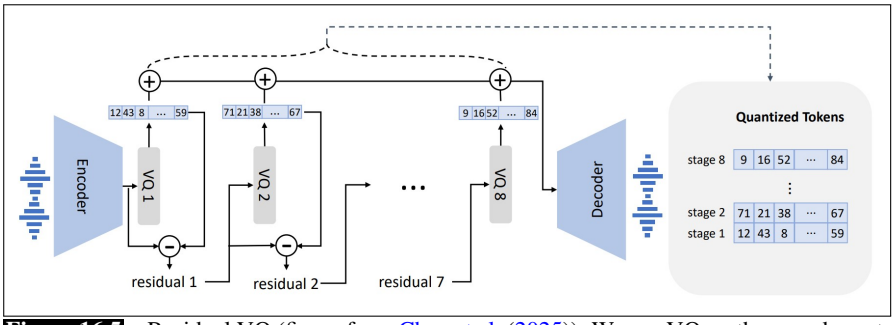


Figure 16.5 Residual VQ (figure from Chen et al. (2025)). We run VQ on the encoder output embedding to produce a discrete symbol and the corresponding codeword. We then look at the **residual**, the difference between the encoder output embedding z_t and the codeword chosen by VQ. We then take a second codebook and run VQ on this residual. We repeat the process until we have 8 tokens.

The idea is very simple. We run standard VQ with a codebook just as in Fig. 16.4 in the prior section. Then for an input embedding $\mathbf{z}_t$ we take the codeword vector that is produced, let's call it $\mathbf{z_{q1}}_t$ for the $\mathbf{z}_t$ as quantified by codebook 1, and take the difference between the two:

$$\text{residual} = \mathbf{z}_t - \mathbf{z}_{\mathbf{q}}^{(\mathbf{1})}{}_t. \tag{16.1}$$

residual

This **residual** is the error in the VQ; the part of the original vector that the VQ didn't capture. The residual is kind of a rounding error; it's as if in VQ we 'round' the vector to the nearest codeword, and that creates some error. So we then take that residual vector and pass it through another vector quantizer! That gives us a second codeword that represents the residual part of the vector. We then take the residual from the second codeword, and do this again. The total result is 8 codewords (the original codeword and the 7 residuals).

That means for RVQ we represent the original speech span by a sequence of 8 discrete symbols (instead of 1 discrete symbol in basic VQ). Fig. 16.5 shows the intuition.

What do we do when we want to reconstruct the speech? The method used in ENCODEC RVQ is again simple: we take the 8 codewords and add them together! The resulting vector $\mathbf{z_q}_t$ is then passed through the decoder to generate a waveform.

.

16.2.4 Training the ENCODEC model of audio tokens

The ENCODEC model (like similar audio tokenizer models) is trained end to end. The input is a waveform, a span of speech of perhaps 1 or 10 seconds extracted from a longer original waveform. The desired output is the same waveform span, since

the model is a kind of autoencoder that learns to map to itself. The model is trained to do this reconstruction on large speech datasets like Common Voice (Ardila et al., 2020) (over 30,000 hours of speech in 133 languages) as well as other audio data like Audio Set (Gemmeke et al., 2017) (1.7 million 10 sec excerpts from YouTube videos labeled from a large ontology including natural, animal, and machine sounds, music, and so on).

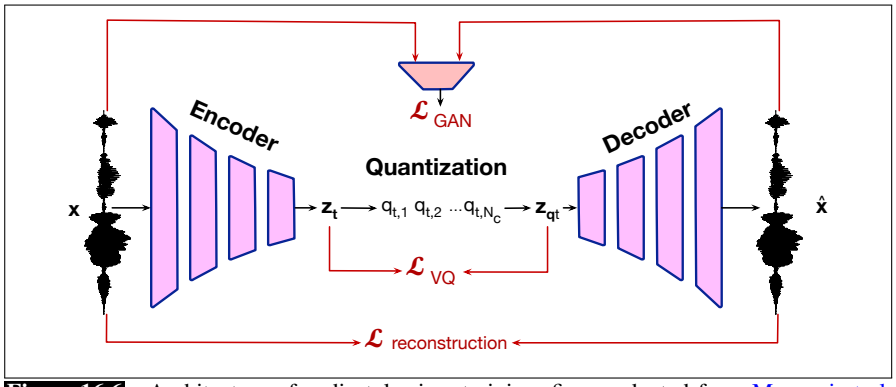


Figure 16.6 Architecture of audio tokenizer training, figure adapted from Mousavi et al. (2025). The audio tokenizer is trained with a weighted combination of various loss functions, summarized in the figure and described below.

reconstruction loss

The ENCODEC model, like most audio tokenizers, is trained with a number of loss functions, as suggested in Fig. 16.6. The **reconstruction loss** $L_{\text{reconstruction}}$ measures how similar the output waveform is to the input waveform, for example by the sum-squared difference between the original and reconstructed audio:

$$L_{\text{reconstruction}}(\mathbf{x},\hat{\mathbf{x}}) = \sum_{t=1}^{T} ||x_t - \hat{x}_t||^2 \tag{16.2}$$

Similarity can additionally be measured in the frequency domain, by comparing the original and reconstructed mel-spectrogram, again using sum-squared (L2) distance or L1 distance or some combination.

adversarial loss

Another kind of loss is the **adversarial loss** L_{GAN}. For this loss we train a generative adversarial network, a generator and a binary discriminator D, which is a classifier to distinguish between the true wavefile $\mathbf{x}$ and a generated one. We want to train the model to fool this discriminator, so the better the discriminator, the worse our reconstruction must be, and so we use the discriminator's success as a loss function, We can also incorporate various features from the generator.

Finally, we need a loss for the quantizer. This is because having a quantizer in the middle of end-to-end training causes problems in propagation of the gradient in the backward pass of training, because the quantization step is not differentiable. We deal with this problem in two ways. First, we ignore the quantization step in the backward pass. Instead we copy the gradients from the output of the quantizer ($\mathbf{z}_{\mathbf{q}_t}$) back to the input of the quantizer ($\mathbf{z}_t$), a method called the **straight-through estimator** (Van Den Oord et al., 2017).

But then we need a method to make sure the code words in the vector quantizer step get updated during training. One method is to start these off using k-means clustering of the vectors $\mathbf{z_t}$ to get an initial clustering. Then we can add to a loss component, L_{VQ}, which will be a function of the difference between the encoder

output vector $\mathbf{z}_t$ and the reconstructed vector after the quantization $\mathbf{z_q}_t$, i.e. the codeword, summed over all the N_c codebooks and residuals.

$$L_{\text{VQ}}(\mathbf{x},\hat{\mathbf{x}}) = \sum_{t=1}^{T}\sum_{c=1}^{N_c} ||\mathbf{z}^{(\mathbf{c})}{}_t - \mathbf{z}^{(\mathbf{c})}_{\mathbf{q}}{}_t|| \tag{16.3}$$

The total loss function can then just be a weighted sum of these losses:

$$L(\mathbf{x},\hat{\mathbf{x}}) = \lambda_1 L_{\text{reconstruction}}(\mathbf{x},\hat{\mathbf{x}}) + \lambda_2 L_{\text{GAN}}(\mathbf{x},\hat{\mathbf{x}}) + \lambda_3 L_{\text{VQ}}(\mathbf{x},\hat{\mathbf{x}}) \tag{16.4}$$

16.3 VALL-E: Generating audio with 2-stage LM

As we summarized in the introduction, the structure of TTS systems like VALL-E is to take as input a text to be synthesized and a sample of the voice to be used, and tokenize both, using BPE for the text and an audio codec for the speech. We then use a language model to conditionally generate discrete audio tokens corresponding to the text prompt, in the voice of the speech sample.

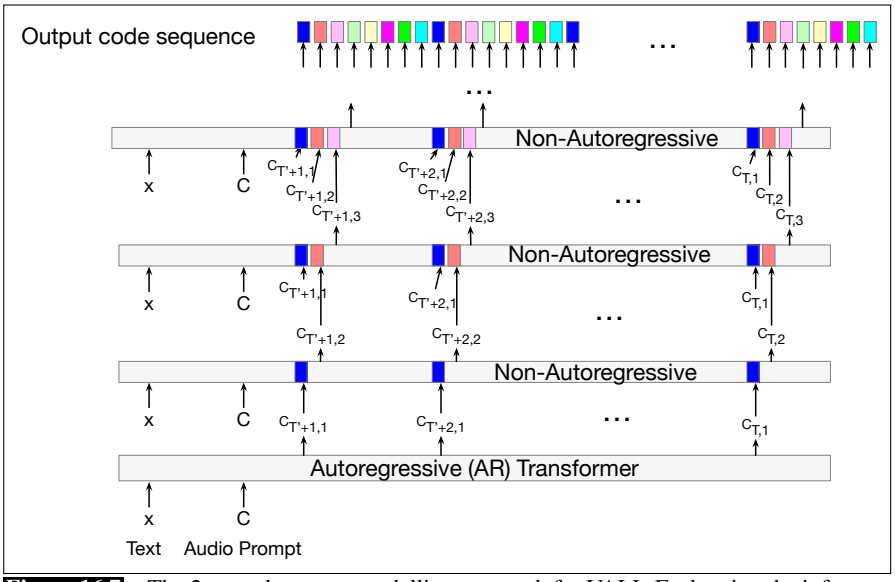


Figure 16.7 The 2-stage language modelling approach for VALL-E, showing the inference stage for the autoregressive transformer and the first 3 of the 7 non-autoregressive transformers. The output sequence of discrete audio codes is generated in two stages. First the autoregressive LM generates all the codes for the first quantizer from left to right. Then the non-autoregressive model is called 7 times to generate the remaining codes conditioned on all the codes from the preceding quantizer, including conditioning on the codes to the right.

Instead of doing this conditional generation with a single autoregressive language model, VALL-E does the conditional generation in a 2-stage process, using two distinct language models. This architectural choice is influenced by the hierarchical nature of the RVQ quantizer that generates the audio tokens. The output of the first RVQ quantizer is the most important token to the final speech, while the subsequent quantizers contribute less and less residual information to the final signal. So

the language model generates the acoustic codes in two stages. First, an autoregressive LM generates the first-quantizer codes for the entire output sequence, given the input text and enrolled audio. Then given those codes, a non-autoregressive LM is run 7 times, each time taking as input the output of the initial autoregressive codes and the prior non-autoregressive quantizer and thus generating the codes from the remaining quantizers one by one. Fig. 16.7 shows the intuition for the inference step.

Now let's see the architecture in a bit more detail. For **training**, we are given an audio sample $\mathbf{y}$ and its tokenized text transcription $\mathbf{x} = [x_0, x_1, \ldots, x_L]$. We use a pretrained EnCodec to convert $\mathbf{y}$ into a code matrix $\mathbf{C}$. Let T be the number of downsampled vectors output by EnCodec, with 8 codes per vector. Then we can represent the encoder output as

$$\mathbf{C}^{T\times 8} = \text{EnCodec}(\mathbf{y}) \tag{16.5}$$

Here $\mathbf{C}$ is a two-dimensional acoustic code matrix that has $T \times 8$ entries, where the columns represent time and the rows represent different quantizers. That is, the row vector $\mathbf{c}_{t,:}$ of the matrix contains the 8 codes for the t-th frame, and the column vector $\mathbf{c}_{:,j}$ contains the code sequence from the j-th vector quantizer where $j \in [1, ..., 8]$.

Given the text $\mathbf{x}$ and audio $\mathbf{C}$, we train the TTS as a conditional code language model to maximize the likelihood of $\mathbf{C}$ conditioned on $\mathbf{x}$:

$$\begin{aligned} L &= -\log p(\mathbf{C}|\mathbf{x}) \\ &= -\log \prod_{t=0}^{T} p(\mathbf{c}_{<t,:}, \mathbf{x}) \end{aligned} \tag{16.6}$$

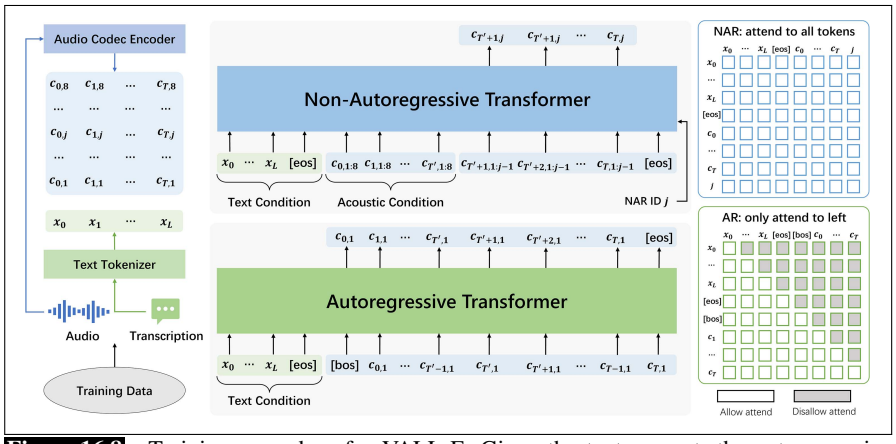


Figure 16.8 Training procedure for VALL-E. Given the text prompt, the autoregressive transformer is first trained to generate each code of the first-quantizer code sequence, autoregressively The the non-autoregressive transformer generates the rest of the codes. Figure from Chen et al. (2025).

Fig. 16.8 shows the intuition. On the left, we have an audio sample and its transcription, and both are tokenized. Then we append an [EOS] and [BOS] token to $\mathbf{x}$ and an [EOS] token to the end of $\mathbf{C}$ and train the autoregressive transformer to predict the acoustic tokens, starting with $\mathbf{c}_{0,1}$, until [EOS], and then the non-autoregressive transformers to fill in the other tokens.

During **inference**, we are given a text sequence to be spoken as well as $\mathbf{y}'$, an enrolled speech sample from some unseen speaker, for which we have the transcription

transcript($\mathbf{y}'$). We first run the codec to get an acoustic code matrix for $\mathbf{y}'$, which will be $\mathbf{C}^P = \mathbf{C}_{:T',:} = [\mathbf{c}_{0,:}, \mathbf{c}_{1,:}, \ldots \mathbf{c}_{T',:}:]$. Next we concatenate the transcription of $\mathbf{y}'$ to the text sequence to be spoken to create the total input text $\mathbf{x}$, which we pass through a text tokenizer. At this stage we thus have a tokenized text $\mathbf{x}$ and a tokenized audio prompt $\mathbf{C}^P$.

Then we generate $\mathbf{C}^T = \mathbf{C}_{>T',:} = [\mathbf{c}_{T'+1,:}, \ldots \mathbf{c}_{T,:}]$ conditioned on the text sequence $\mathbf{x}$ and the prompt $\mathbf{C}^P$:

$$\begin{aligned} \mathbf{C}^T &= \underset{\mathbf{C}^T}{\text{argmax}}\, p(\mathbf{C}^T|\mathbf{C}^P, \mathbf{x}) \\ &= \underset{\mathbf{C}^T}{\text{argmax}} \prod_{t=T'+1}^{T} p(\mathbf{c}_{t,:}|\mathbf{c}_{<t,:}, \mathbf{x}) \end{aligned} \tag{16.7}$$

Then the generated tokens $\mathbf{C}^T$ can be converted by the EnCodec decoder into a waveform. Fig. 16.9 shows the intuition.

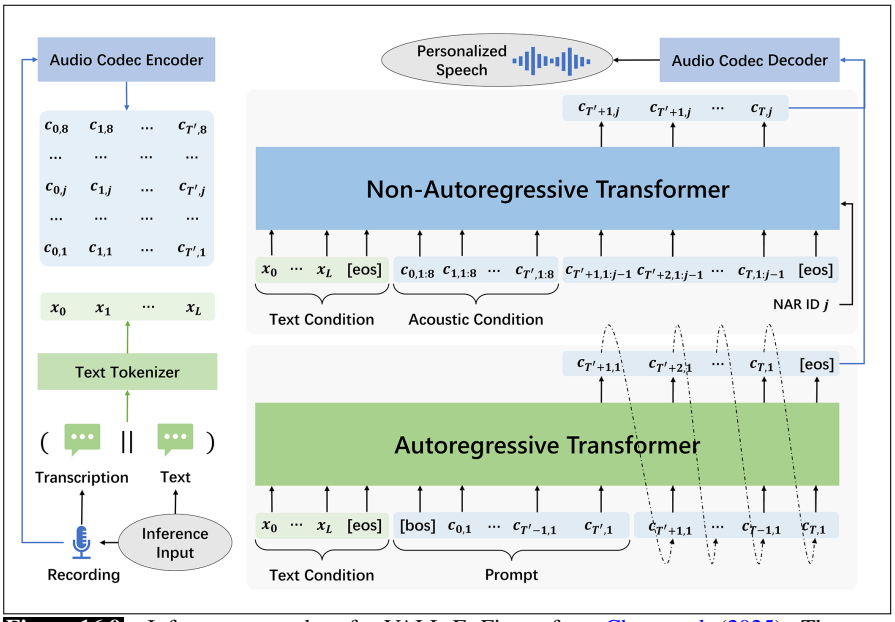


Figure 16.9 Inference procedure for VALL-E. Figure from Chen et al. (2025). The transcript for the 3 seconds of enrolled speech is first prepended to the text to be generated, and both the speech and text are tokenized. Next the autoregressive transformer starts generating the first codes $\mathbf{c}_{t'+1,1}$ conditioned on the transcript and acoustic prompt.

See Chen et al. (2025) for more details on the transformer components and other details of training.

16.4 TTS Evaluation

MOS

TTS systems are evaluated by humans, by playing an utterance to listeners and asking them to give a **mean opinion score** (**MOS**), a rating of how good the synthesized utterances are, usually on a scale from 1–5. We can then compare systems by comparing their MOS scores on the same sentences (using, e.g., paired t-tests to test for significant differences).

CMOS

If we are comparing exactly two systems (perhaps to see if a particular change actually improved the system), we can also compare using **CMOS** (Comparative MOS). where users give their preference on which of the two utterances is better. CMOS scores range from -3 (the system is much worse than the reference) to 3 (the system is better than the reference) Here we play the same sentence synthesized by two different systems. The human listeners choose which of the two utterances they like better. We do this for say 50 sentences (presented in random order) and compare the number of sentences preferred for each system.

Although speech synthesis systems are best evaluated by human listeners, some automatic metrics can be used to add more information. For example we can run the output through an ASR system and compute the word error rate (WER) to see how robust the synthesized output is. Or for measuring how well the voice output of the TTS system matches the enrolled voice, we can treat the task as if it were speaker verification, passing the two voices to a speaker verification system and using the resulting score as a similarity score.

16.5 Other speech tasks

There are a wide variety of other speech-related tasks.

speaker diarization

Speaker diarization is the task of determining 'who spoke when' in a long multi-speaker audio recording, marking the start and end of each speaker's turns in the interaction. This can be useful for transcribing meetings, classroom speech, or medical interactions. Often diarization systems use voice activity detection (VAD) to find segments of continuous speech, extract speaker embedding vectors, and cluster the vectors to group together segments likely from the same speaker. More recent work is investigating end-to-end algorithms to map directly from input speech to a sequence of speaker labels for each frame.

speaker recognition
speaker verification

Speaker recognition, is the task of identifying a speaker. We generally distinguish the subtasks of **speaker verification**, where we make a binary decision (is this speaker X or not?), such as for security when accessing personal information over the telephone, and **speaker identification**, where we make a one of N decision trying to match a speaker's voice against a database of many speakers.

language identification

In the task of **language identification**, we are given a wavefile and must identify which language is being spoken; this is an important part of building multilingual models, creating datasets, and even plays a role in online systems.

wake word

The task of **wake word** detection is to detect a word or short phrase, usually in order to wake up a voice-enable assistant like Alexa, Siri, or the Google Assistant. The goal with wake words is build the detection into small devices at the computing edge, to maintain privacy by transmitting the least amount of user speech to a cloud-based server. Thus wake word detectors need to be fast, small footprint software that can fit into embedded devices. Wake word detectors usually use the same frontend feature extraction we saw for ASR, often followed by a whole-word classifier.

16.6 Spoken Language Models

TBD

16.7 Summary

This chapter introduced the fundamental algorithms of **text-to-speech (TTS)**.

- A common modern algorithm for TTS is to use conditional generation with a language model over audio tokens learned by a codec model.
- A neural audio **codec**, short for **coder/decoder**, is a system that encodes analog speech signals into a digitized, discrete compressed representation for compression.
- The discrete symbols that a codec produces as its compressed representation can be used as discrete codes for language modeling.
- A codec includes an **encoder** that uses convnets to downsample speech into a downsampled embedding, a **quantizer** that converts the embedding into a series of discrete tokens, and a decoder that uses convnets to upsample the tokens/embedding back into a lossy reconstructed waveform.
- **Vector Quantization** (**VQ**) is a method for turning a series of vectors into a series of discrete symbols. This can be done by using k-means clustering, and then creating a **codebook** in which each code is represented by a vector at the centroid of each cluster, called a **codeword**. Input vector can be assigned the nearest codeword cluster.
- **Residual Vector Quantization** (**RVQ**) is a hierarchical version of vector quantization that produces multiple codes for an input vector by first quantizing a vector into a codebook, and then quantizing the residual (the difference between the codeword and the input vector) and then iterating.
- TTS systems like **VALL-E** take a text to be synthesized and a sample of the voice to be used, tokenize with BPE (text) and an audio codec (speech) and then use an LM to conditionally generate discrete audio tokens corresponding to the text prompt, in the voice of the speech sample.
- TTS is evaluated by playing a sentence to human listeners and having them give a **mean opinion score (MOS)**.

Historical Notes

As we noted at the beginning of the chapter, speech synthesis is one of the earliest fields of speech and language processing. The 18th century saw a number of physical models of the articulation process, including the von Kempelen model mentioned above, as well as the 1773 vowel model of Kratzenstein in Copenhagen using organ pipes.

The early 1950s saw the development of three early paradigms of waveform synthesis: formant synthesis, articulatory synthesis, and concatenative synthesis.

Formant synthesizers originally were inspired by attempts to mimic human speech by generating artificial spectrograms. The Haskins Laboratories Pattern Playback Machine generated a sound wave by painting spectrogram patterns on a moving transparent belt and using reflectance to filter the harmonics of a waveform (Cooper et al., 1951); other very early formant synthesizers include those of Lawrence (1953) and Fant (1951). Perhaps the most well-known of the formant synthesizers were the **Klatt formant synthesizer** and its successor systems, including the MITalk system (Allen et al., 1987) and the Klattalk software used in Digital Equipment Corporation's DECtalk (Klatt, 1982). See Klatt (1975) for details.

A second early paradigm, concatenative synthesis, seems to have been first proposed by Harris (1953) at Bell Laboratories; he literally spliced together pieces of magnetic tape corresponding to phones. Soon afterwards, Peterson et al. (1958) proposed a theoretical model based on diphones, including a database with multiple copies of each diphone with differing prosody, each labeled with prosodic features including F0, stress, and duration, and the use of join costs based on F0 and formant distance between neighboring units. But such **diphone synthesis** models were not actually implemented until decades later (Dixon and Maxey 1968, Olive 1977). The 1980s and 1990s saw the invention of **unit selection synthesis**, based on larger units of non-uniform length and the use of a target cost, (Sagisaka 1988, Sagisaka et al. 1992, Hunt and Black 1996, Black and Taylor 1994, Syrdal et al. 2000).

A third paradigm, **articulatory synthesizers** attempt to synthesize speech by modeling the physics of the vocal tract as an open tube. Representative models include Stevens et al. (1953), Flanagan et al. (1975), and Fant (1986). See Klatt (1975) and Flanagan (1972) for more details.

Most early TTS systems used phonemes as input; development of the text analysis components of TTS came somewhat later, drawing on NLP. Indeed the first true text-to-speech system seems to have been the system of Umeda and Teranishi (Umeda et al. 1968, Teranishi and Umeda 1968, Umeda 1976), which included a parser that assigned prosodic boundaries, as well as accent and stress.

History of codecs and modern history of neural TTS TBD.

Exercises

Volume II

ANNOTATING LINGUISTIC STRUCTURE

In the second volume of the book we discuss the task of detecting linguistic structure. In the early history of NLP these structures were an intermediate step toward deeper language processing. In modern NLP, we don't generally make explicit use of parse or other structures inside the large language models we introduced in Part I.

Instead linguistic structure plays a number of new roles. One important role is for **interpretability**: to provide a useful interpretive lens on neural networks. Knowing that a particular layer or neuron may be computing something related to a particular kind of structure can help us break open the 'black box' and understand what the components of our language models are doing.

A second important role for linguistic structure is as a practical tool for **social scientific studies** of text: knowing which adjective modifies which noun, or whether a particular implicit metaphor is being used, can be important for measuring attitudes toward groups or individuals. Detailed semantic structure can be helpful, for example in finding particular clauses that have particular meanings in legal contracts. Word sense labels can help keep any corpus study from measuring facts about the wrong word sense. Relation structures can be used to help build knowledge bases from text.

Finally, computation of linguistic structure is an important tool for answering questions about language itself, a research area called **computational linguistics** that is sometimes distinguished from natural language processing. To answer linguistic questions about how language changes over time or across individuals we'll need to be able, for example, to parse entire documents from different time periods. To understand how certain linguistic structures are learned or processed by people, it's necessary to be able to automatically label structures for arbitrary text.

In our study of linguistic structure, we begin with one of the oldest tasks in computational linguistics: the extraction of syntactic structure, and give two sets of algorithms for **parsing**: extracting syntactic structure, including constituency parsing and dependency parsing. We then introduce a variety of structures related to meaning, including semantic roles, word senses, entity relations, and events. We

conclude with linguistic structures that tend to be related to discourse and meaning over larger texts, including coreference and discourse coherence. In each case we'll give algorithms for automatically annotating the relevant structure.

CHAPTER

17 Sequence Labeling for Parts of Speech and Named Entities

To each word a warbling note
A Midsummer Night's Dream, V.I

Dionysius Thrax of Alexandria (*c.* 100 B.C.), or perhaps someone else (it was a long time ago), wrote a grammatical sketch of Greek (a "*technē*") that summarized the linguistic knowledge of his day. This work is the source of an astonishing proportion of modern linguistic vocabulary, including the words *syntax*, *diphthong*, *clitic*, and *analogy*. Also included are a description of eight **parts of speech**: noun, verb, pronoun, preposition, adverb, conjunction, participle, and article. Although earlier scholars (including Aristotle as well as the Stoics) had their own lists of parts of speech, it was Thrax's set of eight that became the basis for descriptions of European languages for the next 2000 years. (All the way to the *Schoolhouse Rock* educational television shows of our childhood, which had songs about 8 parts of speech, like the late great Bob Dorough's *Conjunction Junction*.) The durability of parts of speech through two millennia speaks to their centrality in models of human language.

parts of speech

Proper names are another important and anciently studied linguistic category. While parts of speech are generally assigned to individual words or morphemes, a proper name is often an entire multiword phrase, like the name "Marie Curie", the location "New York City", or the organization "Stanford University". We'll use the term **named entity** for, roughly speaking, anything that can be referred to with a proper name: a person, a location, an organization, although as we'll see the term is commonly extended to include things that aren't entities per se.

named entity

Parts of speech (also known as **POS**) and named entities are useful clues to sentence structure and meaning. Knowing whether a word is a noun or a verb tells us about likely neighboring words (nouns in English are preceded by determiners and adjectives, verbs by nouns) and syntactic structure (verbs have dependency links to nouns), making part-of-speech tagging a key aspect of parsing. Knowing if a named entity like *Washington* is a name of a person, a place, or a university is important to many natural language processing tasks like question answering, stance detection, or information extraction.

POS

In this chapter we'll introduce the task of **part-of-speech tagging**, taking a sequence of words and assigning each word a part of speech like NOUN or VERB, and the task of **named entity recognition** (**NER**), assigning words or phrases tags like PERSON, LOCATION, or ORGANIZATION.

Such tasks in which we assign, to each word x_i in an input word sequence, a label y_i, so that the output sequence Y has the same length as the input sequence X are called **sequence labeling** tasks. We'll introduce classic sequence labeling algorithms, one generative— the Hidden Markov Model (HMM)—and one discriminative—the Conditional Random Field (CRF). In following chapters we'll introduce modern sequence labelers based on RNNs and Transformers.

sequence labeling

17.1 (Mostly) English Word Classes

Until now we have been using part-of-speech terms like **noun** and **verb** rather freely. In this section we give more complete definitions. While word classes do have semantic tendencies—adjectives, for example, often describe *properties* and nouns *people*— parts of speech are defined instead based on their grammatical relationship with neighboring words or the morphological properties about their affixes.

	Tag	Description	Example
Open Class	**ADJ**	Adjective: noun modifiers describing properties	*red*, *young*, *awesome*
	ADV	Adverb: verb modifiers of time, place, manner	*very*, *slowly*, *home*, *yesterday*
	NOUN	words for persons, places, things, etc.	*algorithm*, *cat*, *mango*, *beauty*
	VERB	words for actions and processes	*draw*, *provide*, *go*
	PROPN	Proper noun: name of a person, organization, place, etc..	*Regina*, *IBM*, *Colorado*
	INTJ	Interjection: exclamation, greeting, yes/no response, etc.	*oh*, *um*, *yes*, *hello*
Closed Class Words	**ADP**	Adposition (Preposition/Postposition): marks a noun's spacial, temporal, or other relation	*in, on, by, under*
	AUX	Auxiliary: helping verb marking tense, aspect, mood, etc.,	*can, may, should, are*
	CCONJ	Coordinating Conjunction: joins two phrases/clauses	*and*, *or*, *but*
	DET	Determiner: marks noun phrase properties	*a, an, the, this*
	NUM	Numeral	*one, two, 2026, 11:00, hundred*
	PART	Particle: a function word that must be associated with another word	*'s, not, (infinitive) to*
	PRON	Pronoun: a shorthand for referring to an entity or event	*she, who, I, others*
	SCONJ	Subordinating Conjunction: joins a main clause with a subordinate clause such as a sentential complement	*whether*, *because*
Other	**PUNCT**	Punctuation	; , ()
	SYM	Symbols like $ or emoji	$, %
	X	Other	asdf, qwfg

Figure 17.1 The 17 parts of speech in the Universal Dependencies tagset (de Marneffe et al., 2021). Features can be added to make finer-grained distinctions (with properties like number, case, definiteness, and so on).

closed class **open class**

Parts of speech fall into two broad categories: **closed class** and **open class**. Closed classes are those with relatively fixed membership, such as prepositions—new prepositions are rarely coined. By contrast, nouns and verbs are open classes—new nouns and verbs like *iPhone* or *to fax* are continually being created or borrowed.

function word

Closed class words are generally **function words** like *of*, *it*, *and*, or *you*, which tend to be very short, occur frequently, and often have structuring uses in grammar.

Four major open classes occur in the languages of the world: **nouns** (including proper nouns), **verbs**, **adjectives**, and **adverbs**, as well as the smaller open class of **interjections**. English has all five, although not every language does.

noun **common noun**

Nouns are words for people, places, or things, but include others as well. **Common nouns** include concrete terms like *cat* and *mango*, abstractions like *algorithm* and *beauty*, and verb-like terms like *pacing* as in *His pacing to and fro became quite annoying*. Nouns in English can occur with determiners (*a goat, this bandwidth*) take possessives (*IBM's annual revenue*), and may occur in the plural (*goats, abaci*).

count noun **mass noun**

Many languages, including English, divide common nouns into **count nouns** and **mass nouns**. Count nouns can occur in the singular and plural (*goat/goats, relationship/relationships*) and can be counted (*one goat, two goats*). Mass nouns are used when something is conceptualized as a homogeneous group. So *snow, salt*, and *communism* are not counted (i.e., **two snows* or **two communisms*).

proper noun

Proper nouns, like *Regina*, *Colorado*, and *IBM*, are names of specific persons or entities.

verb **Verbs** refer to actions and processes, including main verbs like *draw*, *provide*, and *go*. English verbs have inflections (non-third-person-singular (*eat*), third-person-singular (*eats*), progressive (*eating*), past participle (*eaten*)). While many scholars believe that all human languages have the categories of noun and verb, others have argued that some languages, such as Riau Indonesian and Tongan, don't even make this distinction (Broschart 1997; Evans 2000; Gil 2000) .

adjective **Adjectives** often describe properties or qualities of nouns, like color (*white*, *black*), age (*old*, *young*), and value (*good*, *bad*), but there are languages without adjectives. In Korean, for example, the words corresponding to English adjectives act as a subclass of verbs, so what is in English an adjective "beautiful" acts in Korean like a verb meaning "to be beautiful".

adverb **Adverbs** are a hodge-podge. All the italicized words in this example are adverbs:

Actually, I ran *home extremely quickly yesterday*

Adverbs generally modify something (often verbs, hence the name "adverb", but also other adverbs and entire verb phrases). **Directional adverbs** or **locative adverbs** (*home*, *here*, *downhill*) specify the direction or location of some action; **degree adverbs** (*extremely*, *very*, *somewhat*) specify the extent of some action, process, or property; **manner adverbs** (*slowly*, *slinkily*, *delicately*) describe the manner of some action or process; and **temporal adverbs** describe the time that some action or event took place (*yesterday*, *Monday*).

locative degree manner temporal

interjection **Interjections** (*oh, hey, alas, uh, um*) are a smaller open class that also includes greetings (*hello, goodbye*) and question responses (*yes, no, uh-huh*).

preposition English adpositions occur before nouns, hence are called **prepositions**. They can indicate spatial or temporal relations, whether literal (*on it*, *before then*, *by the house*) or metaphorical (*on time*, *with gusto*, *beside herself*), and relations like marking the agent in *Hamlet was written by Shakespeare*.

particle A **particle** resembles a preposition or an adverb and is used in combination with a verb. Particles often have extended meanings that aren't quite the same as the prepositions they resemble, as in the particle *over* in *she turned the paper over*. A verb and a particle acting as a single unit is called a **phrasal verb**. The meaning of phrasal verbs is often **non-compositional**—not predictable from the individual meanings of the verb and the particle. Thus, *turn down* means 'reject', *rule out* 'eliminate', and *go on* 'continue'.

phrasal verb

determiner article **Determiners** like *this* and *that* (*this chapter*, *that page*) can mark the start of an English noun phrase. **Articles** like *a*, *an*, and *the*, are a type of determiner that mark discourse properties of the noun and are quite frequent; *the* is the most common word in written English, with *a* and *an* right behind.

conjunction **Conjunctions** join two phrases, clauses, or sentences. Coordinating conjunctions like *and*, *or*, and *but* join two elements of equal status. Subordinating conjunctions are used when one of the elements has some embedded status. For example, the subordinating conjunction *that* in *"I thought that you might like some milk"* links the main clause *I thought* with the subordinate clause *you might like some milk*. This clause is called subordinate because this entire clause is the "content" of the main verb *thought*. Subordinating conjunctions like *that* which link a verb to its argument in this way are also called **complementizers**.

complementizer

pronoun **Pronouns** act as a shorthand for referring to an entity or event. Personal pronouns refer to persons or entities (*you*, *she*, *I*, *it*, *me*, etc.). Possessive pronouns are forms of personal pronouns that indicate either actual possession or more often just an abstract relation between the person and some object (*my, your, his, her, its, one's, our, their*). **Wh-pronouns** (*what, who, whom, whoever*) are used in certain question

wh

forms, or act as complementizers (*Frida, who married Diego...*).

auxiliary

Auxiliary verbs mark semantic features of a main verb such as its tense, whether it is completed (aspect), whether it is negated (polarity), and whether an action is necessary, possible, suggested, or desired (mood). English auxiliaries include the

copula

copula verb *be*, the two verbs *do* and *have*, forms, as well as **modal verbs** used to

modal

mark the mood associated with the event depicted by the main verb: *can* indicates ability or possibility, *may* permission or possibility, *must* necessity.

An English-specific tagset, the Penn Treebank tagset (Marcus et al., 1993), shown in Fig. 17.2, has been used to label many syntactically annotated corpora like the Penn Treebank corpora, so it is worth knowing about.

Tag	Description	Example	Tag	Description	Example	Tag	Description	Example
CC	coord. conj.	*and, but, or*	NNP	proper noun, sing.	*IBM*	TO	infinitive to	*to*
CD	cardinal number	*one, two*	NNPS	proper noun, plu.	*Carolinas*	UH	interjection	*ah, oops*
DT	determiner	*a, the*	NNS	noun, plural	*llamas*	VB	verb base	*eat*
EX	existential 'there'	*there*	PDT	predeterminer	*all, both*	VBD	verb past tense	*ate*
FW	foreign word	*mea culpa*	POS	possessive ending	*'s*	VBG	verb gerund	*eating*
IN	preposition/ subordin-conj	*of, in, by*	PRP	personal pronoun	*I, you, he*	VBN	verb past participle	*eaten*
JJ	adjective	*yellow*	PRP$	possess. pronoun	*your*	VBP	verb non-3sg-pr	*eat*
JJR	comparative adj	*bigger*	RB	adverb	*quickly*	VBZ	verb 3sg pres	*eats*
JJS	superlative adj	*wildest*	RBR	comparative adv	*faster*	WDT	wh-determ.	*which, that*
LS	list item marker	*1, 2, One*	RBS	superlatv. adv	*fastest*	WP	wh-pronoun	*what, who*
MD	modal	*can, should*	RP	particle	*up, off*	WP$	wh-possess.	*whose*
NN	sing or mass noun	*llama*	SYM	symbol	+, %, &	WRB	wh-adverb	*how, where*

Figure 17.2 Penn Treebank core 36 part-of-speech tags.

Below we show some examples with each word tagged according to both the UD (in blue) and Penn (in red) tagsets. Notice that the Penn tagset distinguishes tense and participles on verbs, and has a special tag for the existential *there* construction in English. Note that since *London Journal of Medicine* is a proper noun, both tagsets mark its component nouns as PROPN/NNP, including *journal* and *medicine*, which might otherwise be labeled as common nouns (NOUN/NN).

(17.1) There/PRON/EX are/VERB/VBP 70/NUM/CD children/NOUN/NNS there/ADV/RB ./PUNC/.

(17.2) Preliminary/ADJ/JJ findings/NOUN/NNS were/AUX/VBD reported/VERB/VBN in/ADP/IN today/NOUN/NN 's/PART/POS London/PROPN/NNP Journal/PROPN/NNP of/ADP/IN Medicine/PROPN/NNP

17.2 Part-of-Speech Tagging

part-of-speech tagging

Part-of-speech tagging is the process of assigning a part-of-speech to each word in a text. The input is a sequence $x_1, x_2, ..., x_n$ of (tokenized) words and a tagset, and the output is a sequence $y_1, y_2, ..., y_n$ of tags, each output y_i corresponding exactly to one input x_i, as shown in the intuition in Fig. 17.3.

ambiguous

Tagging is a **disambiguation** task; words are **ambiguous** —have more than one possible part-of-speech—and the goal is to find the correct tag for the situation. For example, *book* can be a verb (*book that flight*) or a noun (*hand me that book*). *That* can be a determiner (*Does that flight serve dinner*) or a complementizer (*I*

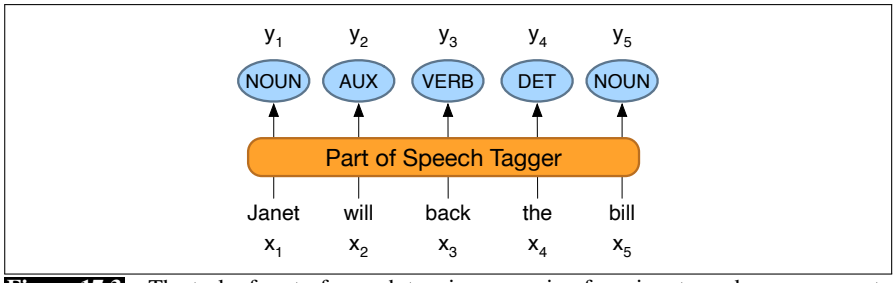


Figure 17.3 The task of part-of-speech tagging: mapping from input words $x_1, x_2, ..., x_n$ to output POS tags $y_1, y_2, ..., y_n$.

ambiguity resolution

thought that your flight was earlier). The goal of POS-tagging is to **resolve** these ambiguities, choosing the proper tag for the context.

accuracy

The **accuracy** of part-of-speech tagging algorithms (the percentage of test set tags that match human gold labels) is extremely high. One study found accuracies over 97% across 15 languages from the Universal Dependency (UD) treebank (Wu and Dredze, 2019). Accuracies on various English treebanks are also 97% (no matter the algorithm; HMMs, CRFs, BERT perform similarly). This 97% number is also about the human performance on this task, at least for English (Manning, 2011).

Types:		WSJ	Brown
Unambiguous	(1 tag)	44,432 (**86%**)	45,799 (**85%**)
Ambiguous	(2+ tags)	7,025 (**14%**)	8,050 (**15%**)
Tokens:			
Unambiguous	(1 tag)	577,421 (**45%**)	384,349 (**33%**)
Ambiguous	(2+ tags)	711,780 (**55%**)	786,646 (**67%**)

Figure 17.4 Tag ambiguity in the Brown and WSJ corpora (Treebank-3 45-tag tagset).

We'll introduce algorithms for the task in the next few sections, but first let's explore the task. Exactly how hard is it? Fig. 17.4 shows that most word types (85-86%) are unambiguous (*Janet* is always NNP, *hesitantly* is always RB). But the ambiguous words, though accounting for only 14-15% of the vocabulary, are very common, and 55-67% of word tokens in running text are ambiguous. Particularly ambiguous common words include *that*, *back*, *down*, *put* and *set*; here are some examples of the 6 different parts of speech for the word *back*:

earnings growth took a **back/JJ** seat
a small building in the **back/NN**
a clear majority of senators **back/VBP** the bill
Dave began to **back/VB** toward the door
enable the country to buy **back/RP** debt
I was twenty-one **back/RB** then

Nonetheless, many words are easy to disambiguate, because their different tags aren't equally likely. For example, *a* can be a determiner or the letter *a*, but the determiner sense is much more likely.

This idea suggests a useful **baseline**: given an ambiguous word, choose the tag which is **most frequent** in the training corpus. This is a key concept:

Most Frequent Class Baseline: Always compare a classifier against a baseline at least as good as the most frequent class baseline (assigning each token to the class it occurred in most often in the training set).

The most-frequent-tag baseline has an accuracy of about 92%[1]. The baseline thus differs from the state-of-the-art and human ceiling (97%) by only 5%.

17.3 Named Entities and Named Entity Tagging

Part of speech tagging can tell us that words like *Janet*, *Stanford University*, and *Colorado* are all proper nouns; being a proper noun is a grammatical property of these words. But viewed from a semantic perspective, these proper nouns refer to different kinds of entities: Janet is a person, Stanford University is an organization, and Colorado is a location.

named entity

Here we re-introduce the concept of a **named entity**, which was also introduced in Section 9.5 for readers who haven't yet read Chapter 9.

named entity

A **named entity** is, roughly speaking, anything that can be referred to with a proper name: a person, a location, an organization. The task of **named entity recognition** (**NER**) is to find spans of text that constitute proper names and tag the type of the entity. Four entity tags are most common: **PER** (person), **LOC** (location), **ORG** (organization), or **GPE** (geo-political entity). However, the term **named entity** is commonly extended to include things that aren't entities per se, including dates, times, and other kinds of temporal expressions, and even numerical expressions like prices. Here's an example of the output of an NER tagger:

named entity recognition
NER

> Citing high fuel prices, [ORG **United Airlines**] said [TIME **Friday**] it has increased fares by [MONEY **$6**] per round trip on flights to some cities also served by lower-cost carriers. [ORG **American Airlines**], a unit of [ORG **AMR Corp.**], immediately matched the move, spokesman [PER **Tim Wagner**] said. [ORG **United**], a unit of [ORG **UAL Corp.**], said the increase took effect [TIME **Thursday**] and applies to most routes where it competes against discount carriers, such as [LOC **Chicago**] to [LOC **Dallas**] and [LOC **Denver**] to [LOC **San Francisco**].

The text contains 13 mentions of named entities including 5 organizations, 4 locations, 2 times, 1 person, and 1 mention of money. Figure 17.5 shows typical generic named entity types. Many applications will also need to use specific entity types like proteins, genes, commercial products, or works of art.

Type	Tag	Sample Categories	Example sentences
People	PER	people, characters	**Turing** is a giant of computer science.
Organization	ORG	companies, sports teams	The **IPCC** warned about the cyclone.
Location	LOC	regions, mountains, seas	**Mt. Sanitas** is in **Sunshine Canyon**.
Geo-Political Entity	GPE	countries, states	**Palo Alto** is raising the fees for parking.

Figure 17.5 A list of generic named entity types with the kinds of entities they refer to.

Named entity tagging is a useful first step in lots of natural language processing tasks. In sentiment analysis we might want to know a consumer's sentiment toward a particular entity. Entities are a useful first stage in question answering, or for linking text to information in structured knowledge sources like Wikipedia. And named entity tagging is also central to tasks involving building semantic representations, like extracting events and the relationship between participants.

[1] In English, on the WSJ corpus, tested on sections 22-24.

Unlike part-of-speech tagging, where there is no segmentation problem since each word gets one tag, the task of named entity recognition is to find and label *spans* of text, and is difficult partly because of the ambiguity of segmentation; we need to decide what's an entity and what isn't, and where the boundaries are. Indeed, most words in a text will not be named entities. Another difficulty is caused by type ambiguity. The mention JFK can refer to a person, the airport in New York, or any number of schools, bridges, and streets around the United States. Some examples of this kind of cross-type confusion are given in Figure 17.6.

[PER Washington] was born into slavery on the farm of James Burroughs.
[ORG Washington] went up 2 games to 1 in the four-game series.
Blair arrived in [LOC Washington] for what may well be his last state visit.
In June, [GPE Washington] passed a primary seatbelt law.

Figure 17.6 Examples of type ambiguities in the use of the name *Washington*.

The standard approach to sequence labeling for a span-recognition problem like NER is **BIO** tagging (Ramshaw and Marcus, 1995). This is a method that allows us to treat NER like a word-by-word sequence labeling task, via tags that capture both the boundary and the named entity type. Consider the following sentence:

[PER **Jane Villanueva**] of [ORG **United**] , a unit of [ORG **United Airlines Holding**] , said the fare applies to the [LOC **Chicago**] route.

BIO

Figure 17.7 shows the same excerpt represented with **BIO** tagging, as well as variants called **IO** tagging and **BIOES** tagging. In BIO tagging we label any token that *begins* a span of interest with the label B, tokens that occur *inside* a span are tagged with an I, and any tokens *outside* of any span of interest are labeled O. While there is only one O tag, we'll have distinct B and I tags for each named entity class. The number of tags is thus $2n+1$ tags, where n is the number of entity types. BIO tagging can represent exactly the same information as the bracketed notation, but has the advantage that we can represent the task in the same simple sequence modeling way as part-of-speech tagging: assigning a single label y_i to each input word x_i:

Words	IO Label	BIO Label	BIOES Label
Jane	I-PER	B-PER	B-PER
Villanueva	I-PER	I-PER	E-PER
of	O	O	O
United	I-ORG	B-ORG	B-ORG
Airlines	I-ORG	I-ORG	I-ORG
Holding	I-ORG	I-ORG	E-ORG
discussed	O	O	O
the	O	O	O
Chicago	I-LOC	B-LOC	S-LOC
route	O	O	O
.	O	O	O

Figure 17.7 NER as a sequence model, showing IO, BIO, and BIOES taggings.

We've also shown two variant tagging schemes: IO tagging, which loses some information by eliminating the B tag, and BIOES tagging, which adds an end tag E for the end of a span, and a span tag S for a span consisting of only one word. A sequence labeler (HMM, CRF, RNN, Transformer, etc.) is trained to label each token in a text with tags that indicate the presence (or absence) of particular kinds of named entities.

17.4 HMM Part-of-Speech Tagging

In this section we introduce our first sequence labeling algorithm, the Hidden Markov Model, and show how to apply it to part-of-speech tagging. Recall that a sequence labeler is a model whose job is to assign a label to each unit in a sequence, thus mapping a sequence of observations to a sequence of labels of the same length. The HMM is a classic model that introduces many of the key concepts of sequence modeling that we will see again in more modern models.

An HMM is a probabilistic sequence model: given a sequence of units (words, letters, morphemes, sentences, whatever), it computes a probability distribution over possible sequences of labels and chooses the best label sequence.

17.4.1 Markov Chains

Markov chain

The HMM is based on augmenting the Markov chain. A **Markov chain** is a model that tells us something about the probabilities of sequences of random variables, *states*, each of which can take on values from some set. These sets can be words, or tags, or symbols representing anything, for example the weather. A Markov chain makes a very strong assumption that if we want to predict the future in the sequence, all that matters is the current state. All the states before the current state have no impact on the future except via the current state. It's as if to predict tomorrow's weather you could examine today's weather but you weren't allowed to look at yesterday's weather.

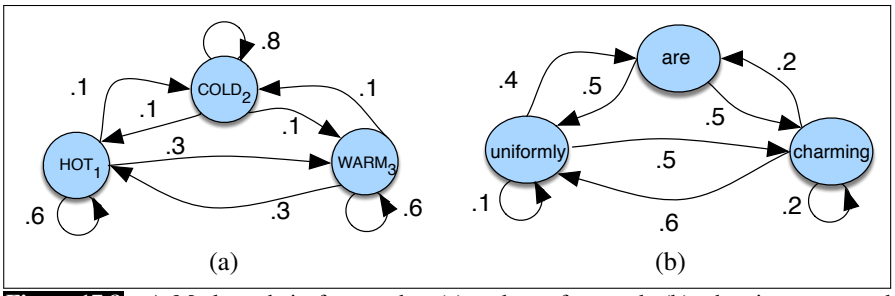


Figure 17.8 A Markov chain for weather (a) and one for words (b), showing states and transitions. A start distribution π is required; setting $\pi = [0.1, 0.7, 0.2]$ for (a) would mean a probability 0.7 of starting in state 2 (cold), probability 0.1 of starting in state 1 (hot), etc.

Markov assumption

More formally, consider a sequence of state variables $q_1, q_2, ..., q_i$. A Markov model embodies the **Markov assumption** on the probabilities of this sequence: that when predicting the future, the past doesn't matter, only the present.

$$\textbf{Markov Assumption:} \quad P(q_i = a|q_1...q_{i-1}) = P(q_i = a|q_{i-1}) \tag{17.3}$$

Figure 17.8a shows a Markov chain for assigning a probability to a sequence of weather events, for which the vocabulary consists of HOT, COLD, and WARM. The states are represented as nodes in the graph, and the transitions, with their probabilities, as edges. The transitions are probabilities: the values of arcs leaving a given state must sum to 1. Figure 17.8b shows a Markov chain for assigning a probability to a sequence of words $w_1...w_t$. This Markov chain should be familiar; in fact, it represents a bigram language model, with each edge expressing the probability $p(w_i|w_j)$! Given the two models in Fig. 17.8, we can assign a probability to any sequence from our vocabulary.

Formally, a Markov chain is specified by the following components:

$Q = q_1 q_2 \ldots q_N$	a set of N **states**
$A = a_{11} a_{12} \ldots a_{N1} \ldots a_{NN}$	a **transition probability matrix** A, each a_{ij} representing the probability of moving from state i to state j, s.t. $\sum_{j=1}^{n} a_{ij} = 1 \quad \forall i$
$\pi = \pi_1, \pi_2, ..., \pi_N$	an **initial probability distribution** over states. π_i is the probability that the Markov chain will start in state i. Some states j may have $\pi_j = 0$, meaning that they cannot be initial states. Also, $\sum_{i=1}^{n} \pi_i = 1$

Before you go on, use the sample probabilities in Fig. 17.8a (with $\pi = [0.1, 0.7, 0.2]$) to compute the probability of each of the following sequences:

(17.4) hot hot hot hot
(17.5) cold hot cold hot

What does the difference in these probabilities tell you about a real-world weather fact encoded in Fig. 17.8a?

17.4.2 The Hidden Markov Model

A Markov chain is useful when we need to compute a probability for a sequence of observable events. In many cases, however, the events we are interested in are **hidden**: we don't observe them directly. For example we don't normally observe part-of-speech tags in a text. Rather, we see words, and must infer the tags from the word sequence. We call the tags **hidden** because they are not observed.

hidden

hidden Markov model

A **hidden Markov model** (**HMM**) allows us to talk about both *observed* events (like words that we see in the input) and *hidden* events (like part-of-speech tags) that we think of as causal factors in our probabilistic model. An HMM is specified by the following components:

$Q = q_1 q_2 \ldots q_N$	a set of N **states**
$A = a_{11} \ldots a_{ij} \ldots a_{NN}$	a **transition probability matrix** A, each a_{ij} representing the probability of moving from state i to state j, s.t. $\sum_{j=1}^{N} a_{ij} = 1 \quad \forall i$
$B = b_i(o_t)$	a sequence of **observation likelihoods**, also called **emission probabilities**, each expressing the probability of an observation o_t (drawn from a vocabulary $V = v_1, v_2, ..., v_V$) being generated from a state q_i
$\pi = \pi_1, \pi_2, ..., \pi_N$	an **initial probability distribution** over states. π_i is the probability that the Markov chain will start in state i. Some states j may have $\pi_j = 0$, meaning that they cannot be initial states. Also, $\sum_{i=1}^{n} \pi_i = 1$

The HMM is given as input $O = o_1 o_2 \ldots o_T$: a sequence of T **observations**, each one drawn from the vocabulary V.

A first-order hidden Markov model instantiates two simplifying assumptions. First, as with a first-order Markov chain, the probability of a particular state depends only on the previous state:

$$\textbf{Markov Assumption:} \quad P(q_i|q_1, ..., q_{i-1}) = P(q_i|q_{i-1}) \tag{17.6}$$

Second, the probability of an output observation o_i depends only on the state that produced the observation q_i and not on any other states or any other observations:

$$\textbf{Output Independence:} \quad P(o_i|q_1, \ldots q_i, \ldots, q_T, o_1, \ldots, o_i, \ldots, o_T) = P(o_i|q_i) \tag{17.7}$$

17.4.3 The components of an HMM tagger

An HMM has two components, the A and B probabilities, both estimated by counting on a tagged training corpus. (For this example we'll use the tagged WSJ corpus.)

The A matrix contains the tag transition probabilities $P(t_i|t_{i-1})$ which represent the probability of a tag occurring given the previous tag. For example, modal verbs like *will* are very likely to be followed by a verb in the base form, a VB, like *race*, so we expect this probability to be high. We compute the maximum likelihood estimate of this transition probability by counting, out of the times we see the first tag in a labeled corpus, how often the first tag is followed by the second:

$$P(t_i|t_{i-1}) = \frac{C(t_{i-1},t_i)}{C(t_{i-1})} \tag{17.8}$$

In the WSJ corpus, for example, MD occurs 13124 times of which it is followed by VB 10471, for an MLE estimate of

$$P(VB|MD) = \frac{C(MD,VB)}{C(MD)} = \frac{10471}{13124} = .80 \tag{17.9}$$

The B emission probabilities, $P(w_i|t_i)$, represent the probability, given a tag (say MD), that it will be associated with a given word (say *will*). The MLE of the emission probability is

$$P(w_i|t_i) = \frac{C(t_i,w_i)}{C(t_i)} \tag{17.10}$$

Of the 13124 occurrences of MD in the WSJ corpus, it is associated with *will* 4046 times:

$$P(will|MD) = \frac{C(MD,will)}{C(MD)} = \frac{4046}{13124} = .31 \tag{17.11}$$

We saw this kind of Bayesian modeling in Appendix K; recall that this likelihood term is not asking "which is the most likely tag for the word *will*?" That would be the posterior $P(\text{MD}|\text{will})$. Instead, $P(\text{will}|\text{MD})$ answers the slightly counterintuitive question "If we were going to generate a MD, how likely is it that this modal would be *will*?"

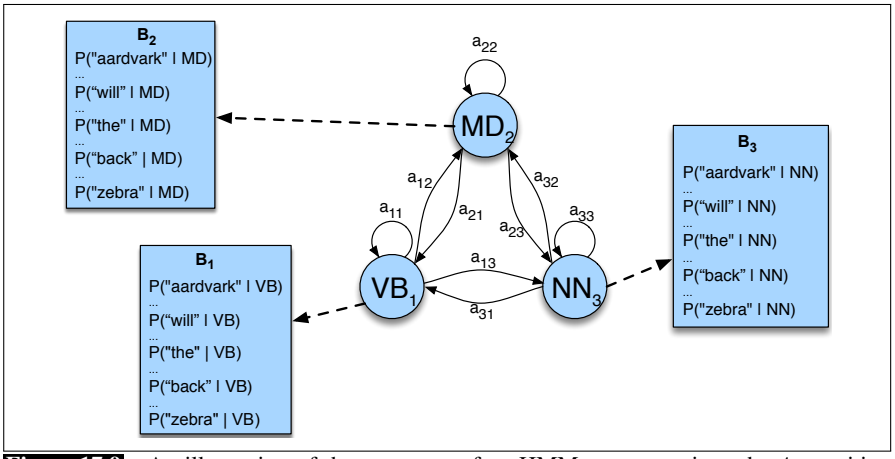


Figure 17.9 An illustration of the two parts of an HMM representation: the A transition probabilities used to compute the prior probability, and the B observation likelihoods that are associated with each state, one likelihood for each possible observation word.

The A transition probabilities, and B observation likelihoods of the HMM are illustrated in Fig. 17.9 for three states in an HMM part-of-speech tagger; the full tagger would have one state for each tag.

17.4.4 HMM tagging as decoding

For any model, such as an HMM, that contains hidden variables, the task of determining the hidden variables sequence corresponding to the sequence of observations is called **decoding**. More formally,

decoding

> **Decoding**: Given as input an HMM $\lambda = (A,B)$ and a sequence of observations $O = o_1, o_2, ..., o_T$, find the most probable sequence of states $Q = q_1 q_2 q_3 \dots q_T$.

For part-of-speech tagging, the goal of HMM decoding is to choose the tag sequence $t_1 \dots t_n$ that is most probable given the observation sequence of n words $w_1 \dots w_n$:

$$\hat{t}_{1:n} = \underset{t_1 \dots t_n}{\operatorname{argmax}} P(t_1 \dots t_n | w_1 \dots w_n) \tag{17.12}$$

The way we'll do this in the HMM is to use Bayes' rule to instead compute:

$$\hat{t}_{1:n} = \underset{t_1 \dots t_n}{\operatorname{argmax}} \frac{P(w_1 \dots w_n | t_1 \dots t_n) P(t_1 \dots t_n)}{P(w_1 \dots w_n)} \tag{17.13}$$

Furthermore, we simplify Eq. 17.13 by dropping the denominator $P(w_1^n)$:

$$\hat{t}_{1:n} = \underset{t_1 \dots t_n}{\operatorname{argmax}} P(w_1 \dots w_n | t_1 \dots t_n) P(t_1 \dots t_n) \tag{17.14}$$

HMM taggers make two further simplifying assumptions. The first (output independence, from Eq. 17.7) is that the probability of a word appearing depends only on its own tag and is independent of neighboring words and tags:

$$P(w_1 \dots w_n | t_1 \dots t_n) \approx \prod_{i=1}^{n} P(w_i | t_i) \tag{17.15}$$

The second assumption (the **Markov** assumption, Eq. 17.6) is that the probability of a tag is dependent only on the previous tag, rather than the entire tag sequence;

$$P(t_1 \dots t_n) \approx \prod_{i=1}^{n} P(t_i | t_{i-1}) \tag{17.16}$$

Plugging the simplifying assumptions from Eq. 17.15 and Eq. 17.16 into Eq. 17.14 results in the following equation for the most probable tag sequence from a bigram tagger:

$$\hat{t}_{1:n} = \underset{t_1 \dots t_n}{\operatorname{argmax}} P(t_1 \dots t_n | w_1 \dots w_n) \approx \underset{t_1 \dots t_n}{\operatorname{argmax}} \prod_{i=1}^{n} \overbrace{P(w_i | t_i)}^{\text{emission}} \overbrace{P(t_i | t_{i-1})}^{\text{transition}} \tag{17.17}$$

The two parts of Eq. 17.17 correspond neatly to the B **emission probability** and A **transition probability** that we just defined above!

function VITERBI(*observations* of len T,*state-graph* of len N) **returns** *best-path*, *path-prob*

create a path probability matrix *viterbi[N,T]*
create a backpointer matrix *backpointer[N,T]*
for each state s **from** 1 **to** N **do** ; initialization step
  viterbi[s,1] $\leftarrow \pi_s * b_s(o_1)$
  backpointer[s,1] $\leftarrow 0$
for each time step t **from** 2 **to** T **do** ; recursion step
  for each state s **from** 1 **to** N **do**
    viterbi[s,t] $\leftarrow \max_{s'=1}^{N}$ *viterbi*$[s',t-1] * a_{s',s} * b_s(o_t)$
    backpointer[s,t] $\leftarrow \operatorname{argmax}_{s'=1}^{N}$ *viterbi*$[s',t-1] * a_{s',s} * b_s(o_t)$
bestpathprob $\leftarrow \max_{s=1}^{N}$ *viterbi*$[s,T]$; termination step
bestpathpointer $\leftarrow \operatorname{argmax}_{s=1}^{N}$ *viterbi*$[s,T]$; termination step
bestpath $\leftarrow$ the path starting at state *bestpathpointer*, that follows backpointer[] to states back in time
return *bestpath*, *bestpathprob*

Figure 17.10 Viterbi algorithm for finding the optimal sequence of tags. Given an observation sequence and an HMM $\lambda = (A,B)$, the algorithm returns the state path through the HMM that assigns maximum likelihood to the observation sequence.

17.4.5 The Viterbi Algorithm

Viterbi algorithm

The decoding algorithm for HMMs is the **Viterbi algorithm** shown in Fig. 17.10. As an instance of **dynamic programming**, Viterbi resembles the dynamic programming **minimum edit distance** algorithm of Chapter 2.

The Viterbi algorithm first sets up a probability matrix or **lattice**, with one column for each observation o_t and one row for each state in the state graph. Each column thus has a cell for each state q_i in the single combined automaton. Figure 17.11 shows an intuition of this lattice for the sentence *Janet will back the bill*.

Each cell of the lattice, $v_t(j)$, represents the probability that the HMM is in state j after seeing the first t observations and passing through the most probable state sequence $q_1,...,q_{t-1}$, given the HMM λ. The value of each cell $v_t(j)$ is computed by recursively taking the most probable path that could lead us to this cell. Formally, each cell expresses the probability

$$v_t(j) = \max_{q_1,\ldots,q_{t-1}} P(q_1 \ldots q_{t-1}, o_1, o_2 \ldots o_t, q_t = j|\lambda) \tag{17.18}$$

We represent the most probable path by taking the maximum over all possible previous state sequences $\max_{q_1,\ldots,q_{t-1}}$. Like other dynamic programming algorithms, Viterbi fills each cell recursively. Given that we had already computed the probability of being in every state at time $t-1$, we compute the Viterbi probability by taking the most probable of the extensions of the paths that lead to the current cell. For a given state q_j at time t, the value $v_t(j)$ is computed as

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(i)\, a_{ij}\, b_j(o_t) \tag{17.19}$$

The three factors that are multiplied in Eq. 17.19 for extending the previous paths to compute the Viterbi probability at time t are

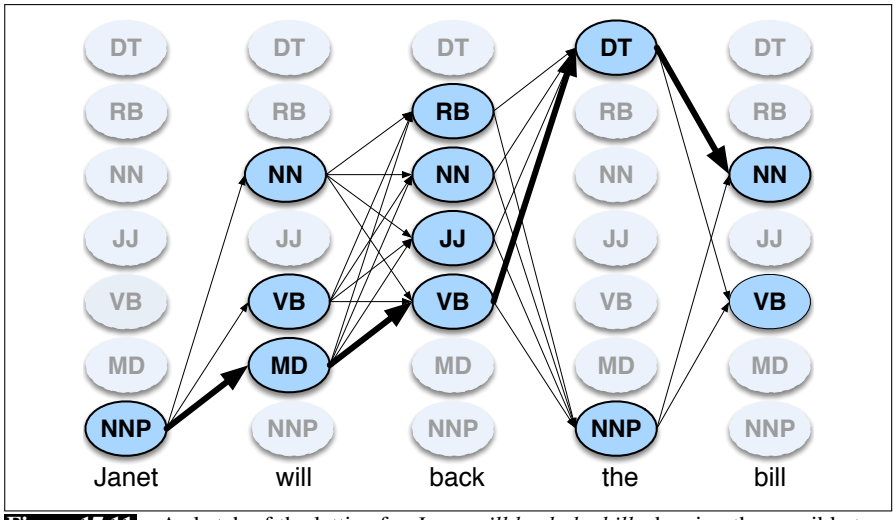


Figure 17.11 A sketch of the lattice for *Janet will back the bill*, showing the possible tags (q_i) for each word and highlighting the path corresponding to the correct tag sequence through the hidden states. States (parts of speech) which have a zero probability of generating a particular word according to the B matrix (such as the probability that a determiner DT will be realized as *Janet*) are greyed out.

$v_{t-1}(i)$	the **previous Viterbi path probability** from the previous time step
a_{ij}	the **transition probability** from previous state q_i to current state q_j
$b_j(o_t)$	the **state observation likelihood** of the observation symbol o_t given the current state j

17.4.6 Working through an example

Let's tag the sentence *Janet will back the bill*; the goal is the correct series of tags (see also Fig. 17.11):

(17.20) Janet/NNP will/MD back/VB the/DT bill/NN

	NNP	MD	VB	JJ	NN	RB	DT
$<s>$	0.2767	0.0006	0.0031	0.0453	0.0449	0.0510	0.2026
NNP	0.3777	0.0110	0.0009	0.0084	0.0584	0.0090	0.0025
MD	0.0008	0.0002	0.7968	0.0005	0.0008	0.1698	0.0041
VB	0.0322	0.0005	0.0050	0.0837	0.0615	0.0514	0.2231
JJ	0.0366	0.0004	0.0001	0.0733	0.4509	0.0036	0.0036
NN	0.0096	0.0176	0.0014	0.0086	0.1216	0.0177	0.0068
RB	0.0068	0.0102	0.1011	0.1012	0.0120	0.0728	0.0479
DT	0.1147	0.0021	0.0002	0.2157	0.4744	0.0102	0.0017

Figure 17.12 The A transition probabilities $P(t_i|t_{i-1})$ computed from the WSJ corpus without smoothing. Rows are labeled with the conditioning event; thus $P(VB|MD)$ is 0.7968. $<s>$ is the start token.

Let the HMM be defined by the two tables in Fig. 17.12 and Fig. 17.13. Figure 17.12 lists the a_{ij} probabilities for transitioning between the hidden states (part-of-speech tags). Figure 17.13 expresses the $b_i(o_t)$ probabilities, the *observation* likelihoods of words given tags. This table is (slightly simplified) from counts in the WSJ corpus. So the word *Janet* only appears as an NNP, *back* has 4 possible parts

	Janet	will	back	the	bill
NNP	0.000032	0	0	0.000048	0
MD	0	0.308431	0	0	0
VB	0	0.000028	0.000672	0	0.000028
JJ	0	0	0.000340	0	0
NN	0	0.000200	0.000223	0	0.002337
RB	0	0	0.010446	0	0
DT	0	0	0	0.506099	0

Figure 17.13 Observation likelihoods B computed from the WSJ corpus without smoothing, simplified slightly.

of speech, and the word *the* can appear as a determiner or as an NNP (in titles like "Somewhere Over the Rainbow" all words are tagged as NNP).

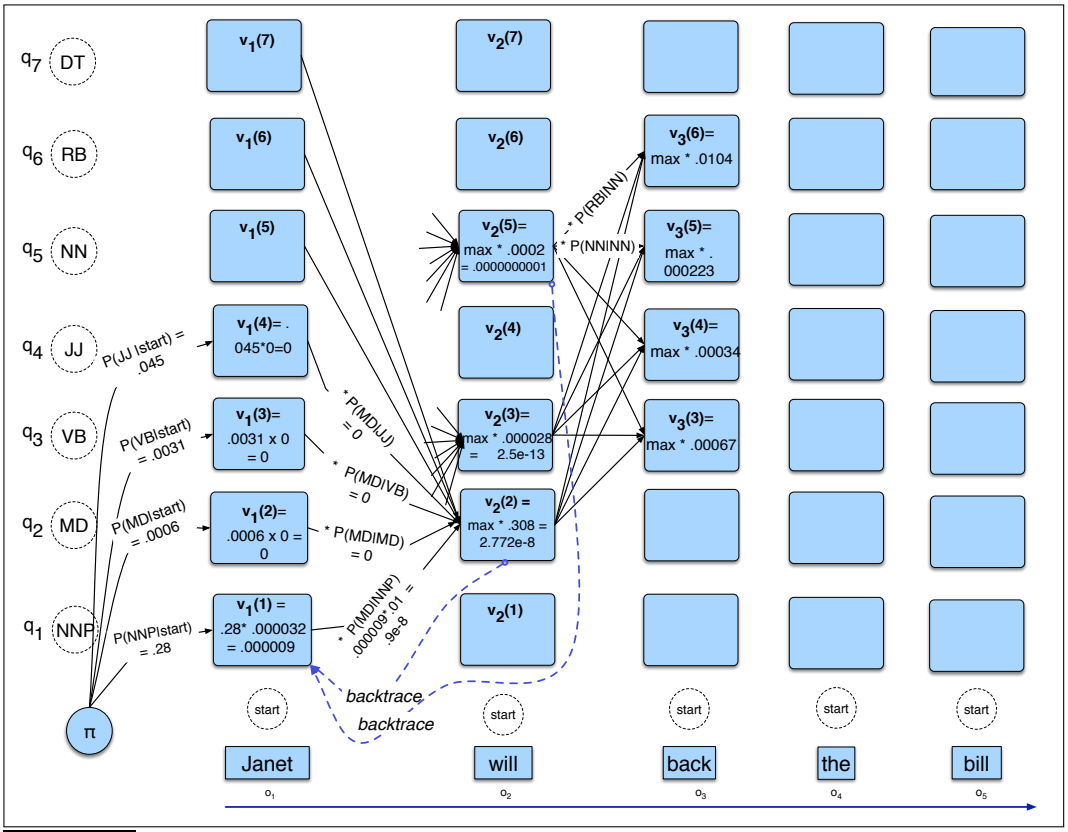


Figure 17.14 The first few entries in the individual state columns for the Viterbi algorithm. Each cell keeps the probability of the best path so far and a pointer to the previous cell along that path. We have only filled out columns 1 and 2; to avoid clutter most cells with value 0 are left empty. The rest is left as an exercise for the reader. After the cells are filled in, backtracing from the *end* state, we should be able to reconstruct the correct state sequence NNP MD VB DT NN.

Figure 17.14 shows a fleshed-out version of the sketch we saw in Fig. 17.11, the Viterbi lattice for computing the best hidden state sequence for the observation sequence *Janet will back the bill*.

There are $N = 5$ state columns. We begin in column 1 (for the word *Janet*) by setting the Viterbi value in each cell to the product of the π transition probability (the start probability for that state i, which we get from the $<s>$ entry of Fig. 17.12), and

the observation likelihood of the word *Janet* given the tag for that cell. Most of the cells in the column are zero since the word *Janet* cannot be any of those tags. The reader should find this in Fig. 17.14.

Next, each cell in the *will* column gets updated. For each state, we compute the value $viterbi[s,t]$ by taking the maximum over the extensions of all the paths from the previous column that lead to the current cell according to Eq. 17.19. We have shown the values for the MD, VB, and NN cells. Each cell gets the max of the 7 values from the previous column, multiplied by the appropriate transition probability; as it happens in this case, most of them are zero from the previous column. The remaining value is multiplied by the relevant observation probability, and the (trivial) max is taken. In this case the final value, 2.772e-8, comes from the NNP state at the previous column. The reader should fill in the rest of the lattice in Fig. 17.14 and backtrace to see whether or not the Viterbi algorithm returns the gold state sequence NNP MD VB DT NN.

17.5 Conditional Random Fields (CRFs)

unknown words

While the HMM is a useful and powerful model, it turns out that HMMs need a number of augmentations to achieve high accuracy. For example, in POS tagging as in other tasks, we often run into **unknown words**: proper names and acronyms are created very often, and even new common nouns and verbs enter the language at a surprising rate. It would be great to have ways to add arbitrary features to help with this, perhaps based on capitalization or morphology (words starting with capital letters are likely to be proper nouns, words ending with *-ed* tend to be past tense (VBD or VBN), etc.) Or knowing the previous or following words might be a useful feature (if the previous word is *the*, the current tag is unlikely to be a verb).

Although we could try to hack the HMM to find ways to incorporate some of these, in general it's hard for generative models like HMMs to add arbitrary features directly into the model in a clean way. We've already seen a model for combining arbitrary features in a principled way: log-linear models like the logistic regression model of Chapter 4! But logistic regression isn't a sequence model; it assigns a class to a single observation.

CRF

Luckily, there is a discriminative sequence model based on log-linear models: the **conditional random field** (**CRF**). We'll describe here the **linear chain CRF**, the version of the CRF most commonly used for language processing, and the one whose conditioning closely matches the HMM.

Assuming we have a sequence of input words $X = x_1 \ldots x_n$ and want to compute a sequence of output tags $Y = y_1 \ldots y_n$. In an HMM to compute the best tag sequence that maximizes $P(Y|X)$ we rely on Bayes' rule and the likelihood $P(X|Y)$:

$$\begin{aligned}\hat{Y} &= \operatorname*{argmax}_Y p(Y|X) \\ &= \operatorname*{argmax}_Y p(X|Y)p(Y) \\ &= \operatorname*{argmax}_Y \prod_i p(x_i|y_i) \prod_i p(y_i|y_{i-1}) \end{aligned} \tag{17.21}$$

In a CRF, by contrast, we compute the posterior $p(Y|X)$ directly, training the CRF

to discriminate among the possible tag sequences:

$$\hat{Y} = \operatorname*{argmax}_{Y \in \mathcal{Y}} P(Y|X) \tag{17.22}$$

However, the CRF does not compute a probability for each tag at each time step. Instead, at each time step the CRF computes log-linear functions over a set of relevant features, and these local features are aggregated and normalized to produce a global probability for the whole sequence.

Let's introduce the CRF more formally, again using X and Y as the input and output sequences. A CRF is a log-linear model that assigns a probability to an entire output (tag) sequence Y, out of all possible sequences $\mathcal{Y}$, given the entire input (word) sequence X. We can think of a CRF as like a giant sequential version of the multinomial logistic regression algorithm we saw for text categorization. Recall that we introduced the feature function f in regular multinomial logistic regression for text categorization as a function of a tuple: the input text x and a single class y (page 80). In a CRF, we're dealing with a sequence, so the function F maps an entire input sequence X and an entire output sequence Y to a feature vector. Let's assume we have K features, with a weight w_k for each feature F_k:

$$p(Y|X) = \frac{\exp\left(\sum_{k=1}^{K} w_k F_k(X,Y)\right)}{\sum_{Y' \in \mathcal{Y}} \exp\left(\sum_{k=1}^{K} w_k F_k(X,Y')\right)} \tag{17.23}$$

It's common to also describe the same equation by pulling out the denominator into a function Z(X):

$$p(Y|X) = \frac{1}{Z(X)} \exp\left(\sum_{k=1}^{K} w_k F_k(X,Y)\right) \tag{17.24}$$

$$Z(X) = \sum_{Y' \in \mathcal{Y}} \exp\left(\sum_{k=1}^{K} w_k F_k(X,Y')\right) \tag{17.25}$$

We'll call these K functions $F_k(X,Y)$ **global features**, since each one is a property of the entire input sequence X and output sequence Y. We compute them by decomposing into a sum of **local** features for each position i in Y:

$$F_k(X,Y) = \sum_{i=1}^{n} f_k(y_{i-1}, y_i, X, i) \tag{17.26}$$

Each of these local features f_k in a linear-chain CRF is allowed to make use of the current output token y_i, the previous output token y_{i-1}, the entire input string X (or any subpart of it), and the current position i. This constraint to only depend on the current and previous output tokens y_i and y_{i-1} are what characterizes a **linear chain CRF**. As we will see, this limitation makes it possible to use versions of the efficient Viterbi and Forward-Backwards algorithms from the HMM. A general CRF, by contrast, allows a feature to make use of any output token, and are thus necessary for tasks in which the decision depend on distant output tokens, like y_{i-4}. General CRFs require more complex inference, and are less commonly used for language processing.

linear chain CRF

17.5.1 Features in a CRF POS Tagger

Let's look at some of these features in detail, since the reason to use a discriminative sequence model is that it's easier to incorporate a lot of features.[2]

Again, in a linear-chain CRF, each local feature f_k at position i can depend on any information from: (y_{i-1}, y_i, X, i). So some legal features representing common situations might be the following:

$$\mathbb{1}\{x_i = \textit{the},\ y_i = \text{DET}\}$$
$$\mathbb{1}\{y_i = \text{PROPN},\ x_{i+1} = \textit{Street},\ y_{i-1} = \text{NUM}\}$$
$$\mathbb{1}\{y_i = \text{VERB},\ y_{i-1} = \text{AUX}\}$$

For simplicity, we'll assume all CRF features take on the value 1 or 0. Above, we explicitly use the notation $\mathbb{1}\{x\}$ to mean "1 if x is true, and 0 otherwise". From now on, we'll leave off the $\mathbb{1}$ when we define features, but you can assume each feature has it there implicitly.

feature templates

Although the idea of what features to use is done by the system designer by hand, the specific features are automatically populated by using **feature templates** as we briefly mentioned in Chapter 4. Here are some templates that only use information from (y_{i-1}, y_i, X, i):

$$\langle y_i, x_i \rangle, \langle y_i, y_{i-1} \rangle, \langle y_i, x_{i-1}, x_{i+2} \rangle$$

These templates automatically populate the set of features from every instance in the training and test set. Thus for our example *Janet/NNP will/MD back/VB the/DT bill/NN*, when x_i is the word *back*, the following features would be generated and have the value 1 (we've assigned them arbitrary feature numbers):

f_{3743}: y_i = VB and x_i = back
f_{156}: y_i = VB and y_{i-1} = MD
f_{99732}: y_i = VB and x_{i-1} = will and x_{i+2} = bill

word shape

It's also important to have features that help with unknown words. One of the most important is **word shape** features, which represent the abstract letter pattern of the word by mapping lower-case letters to 'x', upper-case to 'X', numbers to 'd', and retaining punctuation. Thus for example I.M.F. would map to X.X.X. and DC10-30 would map to XXdd-dd. A second class of shorter word shape features is also used. In these features consecutive character types are removed, so words in all caps map to X, words with initial-caps map to Xx, DC10-30 would be mapped to Xd-d but I.M.F would still map to X.X.X. Prefix and suffix features are also useful. In summary, here are some sample feature templates that help with unknown words:

x_i contains a particular prefix (perhaps from all prefixes of length ≤ 2)
x_i contains a particular suffix (perhaps from all suffixes of length ≤ 2)
x_i's word shape
x_i's short word shape

For example the word *well-dressed* might generate the following non-zero valued feature values:

[2] Because in HMMs all computation is based on the two probabilities $P(\text{tag}|\text{tag})$ and $P(\text{word}|\text{tag})$, if we want to include some source of knowledge into the tagging process, we must find a way to encode the knowledge into one of these two probabilities. Each time we add a feature we have to do a lot of complicated conditioning which gets harder and harder as we have more and more such features.

prefix(x_i) = `w`
prefix(x_i) = `we`
suffix(x_i) = `ed`
suffix(x_i) = `d`
word-shape(x_i) = `xxxx-xxxxxxx`
short-word-shape(x_i) = `x-x`

The known-word templates are computed for every word seen in the training set; the unknown word features can also be computed for all words in training, or only on training words whose frequency is below some threshold. The result of the known-word templates and word-signature features is a very large set of features. Generally a feature cutoff is used in which features are thrown out if they have count < 5 in the training set.

Remember that in a CRF we don't learn weights for each of these local features f_k. Instead, we first sum the values of each local feature (for example feature f_{3743}) over the entire sentence, to create each global feature (for example F_{3743}). It is those global features that will then be multiplied by weight w_{3743}. Thus for training and inference there is always a fixed set of K features with K weights, even though the length of each sentence is different.

17.5.2 Features for CRF Named Entity Recognizers

A CRF for NER makes use of very similar features to a POS tagger, as shown in Figure 17.15.

identity of w_i, identity of neighboring words
embeddings for w_i, embeddings for neighboring words
part of speech of w_i, part of speech of neighboring words
presence of w_i in a **gazetteer**
w_i contains a particular prefix (from all prefixes of length ≤ 4)
w_i contains a particular suffix (from all suffixes of length ≤ 4)
word shape of w_i, word shape of neighboring words
short word shape of w_i, short word shape of neighboring words
gazetteer features

Figure 17.15 Typical features for a feature-based NER system.

gazetteer

One feature that is especially useful for locations is a **gazetteer**, a list of place names, often providing millions of entries for locations with detailed geographical and political information.[3] This can be implemented as a binary feature indicating a phrase appears in the list. Other related resources like **name-lists**, for example from the United States Census Bureau[4], can be used, as can other entity dictionaries like lists of corporations or products, although they may not be as helpful as a gazetteer (Mikheev et al., 1999).

The sample named entity token *L'Occitane* would generate the following non-zero valued feature values (assuming that *L'Occitane* is neither in the gazetteer nor the census).

[3] www.geonames.org

[4] www.census.gov

prefix(x_i) = L	suffix(x_i) = tane
prefix(x_i) = L'	suffix(x_i) = ane
prefix(x_i) = L'O	suffix(x_i) = ne
prefix(x_i) = L'Oc	suffix(x_i) = e
word-shape(x_i) = X'Xxxxxxx	short-word-shape(x_i) = X'Xx

Figure 17.16 illustrates the result of adding part-of-speech tags and some shape information to our earlier example.

Words	POS	Short shape	Gazetteer	BIO Label
Jane	NNP	Xx	0	B-PER
Villanueva	NNP	Xx	1	I-PER
of	IN	x	0	O
United	NNP	Xx	0	B-ORG
Airlines	NNP	Xx	0	I-ORG
Holding	NNP	Xx	0	I-ORG
discussed	VBD	x	0	O
the	DT	x	0	O
Chicago	NNP	Xx	1	B-LOC
route	NN	x	0	O
.	.	.	0	O

Figure 17.16 Some NER features for a sample sentence, assuming that Chicago and Villanueva are listed as locations in a gazetteer. We assume features only take on the values 0 or 1, so the first POS feature, for example, would be represented as $\mathbb{1}\{\text{POS} = \text{NNP}\}$.

17.5.3 Inference and Training for CRFs

How do we find the best tag sequence $\hat{Y}$ for a given input X? We start with Eq. 17.22:

$$\begin{aligned}
\hat{Y} &= \operatorname*{argmax}_{Y \in \mathcal{Y}} P(Y|X) \\
&= \operatorname*{argmax}_{Y \in \mathcal{Y}} \frac{1}{Z(X)} \exp\left(\sum_{k=1}^{K} w_k F_k(X,Y)\right) && (17.27) \\
&= \operatorname*{argmax}_{Y \in \mathcal{Y}} \exp\left(\sum_{k=1}^{K} w_k \sum_{i=1}^{n} f_k(y_{i-1}, y_i, X, i)\right) && (17.28) \\
&= \operatorname*{argmax}_{Y \in \mathcal{Y}} \sum_{k=1}^{K} w_k \sum_{i=1}^{n} f_k(y_{i-1}, y_i, X, i) && (17.29) \\
&= \operatorname*{argmax}_{Y \in \mathcal{Y}} \sum_{i=1}^{n} \sum_{k=1}^{K} w_k f_k(y_{i-1}, y_i, X, i) && (17.30)
\end{aligned}$$

We can ignore the exp function and the denominator $Z(X)$, as we do above, because exp doesn't change the argmax, and the denominator $Z(X)$ is constant for a given observation sequence X.

How should we decode to find this optimal tag sequence $\hat{y}$? Just as with HMMs, we'll turn to the Viterbi algorithm, which works because, like the HMM, the linear-chain CRF depends at each timestep on only one previous output token y_{i-1}.

Concretely, this involves filling an $N \times T$ array with the appropriate values, maintaining backpointers as we proceed. As with HMM Viterbi, when the table is filled, we simply follow pointers back from the maximum value in the final column to retrieve the desired set of labels.

The requisite changes from HMM Viterbi have to do only with how we fill each cell. Recall from Eq. 17.19 that the recursive step of the Viterbi equation computes the Viterbi value of time t for state j as

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(i)\, a_{ij}\, b_j(o_t); \quad 1 \le j \le N, 1 < t \le T \tag{17.31}$$

which is the HMM implementation of

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(i)\, P(s_j|s_i)\, P(o_t|s_j) \quad 1 \le j \le N, 1 < t \le T \tag{17.32}$$

The CRF requires only a slight change to this latter formula, replacing the a and b prior and likelihood probabilities with the CRF features:

$$v_t(j) = \max_{i=1}^{N} \left[v_{t-1}(i) + \sum_{k=1}^{K} w_k f_k(y_{t-1}, y_t, X, t) \right] \quad 1 \le j \le N, 1 < t \le T \tag{17.33}$$

Learning in CRFs relies on the same supervised learning algorithms we presented for logistic regression. Given a sequence of observations, feature functions, and corresponding outputs, we use stochastic gradient descent to train the weights to maximize the log-likelihood of the training corpus. The local nature of linear-chain CRFs means that the forward-backward algorithm introduced for HMMs in Appendix A can be extended to a CRF version that will efficiently compute the necessary derivatives. As with logistic regression, L1 or L2 regularization is important.

17.6 Evaluation of Named Entity Recognition

Part-of-speech taggers are evaluated by the standard metric of **accuracy**. Named entity recognizers are evaluated by **recall**, **precision**, and $\mathbf{F_1}$ **measure**. Recall that recall is the ratio of the number of correctly labeled responses to the total that should have been labeled; precision is the ratio of the number of correctly labeled responses to the total labeled; and F-measure is the harmonic mean of the two.

To know if the difference between the F_1 scores of two NER systems is a significant difference, we use the paired bootstrap test, or the similar randomization test (Section 4.11).

For named entity tagging, the *entity* rather than the word is the unit of response. Thus in the example in Fig. 17.16, the two entities *Jane Villanueva* and *United Airlines Holding* and the non-entity *discussed* would each count as a single response.

The fact that named entity tagging has a segmentation component which is not present in tasks like text categorization or part-of-speech tagging causes some problems with evaluation. For example, a system that labeled *Jane* but not *Jane Villanueva* as a person would cause two errors, a false positive for O and a false negative for I-PER. In addition, using entities as the unit of response but words as the unit of training means that there is a mismatch between the training and test conditions.

17.7 Further Details

In this section we summarize a few remaining details of the data and models for part-of-speech tagging and NER, beginning with data. Since the algorithms we have

presented are supervised, having labeled data is essential for training and testing. A wide variety of datasets exist for part-of-speech tagging and/or NER. The Universal Dependencies (UD) dataset (de Marneffe et al., 2021) has POS tagged corpora in over a hundred languages, as do the Penn Treebanks in English, Chinese, and Arabic. OntoNotes has corpora labeled for named entities in English, Chinese, and Arabic (Hovy et al., 2006). Named entity tagged corpora are also available in particular domains, such as for biomedical (Bada et al., 2012) and literary text (Bamman et al., 2019).

17.7.1 Rule-based Methods

While machine learned (neural or CRF) sequence models are the norm in academic research, commercial approaches to NER are often based on pragmatic combinations of lists and rules, with some smaller amount of supervised machine learning (Chiticariu et al., 2013). For example in the IBM System T architecture, a user specifies declarative constraints for tagging tasks in a formal query language that includes regular expressions, dictionaries, semantic constraints, and other operators, which the system compiles into an efficient extractor (Chiticariu et al., 2018).

One common approach is to make repeated rule-based passes over a text, starting with rules with very high precision but low recall, and, in subsequent stages, using machine learning methods that take the output of the first pass into account (an approach first worked out for coreference (Lee et al., 2017a)):

1. First, use high-precision rules to tag unambiguous entity mentions.
2. Then, search for substring matches of the previously detected names.
3. Use application-specific name lists to find likely domain-specific mentions.
4. Finally, apply supervised sequence labeling techniques that use tags from previous stages as additional features.

Rule-based methods were also the earliest methods for part-of-speech tagging. Rule-based taggers like the English Constraint Grammar system (Karlsson et al. 1995, Voutilainen 1999) use a two-stage formalism invented in the 1950s and 1960s: (1) a morphological analyzer with tens of thousands of word stem entries returns all parts of speech for a word, then (2) a large set of thousands of constraints are applied to the input sentence to rule out parts of speech inconsistent with the context.

17.7.2 POS Tagging for Morphologically Rich Languages

Augmentations to tagging algorithms become necessary when dealing with languages with rich morphology like Czech, Hungarian and Turkish.

These productive word-formation processes result in a large vocabulary for these languages: a 250,000 word token corpus of Hungarian has more than twice as many word types as a similarly sized corpus of English (Oravecz and Dienes, 2002), while a 10 million word token corpus of Turkish contains four times as many word types as a similarly sized English corpus (Hakkani-Tür et al., 2002). Large vocabularies mean many unknown words, and these unknown words cause significant performance degradations in a wide variety of languages (including Czech, Slovene, Estonian, and Romanian) (Hajič, 2000).

Highly inflectional languages also have much more information than English coded in word morphology, like **case** (nominative, accusative, genitive) or **gender** (masculine, feminine). Because this information is important for tasks like parsing and coreference resolution, part-of-speech taggers for morphologically rich lan-

guages need to label words with case and gender information. Tagsets for morphologically rich languages are therefore sequences of morphological tags rather than a single primitive tag. Here's a Turkish example, in which the word *izin* has three possible morphological/part-of-speech tags and meanings (Hakkani-Tür et al., 2002):

1. Yerdeki **izin** temizlenmesi gerek. iz + `Noun+A3sg+Pnon+Gen`
   The trace on the floor should be cleaned.
2. Üzerinde parmak **izin** kalmiş. iz + `Noun+A3sg+P2sg+Nom`
   Your finger **print** is left on (it).
3. Içeri girmek için **izin** alman gerekiyor. izin + `Noun+A3sg+Pnon+Nom`
   You need **permission** to enter.

Using a morphological parse sequence like `Noun+A3sg+Pnon+Gen` as the part-of-speech tag greatly increases the number of parts of speech, and so tagsets can be 4 to 10 times larger than the 50–100 tags we have seen for English. With such large tagsets, each word needs to be morphologically analyzed to generate the list of possible morphological tag sequences (part-of-speech tags) for the word. The role of the tagger is then to disambiguate among these tags. This method also helps with unknown words since morphological parsers can accept unknown stems and still segment the affixes properly.

17.8 Summary

This chapter introduced **parts of speech** and **named entities**, and the tasks of **part-of-speech tagging** and **named entity recognition**:

- Languages generally have a small set of **closed class** words that are highly frequent, ambiguous, and act as **function words**, and **open-class** words like **nouns**, **verbs**, **adjectives**. Various part-of-speech **tagsets** exist, of between 40 and 200 tags.
- **Part-of-speech tagging** is the process of assigning a part-of-speech label to each of a sequence of words.
- **Named entities** are words for proper nouns referring mainly to people, places, and organizations, but extended to many other types that aren't strictly entities or even proper nouns.
- Two common approaches to **sequence modeling** are a **generative** approach, **HMM** tagging, and a **discriminative** approach, **CRF** tagging. We will see a neural approach in following chapters.
- The probabilities in HMM taggers are estimated by maximum likelihood estimation on tag-labeled training corpora. The Viterbi algorithm is used for **decoding**, finding the most likely tag sequence
- **Conditional Random Fields** or **CRF taggers** train a log-linear model that can choose the best tag sequence given an observation sequence, based on features that condition on the output tag, the prior output tag, the entire input sequence, and the current timestep. They use the Viterbi algorithm for inference, to choose the best sequence of tags, and a version of the Forward-Backward algorithm (see Appendix A) for training,

Historical Notes

What is probably the earliest part-of-speech tagger was part of the parser in Zellig Harris's Transformations and Discourse Analysis Project (TDAP), implemented between June 1958 and July 1959 at the University of Pennsylvania (Harris, 1962), although earlier systems had used part-of-speech dictionaries. TDAP used 14 handwritten rules for part-of-speech disambiguation; the use of part-of-speech tag sequences and the relative frequency of tags for a word prefigures modern algorithms. The parser was implemented essentially as a cascade of finite-state transducers; see Joshi and Hopely (1999) and Karttunen (1999) for a reimplementation.

The Computational Grammar Coder (CGC) of Klein and Simmons (1963) had three components: a lexicon, a morphological analyzer, and a context disambiguator. The small 1500-word lexicon listed only function words and other irregular words. The morphological analyzer used inflectional and derivational suffixes to assign part-of-speech classes. These were run over words to produce candidate parts of speech which were then disambiguated by a set of 500 context rules by relying on surrounding islands of unambiguous words. For example, one rule said that between an ARTICLE and a VERB, the only allowable sequences were ADJ-NOUN, NOUN-ADVERB, or NOUN-NOUN. The TAGGIT tagger (Greene and Rubin, 1971) used the same architecture as Klein and Simmons (1963), with a bigger dictionary and more tags (87). TAGGIT was applied to the Brown corpus and, according to Francis and Kučera (1982, p. 9), accurately tagged 77% of the corpus; the remainder of the Brown corpus was then tagged by hand. All these early algorithms were based on a two-stage architecture in which a dictionary was first used to assign each word a set of potential parts of speech, and then lists of handwritten disambiguation rules winnowed the set down to a single part of speech per word.

Probabilities were used in tagging by Stolz et al. (1965) and a complete probabilistic tagger with Viterbi decoding was sketched by Bahl and Mercer (1976). The Lancaster-Oslo/Bergen (LOB) corpus, a British English equivalent of the Brown corpus, was tagged in the early 1980's with the CLAWS tagger (Marshall 1983; Marshall 1987; Garside 1987), a probabilistic algorithm that approximated a simplified HMM tagger. The algorithm used tag bigram probabilities, but instead of storing the word likelihood of each tag, the algorithm marked tags either as *rare* ($P(\text{tag}|\text{word}) < .01$) *infrequent* ($P(\text{tag}|\text{word}) < .10$) or *normally frequent* ($P(\text{tag}|\text{word}) > .10$).

DeRose (1988) developed a quasi-HMM algorithm, including the use of dynamic programming, although computing $P(t|w)P(w)$ instead of $P(w|t)P(w)$. The same year, the probabilistic PARTS tagger of Church 1988, 1989 was probably the first implemented HMM tagger, described correctly in Church (1989), although Church (1988) also described the computation incorrectly as $P(t|w)P(w)$ instead of $P(w|t)P(w)$. Church (p.c.) explained that he had simplified for pedagogical purposes because using the probability $P(t|w)$ made the idea seem more understandable as "storing a lexicon in an almost standard form".

Later taggers explicitly introduced the use of the hidden Markov model (Kupiec 1992; Weischedel et al. 1993; Schütze and Singer 1994). Merialdo (1994) showed that fully unsupervised EM didn't work well for the tagging task and that reliance on hand-labeled data was important. Charniak et al. (1993) showed the importance of the most frequent tag baseline; the 92.3% number we give above was from Abney et al. (1999). See Brants (2000) for HMM tagger implementation details, including the extension to trigram contexts, and the use of sophisticated unknown word features; its performance is still close to state of the art taggers.

Log-linear models for POS tagging were introduced by Ratnaparkhi (1996), who introduced a system called MXPOST which implemented a maximum entropy Markov model (MEMM), a slightly simpler version of a CRF. Around the same time, sequence labelers were applied to the task of named entity tagging, first with HMMs (Bikel et al., 1997) and MEMMs (McCallum et al., 2000), and then once CRFs were developed (Lafferty et al. 2001), they were also applied to NER (McCallum and Li, 2003). A wide exploration of features followed (Zhou et al., 2005). Neural approaches to NER mainly follow from the pioneering results of Collobert et al. (2011), who applied a CRF on top of a convolutional net. BiLSTMs with word and character-based embeddings as input followed shortly and became a standard neural algorithm for NER (Huang et al. 2015, Ma and Hovy 2016, Lample et al. 2016) followed by the more recent use of Transformers and BERT.

The idea of using letter suffixes for unknown words is quite old; the early Klein and Simmons (1963) system checked all final letter suffixes of lengths 1-5. The unknown word features described on page 398 come mainly from Ratnaparkhi (1996), with augmentations from Toutanova et al. (2003) and Manning (2011).

State of the art POS taggers use neural algorithms, either bidirectional RNNs or Transformers like BERT; see Chapter 13 to Chapter 9. HMM (Brants 2000; Thede and Harper 1999) and CRF tagger accuracies are likely just a tad lower.

Manning (2011) investigates the remaining 2.7% of errors in a high-performing tagger (Toutanova et al., 2003). He suggests that a third or half of these remaining errors are due to errors or inconsistencies in the training data, a third might be solvable with richer linguistic models, and for the remainder the task is underspecified or unclear.

Supervised tagging relies heavily on in-domain training data hand-labeled by experts. Ways to relax this assumption include unsupervised algorithms for clustering words into part-of-speech-like classes, summarized in Christodoulopoulos et al. (2010), and ways to combine labeled and unlabeled data, for example by co-training (Clark et al. 2003; Søgaard 2010).

See Householder (1995) for historical notes on parts of speech, and Sampson (1987) and Garside et al. (1997) on the provenance of the Brown and other tagsets.

Exercises

17.1 Find one tagging error in each of the following sentences that are tagged with the Penn Treebank tagset:

1. I/PRP need/VBP a/DT flight/NN from/IN Atlanta/NN
2. Does/VBZ this/DT flight/NN serve/VB dinner/NNS
3. I/PRP have/VB a/DT friend/NN living/VBG in/IN Denver/NNP
4. Can/VBP you/PRP list/VB the/DT nonstop/JJ afternoon/NN flights/NNS

17.2 Use the Penn Treebank tagset to tag each word in the following sentences from Damon Runyon's short stories. You may ignore punctuation. Some of these are quite difficult; do your best.

1. It is a nice night.
2. This crap game is over a garage in Fifty-second Street...
3. ...Nobody ever takes the newspapers she sells ...
4. He is a tall, skinny guy with a long, sad, mean-looking kisser, and a mournful voice.

5. . . . I am sitting in Mindy's restaurant putting on the gefillte fish, which is a dish I am very fond of, . . .
6. When a guy and a doll get to taking peeks back and forth at each other, why there you are indeed.

17.3 Now compare your tags from the previous exercise with one or two friend's answers. On which words did you disagree the most? Why?

17.4 Implement the "most likely tag" baseline. Find a POS-tagged training set, and use it to compute for each word the tag that maximizes $p(t|w)$. You will need to implement a simple tokenizer to deal with sentence boundaries. Start by assuming that all unknown words are NN and compute your error rate on known and unknown words. Now write at least five rules to do a better job of tagging unknown words, and show the difference in error rates.

17.5 Build a bigram HMM tagger. You will need a part-of-speech-tagged corpus. First split the corpus into a training set and test set. From the labeled training set, train the transition and observation probabilities of the HMM tagger directly on the hand-tagged data. Then implement the Viterbi algorithm so you can decode a test sentence. Now run your algorithm on the test set. Report its error rate and compare its performance to the most frequent tag baseline.

17.6 Do an error analysis of your tagger. Build a confusion matrix and investigate the most frequent errors. Propose some features for improving the performance of your tagger on these errors.

17.7 Develop a set of regular expressions to recognize the character shape features described on page 398.

17.8 The BIO and other labeling schemes given in this chapter aren't the only possible one. For example, the B tag can be reserved only for those situations where an ambiguity exists between adjacent entities. Propose a new set of BIO tags for use with your NER system. Experiment with it and compare its performance with the schemes presented in this chapter.

17.9 Names of works of art (books, movies, video games, etc.) are quite different from the kinds of named entities we've discussed in this chapter. Collect a list of names of works of art from a particular category from a Web-based source (e.g., gutenberg.org, amazon.com, imdb.com, etc.). Analyze your list and give examples of ways that the names in it are likely to be problematic for the techniques described in this chapter.

17.10 Develop an NER system specific to the category of names that you collected in the last exercise. Evaluate your system on a collection of text likely to contain instances of these named entities.

CHAPTER

18 Context-Free Grammars and Constituency Parsing

Because the Night by Bruce Springsteen and Patti Smith
The Fire Next Time by James Baldwin
If on a winter's night a traveler by Italo Calvino
Love Actually by Richard Curtis
Suddenly Last Summer by Tennessee Williams
A Scanner Darkly by Philip K. Dick

Six titles that are not constituents, from Geoffrey K. Pullum on Language Log (who was pointing out their incredible rarity).

One morning I shot an elephant in my pajamas.
How he got into my pajamas I don't know.

Groucho Marx, *Animal Crackers*, 1930

The study of grammar has an ancient pedigree. The grammar of Sanskrit was described by the Indian grammarian Pāṇini sometime between the 7th and 4th centuries BCE, in his famous treatise the Aṣṭādhyāyī ('8 books'). And our word **syntax** comes from the Greek *sýntaxis*, meaning "setting out together or arrangement", and refers to the way words are arranged together. We have seen syntactic notions in previous chapters like the use of part-of-speech categories (Chapter 17). In this chapter and the next one we introduce formal models for capturing more sophisticated notions of grammatical structure and algorithms for parsing these structures.

syntax

Our focus in this chapter is **context-free grammars** and the **CKY algorithm** for parsing them. Context-free grammars are the backbone of many formal models of the syntax of natural language (and, for that matter, of computer languages). Syntactic parsing is the task of assigning a syntactic structure to a sentence. Parse trees (whether for context-free grammars or for the dependency or CCG formalisms we introduce in following chapters) can be used in applications such as **grammar checking**: sentence that cannot be parsed may have grammatical errors (or at least be hard to read). Parse trees can be an intermediate stage of representation for **formal semantic analysis**. And parsers and the grammatical structure they assign a sentence are a useful text analysis tool for text data science applications that require modeling the relationship of elements in sentences.

In this chapter we introduce context-free grammars, give a small sample grammar of English, introduce more formal definitions of context-free grammars and grammar normal form, and talk about **treebanks**: corpora that have been annotated with syntactic structure. We then discuss parse ambiguity and the problems it presents, and turn to parsing itself, giving the famous Cocke-Kasami-Younger (CKY) algorithm (Kasami 1965, Younger 1967), the standard dynamic programming approach to syntactic parsing. The CKY algorithm returns an efficient representation of the set of parse trees for a sentence, but doesn't tell us **which** parse tree is the right one. For that, we need to augment CKY with scores for each possible constituent. We'll see how to do this with neural span-based parsers. Finally, we'll introduce the standard set of metrics for evaluating parser accuracy.

18.1 Constituency

Syntactic constituency is the idea that groups of words can behave as single units, or constituents. Part of developing a grammar involves building an inventory of the constituents in the language. How do words group together in English? Consider the **noun phrase**, a sequence of words surrounding at least one noun. Here are some examples of noun phrases (thanks to Damon Runyon):

noun phrase

Harry the Horse	a high-class spot such as Mindy's
the Broadway coppers	the reason he comes into the Hot Box
they	three parties from Brooklyn

What evidence do we have that these words group together (or "form constituents")? One piece of evidence is that they can all appear in similar syntactic environments, for example, before a verb.

three parties from Brooklyn *arrive...*
a high-class spot such as Mindy's *attracts...*
the Broadway coppers *love...*
they *sit*

But while the whole noun phrase can occur before a verb, this is not true of each of the individual words that make up a noun phrase. The following are not grammatical sentences of English (recall that we use an asterisk (*) to mark fragments that are not grammatical English sentences):

*from *arrive...*	*as *attracts...*
*the *is...*	*spot *sat...*

Thus, to correctly describe facts about the ordering of these words in English, we must be able to say things like "*Noun Phrases can occur before verbs*". Let's now see how to do this in a more formal way!

18.2 Context-Free Grammars

A widely used formal system for modeling constituent structure in natural language is the **context-free grammar**, or **CFG**. Context-free grammars are also called **phrase-structure grammars**, and the formalism is equivalent to **Backus-Naur form**, or **BNF**. The idea of basing a grammar on constituent structure dates back to the psychologist Wilhelm Wundt (1900) but was not formalized until Chomsky (1956) and, independently, Backus (1959).

CFG

A context-free grammar consists of a set of **rules** or **productions**, each of which expresses the ways that symbols of the language can be grouped and ordered together, and a **lexicon** of words and symbols. For example, the following productions express that an **NP** (or **noun phrase**) can be composed of either a *ProperNoun* or a determiner (*Det*) followed by a *Nominal*; a *Nominal* in turn can consist of one or

rules

lexicon

NP

more *Nouns*.[1]

$$\begin{aligned} NP &\rightarrow Det\ Nominal \\ NP &\rightarrow ProperNoun \\ Nominal &\rightarrow Noun \mid Nominal\ Noun \end{aligned}$$

Context-free rules can be hierarchically embedded, so we can combine the previous rules with others, like the following, that express facts about the lexicon:

$$\begin{aligned} Det &\rightarrow a \\ Det &\rightarrow the \\ Noun &\rightarrow flight \end{aligned}$$

terminal

non-terminal

The symbols that are used in a CFG are divided into two classes. The symbols that correspond to words in the language ("the", "nightclub") are called **terminal** symbols; the lexicon is the set of rules that introduce these terminal symbols. The symbols that express abstractions over these terminals are called **non-terminals**. In each context-free rule, the item to the right of the arrow ($\rightarrow$) is an ordered list of one or more terminals and non-terminals; to the left of the arrow is a single non-terminal symbol expressing some cluster or generalization. The non-terminal associated with each word in the lexicon is its lexical category, or part of speech.

A CFG can be thought of in two ways: as a device for generating sentences and as a device for assigning a structure to a given sentence. Viewing a CFG as a generator, we can read the $\rightarrow$ arrow as "rewrite the symbol on the left with the string of symbols on the right".

So starting from the symbol:	*NP*
we can use our first rule to rewrite *NP* as:	*Det Nominal*
and then rewrite *Nominal* as:	*Noun*
and finally rewrite these parts-of-speech as:	*a flight*

We say the string *a flight* can be derived from the non-terminal *NP*. Thus, a CFG can be used to generate a set of strings. This sequence of rule expansions is called a **derivation** of the string of words. It is common to represent a derivation by a **parse tree** (commonly shown inverted with the root at the top). Figure 18.1 shows the tree representation of this derivation.

derivation

parse tree

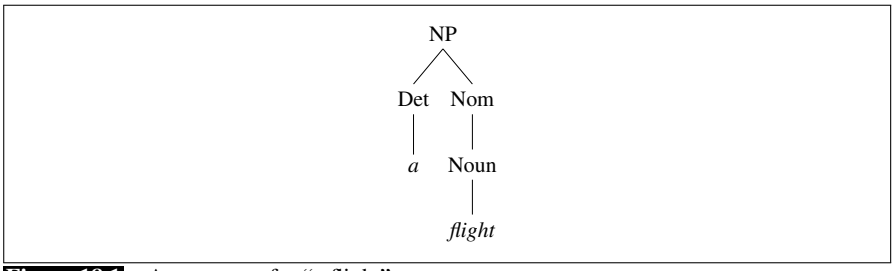


Figure 18.1 A parse tree for "a flight".

dominates

In the parse tree shown in Fig. 18.1, we can say that the node *NP* **dominates** all the nodes in the tree (*Det*, *Nom*, *Noun*, *a*, *flight*). We can say further that it immediately dominates the nodes *Det* and *Nom*.

start symbol

The formal language defined by a CFG is the set of strings that are derivable from the designated **start symbol**. Each grammar must have one designated start

[1] When talking about these rules we can pronounce the rightarrow $\rightarrow$ as "goes to", and so we might read the first rule above as "NP goes to Det Nominal".

symbol, which is often called *S*. Since context-free grammars are often used to define sentences, *S* is usually interpreted as the "sentence" node, and the set of strings that are derivable from *S* is the set of sentences in some simplified version of English.

verb phrase

Let's add a few additional rules to our inventory. The following rule expresses the fact that a sentence can consist of a noun phrase followed by a **verb phrase**:

$$S \rightarrow NP\ VP \quad \text{I prefer a morning flight}$$

A verb phrase in English consists of a verb followed by assorted other things; for example, one kind of verb phrase consists of a verb followed by a noun phrase:

$$VP \rightarrow Verb\ NP \quad \text{prefer a morning flight}$$

Or the verb may be followed by a noun phrase and a prepositional phrase:

$$VP \rightarrow Verb\ NP\ PP \quad \text{leave Boston in the morning}$$

Or the verb phrase may have a verb followed by a prepositional phrase alone:

$$VP \rightarrow Verb\ PP \quad \text{leaving on Thursday}$$

A prepositional phrase generally has a preposition followed by a noun phrase. For example, a common type of prepositional phrase in the ATIS corpus is used to indicate location or direction:

$$PP \rightarrow Preposition\ NP \quad \text{from Los Angeles}$$

The *NP* inside a *PP* need not be a location; *PPs* are often used with times and dates, and with other nouns as well; they can be arbitrarily complex. Here are ten examples from the ATIS corpus:

to Seattle	on these flights
in Minneapolis	about the ground transportation in Chicago
on Wednesday	of the round trip flight on United Airlines
in the evening	of the AP fifty seven flight
on the ninth of July	with a stopover in Nashville

Figure 18.2 gives a sample lexicon, and Fig. 18.3 summarizes the grammar rules we've seen so far, which we'll call $\mathcal{L}_0$. Note that we can use the or-symbol | to indicate that a non-terminal has alternate possible expansions.

Noun →	*flights* | *flight* | *breeze* | *trip* | *morning*
Verb →	*is* | *prefer* | *like* | *need* | *want* | *fly* | *do*
Adjective →	*cheapest* | *non-stop* | *first* | *latest* | *other* | *direct*
Pronoun →	*me* | *I* | *you* | *it*
Proper-Noun →	*Alaska* | *Baltimore* | *Los Angeles* | *Chicago* | *United* | *American*
Determiner →	*the* | *a* | *an* | *this* | *these* | *that*
Preposition →	*from* | *to* | *on* | *near* | *in*
Conjunction →	*and* | *or* | *but*

Figure 18.2 The lexicon for $\mathcal{L}_0$.

We can use this grammar to generate sentences of this "ATIS-language". We start with *S*, expand it to *NP VP*, then choose a random expansion of *NP* (let's say, to

Grammar Rules		Examples
S → *NP VP*		I + want a morning flight
NP → *Pronoun*		I
| *Proper-Noun*		Los Angeles
| *Det Nominal*		a + flight
Nominal → *Nominal Noun*		morning + flight
| *Noun*		flights
VP → *Verb*		do
| *Verb NP*		want + a flight
| *Verb NP PP*		leave + Boston + in the morning
| *Verb PP*		leaving + on Thursday
PP → *Preposition NP*		from + Los Angeles

Figure 18.3 The grammar for $\mathcal{L}_0$, with example phrases for each rule.

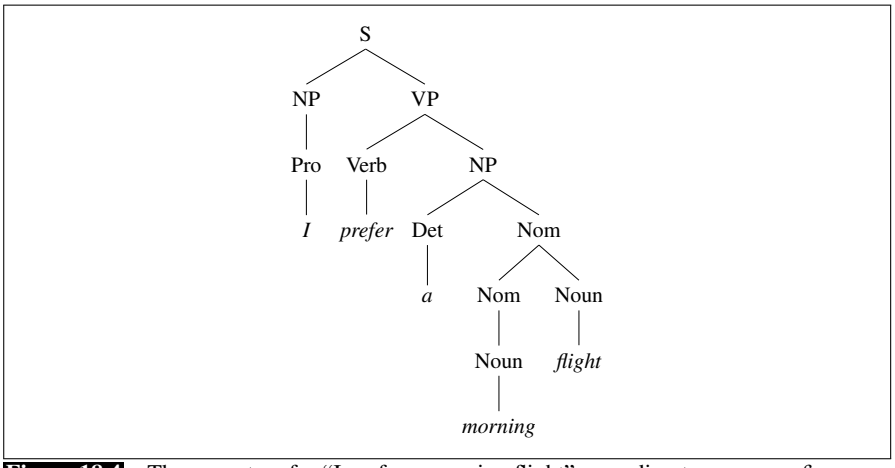


Figure 18.4 The parse tree for "I prefer a morning flight" according to grammar $\mathcal{L}_0$.

I), and a random expansion of *VP* (let's say, to *Verb NP*), and so on until we generate the string *I prefer a morning flight*. Figure 18.4 shows a parse tree that represents a complete derivation of *I prefer a morning flight*.

bracketed notation

We can also represent a parse tree in a more compact format called **bracketed notation**; here is the bracketed representation of the parse tree of Fig. 18.4:

(18.1) [$_S$ [$_{NP}$ [$_{Pro}$ I]] [$_{VP}$ [$_V$ prefer] [$_{NP}$ [$_{Det}$ a] [$_{Nom}$ [$_{Nom}$ [$_N$ morning]] [$_N$ flight]]]]]]

grammatical

ungrammatical

A CFG like that of $\mathcal{L}_0$ defines a formal language. Sentences (strings of words) that can be derived by a grammar are in the formal language defined by that grammar, and are called **grammatical** sentences. Sentences that cannot be derived by a given formal grammar are not in the language defined by that grammar and are referred to as **ungrammatical**. This hard line between "in" and "out" characterizes all formal languages but is only a very simplified model of how natural languages really work. This is because determining whether a given sentence is part of a given natural language (say, English) often depends on the context. In linguistics, the use of formal languages to model natural languages is called **generative grammar** since the language is defined by the set of possible sentences "generated" by the grammar. (Note that this is a different sense of the word 'generate' than when we talk about

generative grammar

language models generating text.)

18.2.1 Formal Definition of Context-Free Grammar

We conclude this section with a quick, formal description of a context-free grammar and the language it generates. A context-free grammar G is defined by four parameters: N, Σ, R, S (technically it is a "4-tuple").

N	a set of **non-terminal symbols** (or **variables**)
Σ	a set of **terminal symbols** (disjoint from N)
R	a set of **rules** or productions, each of the form $A \rightarrow \beta$, where A is a non-terminal, β is a string of symbols from the infinite set of strings $(\Sigma \cup N)^*$
S	a designated **start symbol** and a member of N

For the remainder of the book we adhere to the following conventions when discussing the formal properties of context-free grammars (as opposed to explaining particular facts about English or other languages).

Capital letters like A, B, and S	Non-terminals
S	The start symbol
Lower-case Greek letters like α, β, and γ	Strings drawn from $(\Sigma \cup N)^*$
Lower-case Roman letters like u, v, and w	Strings of terminals

A language is defined through the concept of derivation. One string derives another one if it can be rewritten as the second one by some series of rule applications. More formally, following Hopcroft and Ullman (1979),

directly derives

if $A \rightarrow \beta$ is a production of R and α and γ are any strings in the set $(\Sigma \cup N)^*$, then we say that $\alpha A \gamma$ **directly derives** $\alpha \beta \gamma$, or $\alpha A \gamma \Rightarrow \alpha \beta \gamma$.

Derivation is then a generalization of direct derivation:

Let $\alpha_1, \alpha_2, \ldots, \alpha_m$ be strings in $(\Sigma \cup N)^*, m \geq 1$, such that

$$\alpha_1 \Rightarrow \alpha_2, \alpha_2 \Rightarrow \alpha_3, \ldots, \alpha_{m-1} \Rightarrow \alpha_m$$

derives

We say that α_1 **derives** α_m, or $\alpha_1 \stackrel{*}{\Rightarrow} \alpha_m$.

We can then formally define the language $\mathcal{L}_G$ generated by a grammar G as the set of strings composed of terminal symbols that can be derived from the designated start symbol S.

$$\mathcal{L}_G = \{w | w \text{ is in } \Sigma^* \text{ and } S \stackrel{*}{\Rightarrow} w\}$$

syntactic parsing

The problem of mapping from a string of words to its parse tree is called **syntactic parsing**, as we'll see in Section 18.6.

18.3 Treebanks

treebank

A corpus in which every sentence is annotated with a parse tree is called a **treebank**.

Treebanks play an important role in parsing as well as in linguistic investigations of syntactic phenomena.

Treebanks are generally made by running a parser over each sentence and then having the resulting parse hand-corrected by human linguists. Figure 18.5 shows sentences from the **Penn Treebank** project, which includes various treebanks in English, Arabic, and Chinese. The Penn Treebank part-of-speech tagset was defined in Chapter 17, but we'll see minor formatting differences across treebanks. The use of LISP-style parenthesized notation for trees is extremely common and resembles the bracketed notation we saw earlier in (18.1). For those who are not familiar with it we show a standard node-and-line tree representation in Fig. 18.6.

Penn Treebank

```
((S
   (NP-SBJ (DT That)
     (JJ cold) (, ,)
     (JJ empty) (NN sky) )
   (VP (VBD was)
     (ADJP-PRD (JJ full)
       (PP (IN of)
         (NP (NN fire)
           (CC and)
           (NN light) ))))
   (. .) ))
```

(a)

```
((S
   (NP-SBJ The/DT flight/NN )
   (VP should/MD
     (VP arrive/VB
       (PP-TMP at/IN
         (NP eleven/CD a.m/RB ))
       (NP-TMP tomorrow/NN )))))
```

(b)

Figure 18.5 Parses from the LDC Treebank3 for (a) Brown and (b) ATIS sentences.

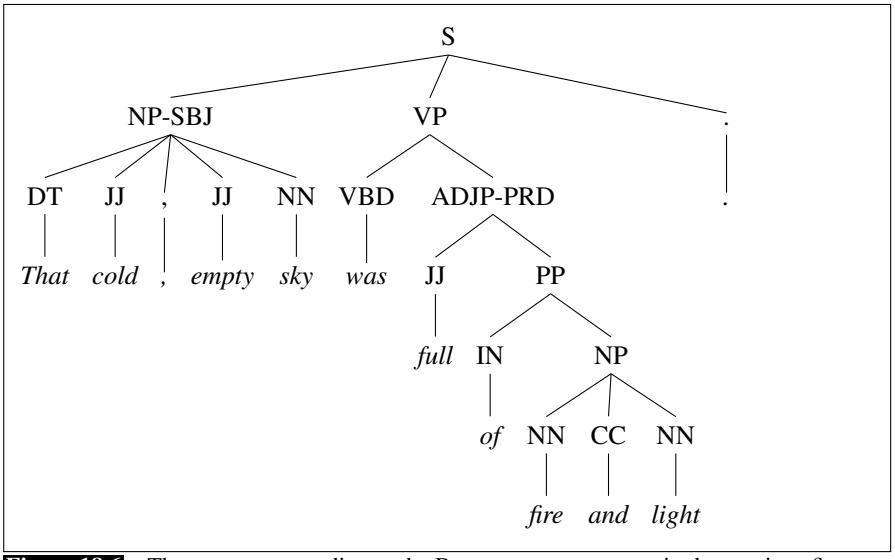


Figure 18.6 The tree corresponding to the Brown corpus sentence in the previous figure.

The sentences in a treebank implicitly constitute a grammar of the language. For example, from the parsed sentences in Fig. 18.5 we can extract the CFG rules shown in Fig. 18.7 (with rule suffixes (`-SBJ`) stripped for simplicity). The grammar used to parse the Penn Treebank is very flat, resulting in very many rules. For example,

Grammar	Lexicon
$S \rightarrow$ *NP VP .*	*DT* $\rightarrow$ *the* | *that*
$S \rightarrow$ *NP VP*	*JJ* $\rightarrow$ *cold* | *empty* | *full*
NP $\rightarrow$ *CD RB*	*NN* $\rightarrow$ *sky* | *fire* | *light* | *flight* | *tomorrow*
NP $\rightarrow$ *DT NN*	*CC* $\rightarrow$ *and*
NP $\rightarrow$ *NN CC NN*	*IN* $\rightarrow$ *of* | *at*
NP $\rightarrow$ *DT JJ , JJ NN*	*CD* $\rightarrow$ *eleven*
NP $\rightarrow$ *NN*	*RB* $\rightarrow$ *a.m.*
VP $\rightarrow$ *MD VP*	*VB* $\rightarrow$ *arrive*
VP $\rightarrow$ *VBD ADJP*	*VBD* $\rightarrow$ *was* | *said*
VP $\rightarrow$ *MD VP*	*MD* $\rightarrow$ *should* | *would*
VP $\rightarrow$ *VB PP NP*	
ADJP $\rightarrow$ *JJ PP*	
PP $\rightarrow$ *IN NP*	

Figure 18.7 CFG grammar rules and lexicon from the treebank sentences in Fig. 18.5.

among the approximately 4,500 different rules for expanding VPs are separate rules for PP sequences of any length and every possible arrangement of verb arguments:

```
VP → VBD PP
VP → VBD PP PP
VP → VBD PP PP PP
VP → VBD PP PP PP PP
VP → VB ADVP PP
VP → VB PP ADVP
VP → ADVP VB PP
```

18.4 Grammar Equivalence and Normal Form

A formal language is defined as a (possibly infinite) set of strings of words. This suggests that we could ask if two grammars are equivalent by asking if they generate the same set of strings. In fact, it is possible to have two distinct context-free grammars generate the same language. We say that two grammars are **strongly equivalent** if they generate the same set of strings *and* if they assign the same phrase structure to each sentence (allowing merely for renaming of the non-terminal symbols). Two grammars are **weakly equivalent** if they generate the same set of strings but do not assign the same phrase structure to each sentence.

strongly equivalent

weakly equivalent

normal form

Chomsky normal form

binary branching

It is sometimes useful to have a **normal form** for grammars, in which each of the productions takes a particular form. For example, a context-free grammar is in **Chomsky normal form** (CNF) (Chomsky, 1963) if it is ϵ-free and if in addition each production is either of the form $A \rightarrow B\ C$ or $A \rightarrow a$. That is, the right-hand side of each rule either has two non-terminal symbols or one terminal symbol. Chomsky normal form grammars are **binary branching**, that is they have binary trees (down to the prelexical nodes). We make use of this binary branching property in the CKY parsing algorithm in Section 18.6.

Any context-free grammar can be converted into a weakly equivalent Chomsky normal form grammar. For example, a rule of the form

$$A \rightarrow B\ C\ D$$

can be converted into the following two CNF rules (Exercise 18.1 asks the reader to

Grammar	Lexicon
$S \rightarrow NP\ VP$	$Det \rightarrow that \mid this \mid the \mid a$
$S \rightarrow Aux\ NP\ VP$	$Noun \rightarrow book \mid flight \mid meal \mid money$
$S \rightarrow VP$	$Verb \rightarrow book \mid include \mid prefer$
$NP \rightarrow Pronoun$	$Pronoun \rightarrow I \mid she \mid me$
$NP \rightarrow Proper\text{-}Noun$	$Proper\text{-}Noun \rightarrow Houston \mid United$
$NP \rightarrow Det\ Nominal$	$Aux \rightarrow does$
$Nominal \rightarrow Noun$	$Preposition \rightarrow from \mid to \mid on \mid near \mid through$
$Nominal \rightarrow Nominal\ Noun$	
$Nominal \rightarrow Nominal\ PP$	
$VP \rightarrow Verb$	
$VP \rightarrow Verb\ NP$	
$VP \rightarrow Verb\ NP\ PP$	
$VP \rightarrow Verb\ PP$	
$VP \rightarrow VP\ PP$	
$PP \rightarrow Preposition\ NP$	

Figure 18.8 The $\mathcal{L}_1$ miniature English grammar and lexicon.

formulate the complete algorithm):

$$A \rightarrow B\ X$$
$$X \rightarrow C\ D$$

Sometimes using binary branching can actually produce smaller grammars. For example, the sentences that might be characterized as

```
VP -> VBD NP PP*
```

are represented in the Penn Treebank by this series of rules:

```
VP → VBD NP PP
VP → VBD NP PP PP
VP → VBD NP PP PP PP
VP → VBD NP PP PP PP PP
...
```

but could also be generated by the following two-rule grammar:

```
VP → VBD NP PP
VP → VP PP
```

Chomsky-adjunction

The generation of a symbol A with a potentially infinite sequence of symbols B with a rule of the form `A → A B` is known as **Chomsky-adjunction**.

18.5 Ambiguity

structural ambiguity

Ambiguity is the most serious problem faced by syntactic parsers. Chapter 17 introduced the notions of **part-of-speech ambiguity** and **part-of-speech disambiguation**. Here, we introduce a new kind of ambiguity, called **structural ambiguity**, illustrated with a new toy grammar $\mathcal{L}_1$, shown in Figure 18.8, which adds a few rules to the $\mathcal{L}_0$ grammar.

Structural ambiguity occurs when the grammar can assign more than one parse to a sentence. Groucho Marx's well-known line as Captain Spaulding in *Animal*

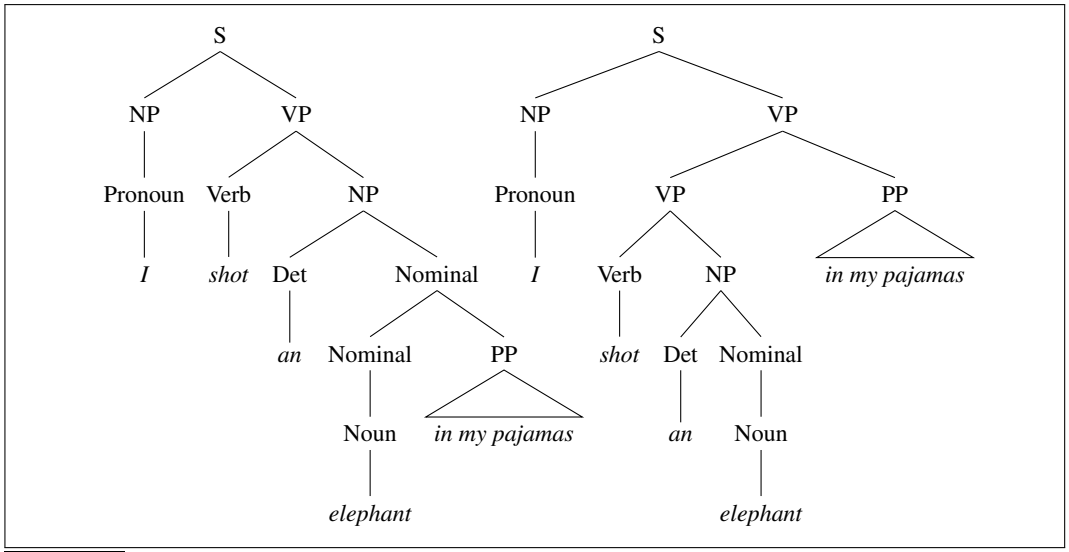


Figure 18.9 Two parse trees for an ambiguous sentence. The parse on the left corresponds to the humorous reading in which the elephant is in the pajamas, the parse on the right corresponds to the reading in which Captain Spaulding did the shooting in his pajamas.

Crackers is ambiguous because the phrase *in my pajamas* can be part of the *NP* headed by *elephant* or a part of the verb phrase headed by *shot*. Figure 18.9 illustrates these two analyses of Marx's line using rules from $\mathcal{L}_1$.

Structural ambiguity, appropriately enough, comes in many forms. Two common kinds of ambiguity are **attachment ambiguity** and **coordination ambiguity**. A sentence has an **attachment ambiguity** if a particular constituent can be attached to the parse tree at more than one place. The Groucho Marx sentence is an example of **PP-attachment ambiguity**: the preposition phrase can be attached either as part of the NP or as part of the VP. Various kinds of adverbial phrases are also subject to this kind of ambiguity. For instance, in the following example the gerundive-VP *flying to Paris* can be part of a gerundive sentence whose subject is *the Eiffel Tower* or it can be an adjunct modifying the VP headed by *saw*:

attachment ambiguity

PP-attachment ambiguity

(18.2) We saw the Eiffel Tower flying to Paris.

In **coordination ambiguity** phrases can be conjoined by a conjunction like *and*. For example, the phrase *old men and women* can be bracketed as *[old [men and women]]*, referring to *old men* and *old women*, or as *[old men] and [women]*, in which case it is only the men who are old. These ambiguities combine in complex ways in real sentences, like the following news sentence from the Brown corpus:

coordination ambiguity

(18.3) President Kennedy today pushed aside other White House business to devote all his time and attention to working on the Berlin crisis address he will deliver tomorrow night to the American people over nationwide television and radio.

This sentence has a number of ambiguities, although since they are semantically unreasonable, it requires a careful reading to see them. The last noun phrase could be parsed *[nationwide [television and radio]]* or *[[nationwide television] and radio]*. The direct object of *pushed aside* should be *other White House business* but could also be the bizarre phrase *[other White House business to devote all his time and attention to working]* (i.e., a structure like *Kennedy affirmed [his intention to propose a new budget to address the deficit]*). Then the phrase *on the Berlin crisis address he*

will deliver tomorrow night to the American people could be an adjunct modifying the verb *pushed*. A *PP* like *over nationwide television and radio* could be attached to any of the higher *VP*s or *NP*s (e.g., it could modify *people* or *night*).

The fact that there are many grammatically correct but semantically unreasonable parses for naturally occurring sentences is an irksome problem that affects all parsers. Fortunately, the CKY algorithm below is designed to efficiently handle structural ambiguities. And as we'll see in the following section, we can augment CKY with neural methods to choose a single correct parse by **syntactic disambiguation**.

syntactic disambiguation

18.6 CKY Parsing: A Dynamic Programming Approach

Dynamic programming provides a powerful framework for addressing the problems caused by ambiguity in grammars. Recall that a dynamic programming approach systematically fills in a table of solutions to subproblems. The complete table has the solution to all the subproblems needed to solve the problem as a whole. In the case of syntactic parsing, these subproblems represent parse trees for all the constituents detected in the input.

The dynamic programming advantage arises from the context-free nature of our grammar rules—once a constituent has been discovered in a segment of the input we can record its presence and make it available for use in any subsequent derivation that might require it. This provides both time and storage efficiencies since subtrees can be looked up in a table, not reanalyzed. This section presents the Cocke-Kasami-Younger (CKY) algorithm, the most widely used dynamic-programming based approach to parsing. **Chart parsing** (Kaplan 1973, Kay 1982) is a related approach, and dynamic programming methods are often referred to as **chart parsing** methods.

chart parsing

18.6.1 Conversion to Chomsky Normal Form

The CKY algorithm requires grammars to first be in Chomsky Normal Form (CNF). Recall from Section 18.4 that grammars in CNF are restricted to rules of the form $A \rightarrow B\ C$ or $A \rightarrow w$. That is, the right-hand side of each rule must expand either to two non-terminals or to a single terminal. Restricting a grammar to CNF does not lead to any loss in expressiveness, since any context-free grammar can be converted into a corresponding CNF grammar that accepts exactly the same set of strings as the original grammar.

Let's start with the process of converting a generic CFG into one represented in CNF. Assuming we're dealing with an ϵ-free grammar, there are three situations we need to address in any generic grammar: rules that mix terminals with non-terminals on the right-hand side, rules that have a single non-terminal on the right-hand side, and rules in which the length of the right-hand side is greater than 2.

The remedy for rules that mix terminals and non-terminals is to simply introduce a new dummy non-terminal that covers only the original terminal. For example, a rule for an infinitive verb phrase such as *INF-VP* $\rightarrow$ *to VP* would be replaced by the two rules *INF-VP* $\rightarrow$ *TO VP* and *TO* $\rightarrow$ *to*.

Unit productions

Rules with a single non-terminal on the right are called **unit productions**. We can eliminate unit productions by rewriting the right-hand side of the original rules with the right-hand side of all the non-unit production rules that they ultimately lead to. More formally, if $A \stackrel{*}{\Rightarrow} B$ by a chain of one or more unit productions and $B \rightarrow \gamma$

is a non-unit production in our grammar, then we add $A \rightarrow \gamma$ for each such rule in the grammar and discard all the intervening unit productions. As we demonstrate with our toy grammar, this can lead to a substantial *flattening* of the grammar and a consequent promotion of terminals to fairly high levels in the resulting trees.

Rules with right-hand sides longer than 2 are normalized through the introduction of new non-terminals that spread the longer sequences over several new rules. Formally, if we have a rule like

$$A \rightarrow B\ C\ \gamma$$

we replace the leftmost pair of non-terminals with a new non-terminal and introduce a new production, resulting in the following new rules:

$$A \rightarrow X1\ \gamma$$
$$X1 \rightarrow B\ C$$

In the case of longer right-hand sides, we simply iterate this process until the offending rule has been replaced by rules of length 2. The choice of replacing the leftmost pair of non-terminals is purely arbitrary; any systematic scheme that results in binary rules would suffice.

In our current grammar, the rule $S \rightarrow Aux\ NP\ VP$ would be replaced by the two rules $S \rightarrow X1\ VP$ and $X1 \rightarrow Aux\ NP$.

The entire conversion process can be summarized as follows:

1. Copy all conforming rules to the new grammar unchanged.
2. Convert terminals within rules to dummy non-terminals.
3. Convert unit productions.
4. Make all rules binary and add them to new grammar.

Figure 18.10 shows the results of applying this entire conversion procedure to the $\mathcal{L}_1$ grammar introduced earlier on page 415. Note that this figure doesn't show the original lexical rules; since these original lexical rules are already in CNF, they all carry over unchanged to the new grammar. Figure 18.10 does, however, show the various places where the process of eliminating unit productions has, in effect, created new lexical rules. For example, all the original verbs have been promoted to both *VP*s and to *S*s in the converted grammar.

18.6.2 CKY Recognition

With our grammar now in CNF, each non-terminal node above the part-of-speech level in a parse tree will have exactly two daughters. A two-dimensional matrix can be used to encode the structure of an entire tree. For a sentence of length n, we will work with the upper-triangular portion of an $(n+1) \times (n+1)$ matrix. Each cell $[i, j]$ in this matrix contains the set of non-terminals that represent all the constituents that span positions i through j of the input. Since our indexing scheme begins with 0, it's natural to think of the indexes as pointing at the gaps between the input words (as in $_0\ Book\ _1\ that\ _2\ flight\ _3$). These gaps are often called **fenceposts**, on the metaphor of the posts between segments of fencing. It follows then that the cell that represents the entire input resides in position $[0, n]$ in the matrix.

fenceposts

Since each non-terminal entry in our table has two daughters in the parse, it follows that for each constituent represented by an entry $[i, j]$, there must be a position in the input, k, where it can be split into two parts such that $i < k < j$. Given such

$\mathcal{L}_1$ **Grammar**	$\mathcal{L}_1$ **in CNF**
$S \rightarrow NP\ VP$	$S \rightarrow NP\ VP$
$S \rightarrow Aux\ NP\ VP$	$S \rightarrow X1\ VP$
	$X1 \rightarrow Aux\ NP$
$S \rightarrow VP$	$S \rightarrow book \mid include \mid prefer$
	$S \rightarrow Verb\ NP$
	$S \rightarrow X2\ PP$
	$S \rightarrow Verb\ PP$
	$S \rightarrow VP\ PP$
$NP \rightarrow Pronoun$	$NP \rightarrow I \mid she \mid me$
$NP \rightarrow Proper\text{-}Noun$	$NP \rightarrow United \mid Houston$
$NP \rightarrow Det\ Nominal$	$NP \rightarrow Det\ Nominal$
$Nominal \rightarrow Noun$	$Nominal \rightarrow book \mid flight \mid meal \mid money$
$Nominal \rightarrow Nominal\ Noun$	$Nominal \rightarrow Nominal\ Noun$
$Nominal \rightarrow Nominal\ PP$	$Nominal \rightarrow Nominal\ PP$
$VP \rightarrow Verb$	$VP \rightarrow book \mid include \mid prefer$
$VP \rightarrow Verb\ NP$	$VP \rightarrow Verb\ NP$
$VP \rightarrow Verb\ NP\ PP$	$VP \rightarrow X2\ PP$
	$X2 \rightarrow Verb\ NP$
$VP \rightarrow Verb\ PP$	$VP \rightarrow Verb\ PP$
$VP \rightarrow VP\ PP$	$VP \rightarrow VP\ PP$
$PP \rightarrow Preposition\ NP$	$PP \rightarrow Preposition\ NP$

Figure 18.10 $\mathcal{L}_1$ Grammar and its conversion to CNF. Note that although they aren't shown here, all the original lexical entries from $\mathcal{L}_1$ carry over unchanged as well.

a position k, the first constituent $[i,k]$ must lie to the left of entry $[i,j]$ somewhere along row i, and the second entry $[k,j]$ must lie beneath it, along column j.

To make this more concrete, consider the following example with its completed parse matrix, shown in Fig. 18.11.

(18.4) Book the flight through Houston.

The superdiagonal row in the matrix contains the parts of speech for each word in the input. The subsequent diagonals above that superdiagonal contain constituents that cover all the spans of increasing length in the input.

Given this setup, CKY recognition consists of filling the parse table in the right way. To do this, we'll proceed in a bottom-up fashion so that at the point where we are filling any cell $[i,j]$, the cells containing the parts that could contribute to this entry (i.e., the cells to the left and the cells below) have already been filled. The algorithm given in Fig. 18.12 fills the upper-triangular matrix a column at a time working from left to right, with each column filled from bottom to top, as the right side of Fig. 18.11 illustrates. This scheme guarantees that at each point in time we have all the information we need (to the left, since all the columns to the left have already been filled, and below since we're filling bottom to top). It also mirrors on-line processing, since filling the columns from left to right corresponds to processing each word one at a time.

The outermost loop of the algorithm given in Fig. 18.12 iterates over the columns, and the second loop iterates over the rows, from the bottom up. The purpose of the innermost loop is to range over all the places where a substring spanning i to j in the input might be split in two. As k ranges over the places where the string can be split, the pairs of cells we consider move, in lockstep, to the right along row i and down along column j. Figure 18.13 illustrates the general case of filling cell $[i,j]$.

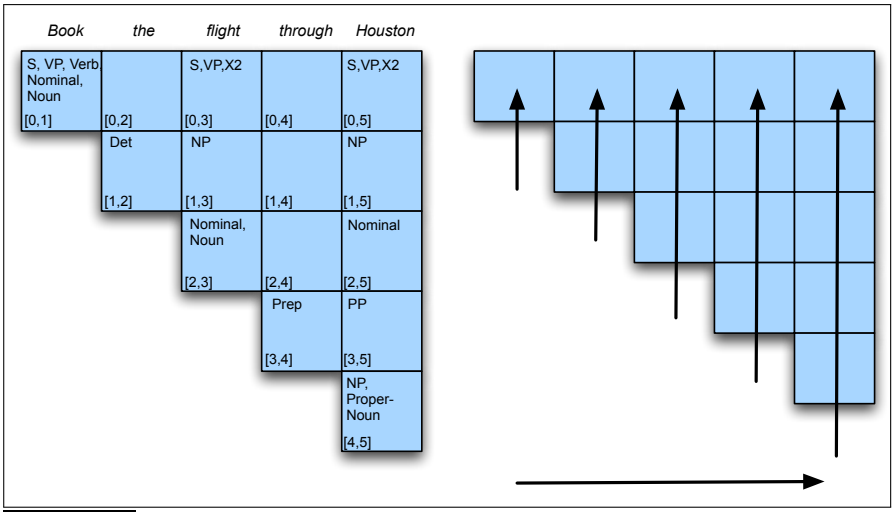


Figure 18.11 Completed parse table for *Book the flight through Houston.*

function CKY-PARSE(*words, grammar*) **returns** *table*

for $j \leftarrow$ **from** 1 **to** LENGTH(*words*) **do**
  for all $\{A \mid A \rightarrow words[j] \in grammar\}$
    $table[j-1, j] \leftarrow table[j-1, j] \cup A$
  for $i \leftarrow$ **from** $j-2$ **down to** 0 **do**
    for $k \leftarrow i+1$ **to** $j-1$ **do**
      for all $\{A \mid A \rightarrow BC \in grammar$ **and** $B \in table[i,k]$ **and** $C \in table[k,j]\}$
        $table[i,j] \leftarrow table[i,j] \cup A$

Figure 18.12 The CKY algorithm.

At each such split, the algorithm considers whether the contents of the two cells can be combined in a way that is sanctioned by a rule in the grammar. If such a rule exists, the non-terminal on its left-hand side is entered into the table.

Figure 18.14 shows how the five cells of column 5 of the table are filled after the word *Houston* is read. The arrows point out the two spans that are being used to add an entry to the table. Note that the action in cell $[0,5]$ indicates the presence of three alternative parses for this input, one where the *PP* modifies the *flight*, one where it modifies the booking, and one that captures the second argument in the original $VP \rightarrow Verb\ NP\ PP$ rule, now captured indirectly with the $VP \rightarrow X2\ PP$ rule.

18.6.3 CKY Parsing

The algorithm given in Fig. 18.12 is a recognizer, not a parser. That is, it can tell us whether a valid parse exists for a given sentence based on whether or not if finds an S in cell $[0,n]$, but it can't provide the derivation, which is the actual job for a parser. To turn it into a parser capable of returning all possible parses for a given input, we can make two simple changes to the algorithm: the first change is to augment the entries in the table so that each non-terminal is paired with pointers to the table entries from which it was derived (more or less as shown in Fig. 18.14), the second change is to permit multiple versions of the same non-terminal to be entered into the table (again as shown in Fig. 18.14). With these changes, the completed table contains all the possible parses for a given input. Returning an arbitrary single

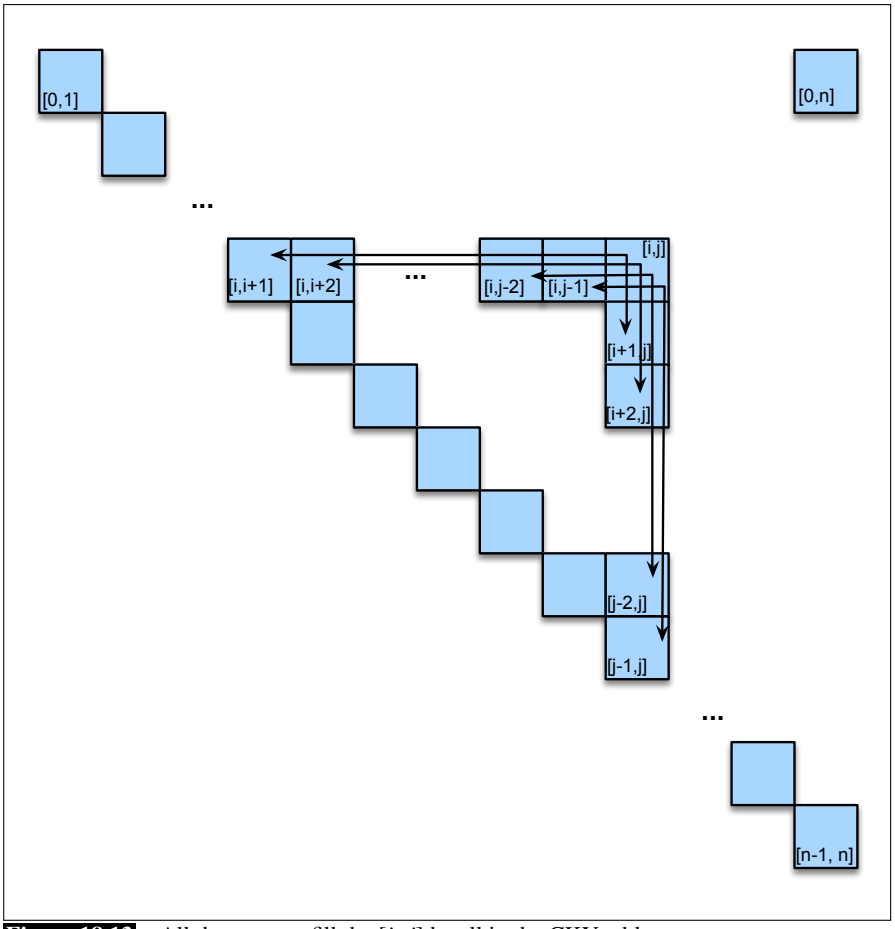


Figure 18.13 All the ways to fill the $[i, j]$th cell in the CKY table.

parse consists of choosing an S from cell $[0,n]$ and then recursively retrieving its component constituents from the table. Of course, instead of returning every parse for a sentence, we usually want just the best parse; we'll see how to do that in the next section.

18.6.4 CKY in Practice

Finally, we should note that while the restriction to CNF does not pose a problem theoretically, it does pose some non-trivial problems in practice. The returned CNF trees may not be consistent with the original grammar built by the grammar developers, and will complicate any syntax-driven approach to semantic analysis.

One approach to getting around these problems is to keep enough information around to transform our trees back to the original grammar as a post-processing step of the parse. This is trivial in the case of the transformation used for rules with length greater than 2. Simply deleting the new dummy non-terminals and promoting their daughters restores the original tree.

In the case of unit productions, it turns out to be more convenient to alter the basic CKY algorithm to handle them directly than it is to store the information needed to recover the correct trees. Exercise 18.3 asks you to make this change. Many of the probabilistic parsers presented in Appendix C use the CKY algorithm altered in

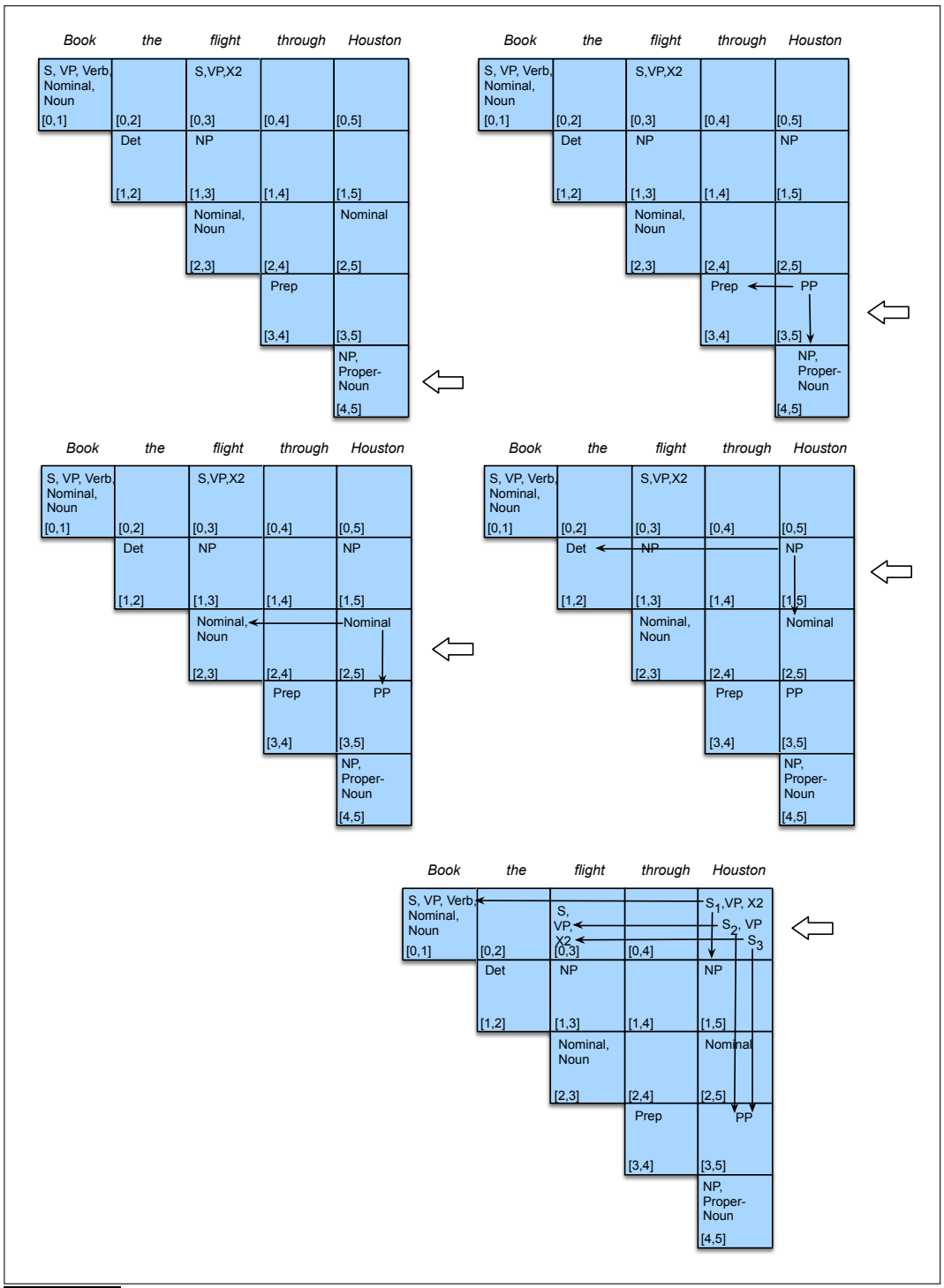


Figure 18.14 Filling the cells of column 5 after reading the word *Houston*.

just this manner.

18.7 Span-Based Neural Constituency Parsing

While the CKY parsing algorithm we've seen so far does great at enumerating all the possible parse trees for a sentence, it has a large problem: it doesn't tell us which parse is the correct one! That is, it doesn't **disambiguate** among the possible parses. To solve the disambiguation problem we'll use a simple neural extension of the CKY algorithm. The intuition of such parsing algorithms (often called **span-based constituency parsing**, or **neural CKY**), is to train a neural classifier to assign a score to each constituent, and then use a modified version of CKY to combine these constituent scores to find the best-scoring parse tree.

Here we'll describe a version of the algorithm from Kitaev et al. (2019). This parser learns to map a span of words to a constituent, and, like CKY, hierarchically combines larger and larger spans to build the parse-tree bottom-up. But unlike classic CKY, this parser doesn't use the hand-written grammar to constrain what constituents can be combined, instead just relying on the learned neural representations of spans to encode likely combinations.

18.7.1 Computing Scores for a Span

span

Let's begin by considering just the constituent (we'll call it a **span**) that lies between fencepost positions i and j with non-terminal symbol label l. We'll build a system to assign a score $s(i, j, l)$ to this constituent span.

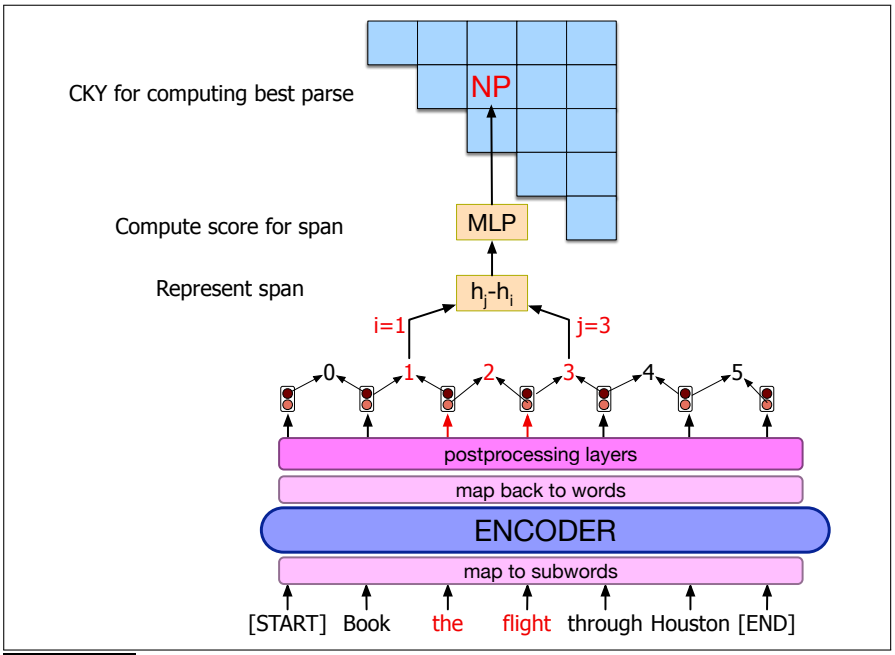
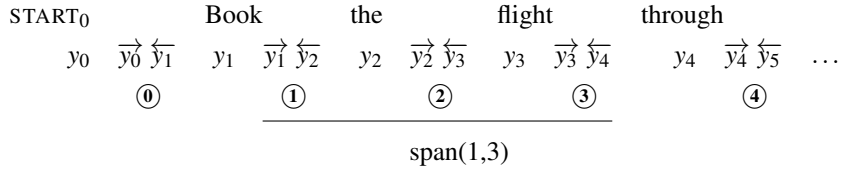


Figure 18.15 A simplified outline of computing the span score for the span *the flight* with the label NP.

Fig. 18.15 sketches the architecture. The input word tokens are embedded by

passing them through a pretrained language model like BERT. Because BERT operates on the level of subword (wordpiece) tokens rather than words, we'll first need to convert the BERT outputs to word representations. One standard way of doing this is to simply use the first subword unit as the representation for the entire word; using the last subword unit, or the sum of all the subword units are also common. The embeddings can then be passed through some postprocessing layers; Kitaev et al. (2019), for example, use 8 Transformer layers.

The resulting word encoder outputs y_t are then used to compute a span score. First, we must map the word encodings (indexed by word positions) to span encodings (indexed by fenceposts). We do this by representing each fencepost with two separate values; the intuition is that a span endpoint to the right of a word represents different information than a span endpoint to the left of a word. We convert each word output y_t into a (leftward-pointing) value for spans ending at this fencepost, $\overleftarrow{y}_t$, and a (rightward-pointing) value $\overrightarrow{y}_t$ for spans beginning at this fencepost, by splitting y_t into two halves. Each span then stretches from one double-vector fencepost to another, as in the following representation of *the flight*, which is span(1,3):



A traditional way to represent a span, developed originally for RNN-based models (Wang and Chang, 2016), but extended also to Transformers, is to take the difference between the embeddings of its start and end, i.e., representing span (i, j) by subtracting the embedding of i from the embedding of j. Here we represent a span by concatenating the difference of each of its fencepost components:

$$v(i, j) = [\overrightarrow{y}_j - \overrightarrow{y}_i \; ; \; \overleftarrow{y}_{j+1} - \overleftarrow{y}_{i+1}] \tag{18.5}$$

The span vector v is then passed through an MLP span classifier, with two fully-connected layers and one ReLU activation function, whose output dimensionality is the number of possible non-terminal labels:

$$s(i, j, \cdot) = \mathbf{W}_2 \, \text{ReLU}(\text{LayerNorm}(\mathbf{W}_1 \mathbf{v}(i, j))) \tag{18.6}$$

The MLP then outputs a score for each possible non-terminal.

18.7.2 Integrating Span Scores into a Parse

Now we have a score for each labeled constituent span $s(i, j, l)$. But we need a score for an entire parse tree. Formally a tree T is represented as a set of $|T|$ such labeled spans, with the t^{th} span starting at position i_t and ending at position j_t, with label l_t:

$$T = \{(i_t, j_t, l_t) : t = 1, \ldots, |T|\} \tag{18.7}$$

Thus once we have a score for each span, the parser can compute a score for the whole tree $s(T)$ simply by summing over the scores of its constituent spans:

$$s(T) = \sum_{(i,j,l) \in T} s(i, j, l) \tag{18.8}$$

And we can choose the final parse tree as the tree with the maximum score:

$$\hat{T} = \operatorname*{argmax}_{T} s(T) \tag{18.9}$$

The simplest method to produce the most likely parse is to greedily choose the highest scoring label for each span. This greedy method is not guaranteed to produce a tree, since the best label for a span might not fit into a complete tree. In practice, however, the greedy method tends to find trees; in their experiments Gaddy et al. (2018) finds that 95% of predicted bracketings form valid trees.

Nonetheless it is more common to use a variant of the CKY algorithm to find the full parse. The variant defined in Gaddy et al. (2018) works as follows. Let's define $s_{\text{best}}(i,j)$ as the score of the best subtree spanning (i,j). For spans of length one, we choose the best label:

$$s_{\text{best}}(i,i+1) = \max_{l} s(i,i+1,l) \tag{18.10}$$

For other spans (i,j), the recursion is:

$$\begin{aligned} s_{\text{best}}(i,j) &= \max_{l} s(i,j,l) \\ &+ \max_{k} [s_{\text{best}}(i,k) + s_{\text{best}}(k,j)] \end{aligned} \tag{18.11}$$

Note that the parser is using the max label for span (i,j) + the max labels for spans (i,k) and (k,j) without worrying about whether those decisions make sense given a grammar. The role of the grammar in classical parsing is to help constrain possible combinations of constituents (NPs like to be followed by VPs). By contrast, the neural model seems to learn these kinds of contextual constraints during its mapping from spans to non-terminals.

For more details on span-based parsing, including the margin-based training algorithm, see Stern et al. (2017), Gaddy et al. (2018), Kitaev and Klein (2018), and Kitaev et al. (2019).

18.8 Evaluating Parsers

PARSEVAL

The standard tool for evaluating parsers that assign a single parse tree to a sentence is the **PARSEVAL** metrics (Black et al., 1991). The PARSEVAL metric measures how much the **constituents** in the hypothesis parse tree look like the constituents in a hand-labeled, **reference** parse. PARSEVAL thus requires a human-labeled reference (or "gold standard") parse tree for each sentence in the test set; we generally draw these reference parses from a treebank like the Penn Treebank.

A constituent in a hypothesis parse C_h of a sentence s is labeled correct if there is a constituent in the reference parse C_r with the same starting point, ending point, and non-terminal symbol. We can then measure the precision and recall just as for tasks we've seen already like named entity tagging:

$$\textbf{labeled recall:} = \frac{\text{\# of correct constituents in hypothesis parse of } s}{\text{\# of total constituents in reference parse of } s}$$

$$\textbf{labeled precision:} = \frac{\text{\# of correct constituents in hypothesis parse of } s}{\text{\# of total constituents in hypothesis parse of } s}$$

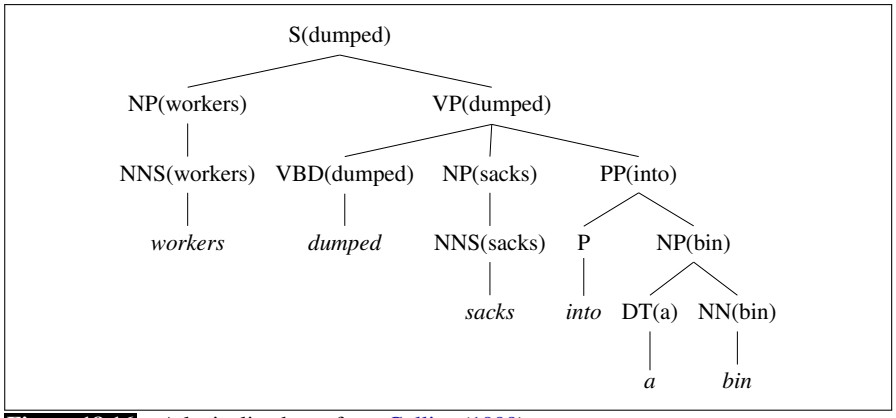


Figure 18.16 A lexicalized tree from Collins (1999).

As usual, we often report a combination of the two, F_1:

$$F_1 = \frac{2PR}{P+R} \tag{18.12}$$

We additionally use a new metric, crossing brackets, for each sentence *s*:

cross-brackets: the number of constituents for which the reference parse has a bracketing such as ((A B) C) but the hypothesis parse has a bracketing such as (A (B C)).

For comparing parsers that use different grammars, the PARSEVAL metric includes a canonicalization algorithm for removing information likely to be grammar-specific (auxiliaries, pre-infinitival "to", etc.) and for computing a simplified score (Black et al., 1991). The canonical implementation of the PARSEVAL metrics is called **evalb** (Sekine and Collins, 1997).

evalb

18.9 Heads and Head-Finding

Syntactic constituents can be associated with a lexical **head**; *N* is the head of an *NP*, *V* is the head of a *VP*. This idea of a head for each constituent dates back to Bloomfield 1914, and is central to the dependency grammars and dependency parsing we'll introduce in Chapter 19. Indeed, heads can be used as a way to map between constituency and dependency parses. Heads are also important in probabilistic parsing (Appendix C) and in constituent-based grammar formalisms like Head-Driven Phrase Structure Grammar (Pollard and Sag, 1994)..

In one simple model of lexical heads, each context-free rule is associated with a head (Charniak 1997, Collins 1999). The head is the word in the phrase that is grammatically the most important. Heads are passed up the parse tree; thus, each non-terminal in a parse tree is annotated with a single word, which is its lexical head. Figure 18.16 shows an example of such a tree from Collins (1999), in which each non-terminal is annotated with its head.

For the generation of such a tree, each CFG rule must be augmented to identify one right-side constituent to be the head child. The headword for a node is then set to the headword of its head child. Choosing these head children is simple for textbook examples (*NN* is the head of *NP*) but is complicated and indeed controversial for

most phrases. (Should the complementizer *to* or the verb be the head of an infinite verb phrase?) Modern linguistic theories of syntax generally include a component that defines heads (see, e.g., (Pollard and Sag, 1994)).

An alternative approach to finding a head is used in most practical computational systems. Instead of specifying head rules in the grammar itself, heads are identified dynamically in the context of trees for specific sentences. In other words, once a sentence is parsed, the resulting tree is walked to decorate each node with the appropriate head. Most current systems rely on a simple set of handwritten rules, such as a practical one for Penn Treebank grammars given in Collins (1999) but developed originally by Magerman (1995). For example, the rule for finding the head of an *NP* is as follows (Collins, 1999, p. 238):

- If the last word is tagged POS, return last-word.
- Else search from right to left for the first child which is an NN, NNP, NNPS, NX, POS, or JJR.
- Else search from left to right for the first child which is an NP.
- Else search from right to left for the first child which is a $, ADJP, or PRN.
- Else search from right to left for the first child which is a CD.
- Else search from right to left for the first child which is a JJ, JJS, RB or QP.
- Else return the last word

Selected other rules from this set are shown in Fig. 18.17. For example, for *VP* rules of the form $VP \rightarrow Y_1 \cdots Y_n$, the algorithm would start from the left of $Y_1 \cdots Y_n$ looking for the first Y_i of type TO; if no TOs are found, it would search for the first Y_i of type VBD; if no VBDs are found, it would search for a VBN, and so on. See Collins (1999) for more details.

Parent	Direction	Priority List
ADJP	Left	NNS QP NN $ ADVP JJ VBN VBG ADJP JJR NP JJS DT FW RBR RBS SBAR RB
ADVP	Right	RB RBR RBS FW ADVP TO CD JJR JJ IN NP JJS NN
PRN	Left	
PRT	Right	RP
QP	Left	$ IN NNS NN JJ RB DT CD NCD QP JJR JJS
S	Left	TO IN VP S SBAR ADJP UCP NP
SBAR	Left	WHNP WHPP WHADVP WHADJP IN DT S SQ SINV SBAR FRAG
VP	Left	TO VBD VBN MD VBZ VB VBG VBP VP ADJP NN NNS NP

Figure 18.17 Some head rules from Collins (1999). The head rules are also called a **head percolation table**.

18.10 Summary

This chapter introduced constituency parsing. Here's a summary of the main points:

- In many languages, groups of consecutive words act as a group or a **constituent**, which can be modeled by **context-free grammars** (which are also known as **phrase-structure grammars**).
- A context-free grammar consists of a set of **rules** or **productions**, expressed over a set of **non-terminal** symbols and a set of **terminal** symbols. Formally, a particular **context-free language** is the set of strings that can be **derived** from a particular **context-free grammar**.

- **Structural ambiguity** is a significant problem for parsers. Common sources of structural ambiguity include **PP-attachment** and **coordination ambiguity**.
- **Dynamic programming** parsing algorithms, such as **CKY**, use a table of partial parses to efficiently parse ambiguous sentences.
- **CKY** restricts the form of the grammar to Chomsky normal form (CNF).
- The basic CKY algorithm compactly represents all possible parses of the sentence but doesn't choose a single best parse.
- Choosing a single parse from all possible parses (**disambiguation**) can be done by **neural constituency parsers**.
- Span-based neural constituency parses train a neural classifier to assign a score to each constituent, and then use a modified version of CKY to combine these constituent scores to find the best-scoring parse tree.
- Parsers are evaluated with three metrics: **labeled recall**, **labeled precision**, and **cross-brackets**.
- **Partial parsing** and **chunking** are methods for identifying shallow syntactic constituents in a text. They are solved by sequence models trained on syntactically-annotated data.

Historical Notes

According to Percival (1976), the idea of breaking up a sentence into a hierarchy of constituents appeared in the *Völkerpsychologie* of the groundbreaking psychologist Wilhelm Wundt (Wundt, 1900):

> *...den sprachlichen Ausdruck für die willkürliche Gliederung einer Gesammtvorstellung in ihre in logische Beziehung zueinander gesetzten Bestandteile*
>
> [the linguistic expression for the arbitrary division of a total idea into its constituent parts placed in logical relations to one another]

Wundt's idea of constituency was taken up into linguistics by Leonard Bloomfield in his early book *An Introduction to the Study of Language* (Bloomfield, 1914). By the time of his later book, *Language* (Bloomfield, 1933), what was then called "immediate-constituent analysis" was a well-established method of syntactic study in the United States. By contrast, traditional European grammar, dating from the Classical period, defined relations between *words* rather than constituents, and European syntacticians retained this emphasis on such **dependency** grammars, the subject of Chapter 19. (And indeed, both dependency and constituency grammars have been in vogue in computational linguistics at different times).

American Structuralism saw a number of specific definitions of the immediate constituent, couched in terms of their search for a "discovery procedure": a methodological algorithm for describing the syntax of a language. In general, these attempt to capture the intuition that "The primary criterion of the immediate constituent is the degree in which combinations behave as simple units" (Bazell, 1952/1966, p. 284). The most well known of the specific definitions is Harris' idea of distributional similarity to individual units, with the *substitutability* test. Essentially, the method proceeded by breaking up a construction into constituents by attempting to substitute simple structures for possible constituents—if a substitution of a simple form, say,

man, was substitutable in a construction for a more complex set (like *intense young man*), then the form *intense young man* was probably a constituent. Harris's test was the beginning of the intuition that a constituent is a kind of equivalence class.

The **context-free grammar** was a formalization of this idea of hierarchical constituency defined in Chomsky (1956) and further expanded upon (and argued against) in Chomsky (1957) and Chomsky (1956/1975). Shortly after Chomsky's initial work, the context-free grammar was reinvented by Backus (1959) and independently by Naur et al. (1960) in their descriptions of the ALGOL programming language; Backus (1996) noted that he was influenced by the productions of Emil Post and that Naur's work was independent of his (Backus') own. After this early work, a great number of computational models of natural language processing were based on context-free grammars because of the early development of efficient parsing algorithms.

Dynamic programming parsing has a history of independent discovery. According to the late Martin Kay (personal communication), a dynamic programming parser containing the roots of the CKY algorithm was first implemented by John Cocke in 1960. Later work extended and formalized the algorithm, as well as proving its time complexity (Kay 1967, Younger 1967, Kasami 1965). The related **well-formed substring table** (**WFST**) seems to have been independently proposed by Kuno (1965) as a data structure that stores the results of all previous computations in the course of the parse. Based on a generalization of Cocke's work, a similar data structure had been independently described in Kay (1967) (and Kay 1973). The top-down application of dynamic programming to parsing was described in Earley's Ph.D. dissertation (Earley 1968, Earley 1970). Sheil (1976) showed the equivalence of the WFST and the Earley algorithm. Norvig (1991) shows that the efficiency offered by dynamic programming can be captured in any language with a *memoization* function (such as in LISP) simply by wrapping the *memoization* operation around a simple top-down parser.

WFST

The earliest disambiguation algorithms for parsing were based on **probabilistic context-free grammars**, first worked out by Booth (1969) and Salomaa (1969); see Appendix C for more history. Neural methods were first applied to parsing at around the same time as statistical parsing methods were developed (Henderson, 1994). In the earliest work neural networks were used to estimate some of the probabilities for statistical constituency parsers (Henderson, 2003, 2004; Emami and Jelinek, 2005) . The next decades saw a wide variety of neural parsing algorithms, including recursive neural architectures (Socher et al., 2011, 2013), encoder-decoder models (Vinyals et al., 2015; Choe and Charniak, 2016), and the idea of focusing on spans (Cross and Huang, 2016). For more on the span-based self-attention approach we describe in this chapter see Stern et al. (2017), Gaddy et al. (2018), Kitaev and Klein (2018), and Kitaev et al. (2019). See Chapter 19 for the parallel history of neural dependency parsing.

probabilistic context-free grammars

The classic reference for parsing algorithms is Aho and Ullman (1972); although the focus of that book is on computer languages, most of the algorithms have been applied to natural language.

Exercises

18.1 Implement the algorithm to convert arbitrary context-free grammars to CNF.

Apply your program to the $\mathcal{L}_1$ grammar.

18.2 Implement the CKY algorithm and test it with your converted $\mathcal{L}_1$ grammar.

18.3 Rewrite the CKY algorithm given in Fig. 18.12 on page 420 so that it can accept grammars that contain unit productions.

18.4 Discuss how to augment a parser to deal with input that may be incorrect, for example, containing spelling errors or mistakes arising from automatic speech recognition.

18.5 Implement the PARSEVAL metrics described in Section 18.8. Next, use a parser and a treebank, compare your metrics against a standard implementation. Analyze the errors in your approach.

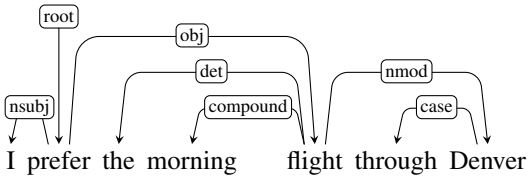
CHAPTER

19 Dependency Parsing

> Tout mot qui fait partie d'une phrase... Entre lui et ses voisins, l'esprit aperçoit des connexions, dont l'ensemble forme la charpente de la phrase.
>
> [Between each word in a sentence and its neighbors, the mind perceives **connections**. These connections together form the scaffolding of the sentence.]
>
> Lucien Tesnière. 1959. Éléments de syntaxe structurale, A.1.§3

dependency grammars

The focus of the last chapter was on context-free grammars and constituent-based representations. Here we present another important family of grammar formalisms called **dependency grammars**. In dependency formalisms, phrasal constituents and phrase-structure rules do not play a direct role. Instead, the syntactic structure of a sentence is described solely in terms of directed binary grammatical relations between the *words*, as in the following dependency parse:



(19.1)

typed dependency

Relations among the words are illustrated above the sentence with directed, labeled arcs from **heads** to **dependents**. We call this a **typed dependency structure** because the labels are drawn from a fixed inventory of grammatical relations. A *root* node explicitly marks the root of the tree, the head of the entire structure.

Figure 19.1 on the next page shows the dependency analysis from Eq. 19.1 but visualized as a tree, alongside its corresponding phrase-structure analysis of the kind given in the prior chapter. Note the absence of nodes corresponding to phrasal constituents or lexical categories in the dependency parse; the internal structure of the dependency parse consists solely of directed relations between words. These head-dependent relationships directly encode important information that is often buried in the more complex phrase-structure parses. For example, the arguments to the verb *prefer* are directly linked to it in the dependency structure, while their connection to the main verb is more distant in the phrase-structure tree. Similarly, *morning* and *Denver*, modifiers of *flight*, are linked to it directly in the dependency structure. This fact that the head-dependent relations are a good proxy for the semantic relationship between predicates and their arguments is an important reason why dependency grammars are currently more common than constituency grammars in natural language processing.

free word order

Another major advantage of dependency grammars is their ability to deal with languages that have a relatively **free word order**. For example, word order in Czech can be much more flexible than in English; a grammatical *object* might occur before or after a *location adverbial*. A phrase-structure grammar would need a separate rule

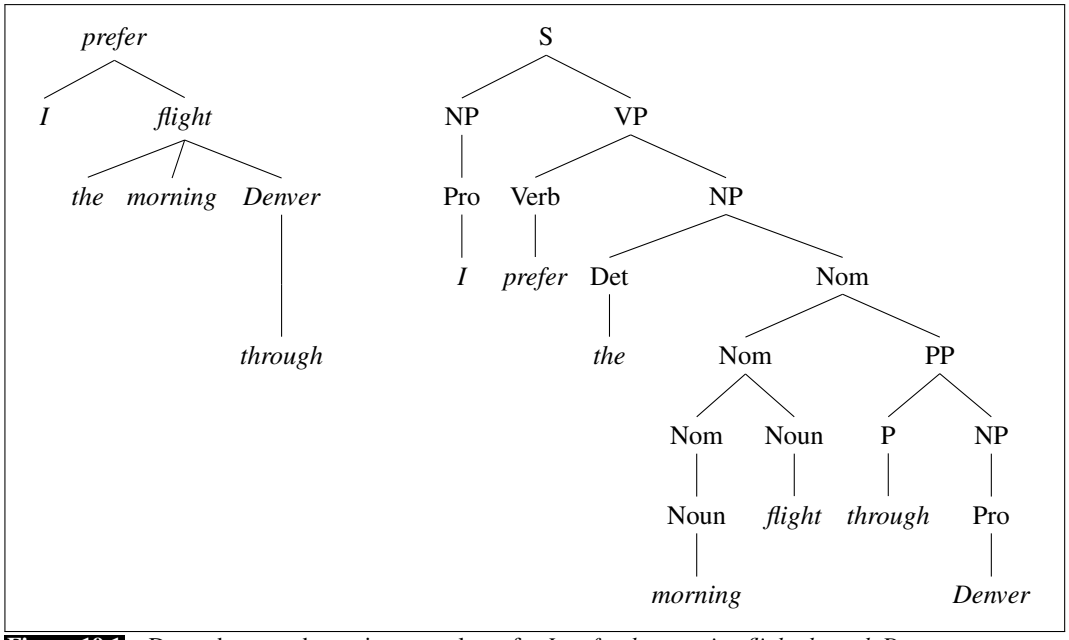


Figure 19.1 Dependency and constituent analyses for *I prefer the morning flight through Denver.*

for each possible place in the parse tree where such an adverbial phrase could occur. A dependency-based approach can have just one link type representing this particular adverbial relation; dependency grammar approaches can thus abstract away a bit more from word order information.

In the following sections, we'll give an inventory of relations used in dependency parsing, discuss two families of parsing algorithms (transition-based, and graph-based), and discuss evaluation.

19.1 Dependency Relations

grammatical relation

head

dependent

The traditional linguistic notion of **grammatical relation** provides the basis for the binary relations that comprise these dependency structures. The arguments to these relations consist of a **head** and a **dependent**. The head plays the role of the central organizing word, and the dependent as a kind of modifier. The head-dependent relationship is made explicit by directly linking heads to the words that are immediately dependent on them.

grammatical function

In addition to specifying the head-dependent pairs, dependency grammars allow us to classify the kinds of grammatical relations, or **grammatical function** that the dependent plays with respect to its head. These include familiar notions such as *subject*, *direct object* and *indirect object*. In English these notions strongly correlate with, but by no means determine, both position in a sentence and constituent type and are therefore somewhat redundant with the kind of information found in phrase-structure trees. However, in languages with more flexible word order, the information encoded directly in these grammatical relations is critical since phrase-based constituent syntax provides little help.

Linguists have developed taxonomies of relations that go well beyond the familiar notions of subject and object. While there is considerable variation from theory

Clausal Argument Relations	Description
NSUBJ	Nominal subject
OBJ	Direct object
IOBJ	Indirect object
CCOMP	Clausal complement
Nominal Modifier Relations	**Description**
NMOD	Nominal modifier
AMOD	Adjectival modifier
APPOS	Appositional modifier
DET	Determiner
CASE	Prepositions, postpositions and other case markers
Other Notable Relations	**Description**
CONJ	Conjunct
CC	Coordinating conjunction

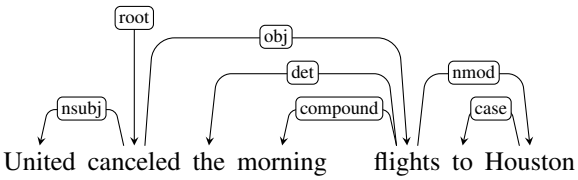
Figure 19.2 Some of the Universal Dependency relations (de Marneffe et al., 2021).

Universal Dependencies

to theory, there is enough commonality that cross-linguistic standards have been developed. The **Universal Dependencies** (UD) project (de Marneffe et al., 2021), an open community effort to annotate dependencies and other aspects of grammar across more than 100 languages, provides an inventory of 37 dependency relations. Fig. 19.2 shows a subset of the UD relations and Fig. 19.3 provides some examples.

The motivation for all of the relations in the Universal Dependency scheme is beyond the scope of this chapter, but the core set of frequently used relations can be broken into two sets: clausal relations that describe syntactic roles with respect to a predicate (often a verb), and modifier relations that categorize the ways that words can modify their heads.

Consider, for example, the following sentence:



(19.2)

Here the clausal relations NSUBJ and OBJ identify the subject and direct object of the predicate *cancel*, while the NMOD, DET, and CASE relations denote modifiers of the nouns *flights* and *Houston*.

19.1.1 Dependency Formalisms

A dependency structure can be represented as a directed graph $G = (V, A)$, consisting of a set of vertices V, and a set of ordered pairs of vertices A, which we'll call arcs.

For the most part we will assume that the set of vertices, V, corresponds exactly to the set of words in a given sentence. However, they might also correspond to punctuation, or when dealing with morphologically complex languages the set of vertices might consist of stems and affixes. The set of arcs, A, captures the head-dependent and grammatical function relationships between the elements in V.

Different grammatical theories or formalisms may place further constraints on these dependency structures. Among the more frequent restrictions are that the structures must be connected, have a designated root node, and be acyclic or planar. Of most relevance to the parsing approaches discussed in this chapter is the common,

Relation	Examples with *head* and **dependent**
NSUBJ	**United** *canceled* the flight.
OBJ	United *diverted* the **flight** to Reno.
	We *booked* her the first **flight** to Miami.
IOBJ	We *booked* **her** the flight to Miami.
COMPOUND	We took the **morning** *flight*.
NMOD	*flight* to **Houston**.
AMOD	Book the **cheapest** *flight*.
APPOS	*United*, a **unit** of UAL, matched the fares.
DET	**The** *flight* was canceled.
	Which *flight* was delayed?
CONJ	We *flew* to Denver and **drove** to Steamboat.
CC	We flew to Denver **and** *drove* to Steamboat.
CASE	Book the flight **through** *Houston*.

Figure 19.3 Examples of some Universal Dependency relations.

dependency tree

computationally-motivated, restriction to rooted trees. That is, a **dependency tree** is a directed graph that satisfies the following constraints:

1. There is a single designated root node that has no incoming arcs.
2. With the exception of the root node, each vertex has exactly one incoming arc.
3. There is a unique path from the root node to each vertex in V.

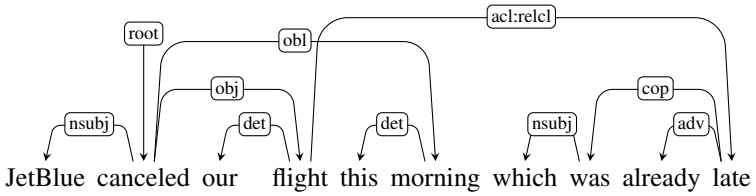
Taken together, these constraints ensure that each word has a single head, that the dependency structure is connected, and that there is a single root node from which one can follow a unique directed path to each of the words in the sentence.

19.1.2 Projectivity

projective

The notion of projectivity imposes an additional constraint that is derived from the order of the words in the input. An arc from a head to a dependent is said to be **projective** if there is a path from the head to every word that lies between the head and the dependent in the sentence. A dependency tree is then said to be projective if all the arcs that make it up are projective. All the dependency trees we've seen thus far have been projective. There are, however, many valid constructions which lead to non-projective trees, particularly in languages with relatively flexible word order.

Consider the following example.



(19.3)

In this example, the arc from *flight* to its modifier *late* is non-projective since there is no path from *flight* to the intervening words *this* and *morning*. As we can see from this diagram, projectivity (and non-projectivity) can be detected in the way we've been drawing our trees. A dependency tree is projective if it can be drawn with no crossing edges. Here there is no way to link *flight* to its dependent *late* without crossing the arc that links *morning* to its head.

Our concern with projectivity arises from two related issues. First, the most widely used English dependency treebanks were automatically derived from phrase-structure treebanks through the use of head-finding rules. The trees generated in such a fashion will always be projective, and hence will be incorrect when non-projective examples like this one are encountered.

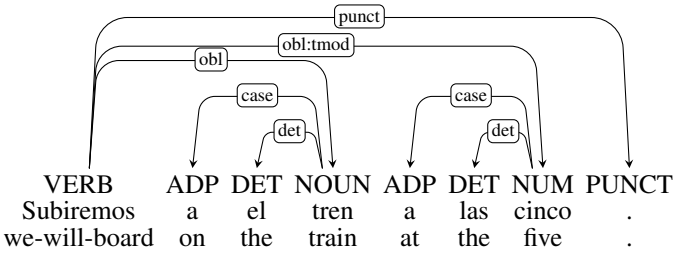
Second, there are computational limitations to the most widely used families of parsing algorithms. The transition-based approaches discussed in Section 19.2 can only produce projective trees, hence any sentences with non-projective structures will necessarily contain some errors. This limitation is one of the motivations for the more flexible graph-based parsing approach described in Section 19.3.

19.1.3 Dependency Treebanks

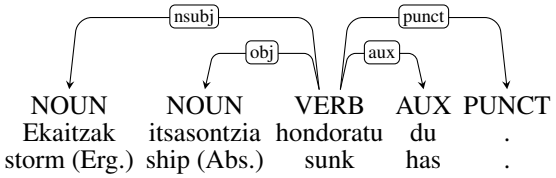
Treebanks play a critical role in the development and evaluation of dependency parsers. They are used for training parsers, they act as the gold labels for evaluating parsers, and they also provide useful information for corpus linguistics studies.

Dependency treebanks are created by having human annotators directly generate dependency structures for a given corpus, or by hand-correcting the output of an automatic parser. A few early treebanks were also based on using a deterministic process to translate existing constituent-based treebanks into dependency trees.

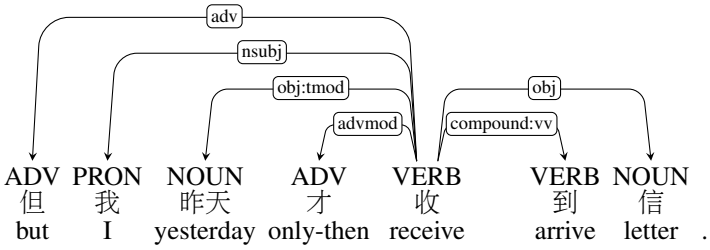
The largest open community project for building dependency trees is the Universal Dependencies project at `https://universaldependencies.org/` introduced above, which currently has almost 200 dependency treebanks in more than 100 languages (de Marneffe et al., 2021). Here are a few UD examples showing dependency trees for sentences in Spanish, Basque, and Mandarin Chinese:



[Spanish] Subiremos al tren a las cinco. "We will be boarding the train at five."(19.4)



[Basque] Ekaitzak itsasontzia hondoratu du. "The storm has sunk the ship."(19.5)



[Chinese] 但我昨天才收到信 "But I didn't receive the letter until yesterday" (19.6)

19.2 Transition-Based Dependency Parsing

transition-based

Our first approach to dependency parsing is called **transition-based** parsing. This architecture draws on **shift-reduce parsing**, a paradigm originally developed for analyzing programming languages (Aho and Ullman, 1972). In transition-based parsing we'll have a **stack** on which we build the parse, a **buffer** of tokens to be parsed, and a parser which takes actions on the parse via a predictor called an **oracle**, as illustrated in Fig. 19.4.

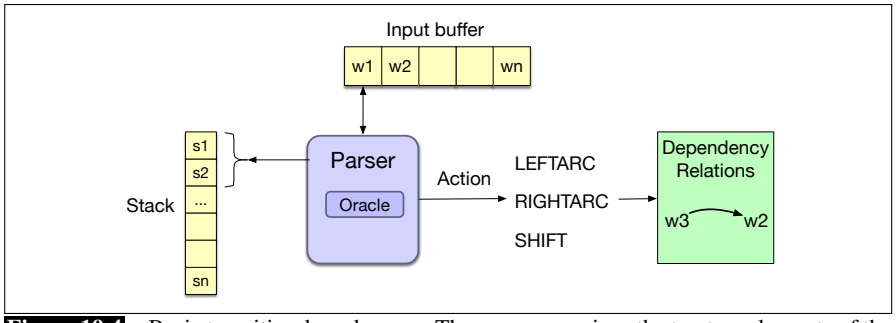


Figure 19.4 Basic transition-based parser. The parser examines the top two elements of the stack and selects an action by consulting an oracle that examines the current configuration.

The parser walks through the sentence left-to-right, successively shifting items from the buffer onto the stack. At each time point we examine the top two elements on the stack, and the oracle makes a decision about what **transition** to apply to build the parse. The possible transitions correspond to the intuitive actions one might take in creating a dependency tree by examining the words in a single pass over the input from left to right (Covington, 2001):

- Assign the current word as the head of some previously seen word,
- Assign some previously seen word as the head of the current word,
- Postpone dealing with the current word, storing it for later processing.

We'll formalize this intuition with the following three transition operators that will operate on the top two elements of the stack:

- LEFTARC: Assert a head-dependent relation between the word at the top of the stack and the second word; remove the second word from the stack.
- RIGHTARC: Assert a head-dependent relation between the second word on the stack and the word at the top; remove the top word from the stack;

- SHIFT: Remove the word from the front of the input buffer and push it onto the stack.

We'll sometimes call operations like LEFTARC and RIGHTARC **reduce** operations, based on a metaphor from shift-reduce parsing, in which reducing means combining elements on the stack. There are some preconditions for using operators. The LEFTARC operator cannot be applied when ROOT is the second element of the stack (since by definition the ROOT node cannot have any incoming arcs). And both the LEFTARC and RIGHTARC operators require two elements to be on the stack to be applied.

arc standard

This particular set of operators implements what is known as the **arc standard** approach to transition-based parsing (Covington 2001, Nivre 2003, Yamada and Matsumoto 2003). In arc standard parsing the transition operators only assert relations between elements at the top of the stack, and once an element has been assigned its head it is removed from the stack and is not available for further processing. As we'll see, there are alternative transition systems which demonstrate different parsing behaviors, but the arc standard approach is quite effective and is simple to implement.

configuration

The specification of a transition-based parser is quite simple, based on representing the current state of the parse as a **configuration**: the stack, an input buffer of words or tokens, and a set of relations representing a dependency tree. Parsing means making a sequence of transitions through the space of possible configurations. We start with an initial configuration in which the stack contains the ROOT node, the buffer has the tokens in the sentence, and an empty set of relations represents the parse. In the final goal state, the stack and the word list should be empty, and the set of relations will represent the final parse. Fig. 19.5 gives the algorithm.

```
function DEPENDENCYPARSE(words) returns dependency tree

  state ← {[root], [words], [] }  ; initial configuration
  while state not final
    t ← ORACLE(state)             ; choose a transition operator to apply
    state ← APPLY(t, state)       ; apply it, creating a new state
  return state
```

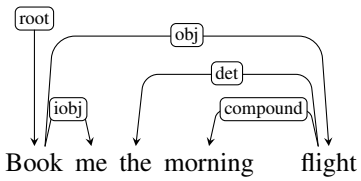
Figure 19.5 A generic transition-based dependency parser

At each step, the parser consults an oracle (we'll come back to this shortly) that provides the correct transition operator to use given the current configuration. It then applies that operator to the current configuration, producing a new configuration. The process ends when all the words in the sentence have been consumed and the ROOT node is the only element remaining on the stack.

The efficiency of transition-based parsers should be apparent from the algorithm. The complexity is linear in the length of the sentence since it is based on a single left to right pass through the words in the sentence. (Each word must first be shifted onto the stack and then later reduced.)

Note that unlike the dynamic programming and search-based approaches discussed in Chapter 18, this approach is a straightforward greedy algorithm—the oracle provides a single choice at each step and the parser proceeds with that choice, no other options are explored, no backtracking is employed, and a single parse is returned in the end.

Figure 19.6 illustrates the operation of the parser with the sequence of transitions leading to a parse for the following example.



(19.7)

Let's consider the state of the configuration at Step 2, after the word *me* has been pushed onto the stack.

Stack	Word List	Relations
[root, book, me]	[the, morning, flight]	

The correct operator to apply here is RIGHTARC which assigns *book* as the head of *me* and pops *me* from the stack resulting in the following configuration.

Stack	Word List	Relations
[root, book]	[the, morning, flight]	(book → me)

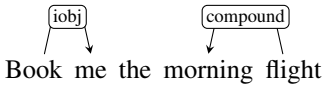
After several subsequent applications of the SHIFT operator, the configuration in Step 6 looks like the following:

Stack	Word List	Relations
[root, book, the, morning, flight]	[]	(book → me)

Here, all the remaining words have been passed onto the stack and all that is left to do is to apply the appropriate reduce operators. In the current configuration, we employ the LEFTARC operator resulting in the following state.

Stack	Word List	Relations
[root, book, the, flight]	[]	(book → me)
		(morning ← flight)

At this point, the parse for this sentence consists of the following structure.



(19.8)

There are several important things to note when examining sequences such as the one in Figure 19.6. First, the sequence given is not the only one that might lead to a reasonable parse. In general, there may be more than one path that leads to the same result, and due to ambiguity, there may be other transition sequences that lead to different equally valid parses.

Second, we are assuming that the oracle always provides the correct operator at each point in the parse—an assumption that is unlikely to be true in practice. As a result, given the greedy nature of this algorithm, incorrect choices will lead to incorrect parses since the parser has no opportunity to go back and pursue alternative choices. Section 19.2.4 will introduce several techniques that allow transition-based approaches to explore the search space more fully.

Step	Stack	Word List	Action	Relation Added
0	[root]	[book, me, the, morning, flight]	SHIFT	
1	[root, book]	[me, the, morning, flight]	SHIFT	
2	[root, book, me]	[the, morning, flight]	RIGHTARC	(book → me)
3	[root, book]	[the, morning, flight]	SHIFT	
4	[root, book, the]	[morning, flight]	SHIFT	
5	[root, book, the, morning]	[flight]	SHIFT	
6	[root, book, the, morning, flight]	[]	LEFTARC	(morning ← flight)
7	[root, book, the, flight]	[]	LEFTARC	(the ← flight)
8	[root, book, flight]	[]	RIGHTARC	(book → flight)
9	[root, book]	[]	RIGHTARC	(root → book)
10	[root]	[]	Done	

Figure 19.6 Trace of a transition-based parse.

Finally, for simplicity, we have illustrated this example without the labels on the dependency relations. To produce labeled trees, we can parameterize the LEFTARC and RIGHTARC operators with dependency labels, as in LEFTARC(NSUBJ) or RIGHTARC(OBJ). This is equivalent to expanding the set of transition operators from our original set of three to a set that includes LEFTARC and RIGHTARC operators for each relation in the set of dependency relations being used, plus an additional one for the SHIFT operator. This, of course, makes the job of the oracle more difficult since it now has a much larger set of operators from which to choose.

19.2.1 Creating an Oracle

The oracle for greedily selecting the appropriate transition is trained by supervised machine learning. As with all supervised machine learning methods, we will need training data: configurations annotated with the correct transition to take. We can draw these from dependency trees. And we need to extract features of the configuration. We'll introduce neural classifiers that represent the configuration via embeddings, as well as classic systems that use hand-designed features.

Generating Training Data

The oracle from the algorithm in Fig. 19.5 takes as input a configuration and returns a transition operator. Therefore, to train a classifier, we will need configurations paired with transition operators (i.e., LEFTARC, RIGHTARC, or SHIFT). Unfortunately, treebanks pair entire sentences with their corresponding trees, not configurations with transitions.

To generate the required training data, we employ the oracle-based parsing algorithm in a clever way. We supply our oracle with the training sentences to be parsed *along with* their corresponding reference parses from the treebank. To produce training instances, we then *simulate* the operation of the parser by running the algorithm and relying on a new **training oracle** to give us correct transition operators for each successive configuration.

training oracle

To see how this works, let's first review the operation of our parser. It begins with a default initial configuration where the stack contains the ROOT, the input list is just the list of words, and the set of relations is empty. The LEFTARC and RIGHTARC operators each add relations between the words at the top of the stack to the set of relations being accumulated for a given sentence. Since we have a gold-standard reference parse for each training sentence, we know which dependency relations are valid for a given sentence. Therefore, we can use the reference parse to guide the

Step	Stack	Word List	Predicted Action
0	[root]	[book, the, flight, through, houston]	SHIFT
1	[root, book]	[the, flight, through, houston]	SHIFT
2	[root, book, the]	[flight, through, houston]	SHIFT
3	[root, book, the, flight]	[through, houston]	LEFTARC
4	[root, book, flight]	[through, houston]	SHIFT
5	[root, book, flight, through]	[houston]	SHIFT
6	[root, book, flight, through, houston]	[]	LEFTARC
7	[root, book, flight, houston]	[]	RIGHTARC
8	[root, book, flight]	[]	RIGHTARC
9	[root, book]	[]	RIGHTARC
10	[root]	[]	Done

Figure 19.7 Generating training items consisting of configuration/predicted action pairs by simulating a parse with a given reference parse.

selection of operators as the parser steps through a sequence of configurations.

To be more precise, given a reference parse and a configuration, the training oracle proceeds as follows:

- Choose LEFTARC if it produces a correct head-dependent relation given the reference parse and the current configuration,
- Otherwise, choose RIGHTARC if (1) it produces a correct head-dependent relation given the reference parse and (2) all of the dependents of the word at the top of the stack have already been assigned,
- Otherwise, choose SHIFT.

The restriction on selecting the RIGHTARC operator is needed to ensure that a word is not popped from the stack, and thus lost to further processing, before all its dependents have been assigned to it.

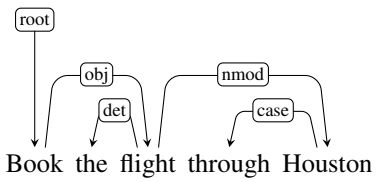
More formally, during training the oracle has access to the following:

- A current configuration with a stack S and a set of dependency relations R_c
- A reference parse consisting of a set of vertices V and a set of dependency relations R_p

Given this information, the oracle chooses transitions as follows:

LEFTARC(r): **if** $(S_1\ r\ S_2) \in R_p$
RIGHTARC(r): **if** $(S_2\ r\ S_1) \in R_p$ **and** $\forall r', w\ s.t. (S_1\ r'\ w) \in R_p$ **then** $(S_1\ r'\ w) \in R_c$
SHIFT: **otherwise**

Let's walk through the processing of the following example as shown in Fig. 19.7.



(19.9)

At Step 1, LEFTARC is not applicable in the initial configuration since it asserts a relation, (root ← book), not in the reference answer; RIGHTARC does assert a relation contained in the final answer (root → book), however *book* has not been attached to any of its dependents yet, so we have to defer, leaving SHIFT as the only

possible action. The same conditions hold in the next two steps. In step 3, LEFTARC is selected to link *the* to its head.

Now consider the situation in Step 4.

Stack	Word buffer	Relations
[root, book, flight]	[through, Houston]	(the ← flight)

Here, we might be tempted to add a dependency relation between *book* and *flight*, which is present in the reference parse. But doing so now would prevent the later attachment of *Houston* since *flight* would have been removed from the stack. Fortunately, the precondition on choosing RIGHTARC prevents this choice and we're again left with SHIFT as the only viable option. The remaining choices complete the set of operators needed for this example.

To recap, we derive appropriate training instances consisting of configuration-transition pairs from a treebank by simulating the operation of a parser in the context of a reference dependency tree. We can deterministically record correct parser actions at each step as we progress through each training example, thereby creating the training set we require.

19.2.2 A feature-based classifier

We'll now introduce two classifiers for choosing transitions, here a classic feature-based algorithm and in the next section a neural classifier using embedding features.

Featured-based classifiers generally use the same features we've seen with part-of-speech tagging and partial parsing: Word forms, lemmas, parts of speech, the head, and the dependency relation to the head. Other features may be relevant for some languages, for example morphosyntactic features like case marking on subjects or objects. The features are extracted from the training *configurations*, which consist of the stack, the buffer and the current set of relations. Most useful are features referencing the top levels of the stack, the words near the front of the buffer, and the dependency relations already associated with any of those elements.

feature template

We'll use a **feature template** as we did for sentiment analysis and part-of-speech tagging. Feature templates allow us to automatically generate large numbers of specific features from a training set. For example, consider the following feature templates that are based on single positions in a configuration.

$$\langle s_1.w, op\rangle, \langle s_2.w, op\rangle \langle s_1.t, op\rangle, \langle s_2.t, op\rangle$$
$$\langle b_1.w, op\rangle, \langle b_1.t, op\rangle \langle s_1.wt, op\rangle \quad (19.10)$$

Here features are denoted as $location.property$, where s = stack, b = the word buffer, w = word forms, t = part-of-speech, and op = operator. Thus the feature for the word form at the top of the stack would be $s_1.w$, the part of speech tag at the front of the buffer $b_1.t$, and the concatenated feature $s_1.wt$ represents the word form concatenated with the part of speech of the word at the top of the stack. Consider applying these templates to the following intermediate configuration derived from a training oracle for (19.2).

Stack	Word buffer	Relations
[root, canceled, flights]	[to Houston]	(canceled → United)
		(flights → morning)
		(flights → the)

The correct transition here is SHIFT (you should convince yourself of this before proceeding). The application of our set of feature templates to this configuration would result in the following set of instantiated features.

$$\begin{aligned}
\langle s_1.w = flights, op = shift \rangle \\
\langle s_2.w = canceled, op = shift \rangle \\
\langle s_1.t = NNS, op = shift \rangle \\
\langle s_2.t = VBD, op = shift \rangle \\
\langle b_1.w = to, op = shift \rangle \\
\langle b_1.t = TO, op = shift \rangle \\
\langle s_1.wt = flightsNNS, op = shift \rangle
\end{aligned} \tag{19.11}$$

Given that the left and right arc transitions operate on the top two elements of the stack, features that *combine* properties from these positions are even more useful. For example, a feature like $s_1.t \circ s_2.t$ concatenates the part of speech tag of the word at the top of the stack with the tag of the word beneath it.

$$\langle s_1.t \circ s_2.t = NNSVBD, op = shift \rangle \tag{19.12}$$

Given the training data and features, any classifier, like multinomial logistic regression or support vector machines, can be used.

19.2.3 A neural classifier

The oracle can also be implemented by a neural classifier. A standard architecture is simply to pass the sentence through an encoder, then take the presentation of the top 2 words on the stack and the first word of the buffer, concatenate them, and present to a feedforward network that predicts the transition to take (Kiperwasser and Goldberg, 2016; Kulmizev et al., 2019). Fig. 19.8 sketches this model. Learning can be done with cross-entropy loss.

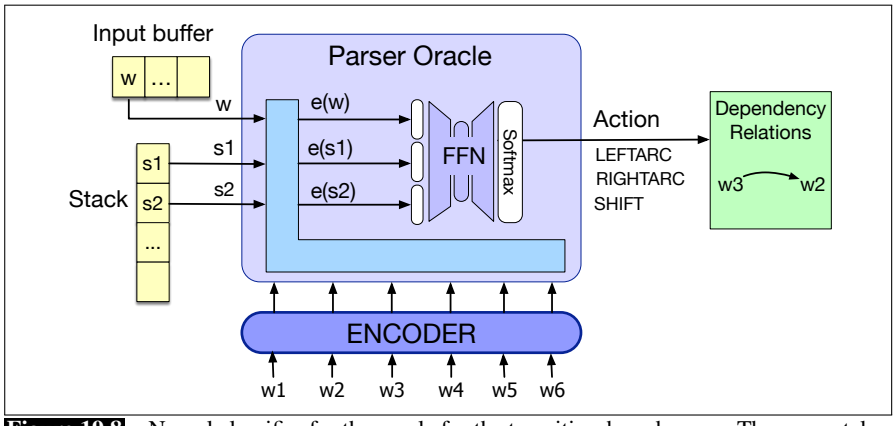


Figure 19.8 Neural classifier for the oracle for the transition-based parser. The parser takes the top 2 words on the stack and the first word of the buffer, represents them by their encodings (from running the whole sentence through the encoder), concatenates the embeddings and passes through a softmax to choose a parser action (transition).

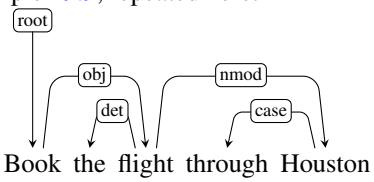
19.2.4 Advanced Methods in Transition-Based Parsing

The basic transition-based approach can be elaborated in a number of ways to improve performance by addressing some of the most obvious flaws in the approach.

Alternative Transition Systems

arc eager

The arc-standard transition system described above is only one of many possible systems. A frequently used alternative is the **arc eager** transition system. The arc eager approach gets its name from its ability to assert rightward relations much sooner than in the arc standard approach. To see this, let's revisit the arc standard trace of Example 19.9, repeated here.



Consider the dependency relation between *book* and *flight* in this analysis. As is shown in Fig. 19.7, an arc-standard approach would assert this relation at Step 8, despite the fact that *book* and *flight* first come together on the stack much earlier at Step 4. The reason this relation can't be captured at this point is due to the presence of the postnominal modifier *through Houston*. In an arc-standard approach, dependents are removed from the stack as soon as they are assigned their heads. If *flight* had been assigned *book* as its head in Step 4, it would no longer be available to serve as the head of *Houston*.

While this delay doesn't cause any issues in this example, in general the longer a word has to wait to get assigned its head the more opportunities there are for something to go awry. The arc-eager system addresses this issue by allowing words to be attached to their heads as early as possible, before all the subsequent words dependent on them have been seen. This is accomplished through minor changes to the LEFTARC and RIGHTARC operators and the addition of a new REDUCE operator.

- LEFTARC: Assert a head-dependent relation between the word at the front of the input buffer and the word at the top of the stack; pop the stack.
- RIGHTARC: Assert a head-dependent relation between the word on the top of the stack and the word at the front of the input buffer; shift the word at the front of the input buffer to the stack.
- SHIFT: Remove the word from the front of the input buffer and push it onto the stack.
- REDUCE: Pop the stack.

The LEFTARC and RIGHTARC operators are applied to the top of the stack and the front of the input buffer, instead of the top two elements of the stack as in the arc-standard approach. The RIGHTARC operator now moves the dependent to the stack from the buffer rather than removing it, thus making it available to serve as the head of following words. The new REDUCE operator removes the top element from the stack. Together these changes permit a word to be eagerly assigned its head and still allow it to serve as the head for later dependents. The trace shown in Fig. 19.9 illustrates the new decision sequence for this example.

In addition to demonstrating the arc-eager transition system, this example demonstrates the power and flexibility of the overall transition-based approach. We were able to swap in a new transition system without having to make any changes to the

Step	Stack	Word List	Action	Relation Added
0	[root]	[book, the, flight, through, houston]	RIGHTARC	(root → book)
1	[root, book]	[the, flight, through, houston]	SHIFT	
2	[root, book, the]	[flight, through, houston]	LEFTARC	(the ← flight)
3	[root, book]	[flight, through, houston]	RIGHTARC	(book → flight)
4	[root, book, flight]	[through, houston]	SHIFT	
5	[root, book, flight, through]	[houston]	LEFTARC	(through ← houston)
6	[root, book, flight]	[houston]	RIGHTARC	(flight → houston)
7	[root, book, flight, houston]	[]	REDUCE	
8	[root, book, flight]	[]	REDUCE	
9	[root, book]	[]	REDUCE	
10	[root]	[]	Done	

Figure 19.9 A processing trace of *Book the flight through Houston* using the arc-eager transition operators.

underlying parsing algorithm. This flexibility has led to the development of a diverse set of transition systems that address different aspects of syntax and semantics including: assigning part of speech tags (Choi and Palmer, 2011a), allowing the generation of non-projective dependency structures (Nivre, 2009), assigning semantic roles (Choi and Palmer, 2011b), and parsing texts containing multiple languages (Bhat et al., 2017).

Beam Search

The computational efficiency of the transition-based approach discussed earlier derives from the fact that it makes a single pass through the sentence, greedily making decisions without considering alternatives. Of course, this is also a weakness – once a decision has been made it can not be undone, even in the face of overwhelming evidence arriving later in a sentence. We can use **beam search** to explore alternative decision sequences. Recall from Chapter 12 that beam search uses a breadth-first search strategy with a heuristic filter that prunes the search frontier to stay within a fixed-size **beam width**.

beam search

beam width

In applying beam search to transition-based parsing, we'll elaborate on the algorithm given in Fig. 19.5. Instead of choosing the single best transition operator at each iteration, we'll apply all applicable operators to each state on an agenda and then score the resulting configurations. We then add each of these new configurations to the frontier, subject to the constraint that there has to be room within the beam. As long as the size of the agenda is within the specified beam width, we can add new configurations to the agenda. Once the agenda reaches the limit, we only add new configurations that are better than the worst configuration on the agenda (removing the worst element so that we stay within the limit). Finally, to insure that we retrieve the best possible state on the agenda, the while loop continues as long as there are non-final states on the agenda.

The beam search approach requires a more elaborate notion of scoring than we used with the greedy algorithm. There, we assumed that the oracle would be a supervised classifier that chose the best transition operator based on features of the current configuration. This choice can be viewed as assigning a score to all the possible transitions and picking the best one.

$$\hat{T}(c) = \text{argmax}\,\text{Score}(t,c)$$

With beam search we are now searching through the space of decision sequences, so it makes sense to base the score for a configuration on its entire history. So we can define the score for a new configuration as the score of its predecessor plus the

score of the operator used to produce it.

$$\begin{aligned} \mathit{ConfigScore}(c_0) &= 0.0 \\ \mathit{ConfigScore}(c_i) &= \mathit{ConfigScore}(c_{i-1}) + \mathit{Score}(t_i, c_{i-1}) \end{aligned}$$

This score is used both in filtering the agenda and in selecting the final answer. The new beam search version of transition-based parsing is given in Fig. 19.10.

```
function DEPENDENCYBEAMPARSE(words, width) returns dependency tree

  state ← {[root], [words], [], 0.0}     ;initial configuration
  agenda ← ⟨state⟩                       ;initial agenda

  while agenda contains non-final states
    newagenda ← ⟨⟩
    for each state ∈ agenda do
      for all {t | t ∈ VALIDOPERATORS(state)} do
        child ← APPLY(t, state)
        newagenda ← ADDTOBEAM(child, newagenda, width)
    agenda ← newagenda
  return BESTOF(agenda)

function ADDTOBEAM(state, agenda, width) returns updated agenda

  if LENGTH(agenda) < width then
      agenda ← INSERT(state, agenda)
  else if SCORE(state) > SCORE(WORSTOF(agenda))
      agenda ← REMOVE(WORSTOF(agenda))
      agenda ← INSERT(state, agenda)
  return agenda
```

Figure 19.10 Beam search applied to transition-based dependency parsing.

19.3 Graph-Based Dependency Parsing

Graph-based methods are the second important family of dependency parsing algorithms. Graph-based parsers are more accurate than transition-based parsers, especially on long sentences; transition-based methods have trouble when the heads are very far from the dependents (McDonald and Nivre, 2011). Graph-based methods avoid this difficulty by scoring entire trees, rather than relying on greedy local decisions. Furthermore, unlike transition-based approaches, graph-based parsers can produce non-projective trees. Although projectivity is not a significant issue for English, it is definitely a problem for many of the world's languages.

Graph-based dependency parsers search through the space of possible trees for a given sentence for a tree (or trees) that maximize some score. These methods encode the search space as directed graphs and employ methods drawn from graph theory to search the space for optimal solutions. More formally, given a sentence S we're looking for the best dependency tree in $\mathcal{G}_s$, the space of all possible trees for that sentence, that maximizes some score.

$$\hat{T}(S) = \underset{t \in \mathcal{G}_S}{\operatorname{argmax}} \operatorname{Score}(t, S)$$

edge-factored

We'll make the simplifying assumption that this score can be **edge-factored**, meaning that the overall score for a tree is the sum of the scores of each of the scores of the edges that comprise the tree.

$$\text{Score}(t,S) = \sum_{e \in t} \text{Score}(e)$$

Graph-based algorithms have to solve two problems: (1) assigning a score to each edge, and (2) finding the best parse tree given the scores of all potential edges. In the next few sections we'll introduce solutions to these two problems, beginning with the second problem of finding trees, and then giving a feature-based and a neural algorithm for solving the first problem of assigning scores.

19.3.1 Parsing via finding the maximum spanning tree

In graph-based parsing, given a sentence S we start by creating a graph G which is a fully-connected, weighted, directed graph where the vertices are the input words and the directed edges represent *all possible* head-dependent assignments. We'll include an additional ROOT node with outgoing edges directed at all of the other vertices. The weights of each edge in G reflect the score for each possible head-dependent relation assigned by some scoring algorithm.

maximum spanning tree

It turns out that finding the best dependency parse for S is equivalent to finding the **maximum spanning tree** over G. A spanning tree over a graph G is a subset of G that is a tree and covers all the vertices in G; a spanning tree over G that starts from the ROOT is a valid parse of S. A maximum spanning tree is the spanning tree with the highest score. Thus a maximum spanning tree of G emanating from the ROOT is the optimal dependency parse for the sentence.

A directed graph for the example *Book that flight* is shown in Fig. 19.11, with the maximum spanning tree corresponding to the desired parse shown in blue. For ease of exposition, we'll describe here the algorithm for *unlabeled* dependency parsing.

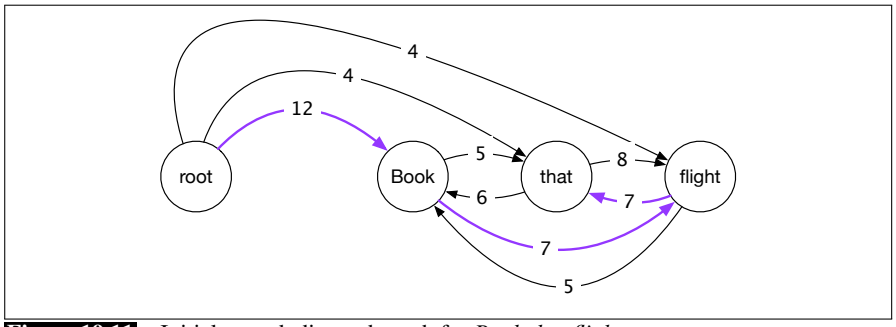


Figure 19.11 Initial rooted, directed graph for *Book that flight*.

Before describing the algorithm it's useful to consider two intuitions about directed graphs and their spanning trees. The first intuition begins with the fact that every vertex in a spanning tree has exactly one incoming edge. It follows from this that every *connected component* of a spanning tree (i.e., every set of vertices that are linked to each other by paths over edges) will also have one incoming edge. The second intuition is that the absolute values of the edge scores are not critical to determining its maximum spanning tree. Instead, it is the relative weights of the edges entering each vertex that matters. If we were to subtract a constant amount from each edge entering a given vertex it would have no impact on the choice of

the maximum spanning tree since every possible spanning tree would decrease by exactly the same amount.

The first step of the algorithm itself is quite straightforward. For each vertex in the graph, an incoming edge (representing a possible head assignment) with the highest score is chosen. If the resulting set of edges produces a spanning tree then we're done. More formally, given the original fully-connected graph $G = (V, E)$, a subgraph $T = (V, F)$ is a spanning tree if it has no cycles and each vertex (other than the root) has exactly one edge entering it. If the greedy selection process produces such a tree then it is the best possible one.

Unfortunately, this approach doesn't always lead to a tree since the set of edges selected may contain cycles. Fortunately, in yet another case of multiple discovery, there is a straightforward way to eliminate cycles generated during the greedy selection phase. Chu and Liu (1965) and Edmonds (1967) independently developed an approach that begins with greedy selection and follows with an elegant recursive cleanup phase that eliminates cycles.

The cleanup phase begins by adjusting all the weights in the graph by subtracting the score of the maximum edge entering each vertex from the score of all the edges entering that vertex. This is where the intuitions mentioned earlier come into play. We have scaled the values of the edges so that the weights of the edges in the cycle have no bearing on the weight of *any* of the possible spanning trees. Subtracting the value of the edge with maximum weight from each edge entering a vertex results in a weight of zero for all of the edges selected during the greedy selection phase, *including all of the edges involved in the cycle*.

Having adjusted the weights, the algorithm creates a new graph by selecting a cycle and collapsing it into a single new node. Edges that enter or leave the cycle are altered so that they now enter or leave the newly collapsed node. Edges that do not touch the cycle are included and edges within the cycle are dropped.

Now, if we knew the maximum spanning tree of this new graph, we would have what we need to eliminate the cycle. The edge of the maximum spanning tree directed towards the vertex representing the collapsed cycle tells us which edge to delete in order to eliminate the cycle. How do we find the maximum spanning tree of this new graph? We recursively apply the algorithm to the new graph. This will either result in a spanning tree or a graph with a cycle. The recursions can continue as long as cycles are encountered. When each recursion completes we expand the collapsed vertex, restoring all the vertices and edges from the cycle *with the exception of the single edge to be deleted.*

Putting all this together, the maximum spanning tree algorithm consists of greedy edge selection, re-scoring of edge costs and a recursive cleanup phase when needed. The full algorithm is shown in Fig. 19.12.

Fig. 19.13 steps through the algorithm with our *Book that flight* example. The first row of the figure illustrates greedy edge selection with the edges chosen shown in blue (corresponding to the set F in the algorithm). This results in a cycle between *that* and *flight*. The scaled weights using the maximum value entering each node are shown in the graph to the right.

Collapsing the cycle between *that* and *flight* to a single node (labeled *tf*) and recursing with the newly scaled costs is shown in the second row. The greedy selection step in this recursion yields a spanning tree that links *root* to *book*, as well as an edge that links *book* to the contracted node. Expanding the contracted node, we can see that this edge corresponds to the edge from *book* to *flight* in the original graph. This in turn tells us which edge to drop to eliminate the cycle.

```
function MAXSPANNINGTREE(G=(V,E), root, score) returns spanning tree

  F ← []
  T' ← []
  score' ← []
  for each v ∈ V do
    bestInEdge ← argmax_{e=(u,v)∈ E} score[e]
    F ← F ∪ bestInEdge
    for each e=(u,v) ∈ E do
      score'[e] ← score[e] − score[bestInEdge]

  if T=(V,F) is a spanning tree then return it
  else
    C ← a cycle in F
    G' ← CONTRACT(G, C)
    T' ← MAXSPANNINGTREE(G', root, score')
    T ← EXPAND(T', C)
    return T

function CONTRACT(G, C) returns contracted graph

function EXPAND(T, C) returns expanded graph
```

Figure 19.12 The Chu-Liu Edmonds algorithm for finding a maximum spanning tree in a weighted directed graph.

On arbitrary directed graphs, this version of the CLE algorithm runs in $O(mn)$ time, where m is the number of edges and n is the number of nodes. Since this particular application of the algorithm begins by constructing a fully connected graph $m = n^2$ yielding a running time of $O(n^3)$. Gabow et al. (1986) present a more efficient implementation with a running time of $O(m + nlogn)$.

19.3.2 A feature-based algorithm for assigning scores

Recall that given a sentence, S, and a candidate tree, T, edge-factored parsing models make the simplification that the score for the tree is the sum of the scores of the edges that comprise the tree:

$$\text{score}(S,T) \;=\; \sum_{e \in T} \text{score}(S,e)$$

In a feature-based algorithm we compute the edge score as a weighted sum of features extracted from it:

$$\text{score}(S,e) \;=\; \sum_{i=1}^{N} w_i f_i(S,e)$$

Or more succinctly.

$$\text{score}(S,e) \;=\; w \cdot f$$

Given this formulation, we need to identify relevant features and train the weights.

The features (and feature combinations) used to train edge-factored models mirror those used in training transition-based parsers, such as

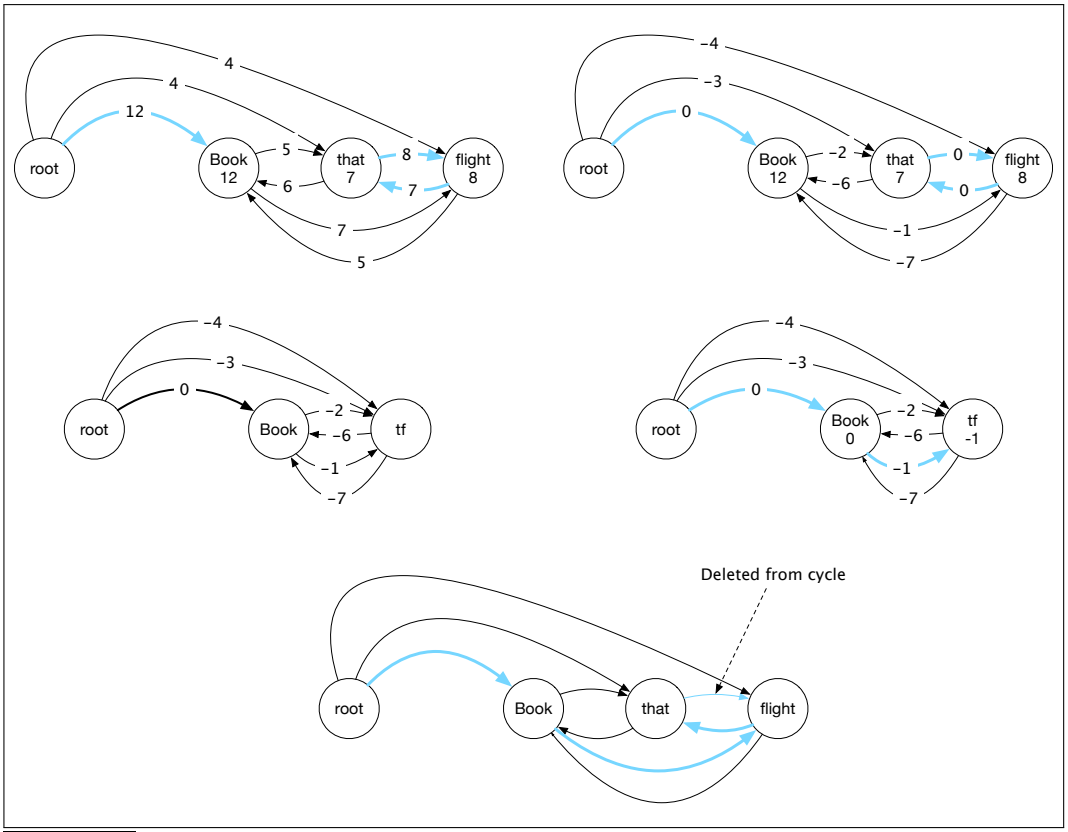


Figure 19.13 Chu-Liu-Edmonds graph-based example for *Book that flight*

- Wordforms, lemmas, and parts of speech of the headword and its dependent.
- Corresponding features from the contexts before, after and between the words.
- Word embeddings.
- The dependency relation itself.
- The direction of the relation (to the right or left).
- The distance from the head to the dependent.

Given a set of features, our next problem is to learn a set of weights corresponding to each. Unlike many of the learning problems discussed in earlier chapters, here we are not training a model to associate training items with class labels, or parser actions. Instead, we seek to train a model that assigns higher scores to correct trees than to incorrect ones. An effective framework for problems like this is to use **inference-based learning** combined with the perceptron learning rule. In this framework, we parse a sentence (i.e, perform inference) from the training set using some initially random set of initial weights. If the resulting parse matches the corresponding tree in the training data, we do nothing to the weights. Otherwise, we find those features in the incorrect parse that are *not* present in the reference parse and we lower their weights by a small amount based on the learning rate. We do this incrementally for each sentence in our training data until the weights converge.

inference-based learning

19.3.3 A neural algorithm for assigning scores

State-of-the-art graph-based multilingual parsers are based on neural networks. Instead of extracting hand-designed features to represent each edge between words w_i and w_j, these parsers run the sentence through an encoder, and then pass the encoded representation of the two words w_i and w_j through a network that estimates a score for the edge $i \rightarrow j$.

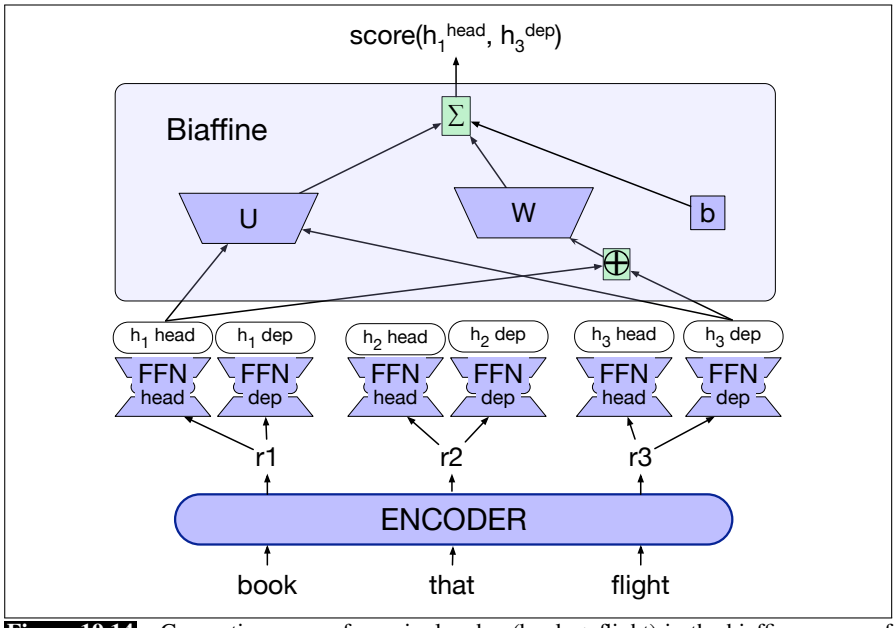


Figure 19.14 Computing scores for a single edge (book→ flight) in the biaffine parser of Dozat and Manning (2017); Dozat et al. (2017). The parser uses distinct feedforward networks to turn the encoder output for each word into a head and dependent representation for the word. The biaffine function turns the head embedding of the head and the dependent embedding of the dependent into a score for the dependency edge.

Here we'll sketch the biaffine algorithm of Dozat and Manning (2017) and Dozat et al. (2017) shown in Fig. 19.14, drawing on the work of Grünewald et al. (2021) who tested many versions of the algorithm via their STEPS system. The algorithm first runs the sentence $X = x_1, ..., x_n$ through an encoder to produce a contextual embedding representation for each token $R = r_1, ..., r_n$. The embedding for each token is now passed through two separate feedforward networks, one to produce a representation of this token as a head, and one to produce a representation of this token as a dependent:

$$\mathbf{h}_i^{head} = \text{FFN}^{head}(\mathbf{r}_i) \tag{19.13}$$

$$\mathbf{h}_i^{dep} = \text{FFN}^{dep}(\mathbf{r}_i) \tag{19.14}$$

Now to assign a score to the directed edge $i \rightarrow j$, (w_i is the head and w_j is the dependent), we feed the head representation of i, $\mathbf{h}_i^{head}$, and the dependent representation of j, $\mathbf{h}_j^{dep}$, into a biaffine scoring function:

$$\text{Score}(i \rightarrow j) = \text{Biaff}(\mathbf{h}_i^{head}, \mathbf{h}_j^{dep}) \tag{19.15}$$

$$\text{Biaff}(\mathbf{x}, \mathbf{y}) = \mathbf{x}^\intercal \mathbf{U} \mathbf{y} + \mathbf{W}(\mathbf{x} \oplus \mathbf{y}) + b \tag{19.16}$$

where $\mathbf{U}$, $\mathbf{W}$, and b are weights learned by the model. The idea of using a biaffine function is to allow the system to learn multiplicative interactions between the vectors $\mathbf{x}$ and $\mathbf{y}$.

If we pass $\text{Score}(i \rightarrow j)$ through a softmax, we end up with a probability distribution, for each token j, over potential heads i (all other tokens in the sentence):

$$p(i \rightarrow j) = \text{softmax}([\text{Score}(k \rightarrow j); \forall k \neq j, 1 \leq k \leq n]) \tag{19.17}$$

This probability can then be passed to the maximum spanning tree algorithm of Section 19.3.1 to find the best tree.

This $p(i \rightarrow j)$ classifier is trained by optimizing the cross-entropy loss.

Note that the algorithm as we've described it is unlabeled. To make this into a labeled algorithm, the Dozat and Manning (2017) algorithm actually trains two classifiers. The first classifier, the **edge-scorer**, the one we described above, assigns a probability $p(i \rightarrow j)$ to each word w_i and w_j. Then the Maximum Spanning Tree algorithm is run to get a single best dependency parse tree for the second. We then apply a second classifier, the **label-scorer**, whose job is to find the maximum probability label for each edge in this parse. This second classifier has the same form as (19.15-19.17), but instead of being trained to predict with binary softmax the probability of an edge existing between two words, it is trained with a softmax over dependency labels to predict the dependency label between the words.

19.4 Evaluation

As with phrase structure-based parsing, the evaluation of dependency parsers proceeds by measuring how well they work on a test set. An obvious metric would be exact match (EM)—how many sentences are parsed correctly. This metric is quite pessimistic, with most sentences being marked wrong. Such measures are not fine-grained enough to guide the development process. Our metrics need to be sensitive enough to tell if actual improvements are being made.

For these reasons, the most common method for evaluating dependency parsers are labeled and unlabeled attachment accuracy. Labeled attachment refers to the proper assignment of a word to its head along with the correct dependency relation. Unlabeled attachment simply looks at the correctness of the assigned head, ignoring the dependency relation. Given a system output and a corresponding reference parse, accuracy is simply the percentage of words in an input that are assigned the correct head with the correct relation. These metrics are usually referred to as the labeled attachment score (LAS) and unlabeled attachment score (UAS). Finally, we can make use of a label accuracy score (LS), the percentage of tokens with correct labels, ignoring where the relations are coming from.

As an example, consider the reference parse and system parse for the following example shown in Fig. 19.15.

(19.18) Book me the flight through Houston.

The system correctly finds 4 of the 6 dependency relations present in the reference parse and receives an LAS of 2/3. However, one of the 2 incorrect relations found by the system holds between *book* and *flight*, which are in a head-dependent relation in the reference parse; the system therefore achieves a UAS of 5/6.

Beyond attachment scores, we may also be interested in how well a system is performing on a particular kind of dependency relation, for example NSUBJ, across

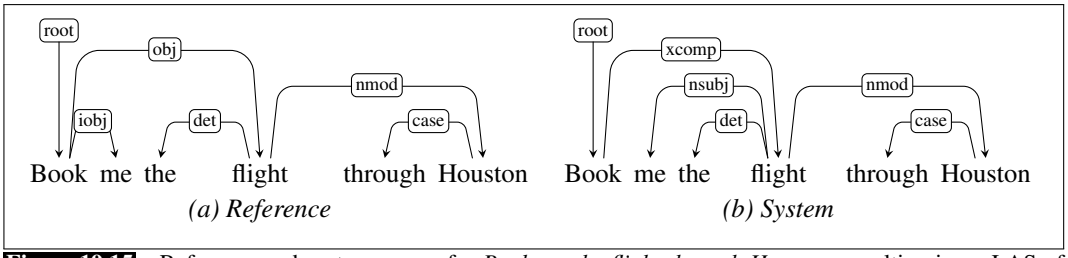


Figure 19.15 Reference and system parses for *Book me the flight through Houston*, resulting in an LAS of 2/3 and an UAS of 5/6.

a development corpus. Here we can make use of the notions of precision and recall introduced in Chapter 17, measuring the percentage of relations labeled NSUBJ by the system that were correct (precision), and the percentage of the NSUBJ relations present in the development set that were in fact discovered by the system (recall). We can employ a confusion matrix to keep track of how often each dependency type was confused for another.

19.5 Summary

This chapter has introduced the concept of dependency grammars and dependency parsing. Here's a summary of the main points that we covered:

- In dependency-based approaches to syntax, the structure of a sentence is described in terms of a set of binary relations that hold between the words in a sentence. Larger notions of constituency are not directly encoded in dependency analyses.
- The relations in a dependency structure capture the head-dependent relationship among the words in a sentence.
- Dependency-based analysis provides information directly useful in further language processing tasks including information extraction, semantic parsing and question answering.
- Transition-based parsing systems employ a greedy stack-based algorithm to create dependency structures.
- Graph-based methods for creating dependency structures are based on the use of maximum spanning tree methods from graph theory.
- Both transition-based and graph-based approaches are developed using supervised machine learning techniques.
- Treebanks provide the data needed to train these systems. Dependency treebanks can be created directly by human annotators or via automatic transformation from phrase-structure treebanks.
- Evaluation of dependency parsers is based on labeled and unlabeled accuracy scores as measured against withheld development and test corpora.

Historical Notes

The dependency-based approach to grammar is much older than the relatively recent phrase-structure or constituency grammars, which date only to the 20th century. Dependency grammar dates back to the Indian grammarian Pāṇini sometime between the 7th and 4th centuries BCE, as well as the ancient Greek linguistic traditions. Contemporary theories of dependency grammar all draw heavily on the 20th century work of Tesnière (1959).

Automatic parsing using dependency grammars was first introduced into computational linguistics by early work on machine translation at the RAND Corporation led by David Hays. This work on dependency parsing closely paralleled work on constituent parsing and made explicit use of grammars to guide the parsing process. After this early period, computational work on dependency parsing remained intermittent over the following decades. Notable implementations of dependency parsers for English during this period include Link Grammar (Sleator and Temperley, 1993), Constraint Grammar (Karlsson et al., 1995), and MINIPAR (Lin, 2003).

Dependency parsing saw a major resurgence in the late 1990's with the appearance of large dependency-based treebanks and the associated advent of data driven approaches described in this chapter. Eisner (1996) developed an efficient dynamic programming approach to dependency parsing based on bilexical grammars derived from the Penn Treebank. Covington (2001) introduced the deterministic word by word approach underlying current transition-based approaches. Yamada and Matsumoto (2003) and Kudo and Matsumoto (2002) introduced both the shift-reduce paradigm and the use of supervised machine learning in the form of support vector machines to dependency parsing.

Transition-based parsing is based on the **shift-reduce** parsing algorithm originally developed for analyzing programming languages (Aho and Ullman, 1972). Shift-reduce parsing also makes use of a context-free grammar. Input tokens are successively shifted onto the stack and the top two elements of the stack are matched against the right-hand side of the rules in the grammar; when a match is found the matched elements are replaced on the stack (reduced) by the non-terminal from the left-hand side of the rule being matched. In transition-based dependency parsing we skip the grammar, and alter the reduce operation to add a dependency relation between a word and its head.

Nivre (2003) defined the modern, deterministic, transition-based approach to dependency parsing. Subsequent work by Nivre and his colleagues formalized and analyzed the performance of numerous transition systems, training methods, and methods for dealing with non-projective language (Nivre and Scholz 2004, Nivre 2006, Nivre and Nilsson 2005, Nivre et al. 2007b, Nivre 2007). The neural approach was pioneered by Chen and Manning (2014) and extended by Kiperwasser and Goldberg (2016); Kulmizev et al. (2019).

The graph-based maximum spanning tree approach to dependency parsing was introduced by McDonald et al. 2005a, McDonald et al. 2005b. The neural classifier was introduced by (Kiperwasser and Goldberg, 2016).

The long-running Prague Dependency Treebank project (Hajič, 1998) is the most significant effort to directly annotate a corpus with multiple layers of morphological, syntactic and semantic information. PDT 3.0 contains over 1.5 M tokens (Bejček et al., 2013).

Universal Dependencies (UD) (de Marneffe et al., 2021) is an open community

project to create a framework for dependency treebank annotation, with nearly 200 treebanks in over 100 languages. The UD annotation scheme evolved out of several distinct efforts including Stanford dependencies (de Marneffe et al. 2006, de Marneffe and Manning 2008, de Marneffe et al. 2014), Google's universal part-of-speech tags (Petrov et al., 2012), and the Interset interlingua for morphosyntactic tagsets (Zeman, 2008).

The Conference on Natural Language Learning (CoNLL) has conducted an influential series of shared tasks related to dependency parsing over the years (Buchholz and Marsi 2006, Nivre et al. 2007a, Surdeanu et al. 2008, Hajič et al. 2009). More recent evaluations have focused on parser robustness with respect to morphologically rich languages (Seddah et al., 2013), and non-canonical language forms such as social media, texts, and spoken language (Petrov and McDonald, 2012). Choi et al. (2015) presents a performance analysis of 10 dependency parsers across a range of metrics, as well as DEPENDABLE, a robust parser evaluation tool.

Exercises

CHAPTER

20 Information Extraction: Relations, Events, and Time

Time will explain.
Jane Austen, *Persuasion*

Imagine that you are an analyst with an investment firm that tracks airline stocks. You're given the task of determining the relationship (if any) between airline announcements of fare increases and the behavior of their stocks the next day. Historical data about stock prices is easy to come by, but what about the airline announcements? You will need to know at least the name of the airline, the nature of the proposed fare hike, the dates of the announcement, and possibly the response of other airlines. Fortunately, these can be all found in news articles like this one:

> Citing high fuel prices, United Airlines said Friday it has increased fares by $6 per round trip on flights to some cities also served by lower-cost carriers. American Airlines, a unit of AMR Corp., immediately matched the move, spokesman Tim Wagner said. United, a unit of UAL Corp., said the increase took effect Thursday and applies to most routes where it competes against discount carriers, such as Chicago to Dallas and Denver to San Francisco.

information extraction

This chapter presents techniques for extracting limited kinds of semantic content from text. This process of **information extraction** (IE) turns the unstructured information embedded in texts into structured data, for example for populating a relational database to enable further processing.

relation extraction

We begin with the task of **relation extraction**: finding and classifying semantic relations among entities mentioned in a text, like child-of (X is the child-of Y), or part-whole or geospatial relations. Relation extraction has close links to populating a relational database, and **knowledge graphs**, datasets of structured relational knowledge, are a useful way for search engines to present information to users.

knowledge graphs

event extraction

Next, we discuss **event extraction**, the task of finding events in which these entities participate, like, in our sample text, the fare increases by *United* and *American* and the reporting events *said* and *cite*. Events are also situated in **time**, occurring at a particular date or time, and events can be related temporally, happening before or after or simultaneously with each other. We'll need to recognize temporal expressions like *Friday*, *Thursday* or *two days from now* and times such as *3:30 P.M.*, and **normalize** them onto specific calendar dates or times. We'll need to link *Friday* to the time of United's announcement, *Thursday* to the previous day's fare increase, and we'll need to produce a timeline in which United's announcement follows the fare increase and American's announcement follows both of those events.

template filling

The related task of **template filling** is to find recurring stereotypical events or situations in documents and fill in the template slots. These slot-fillers may consist of text segments extracted directly from the text, or concepts like times, amounts, or ontology entities that have been inferred through additional processing. Our airline

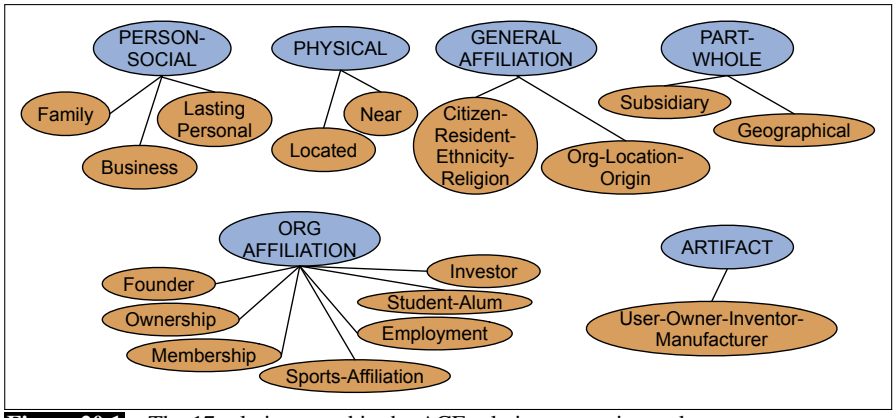


Figure 20.1 The 17 relations used in the ACE relation extraction task.

text presents such a stereotypical situation since airlines often raise fares and then wait to see if competitors follow along. Here we can identify *United* as a lead airline that initially raised its fares, $6 as the amount, *Thursday* as the increase date, and *American* as an airline that followed along, leading to a filled template like the following:

FARE-RAISE ATTEMPT:	LEAD AIRLINE:	UNITED AIRLINES
	AMOUNT:	$6
	EFFECTIVE DATE:	2006-10-26
	FOLLOWER:	AMERICAN AIRLINES

20.1 Relation Extraction

Let's assume that we have detected the named entities in our sample text (perhaps using the techniques of Chapter 17), and would like to discern the relationships that exist among the detected entities:

> Citing high fuel prices, [ORG **United Airlines**] said [TIME **Friday**] it has increased fares by [MONEY **$6**] per round trip on flights to some cities also served by lower-cost carriers. [ORG **American Airlines**], a unit of [ORG **AMR Corp.**], immediately matched the move, spokesman [PER **Tim Wagner**] said. [ORG **United**], a unit of [ORG **UAL Corp.**], said the increase took effect [TIME **Thursday**] and applies to most routes where it competes against discount carriers, such as [LOC **Chicago**] to [LOC **Dallas**] and [LOC **Denver**] to [LOC **San Francisco**].

The text tells us, for example, that *Tim Wagner* is a spokesman for *American Airlines*, that *United* is a unit of *UAL Corp.*, and that *American* is a unit of *AMR*. These binary relations are instances of more generic relations such as **part-of** or **employs** that are fairly frequent in news-style texts. Figure 20.1 lists the 17 relations used in the ACE relation extraction evaluations and Fig. 20.2 shows some sample relations. We might also extract more domain-specific relations such as the notion of an airline route. For example from this text we can conclude that United has routes to Chicago, Dallas, Denver, and San Francisco.

Relations	Types	Examples
Physical-Located	PER-GPE	**He** was in **Tennessee**
Part-Whole-Subsidiary	ORG-ORG	**XYZ**, the parent company of **ABC**
Person-Social-Family	PER-PER	**Yoko**'s husband **John**
Org-AFF-Founder	PER-ORG	**Steve Jobs**, co-founder of **Apple**...

Figure 20.2 Semantic relations with examples and the named entity types they involve.

Sets of relations have been defined for many other domains as well. For example UMLS, the Unified Medical Language System from the US National Library of Medicine has a network that defines 134 broad subject categories, entity types, and 54 relations between the entities, such as the following:

Entity	Relation	Entity
Injury	disrupts	Physiological Function
Bodily Location	location-of	Biologic Function
Anatomical Structure	part-of	Organism
Pharmacologic Substance	causes	Pathological Function
Pharmacologic Substance	treats	Pathologic Function

Given a medical sentence like this one:

(20.1) Doppler echocardiography can be used to diagnose left anterior descending artery stenosis in patients with type 2 diabetes

We could thus extract the UMLS relation:

Echocardiography, Doppler Diagnoses *Acquired stenosis*

infoboxes

Wikipedia also offers a large supply of relations, drawn from **infoboxes**, structured tables associated with certain Wikipedia articles. For example, the Wikipedia infobox for **Stanford** includes structured facts like `state = "California"` or `president = "Marc Tessier-Lavigne"`. These facts can be turned into relations like *president-of* or *located-in*. or into relations in a metalanguage called **RDF** (Resource Description Framework). An **RDF triple** is a tuple of entity-relation-entity, called a subject-predicate-object expression. Here's a sample RDF triple:

RDF

RDF triple

subject	predicate	object
Golden Gate Park	location	San Francisco

For example the crowdsourced DBpedia (Bizer et al., 2009) is an ontology derived from Wikipedia containing over 2 billion RDF triples. Another dataset from Wikipedia infoboxes, **Freebase** (Bollacker et al., 2008), now part of Wikidata (Vrandečić and Krötzsch, 2014), has relations between people and their nationality, or locations, and other locations they are contained in.

Freebase

WordNet or other ontologies offer useful ontological relations that express hierarchical relations between words or concepts. For example WordNet has the **is-a** or **hypernym** relation between classes,

is-a

hypernym

Giraffe is-a ruminant is-a ungulate is-a mammal is-a vertebrate ...

WordNet also has *Instance-of* relation between individuals and classes, so that for example *San Francisco* is in the *Instance-of* relation with *city*. Extracting these relations is an important step in extending or building ontologies.

Finally, there are large datasets that contain sentences hand-labeled with their relations, designed for training and testing relation extractors. The TACRED dataset (Zhang et al., 2017) contains 106,264 examples of relation triples about particular people or organizations, labeled in sentences from news and web text drawn from the

annual TAC Knowledge Base Population (TAC KBP) challenges. TACRED contains 41 relation types (like per:city of birth, org:subsidiaries, org:member of, per:spouse), plus a no relation tag; examples are shown in Fig. 20.3. About 80% of all examples are annotated as no relation; having sufficient negative data is important for training supervised classifiers.

Example	Entity Types & Label
Carey will succeed Cathleen P. Black, who held the position for 15 years and will take on a new role as chairwoman of Hearst Magazines, the company said.	PERSON/TITLE Relation: *per:title*
Irene Morgan Kirkaldy, who was born and reared in Baltimore, lived on Long Island and ran a child-care center in Queens with her second husband, Stanley Kirkaldy.	PERSON/CITY Relation: *per:city_of_birth*
Baldwin declined further comment, and said JetBlue chief executive Dave Barger was unavailable.	Types: PERSON/TITLE Relation: *no_relation*

Figure 20.3 Example sentences and labels from the TACRED dataset (Zhang et al., 2017).

A standard dataset was also produced for the SemEval 2010 Task 8, detecting relations between nominals (Hendrickx et al., 2009). The dataset has 10,717 examples, each with a pair of nominals (untyped) hand-labeled with one of 9 directed relations like *product-producer* (a *factory* manufactures *suits*) or *component-whole* (my *apartment* has a large *kitchen*).

20.2 Relation Extraction Algorithms

There are five main classes of algorithms for relation extraction: **handwritten patterns**, **supervised machine learning**, **semi-supervised** (via **bootstrapping** or **distant supervision**), and **unsupervised**. We'll introduce each of these in the next sections.

20.2.1 Using Patterns to Extract Relations

Hearst patterns

The earliest and still common algorithm for relation extraction is lexico-syntactic patterns, first developed by Hearst (1992a), and therefore often called **Hearst patterns**. Consider the following sentence:

> Agar is a substance prepared from a mixture of red algae, such as Gelidium, for laboratory or industrial use.

Hearst points out that most human readers will not know what *Gelidium* is, but that they can readily infer that it is a kind of (a **hyponym** of) *red algae*, whatever that is. She suggests that the following **lexico-syntactic pattern**

$$NP_0 \text{ such as } NP_1\{,NP_2\ldots,(and|or)NP_i\}, i \geq 1 \tag{20.2}$$

implies the following semantics

$$\forall NP_i, i \geq 1, \text{hyponym}(NP_i, NP_0) \tag{20.3}$$

allowing us to infer

$$\text{hyponym}(\text{Gelidium}, \text{red algae}) \tag{20.4}$$

NP {, NP}* {,} (and|or) other NP_H	temples, treasuries, and other important civic buildings
NP_H such as {NP,}* {(or|and)} NP	red algae such as Gelidium
such NP_H as {NP,}* {(or|and)} NP	such authors as Herrick, Goldsmith, and Shakespeare
NP_H {,} including {NP,}* {(or|and)} NP	common-law countries, including Canada and England
NP_H {,} especially {NP}* {(or|and)} NP	European countries, especially France, England, and Spain

Figure 20.4 Hand-built lexico-syntactic patterns for finding hypernyms, using {} to mark optionality (Hearst 1992a, Hearst 1998).

Figure 20.4 shows five patterns Hearst (1992a, 1998) suggested for inferring the hyponym relation; we've shown NP_H as the parent/hyponym. Modern versions of the pattern-based approach extend it by adding named entity constraints. For example if our goal is to answer questions about "Who holds what office in which organization?", we can use patterns like the following:

PER, POSITION of ORG:
George Marshall, Secretary of State of the United States

PER (named|appointed|chose|etc.) PER Prep? POSITION
Truman appointed Marshall Secretary of State

PER [be]? (named|appointed|etc.) Prep? ORG POSITION
George Marshall was named US Secretary of State

Hand-built patterns have the advantage of high-precision and they can be tailored to specific domains. On the other hand, they are often low-recall, and it's a lot of work to create them for all possible patterns.

20.2.2 Relation Extraction via Supervised Learning

Supervised machine learning approaches to relation extraction follow a scheme that should be familiar by now. A fixed set of relations and entities is chosen, a training corpus is hand-annotated with the relations and entities, and the annotated texts are then used to train classifiers to annotate an unseen test set.

The most straightforward approach, illustrated in Fig. 20.5 is: (1) Find pairs of named entities (usually in the same sentence). (2): Apply a relation-classification on each pair. The classifier can use any supervised technique (logistic regression, RNN, Transformer, random forest, etc.).

An optional intermediate filtering classifier can be used to speed up the processing by making a binary decision on whether a given pair of named entities are related (by any relation). It's trained on positive examples extracted directly from all relations in the annotated corpus, and negative examples generated from within-sentence entity pairs that are not annotated with a relation.

Feature-based supervised relation classifiers. Let's consider sample features for a feature-based classifier (like logistic regression or random forests), classifying the relationship between *American Airlines* (Mention 1, or M1) and *Tim Wagner* (Mention 2, M2) from this sentence:

(20.5) **American Airlines**, a unit of AMR, immediately matched the move, spokesman **Tim Wagner** said

These include **word** features (as embeddings, or 1-hot, stemmed or not):

- The headwords of M1 and M2 and their concatenation
  Airlines Wagner Airlines-Wagner

```
function FINDRELATIONS(words) returns relations

    relations ← nil
    entities ← FINDENTITIES(words)
    forall entity pairs ⟨e1, e2⟩ in entities do
        if RELATED?(e1, e2)
            relations ← relations+CLASSIFYRELATION(e1, e2)
```

Figure 20.5 Finding and classifying the relations among entities in a text.

- Bag-of-words and bigrams in M1 and M2
  American, Airlines, Tim, Wagner, American Airlines, Tim Wagner
- Words or bigrams in particular positions
  M2: -1 spokesman
  M2: +1 said
- Bag of words or bigrams between M1 and M2:
  a, AMR, of, immediately, matched, move, spokesman, the, unit

Named entity features:

- Named-entity types and their concatenation
  (M1: ORG, M2: PER, M1M2: ORG-PER)
- Entity Level of M1 and M2 (from the set NAME, NOMINAL, PRONOUN)
  M1: NAME [it or he would be PRONOUN]
  M2: NAME [the company would be NOMINAL]
- Number of entities between the arguments (in this case 1, for AMR)

Syntactic structure is a useful signal, often represented as the dependency or constituency **syntactic path** traversed through the tree between the entities.

- Constituent paths between M1 and M2
  $NP \uparrow NP \uparrow S \uparrow S \downarrow NP$
- Dependency-tree paths
  $Airlines \leftarrow_{subj} matched \leftarrow_{comp} said \rightarrow_{subj} Wagner$

Neural supervised relation classifiers Neural models for relation extraction similarly treat the task as supervised classification. Let's consider a typical system applied to the TACRED relation extraction dataset and task (Zhang et al., 2017). In TACRED we are given a sentence and two spans within it: a subject, which is a person or organization, and an object, which is any other entity. The task is to assign a relation from the 42 TAC relations, including no relation.

A typical Transformer-encoder algorithm, shown in Fig. 20.6, simply takes a pretrained encoder like BERT and adds a linear layer on top of the sentence representation (for example the BERT `[CLS]` token), a linear layer that is finetuned as a 1-of-N classifier to assign one of the 43 labels. The input to the BERT encoder is partially de-lexified; the subject and object entities are replaced in the input by their NER tags. This helps keep the system from overfitting to the individual lexical items (Zhang et al., 2017). When using BERT-type Transformers for relation extraction, it helps to use versions of BERT like RoBERTa (Liu et al., 2019) or spanBERT (Joshi et al., 2020) that don't have two sequences separated by a `[SEP]` token, but instead form the input from a single long sequence of sentences.

In general, if the test set is similar enough to the training set, and if there is enough hand-labeled data, supervised relation extraction systems can get high ac-

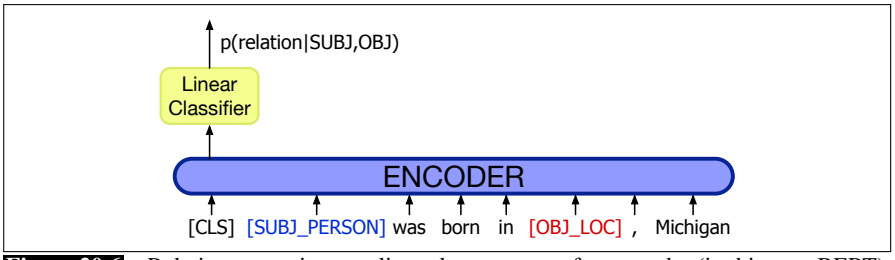


Figure 20.6 Relation extraction as a linear layer on top of an encoder (in this case BERT), with the subject and object entities replaced in the input by their NER tags (Zhang et al. 2017, Joshi et al. 2020).

curacies. But labeling a large training set is extremely expensive and supervised models are brittle: they don't generalize well to different text genres. For this reason, much research in relation extraction has focused on the semi-supervised and unsupervised approaches we turn to next.

20.2.3 Semisupervised Relation Extraction via Bootstrapping

seed patterns
seed tuples
bootstrapping

Supervised machine learning assumes that we have lots of labeled data. Unfortunately, this is expensive. But suppose we just have a few high-precision **seed patterns**, like those in Section 20.2.1, or perhaps a few **seed tuples**. That's enough to bootstrap a classifier! **Bootstrapping** proceeds by taking the entities in the seed pair, and then finding sentences (on the web, or whatever dataset we are using) that contain both entities. From all such sentences, we extract and generalize the context around the entities to learn new patterns. Fig. 20.7 sketches a basic algorithm.

function BOOTSTRAP(*Relation R*) **returns** *new relation tuples*

tuples ← Gather a set of seed tuples that have relation *R*
iterate
  sentences ← find sentences that contain entities in *tuples*
  patterns ← generalize the context between and around entities in *sentences*
  newpairs ← use *patterns* to identify more tuples
  newpairs ← *newpairs* with high confidence
  tuples ← *tuples* + *newpairs*
return tuples

Figure 20.7 Bootstrapping from seed entity pairs to learn relations.

Suppose, for example, that we need to create a list of airline/hub pairs, and we know only that Ryanair has a hub at Charleroi. We can use this seed fact to discover new patterns by finding other mentions of this relation in our corpus. We search for the terms *Ryanair*, *Charleroi* and *hub* in some proximity. Perhaps we find the following set of sentences:

(20.6) Budget airline Ryanair, which uses Charleroi as a hub, scrapped all weekend flights out of the airport.

(20.7) All flights in and out of Ryanair's hub at Charleroi airport were grounded on Friday...

(20.8) A spokesman at Charleroi, a main hub for Ryanair, estimated that 8000 passengers had already been affected.

From these results, we can use the context of words between the entity mentions, the words before mention one, the word after mention two, and the named entity types of the two mentions, and perhaps other features, to extract general patterns such as the following:

```
/ [ORG], which uses [LOC] as a hub /
/ [ORG]'s hub at [LOC] /
/ [LOC], a main hub for [ORG] /
```

These new patterns can then be used to search for additional tuples.

confidence values
semantic drift

Bootstrapping systems also assign **confidence values** to new tuples to avoid **semantic drift**. In semantic drift, an erroneous pattern leads to the introduction of erroneous tuples, which, in turn, lead to the creation of problematic patterns and the meaning of the extracted relations 'drifts'. Consider the following example:

(20.9) Sydney has a ferry hub at Circular Quay.

If accepted as a positive example, this expression could lead to the incorrect introduction of the tuple $\langle Sydney, CircularQuay \rangle$. Patterns based on this tuple could propagate further errors into the database.

Confidence values for patterns are based on balancing two factors: the pattern's performance with respect to the current set of tuples and the pattern's productivity in terms of the number of matches it produces in the document collection. More formally, given a document collection $\mathcal{D}$, a current set of tuples T, and a proposed pattern p, we need to track two factors:

- $hits(p)$: the set of tuples in T that p matches while looking in $\mathcal{D}$
- $finds(p)$: The total set of tuples that p finds in $\mathcal{D}$

The following equation balances these considerations (Riloff and Jones, 1999).

$$Conf_{RlogF}(p) = \frac{|hits(p)|}{|finds(p)|} log(|finds(p)|) \tag{20.10}$$

This metric is generally normalized to produce a probability.

noisy-or

We can assess the confidence in a proposed new tuple by combining the evidence supporting it from all the patterns P' that match that tuple in $\mathcal{D}$ (Agichtein and Gravano, 2000). One way to combine such evidence is the **noisy-or** technique. Assume that a given tuple is supported by a subset of the patterns in P, each with its own confidence assessed as above. In the noisy-or model, we make two basic assumptions. First, that for a proposed tuple to be false, *all* of its supporting patterns must have been in error, and second, that the sources of their individual failures are all independent. If we loosely treat our confidence measures as probabilities, then the probability of any individual pattern p failing is $1 - Conf(p)$; the probability of all of the supporting patterns for a tuple being wrong is the product of their individual failure probabilities, leaving us with the following equation for our confidence in a new tuple.

$$Conf(t) = 1 - \prod_{p \in P'} (1 - Conf(p)) \tag{20.11}$$

Setting conservative confidence thresholds for the acceptance of new patterns and tuples during the bootstrapping process helps prevent the system from drifting away from the targeted relation.

20.2.4 Distant Supervision for Relation Extraction

distant supervision

Although hand-labeling text with relation labels is expensive to produce, there are ways to find indirect sources of training data. The **distant supervision** method (Mintz et al., 2009) combines the advantages of bootstrapping with supervised learning. Instead of just a handful of seeds, distant supervision uses a large database to acquire a huge number of seed examples, creates lots of noisy pattern features from all these examples and then combines them in a supervised classifier.

For example suppose we are trying to learn the *place-of-birth* relationship between people and their birth cities. In the seed-based approach, we might have only 5 examples to start with. But Wikipedia-based databases like DBPedia or Freebase have tens of thousands of examples of many relations; including over 100,000 examples of *place-of-birth*, (<Edwin Hubble, Marshfield>, <Albert Einstein, Ulm>, etc.,). The next step is to run named entity taggers on large amounts of text—Mintz et al. (2009) used 800,000 articles from Wikipedia—and extract all sentences that have two named entities that match the tuple, like the following:

...Hubble was born in Marshfield...
...Einstein, born (1879), Ulm...
...Hubble's birthplace in Marshfield...

Training instances can now be extracted from this data, one training instance for each identical tuple <relation, entity1, entity2>. Thus there will be one training instance for each of:

<born-in, Edwin Hubble, Marshfield>
<born-in, Albert Einstein, Ulm>
<born-year, Albert Einstein, 1879>

and so on.

We can then apply feature-based or neural classification. For feature-based classification, we can use standard supervised relation extraction features like the named entity labels of the two mentions, the words and dependency paths in between the mentions, and neighboring words. Each tuple will have features collected from many training instances; the feature vector for a single training instance like (<born-in,Albert Einstein, Ulm> will have lexical and syntactic features from many different sentences that mention Einstein and Ulm.

Because distant supervision has very large training sets, it is also able to use very rich features that are conjunctions of these individual features. So we will extract thousands of patterns that conjoin the entity types with the intervening words or dependency paths like these:

PER was born in LOC
PER, born (XXXX), LOC
PER's birthplace in LOC

To return to our running example, for this sentence:

(20.12) **American Airlines**, a unit of AMR, immediately matched the move, spokesman **Tim Wagner** said

we would learn rich conjunction features like this one:

M1 = ORG & M2 = PER & nextword="said"& path= $NP \uparrow NP \uparrow S \uparrow S \downarrow NP$

The result is a supervised classifier that has a huge rich set of features to use in detecting relations. Since not every test sentence will have one of the training

relations, the classifier will also need to be able to label an example as *no-relation*. This label is trained by randomly selecting entity pairs that do not appear in any Freebase relation, extracting features for them, and building a feature vector for each such tuple. The final algorithm is sketched in Fig. 20.8.

function DISTANT SUPERVISION(*Database D, Text T*) **returns** *relation classifier C*

foreach relation *R*
  foreach tuple (*e1,e2*) of entities with relation *R* in *D*
    sentences ← Sentences in *T* that contain *e1* and *e2*
    f ← Frequent features in *sentences*
    observations ← observations + new training tuple (*e1, e2, f, R*)
C ← Train supervised classifier on *observations*
return *C*

Figure 20.8 The distant supervision algorithm for relation extraction. A neural classifier would skip the feature set *f*.

Distant supervision shares advantages with each of the methods we've examined. Like supervised classification, distant supervision uses a classifier with lots of features, and supervised by detailed hand-created knowledge. Like pattern-based classifiers, it can make use of high-precision evidence for the relation between entities. Indeed, distance supervision systems learn patterns just like the hand-built patterns of early relation extractors. For example the *is-a* or *hypernym* extraction system of Snow et al. (2005) used hypernym/hyponym NP pairs from WordNet as distant supervision, and then learned new patterns from large amounts of text. Their system induced exactly the original 5 template patterns of Hearst (1992a), but also 70,000 additional patterns including these four:

NP_H like NP	*Many hormones like leptin...*
NP_H called NP	*...using a markup language called XHTML*
NP is a NP_H	Ruby is a programming language...
NP, a NP_H	*IBM, a company with a long...*

This ability to use a large number of features simultaneously means that, unlike the iterative expansion of patterns in seed-based systems, there's no semantic drift. Like unsupervised classification, it doesn't use a labeled training corpus of texts, so it isn't sensitive to genre issues in the training corpus, and relies on very large amounts of unlabeled data. Distant supervision also has the advantage that it can create training tuples to be used with neural classifiers, where features are not required.

The main problem with distant supervision is that it tends to produce low-precision results, and so current research focuses on ways to improve precision. Furthermore, distant supervision can only help in extracting relations for which a large enough database already exists. To extract new relations without datasets, or relations for new domains, purely unsupervised methods must be used.

20.2.5 Unsupervised Relation Extraction

open information extraction

The goal of unsupervised relation extraction is to extract relations from the web when we have no labeled training data, and not even any list of relations. This task is often called **open information extraction** or **Open IE**. In Open IE, the relations

are simply strings of words (usually beginning with a verb).

For example, the **ReVerb** system (Fader et al., 2011) extracts a relation from a sentence s in 4 steps:

1. Run a part-of-speech tagger and entity chunker over s
2. For each verb in s, find the longest sequence of words w that start with a verb and satisfy syntactic and lexical constraints, merging adjacent matches.
3. For each phrase w, find the nearest noun phrase x to the left which is not a relative pronoun, wh-word or existential "there". Find the nearest noun phrase y to the right.
4. Assign confidence c to the relation $r = (x, w, y)$ using a confidence classifier and return it.

A relation is only accepted if it meets syntactic and lexical constraints. The syntactic constraints ensure that it is a verb-initial sequence that might also include nouns (relations that begin with light verbs like *make*, *have*, or *do* often express the core of the relation with a noun, like *have a hub in*):

V | VP | VW*P
V = verb particle? adv?
W = (noun | adj | adv | pron | det)
P = (prep | particle | infinitive "to")

The lexical constraints are based on a dictionary D that is used to prune very rare, long relation strings. The intuition is to eliminate candidate relations that don't occur with sufficient number of distinct argument types and so are likely to be bad examples. The system first runs the above relation extraction algorithm offline on 500 million web sentences and extracts a list of all the relations that occur after normalizing them (removing inflection, auxiliary verbs, adjectives, and adverbs). Each relation r is added to the dictionary if it occurs with at least 20 different arguments. Fader et al. (2011) used a dictionary of 1.7 million normalized relations.

Finally, a confidence value is computed for each relation using a logistic regression classifier. The classifier is trained by taking 1000 random web sentences, running the extractor, and hand labeling each extracted relation as correct or incorrect. A confidence classifier is then trained on this hand-labeled data, using features of the relation and the surrounding words. Fig. 20.9 shows some sample features used in the classification.

(x,r,y) covers all words in s
the last preposition in r is *for*
the last preposition in r is *on*
len(s) $\leq$ 10
there is a coordinating conjunction to the left of r in s
r matches a lone V in the syntactic constraints
there is preposition to the left of x in s
there is an NP to the right of y in s

Figure 20.9 Features for the classifier that assigns confidence to relations extracted by the Open Information Extraction system REVERB (Fader et al., 2011).

For example the following sentence:

(20.13) United has a hub in Chicago, which is the headquarters of United Continental Holdings.

has the relation phrases *has a hub in* and *is the headquarters of* (it also has *has* and *is*, but longer phrases are preferred). Step 3 finds *United* to the left and *Chicago* to the right of *has a hub in*, and skips over *which* to find Chicago to the left of *is the headquarters of*. The final output is:

```
r1:  <United, has a hub in, Chicago>
r2:  <Chicago, is the headquarters of, United Continental Holdings>
```

The great advantage of unsupervised relation extraction is its ability to handle a huge number of relations without having to specify them in advance. The disadvantage is the need to map all the strings into some canonical form for adding to databases or knowledge graphs. Current methods focus heavily on relations expressed with verbs, and so will miss many relations that are expressed nominally.

20.2.6 Evaluation of Relation Extraction

Supervised relation extraction systems are evaluated by using test sets with human-annotated, gold-standard relations and computing precision, recall, and F-measure. Labeled precision and recall require the system to classify the relation correctly, whereas unlabeled methods simply measure a system's ability to detect entities that are related.

Semi-supervised and **unsupervised** methods are much more difficult to evaluate, since they extract totally new relations from the web or a large text. Because these methods use very large amounts of text, it is generally not possible to run them solely on a small labeled test set, and as a result it's not possible to pre-annotate a gold set of correct instances of relations.

For these methods it's possible to approximate (only) precision by drawing a random sample of relations from the output, and having a human check the accuracy of each of these relations. Usually this approach focuses on the **tuples** to be extracted from a body of text rather than on the relation **mentions**; systems need not detect every mention of a relation to be scored correctly. Instead, the evaluation is based on the set of tuples occupying the database when the system is finished. That is, we want to know if the system can discover that Ryanair has a hub at Charleroi; we don't really care how many times it discovers it. The estimated precision $\hat{P}$ is then

$$\hat{P} = \frac{\text{\# of correctly extracted relation tuples in the sample}}{\text{total \# of extracted relation tuples in the sample.}} \tag{20.14}$$

Another approach that gives us a little bit of information about recall is to compute precision at different levels of recall. Assuming that our system is able to rank the relations it produces (by probability, or confidence) we can separately compute precision for the top 1000 new relations, the top 10,000 new relations, the top 100,000, and so on. In each case we take a random sample of that set. This will show us how the precision curve behaves as we extract more and more tuples. But there is no way to directly evaluate recall.

20.3 Extracting Events

event extraction

The task of **event extraction** is to identify mentions of events in texts. For the purposes of this task, an event mention is any expression denoting an event or state that can be assigned to a particular point, or interval, in time. The following markup of the sample text on page 455 shows all the events in this text.

[EVENT Citing] high fuel prices, United Airlines [EVENT said] Friday it has [EVENT increased] fares by $6 per round trip on flights to some cities also served by lower-cost carriers. American Airlines, a unit of AMR Corp., immediately [EVENT matched] [EVENT the move], spokesman Tim Wagner [EVENT said]. United, a unit of UAL Corp., [EVENT said] [EVENT the increase] took effect Thursday and [EVENT applies] to most routes where it [EVENT competes] against discount carriers, such as Chicago to Dallas and Denver to San Francisco.

In English, most event mentions correspond to verbs, and most verbs introduce events. However, as we can see from our example, this is not always the case. Events can be introduced by noun phrases, as in *the move* and *the increase*, and some verbs fail to introduce events, as in the phrasal verb *took effect*, which refers to when the event began rather than to the event itself. Similarly, **light verbs** such as *make*, *take*, and *have* often fail to denote events. A light verb is a verb that has very little meaning itself, and the associated event is instead expressed by its direct object noun. In light verb examples like *took a flight*, it's the word *flight* that defines the event; these light verbs just provide a syntactic structure for the noun's arguments.

light verbs

Various versions of the event extraction task exist, depending on the goal. For example in the TempEval shared tasks (Verhagen et al. 2009) the goal is to extract events and aspects like their aspectual and temporal properties. Events are to be classified as actions, states, **reporting events** (*say, report, tell, explain*), perception events, and so on. The aspect, tense, and modality of each event also needs to be extracted. Thus for example the various *said* events in the sample text would be annotated as (class=REPORTING, tense=PAST, aspect=PERFECTIVE).

reporting events

Event extraction is generally modeled via supervised learning, detecting events via IOB sequence models and assigning event classes and attributes with multi-class classifiers. The input can be neural models starting from encoders; or classic feature-based models using features like those in Fig. 20.10.

Feature	Explanation
Character affixes	Character-level prefixes and suffixes of target word
Nominalization suffix	Character-level suffixes for nominalizations (e.g., *-tion*)
Part of speech	Part of speech of the target word
Light verb	Binary feature indicating that the target is governed by a light verb
Subject syntactic category	Syntactic category of the subject of the sentence
Morphological stem	Stemmed version of the target word
Verb root	Root form of the verb basis for a nominalization
WordNet hypernyms	Hypernym set for the target

Figure 20.10 Features commonly used in classic feature-based approaches to event detection.

20.4 Representing Time

temporal logic

Let's begin by introducing the basics of **temporal logic** and how human languages convey temporal information. The most straightforward theory of time holds that it flows inexorably forward and that events are associated with either points or intervals in time, as on a timeline. We can order distinct events by situating them on the timeline; one event *precedes* another if the flow of time leads from the first event

to the second. Accompanying these notions in most theories is the idea of the current moment in time. Combining this notion with the idea of a temporal ordering relationship yields the familiar notions of past, present, and future.

Various kinds of temporal representation systems can be used to talk about temporal ordering relationship. One of the most commonly used in computational modeling is the **interval algebra** of Allen (1984). Allen models all events and time expressions as intervals there is no representation for points (although intervals can be very short). In order to deal with intervals without points, he identifies 13 primitive relations that can hold between these temporal intervals. Fig. 20.11 shows these 13 **Allen relations**.

interval algebra

Allen relations

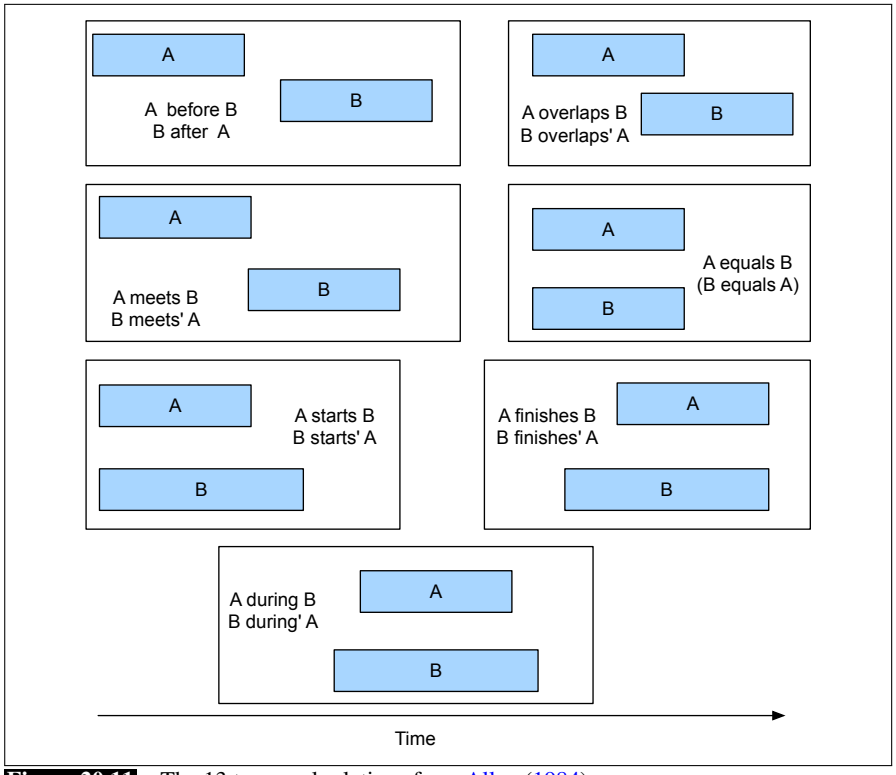


Figure 20.11 The 13 temporal relations from Allen (1984).

20.4.1 Reichenbach's reference point

The relation between simple verb tenses and points in time is by no means straightforward. The present tense can be used to refer to a future event, as in this example:

(20.15) Ok, we fly from San Francisco to Boston at 10.

Or consider the following examples:

(20.16) Flight 1902 arrived late.

(20.17) Flight 1902 had arrived late.

Although both refer to events in the past, representing them in the same way seems wrong. The second example seems to have another unnamed event lurking in the background (e.g., Flight 1902 had already arrived late *when* something else happened).

reference point

To account for this phenomena, Reichenbach (1947) introduced the notion of a **reference point**. In our simple temporal scheme, the current moment in time is equated with the time of the utterance and is used as a reference point for when the event occurred (before, at, or after). In Reichenbach's approach, the notion of the reference point is separated from the utterance time and the event time. The following examples illustrate the basics of this approach:

(20.18) When Mary's flight departed, I ate lunch.

(20.19) When Mary's flight departed, I had eaten lunch.

In both of these examples, the eating event has happened in the past, that is, prior to the utterance. However, the verb tense in the first example indicates that the eating event began when the flight departed, while the second example indicates that the eating was accomplished prior to the flight's departure. Therefore, in Reichenbach's terms the *departure* event specifies the reference point. These facts can be accommodated by additional constraints relating the *eating* and *departure* events. In the first example, the reference point precedes the *eating* event, and in the second example, the eating precedes the reference point. Figure 20.12 illustrates Reichenbach's approach with the primary English tenses. Exercise 20.4 asks you to represent these examples in FOL.

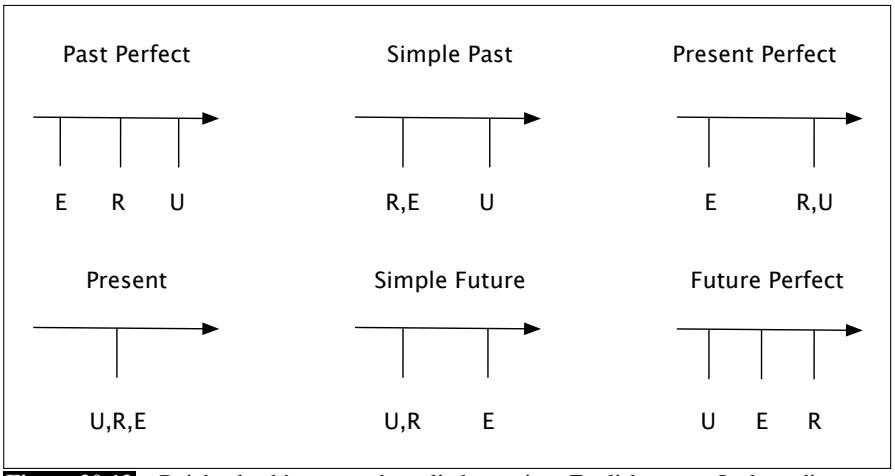


Figure 20.12 Reichenbach's approach applied to various English tenses. In these diagrams, time flows from left to right, **E** denotes the time of the event, **R** denotes the reference time, and **U** denotes the time of the utterance.

Languages have many other ways to convey temporal information besides tense. Most useful for our purposes will be temporal expressions like *in the morning* or *6:45* or *afterwards*.

(20.20) I'd like to go at 6:45 in the morning.

(20.21) Somewhere around noon, please.

(20.22) I want to take the train back afterwards.

Incidentally, temporal expressions display a fascinating metaphorical conceptual organization. Temporal expressions in English are frequently expressed in spatial terms, as is illustrated by the various uses of *at*, *in*, *somewhere*, and *near* in these examples (Lakoff and Johnson 1980, Jackendoff 1983). Metaphorical organizations such as these, in which one domain is systematically expressed in terms of another, are very common in languages of the world.

20.5 Representing Aspect

aspect

A related notion to time is **aspect**, which is what we call the way events can be categorized by their internal temporal structure or temporal contour. By this we mean questions like whether events are ongoing or have ended, or whether they are conceptualized as happening at a point in time or over some interval. Such notions of temporal contour have been used to divide event expressions into classes since Aristotle, although the set of four classes we'll introduce here is due to Vendler (1967) (you may also see the German term **aktionsart** used to refer to these classes).

aktionsart

events

states

stative

The most basic aspectual distinction is between **events** (which involve change) and **states** (which do not involve change). **Stative expressions** represent the notion of an event participant being in a **state**, or having a particular property, at a given point in time. Stative expressions capture aspects of the world at a single point in time, and conceptualize the participant as unchanging and continuous. Consider the following ATIS examples.

(20.23) I like express trains.

(20.24) I need the cheapest fare.

(20.25) I want to go first class.

In examples like these, the event participant denoted by the subject can be seen as experiencing something at a specific point in time, and don't involve any kind of internal change over time (the liking or needing is conceptualized as continuous and unchanging).

activity

Non-states (which we'll refer to as **events**) are divided into subclasses; we'll introduce three here. **Activity expressions** describe events undertaken by a participant that occur over a span of time (rather than being conceptualized as a single point in time like stative expressions), and have no particular end point. Of course in practice all things end, but the meaning of the expression doesn't represent this fact. Consider the following examples:

(20.26) She drove a Mazda.

(20.27) I live in Brooklyn.

These examples both specify that the subject is engaged in, or has engaged in, the activity specified by the verb for some period of time, but doesn't specify when the driving or living might have stopped.

Two more classes of expressions, **achievement** expressions and **accomplishment** expressions, describe events that take place over time, but also conceptualize the event as having a particular kind of endpoint or goal. The Greek word *telos* means 'end' or 'goal' and so the events described by these kinds of expressions are often called **telic** events.

telic

accomplishment expressions

Accomplishment expressions describe events that have a natural end point and result in a particular state. Consider the following examples:

(20.28) He booked me a reservation.

(20.29) The 7:00 train got me to New York City.

In these examples, an event is seen as occurring over some period of time that ends when the intended state is accomplished (i.e., the state of me having a reservation, or me being in New York City).

achievement expressions

The final aspectual class, **achievement expressions**, is only subtly different than accomplishments. Consider the following:

(20.30) She found her gate.

(20.31) I reached New York.

Like accomplishment expressions, achievement expressions result in a state. But unlike accomplishments, achievement events are 'punctual': they are thought of as happening in an instant and the verb doesn't conceptualize the process or activity leading up the state. Thus the events in these examples may in fact have been preceded by extended *searching* or *traveling* events, but the verb doesn't conceptualize these preceding processes, but rather conceptualizes the events corresponding to *finding* and *reaching* as points, not intervals.

In summary, a standard way of categorizing event expressions by their temporal contours is via these four general classes:

Stative: I know my departure gate.

Activity: John is flying.

Accomplishment: Sally booked her flight.

Achievement: She found her gate.

Before moving on, note that event expressions can easily be shifted from one class to another. Consider the following examples:

(20.32) I flew.

(20.33) I flew to New York.

The first example is a simple activity; it has no natural end point. The second example is clearly an accomplishment event since it has an end point, and results in a particular state. Clearly, the classification of an event is not solely governed by the verb, but by the semantics of the entire expression in context.

20.6 Temporally Annotated Datasets: TimeBank

TimeBank

The **TimeBank** corpus consists of American English text annotated with temporal information (Pustejovsky et al., 2003). The annotations use TimeML (Saurí et al., 2006), a markup language for time based on Allen's interval algebra discussed above (Allen, 1984). There are three types of TimeML objects: an EVENT represent events and states, a TIME represents time expressions like dates, and a LINK represents various relationships between events and times (event-event, event-time, and time-time). The links include temporal links (TLINK) for the 13 Allen relations, aspectual links (ALINK) for aspectual relationships between events and subevents, and SLINKS which mark factuality.

Consider the following sample sentence and its corresponding markup shown in Fig. 20.13, selected from one of the TimeBank documents.

(20.34) Delta Air Lines earnings soared 33% to a record in the fiscal first quarter, bucking the industry trend toward declining profits.

This text has three events and two temporal expressions (including the creation time of the article, which serves as the document time), and four temporal links that capture the using the Allen relations:

- Soaring$_{e1}$ is **included** in the fiscal first quarter$_{t58}$
- Soaring$_{e1}$ is **before** 1989-10-26$_{t57}$
- Soaring$_{e1}$ is **simultaneous** with the bucking$_{e3}$

```
<TIMEX3 tid="t57" type="DATE" value="1989-10-26"  functionInDocument="CREATION_TIME">
10/26/89  </TIMEX3>

Delta Air Lines earnings <EVENT eid="e1" class="OCCURRENCE"> soared </EVENT> 33% to a
record in  <TIMEX3 tid="t58" type="DATE" value="1989-Q1" anchorTimeID="t57"> the
fiscal first quarter </TIMEX3>, <EVENT eid="e3"  class="OCCURRENCE">bucking</EVENT>
the industry trend toward <EVENT eid="e4" class="OCCURRENCE">declining</EVENT>
profits.
```

Figure 20.13 Example from the TimeBank corpus.

- Declining$_{e4}$ **includes** soaring$_{e1}$

We can also visualize the links as a graph. The TimeBank snippet in Eq. 20.35 would be represented with a graph like Fig. 20.14.

(20.35) **[DCT:11/02/891]**$_1$: Pacific First Financial Corp. **said**$_2$ shareholders **approved**$_3$ its **acquisition**$_4$ by Royal Trustco Ltd. of Toronto for $27 a share, or $212 million. The thrift holding company **said**$_5$ it **expects**$_6$ to **obtain**$_7$ regulatory **approval**$_8$ and **complete**$_9$ the **transaction**$_{10}$ by **year-end**$_{11}$.

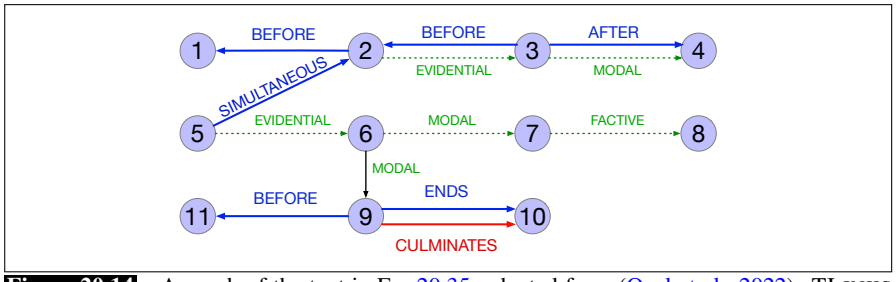


Figure 20.14 A graph of the text in Eq. 20.35, adapted from (Ocal et al., 2022). TLINKS are shown in blue, ALINKS in red, and SLINKS in green.

20.7 Automatic Temporal Analysis

Here we introduce the three common steps used in analyzing time in text:

1. Extracting **temporal expressions**
2. **Normalizing** these expressions, by converting them to a standard format.
3. **Linking** events to times and extracting time graphs and timelines

20.7.1 Extracting Temporal Expressions

absolute **relative** **duration**

Temporal expressions are phrases that refer to absolute points in time, relative times, durations, and sets of these. **Absolute** temporal expressions are those that can be mapped directly to calendar dates, times of day, or both. **Relative** temporal expressions map to particular times through some other reference point (as in *a week from last Tuesday*). Finally, **durations** denote spans of time at varying levels of granularity (seconds, minutes, days, weeks, centuries, etc.). Figure 20.15 lists some sample temporal expressions in each of these categories.

lexical triggers

Temporal expressions are grammatical constructions that often have temporal **lexical triggers** as their heads, making them easy to find. Lexical triggers might

Absolute	Relative	Durations
April 24, 1916	yesterday	four hours
The summer of '77	next semester	three weeks
10:15 AM	two weeks from yesterday	six days
The 3rd quarter of 2006	last quarter	the last three quarters

Figure 20.15 Examples of absolute, relational and durational temporal expressions.

be nouns, proper nouns, adjectives, and adverbs; full temporal expressions consist of their phrasal projections: noun phrases, adjective phrases, and adverbial phrases (Figure 20.16).

Category	Examples
Noun	*morning*, *noon*, *night*, *winter*, *dusk*, *dawn*
Proper Noun	*January, Monday, Ides, Easter, Rosh Hashana, Ramadan, Tet*
Adjective	*recent, past, annual, former*
Adverb	*hourly, daily, monthly, yearly*

Figure 20.16 Examples of temporal lexical triggers.

The task is to detect temporal expressions in running text, like this examples, shown with TIMEX3 tags (Pustejovsky et al. 2005, Ferro et al. 2005).

> A fare increase initiated <TIMEX3>last week</TIMEX3> by UAL Corp's United Airlines was matched by competitors over <TIMEX3>the weekend</TIMEX3>, marking the second successful fare increase in <TIMEX3>two weeks</TIMEX3>.

Rule-based approaches use cascades of regular expressions to recognize larger and larger chunks from previous stages, based on patterns containing parts of speech, trigger words (e.g., *February*) or classes (e.g., *MONTH*) (Chang and Manning, 2012; Strötgen and Gertz, 2013; Chambers, 2013). Here's a rule from SUTime (Chang and Manning, 2012) for detecting expressions like *3 years old*:

```
/(\d+)[-\s]($TEUnits)(s)?([-\s]old)?/
```

Sequence-labeling approaches use the standard IOB scheme, marking words that are either (I)nside, (O)utside or at the (B)eginning of a temporal expression:

A fare increase initiated last week by UAL Corp's...
O O O O B I O O O

A statistical sequence labeler is trained, using either embeddings or a fine-tuned encoder, or classic features extracted from the token and context including words, lexical triggers, and POS.

Temporal expression recognizers are evaluated with the usual recall, precision, and *F*-measures. A major difficulty for all of these very lexicalized approaches is avoiding expressions that trigger false positives:

(20.36) *1984* tells the story of Winston Smith...

(20.37) ...U2's classic *Sunday* Bloody *Sunday*

20.7.2 Temporal Normalization

temporal normalization

Temporal normalization is the task of mapping a temporal expression to a point in time or to a duration. Points in time correspond to calendar dates, to times of day, or both. Durations primarily consist of lengths of time. Normalized times

```
<TIMEX3 id="t1" type="DATE" value="2007-07-02" functionInDocument="CREATION_TIME">
    July 2, 2007 </TIMEX3> A fare increase initiated <TIMEX3 id="t2" type="DATE"
    value="2007-W26" anchorTimeID="t1">last week</TIMEX3> by United Airlines was
    matched by competitors over <TIMEX3 id="t3" type="DURATION" value="P1WE"
    anchorTimeID="t1"> the weekend </TIMEX3>, marking the second successful fare
    increase in <TIMEX3 id="t4" type="DURATION" value="P2W" anchorTimeID="t1"> two
    weeks </TIMEX3>.
```

Figure 20.17 TimeML markup including normalized values for temporal expressions.

are represented via the ISO 8601 standard for encoding temporal values (ISO8601, 2004). Fig. 20.17 reproduces our earlier example with these value attributes.

The dateline, or document date, for this text was *July 2, 2007*. The ISO representation for this kind of expression is YYYY-MM-DD, or in this case, 2007-07-02. The encodings for the temporal expressions in our sample text all follow from this date, and are shown here as values for the VALUE attribute.

The first temporal expression in the text proper refers to a particular week of the year. In the ISO standard, weeks are numbered from 01 to 53, with the first week of the year being the one that has the first Thursday of the year. These weeks are represented with the template YYYY-Wnn. The ISO week for our document date is week 27; thus the value for *last week* is represented as "2007-W26".

The next temporal expression is *the weekend*. ISO weeks begin on Monday; thus, weekends occur at the end of a week and are fully contained within a single week. Weekends are treated as durations, so the value of the VALUE attribute has to be a length. Durations are represented according to the pattern P*nx*, where *n* is an integer denoting the length and *x* represents the unit, as in P3Y for *three years* or P2D for *two days*. In this example, one weekend is captured as P1WE. In this case, there is also sufficient information to anchor this particular weekend as part of a particular week. Such information is encoded in the ANCHORTIMEID attribute. Finally, the phrase *two weeks* also denotes a duration captured as P2W. Figure 20.18 give some more examples, but there is a lot more to the various temporal annotation standards; consult ISO8601 (2004), Ferro et al. (2005), and Pustejovsky et al. (2005) for more details.

Unit	Pattern	Sample Value
Fully specified dates	YYYY-MM-DD	1991-09-28
Weeks	YYYY-Wnn	2007-W27
Weekends	PnWE	P1WE
24-hour clock times	HH:MM:SS	11:13:45
Dates and times	YYYY-MM-DDTHH:MM:SS	1991-09-28T11:00:00
Financial quarters	Qn	1999-Q3

Figure 20.18 Sample ISO patterns for representing various times and durations.

Most current approaches to temporal normalization are rule-based (Chang and Manning 2012, Strötgen and Gertz 2013). Patterns that match temporal expressions are associated with semantic analysis procedures. For example, the pattern above for recognizing phrases like *3 years old* can be associated with the predicate *Duration* that takes two arguments, the length and the unit of time:

```
pattern: /(\d+)[-\s]($TEUnits)(s)?([-\s]old)?/
result: Duration($1, $2)
```

The task is difficult because fully qualified temporal expressions are fairly rare in real texts. Most temporal expressions in news articles are incomplete and are only implicitly anchored, often with respect to the dateline of the article, which we refer

temporal anchor

to as the document's **temporal anchor**. The values of temporal expressions such as *today*, *yesterday*, or *tomorrow* can all be computed with respect to this temporal anchor. The semantic procedure for *today* simply assigns the anchor, and the attachments for *tomorrow* and *yesterday* add a day and subtract a day from the anchor, respectively. Of course, given the cyclic nature of our representations for months, weeks, days, and times of day, our temporal arithmetic procedures must use modulo arithmetic appropriate to the time unit being used.

Unfortunately, even simple expressions such as *the weekend* or *Wednesday* introduce a fair amount of complexity. In our current example, *the weekend* clearly refers to the weekend of the week that immediately precedes the document date. But this won't always be the case, as is illustrated in the following example.

(20.38) Random security checks that began yesterday at Sky Harbor will continue at least through the weekend.

In this case, the expression *the weekend* refers to the weekend of the week that the anchoring date is part of (i.e., the coming weekend). The information that signals this meaning comes from the tense of *continue*, the verb governing *the weekend*.

Relative temporal expressions are handled with temporal arithmetic similar to that used for *today* and *yesterday*. The document date indicates that our example article is ISO week 27, so the expression *last week* normalizes to the current week minus 1. To resolve ambiguous *next* and *last* expressions we consider the distance from the anchoring date to the nearest unit. *Next Friday* can refer either to the immediately next Friday or to the Friday following that, but the closer the document date is to a Friday, the more likely it is that the phrase will skip the nearest one. Such ambiguities are handled by encoding language and domain-specific heuristics into the temporal attachments.

20.7.3 Temporal Ordering of Events

The goal of temporal analysis, is to link times to events and then fit all these events into a complete timeline. This ambitious task is the subject of considerable current research but solving it with a high level of accuracy is beyond the capabilities of current systems. A somewhat simpler, but still useful, task is to impose a partial ordering on the events and temporal expressions mentioned in a text. Such an ordering can provide many of the same benefits as a true timeline. An example of such a partial ordering is the determination that the fare increase by *American Airlines* came *after* the fare increase by *United* in our sample text. Determining such an ordering can be viewed as a binary relation detection and classification task.

Even this partial ordering task assumes that in addition to the detecting and normalizing time expressions steps described above, we have already detected all the events in the text. Indeed, many temporal expressions are anchored to events mentioned in a text and not directly to other temporal expressions. Consider the following example:

(20.39) One week after the storm, JetBlue issued its customer bill of rights.

To determine when JetBlue issued its customer bill of rights we need to determine the time of *the storm* event, and then we need to modify that time by the temporal expression *one week after*.

Thus once the events and times have been detected, our goal next is to assert links between all the times and events: i.e. creating event-event, event-time, time-time, DCT-event, and DCT-time TimeML TLINKS. This can be done by training time relation classifiers to predict the correct T:INK between each pair of times/events,

supervised by the gold labels in the TimeBank corpus with features like words/embeddings, parse paths, tense and aspect The sieve-based architecture using precision-ranked sets of classifiers, which we'll introduce in Chapter 23, is also commonly used.

Systems that perform all 4 tasks (time extraction creation and normalization, event extraction, and time/event linking) include TARSQI (Verhagen et al., 2005) CLEARTK (Bethard, 2013), CAEVO (Chambers et al., 2014), and CATENA (Mirza and Tonelli, 2016).

20.8 Template Filling

Many texts contain reports of events, and possibly sequences of events, that often correspond to fairly common, stereotypical situations in the world. These abstract situations or stories, related to what have been called **scripts** (Schank and Abelson, 1977), consist of prototypical sequences of sub-events, participants, and their roles. The strong expectations provided by these scripts can facilitate the proper classification of entities, the assignment of entities into roles and relations, and most critically, the drawing of inferences that fill in things that have been left unsaid. In their simplest form, such scripts can be represented as **templates** consisting of fixed sets of **slots** that take as values **slot-fillers** belonging to particular classes. The task of **template filling** is to find documents that invoke particular scripts and then fill the slots in the associated templates with fillers extracted from the text. These slot-fillers may consist of text segments extracted directly from the text, or they may consist of concepts that have been inferred from text elements through some additional processing.

scripts

templates

template filling

A filled template from our original airline story might look like the following.

FARE-RAISE ATTEMPT:	LEAD AIRLINE:	UNITED AIRLINES
	AMOUNT:	$6
	EFFECTIVE DATE:	2006-10-26
	FOLLOWER:	AMERICAN AIRLINES

This template has four slots (LEAD AIRLINE, AMOUNT, EFFECTIVE DATE, FOLLOWER). The next section describes a standard sequence-labeling approach to filling slots. Section 20.8.2 then describes an older system based on the use of cascades of finite-state transducers and designed to address a more complex template-filling task that current learning-based systems don't yet address.

20.8.1 Machine Learning Approaches to Template Filling

In the standard paradigm for template filling, we are given training documents with text spans annotated with predefined templates and their slot fillers. Our goal is to create one template for each event in the input, filling in the slots with text spans.

The task is generally modeled by training two separate supervised systems. The first system decides whether the template is present in a particular sentence. This task is called **template recognition** or sometimes, in a perhaps confusing bit of terminology, *event recognition*. Template recognition can be treated as a text classification task, with features extracted from every sequence of words that was labeled in training documents as filling any slot from the template being detected. The usual

template recognition

set of features can be used: tokens, embeddings, word shapes, part-of-speech tags, syntactic chunk tags, and named entity tags.

role-filler extraction

The second system has the job of **role-filler extraction**. A separate classifier is trained to detect each role (LEAD-AIRLINE, AMOUNT, and so on). This can be a binary classifier that is run on every noun-phrase in the parsed input sentence, or a sequence model run over sequences of words. Each role classifier is trained on the labeled data in the training set. Again, the usual set of features can be used, but now trained only on an individual noun phrase or the fillers of a single slot.

Multiple non-identical text segments might be labeled with the same slot label. For example in our sample text, the strings *United* or *United Airlines* might be labeled as the LEAD AIRLINE. These are not incompatible choices and the coreference resolution techniques introduced in Chapter 23 can provide a path to a solution.

A variety of annotated collections have been used to evaluate this style of approach to template filling, including sets of job announcements, conference calls for papers, restaurant guides, and biological texts. A key open question is extracting templates in cases where there is no training data or even predefined templates, by inducing templates as sets of linked events (Chambers and Jurafsky, 2011).

20.8.2 Earlier Finite-State Template-Filling Systems

The templates above are relatively simple. But consider the task of producing a template that contained all the information in a text like this one (Grishman and Sundheim, 1995):

> Bridgestone Sports Co. said Friday it has set up a joint venture in Taiwan with a local concern and a Japanese trading house to produce golf clubs to be shipped to Japan. The joint venture, Bridgestone Sports Taiwan Co., capitalized at 20 million new Taiwan dollars, will start production in January 1990 with production of 20,000 iron and "metal wood" clubs a month.

The MUC-5 'joint venture' task (the *Message Understanding Conferences* were a series of U.S. government-organized information-extraction evaluations) was to produce hierarchically linked templates describing joint ventures. Figure 20.19 shows a structure produced by the FASTUS system (Hobbs et al., 1997). Note how the filler of the ACTIVITY slot of the TIE-UP template is itself a template with slots.

Tie-up-1		**Activity-1**:	
RELATIONSHIP	tie-up	COMPANY	Bridgestone Sports Taiwan Co.
ENTITIES	Bridgestone Sports Co.	PRODUCT	iron and "metal wood" clubs
	a local concern	START DATE	DURING: January 1990
	a Japanese trading house		
JOINT VENTURE	Bridgestone Sports Taiwan Co.		
ACTIVITY	Activity-1		
AMOUNT	NT$20000000		

Figure 20.19 The templates produced by FASTUS given the input text on page 477.

Early systems for dealing with these complex templates were based on cascades of transducers based on handwritten rules, as sketched in Fig. 20.20.

The first four stages use handwritten regular expression and grammar rules to do basic tokenization, chunking, and parsing. Stage 5 then recognizes entities and events with a recognizer based on finite-state transducers (FSTs), and inserts the recognized objects into the appropriate slots in templates. This FST recognizer is based

No.	Step	Description
1	**Tokens**	Tokenize input stream of characters
2	**Complex Words**	Multiword phrases, numbers, and proper names.
3	**Basic phrases**	Segment sentences into noun and verb groups
4	**Complex phrases**	Identify complex noun groups and verb groups
5	**Semantic Patterns**	Identify entities and events, insert into templates.
6	**Merging**	Merge references to the same entity or event

Figure 20.20 Levels of processing in FASTUS (Hobbs et al., 1997). Each level extracts a specific type of information which is then passed on to the next higher level.

on hand-built regular expressions like the following (NG indicates Noun-Group and VG Verb-Group), which matches the first sentence of the news story above.

```
NG(Company/ies) VG(Set-up) NG(Joint-Venture) with NG(Company/ies)
VG(Produce) NG(Product)
```

The result of processing these two sentences is the five draft templates (Fig. 20.21) that must then be merged into the single hierarchical structure shown in Fig. 20.19. The merging algorithm, after performing coreference resolution, merges two activities that are likely to be describing the same events.

#	Template/Slot	Value
1	RELATIONSHIP:	TIE-UP
	ENTITIES:	Bridgestone Co., a local concern, a Japanese trading house
2	ACTIVITY:	PRODUCTION
	PRODUCT:	"golf clubs"
3	RELATIONSHIP:	TIE-UP
	JOINT VENTURE:	"Bridgestone Sports Taiwan Co."
	AMOUNT:	NT$20000000
4	ACTIVITY:	PRODUCTION
	COMPANY:	"Bridgestone Sports Taiwan Co."
	STARTDATE:	DURING: January 1990
5	ACTIVITY:	PRODUCTION
	PRODUCT:	"iron and "metal wood" clubs"

Figure 20.21 The five partial templates produced by stage 5 of FASTUS. These templates are merged in stage 6 to produce the final template shown in Fig. 20.19 on page 477.

20.9 Summary

This chapter has explored techniques for extracting limited forms of semantic content from texts.

- **Relations among entities** can be extracted by pattern-based approaches, supervised learning methods when annotated training data is available, lightly supervised **bootstrapping** methods when small numbers of **seed tuples** or **seed patterns** are available, **distant supervision** when a database of relations is available, and **unsupervised** or **Open IE** methods.
- Reasoning about time can be facilitated by detection and normalization of **temporal expressions**.

- **Events** can be ordered in time using sequence models and classifiers trained on temporally- and event-labeled data like the **TimeBank corpus**.
- **Template-filling** applications can recognize stereotypical situations in texts and assign elements from the text to roles represented as **fixed sets of slots**.

Historical Notes

The earliest work on information extraction addressed the template-filling task in the context of the Frump system (DeJong, 1982). Later work was stimulated by the U.S. government-sponsored MUC conferences (Sundheim 1991, Sundheim 1992, Sundheim 1993, Sundheim 1995). Early MUC systems like CIRCUS system (Lehnert et al., 1991) and SCISOR (Jacobs and Rau, 1990) were quite influential and inspired later systems like FASTUS (Hobbs et al., 1997). Chinchor et al. (1993) describe the MUC evaluation techniques.

Due to the difficulty of porting systems from one domain to another, attention shifted to machine learning approaches. Early supervised learning approaches to IE (Cardie 1993, Cardie 1994, Riloff 1993, Soderland et al. 1995, Huffman 1996) focused on automating the knowledge acquisition process, mainly for finite-state rule-based systems. Their success, and the earlier success of HMM-based speech recognition, led to the use of sequence labeling (HMMs: Bikel et al. 1997; MEMMs McCallum et al. 2000; CRFs: Lafferty et al. 2001), and a wide exploration of features (Zhou et al., 2005). Neural approaches followed from the pioneering results of Collobert et al. (2011), who applied a CRF on top of a convolutional net.

Progress in this area continues to be stimulated by formal evaluations with shared benchmark datasets, including the Automatic Content Extraction (ACE) evaluations of 2000-2007 on named entity recognition, relation extraction, and temporal expressions[1], the **KBP** (**Knowledge Base Population**) evaluations (Ji et al. 2010, Surdeanu 2013) of relation extraction tasks like **slot filling** (extracting attributes ('slots') like age, birthplace, and spouse for a given entity) and a series of SemEval workshops (Hendrickx et al., 2009).

KBP
slot filling

Semisupervised relation extraction was first proposed by Hearst (1992b), and extended by systems like AutoSlog-TS (Riloff, 1996), DIPRE (Brin, 1998), SNOWBALL (Agichtein and Gravano, 2000), and Jones et al. (1999). The distant supervision algorithm we describe was drawn from Mintz et al. (2009), who first used the term 'distant supervision' (which was suggested to them by Chris Manning) but similar ideas had occurred in earlier systems like Craven and Kumlien (1999) and Morgan et al. (2004) under the name *weakly labeled data*, as well as in Snow et al. (2005) and Wu and Weld (2007). Among the many extensions are Wu and Weld (2010), Riedel et al. (2010), and Ritter et al. (2013). Open IE systems include KNOWITALL Etzioni et al. (2005), TextRunner (Banko et al., 2007), and REVERB (Fader et al., 2011). See Riedel et al. (2013) for a universal schema that combines the advantages of distant supervision and Open IE.

[1] www.nist.gov/speech/tests/ace/

Exercises

20.1 Acronym expansion, the process of associating a phrase with an acronym, can be accomplished by a simple form of relational analysis. Develop a system based on the relation analysis approaches described in this chapter to populate a database of acronym expansions. If you focus on English **Three Letter Acronyms** (TLAs) you can evaluate your system's performance by comparing it to Wikipedia's TLA page.

20.2 Acquire the CMU seminar corpus and develop a template-filling system by using any of the techniques mentioned in Section 20.8. Analyze how well your system performs as compared with state-of-the-art results on this corpus.

20.3 A useful functionality in newer email and calendar applications is the ability to associate temporal expressions connected with events in email (doctor's appointments, meeting planning, party invitations, etc.) with specific calendar entries. Collect a corpus of email containing temporal expressions related to event planning. How do these expressions compare to the kinds of expressions commonly found in news text that we've been discussing in this chapter?

20.4 For the following sentences, give FOL translations that capture the temporal relationships between the events.

1. When Mary's flight departed, I ate lunch.
2. When Mary's flight departed, I had eaten lunch.

CHAPTER

21 Semantic Role Labeling

"Who, What, Where, When, With what, Why, How"
The seven circumstances, associated with Hermagoras and Aristotle (Sloan, 2010)

Sometime between the 7th and 4th centuries BCE, the Indian grammarian Pāṇini[1] wrote a famous treatise on Sanskrit grammar, the Aṣṭādhyāyī ('8 books'), a treatise that has been called "one of the greatest monuments of human intelligence" (Bloomfield, 1933, 11). The work describes the linguistics of the Sanskrit language in the form of 3959 sutras, each very efficiently (since it had to be memorized!) expressing part of a formal rule system that brilliantly prefigured modern mechanisms of formal language theory (Penn and Kiparsky, 2012). One set of rules describes the **kārakas**, semantic relationships between a verb and noun arguments, roles like *agent*, *instrument*, or *destination*. Pāṇini's work was the earliest we know of that modeled the linguistic realization of events and their participants. This task of understanding how participants relate to events—being able to answer the question "Who did what to whom" (and perhaps also "when and where")—is a central question of natural language processing.

Let's move forward 2.5 millennia to the present and consider the very mundane goal of understanding text about a purchase of stock by XYZ Corporation. This purchasing event and its participants can be described by a wide variety of surface forms. The event can be described by a verb (*sold, bought*) or a noun (*purchase*), and XYZ Corp can be the syntactic subject (of *bought*), the indirect object (of *sold*), or in a genitive or noun compound relation (with the noun *purchase*) despite having notionally the same role in all of them:

- XYZ corporation bought the stock.
- They sold the stock to XYZ corporation.
- The stock was bought by XYZ corporation.
- The purchase of the stock by XYZ corporation...
- The stock purchase by XYZ corporation...

In this chapter we introduce a level of representation that captures the commonality between these sentences: there was a purchase event, the participants were XYZ Corp and some stock, and XYZ Corp was the buyer. These shallow semantic representations , **semantic roles**, express the role that arguments of a predicate take in the event, codified in databases like PropBank and FrameNet. We'll introduce **semantic role labeling**, the task of assigning roles to spans in sentences, and **selectional restrictions**, the preferences that predicates express about their arguments, such as the fact that the theme of *eat* is generally something edible.

[1] Figure shows a birch bark manuscript from Kashmir of the Rupavatra, a grammatical textbook based on the Sanskrit grammar of Panini. Image from the Wellcome Collection.

21.1 Semantic Roles

Consider the meanings of the arguments *Sasha*, *Pat*, *the window*, and *the door* in these two sentences.

(21.1) Sasha broke the window.

(21.2) Pat opened the door.

The subjects *Sasha* and *Pat*, what we might call the *breaker* of the window-breaking event and the *opener* of the door-opening event, have something in common. They are both volitional actors, often animate, and they have direct causal responsibility for their events.

thematic roles
agents
theme

Thematic roles are a way to capture this semantic commonality between *breakers* and *openers*. We say that the subjects of both these verbs are **agents**. Thus, AGENT is the thematic role that represents an abstract idea such as volitional causation. Similarly, the direct objects of both these verbs, the *BrokenThing* and *OpenedThing*, are both prototypically inanimate objects that are affected in some way by the action. The semantic role for these participants is **theme**.

Thematic Role	Definition
AGENT	The volitional causer of an event
EXPERIENCER	The experiencer of an event
FORCE	The non-volitional causer of the event
THEME	The participant most directly affected by an event
RESULT	The end product of an event
CONTENT	The proposition or content of a propositional event
INSTRUMENT	An instrument used in an event
BENEFICIARY	The beneficiary of an event
SOURCE	The origin of the object of a transfer event
GOAL	The destination of an object of a transfer event

Figure 21.1 Some commonly used thematic roles with their definitions.

Although thematic roles are one of the oldest linguistic models, as we saw above, their modern formulation is due to Fillmore (1968) and Gruber (1965). Although there is no universally agreed-upon set of roles, Figs. 21.1 and 21.2 list some thematic roles that have been used in various computational papers, together with rough definitions and examples. Most thematic role sets have about a dozen roles, but we'll see sets with smaller numbers of roles with even more abstract meanings, and sets with very large numbers of roles that are specific to situations. We'll use the general term **semantic roles** for all sets of roles, whether small or large.

semantic roles

21.2 Diathesis Alternations

The main reason computational systems use semantic roles is to act as a shallow meaning representation that can let us make simple inferences that aren't possible from the pure surface string of words, or even from the parse tree. To extend the earlier examples, if a document says that *Company A acquired Company B*, we'd like to know that this answers the query *Was Company B acquired?* despite the fact that the two sentences have very different surface syntax. Similarly, this shallow semantics might act as a useful intermediate language in machine translation.

Thematic Role	Example
AGENT	*The waiter* spilled the soup.
EXPERIENCER	*John* has a headache.
FORCE	*The wind* blows debris from the mall into our yards.
THEME	Only after Benjamin Franklin broke *the ice*...
RESULT	The city built a *regulation-size baseball diamond*...
CONTENT	Mona asked *"You met Mary Ann at a supermarket?"*
INSTRUMENT	He poached catfish, stunning them *with a shocking device*...
BENEFICIARY	Whenever Ann Callahan makes hotel reservations *for her boss*...
SOURCE	I flew in *from Boston*.
GOAL	I drove *to Portland*.

Figure 21.2 Some prototypical examples of various thematic roles.

Semantic roles thus help generalize over different surface realizations of predicate arguments. For example, while the AGENT is often realized as the subject of the sentence, in other cases the THEME can be the subject. Consider these possible realizations of the thematic arguments of the verb *break*:

(21.3) *John broke the window.*
AGENT THEME

(21.4) *John broke the window with a rock.*
AGENT THEME INSTRUMENT

(21.5) *The rock broke the window.*
INSTRUMENT THEME

(21.6) *The window broke.*
THEME

(21.7) *The window was broken by John.*
THEME AGENT

These examples suggest that *break* has (at least) the possible arguments AGENT, THEME, and INSTRUMENT. The set of thematic role arguments taken by a verb is often called the **thematic grid**, θ-grid, or **case frame**. We can see that there are (among others) the following possibilities for the realization of these arguments of *break*:

thematic grid
case frame

AGENT/Subject, THEME/Object
AGENT/Subject, THEME/Object, INSTRUMENT/PP_{with}
INSTRUMENT/Subject, THEME/Object
THEME/Subject

It turns out that many verbs allow their thematic roles to be realized in various syntactic positions. For example, verbs like *give* can realize the THEME and GOAL arguments in two different ways:

(21.8) a. *Doris gave the book to Cary.*
AGENT THEME GOAL

b. *Doris gave Cary the book.*
AGENT GOAL THEME

These multiple argument structure realizations (the fact that *break* can take AGENT, INSTRUMENT, or THEME as subject, and *give* can realize its THEME and GOAL in either order) are called **verb alternations** or **diathesis alternations**. The alternation we showed above for *give*, the **dative alternation**, seems to occur with particular semantic classes of verbs, including "verbs of future having" (*advance*, *allocate*, *offer*,

verb alternation
dative alternation

owe), "send verbs" (*forward*, *hand*, *mail*), "verbs of throwing" (*kick*, *pass*, *throw*), and so on. Levin (1993) lists for 3100 English verbs the semantic classes to which they belong (47 high-level classes, divided into 193 more specific classes) and the various alternations in which they participate. These lists of verb classes have been incorporated into the online resource VerbNet (Kipper et al., 2000), which links each verb to both WordNet and FrameNet entries.

21.3 Semantic Roles: Problems with Thematic Roles

Representing meaning at the thematic role level seems like it should be useful in dealing with complications like diathesis alternations. Yet it has proved quite difficult to come up with a standard set of roles, and equally difficult to produce a formal definition of roles like AGENT, THEME, or INSTRUMENT.

For example, researchers attempting to define role sets often find they need to fragment a role like AGENT or THEME into many specific roles. Levin and Rappaport Hovav (2005) summarize a number of such cases, such as the fact there seem to be at least two kinds of INSTRUMENTS, *intermediary* instruments that can appear as subjects and *enabling* instruments that cannot:

(21.9) a. Shelly cut the banana with a knife.
b. The knife cut the banana.

(21.10) a. Shelly ate the sliced banana with a fork.
b. *The fork ate the sliced banana.

In addition to the fragmentation problem, there are cases in which we'd like to reason about and generalize across semantic roles, but the finite discrete lists of roles don't let us do this.

Finally, it has proved difficult to formally define the thematic roles. Consider the AGENT role; most cases of AGENTS are animate, volitional, sentient, causal, but any individual noun phrase might not exhibit all of these properties.

semantic role

These problems have led to alternative **semantic role** models that use either many fewer or many more roles.

proto-agent
proto-patient

The first of these options is to define **generalized semantic roles** that abstract over the specific thematic roles. For example, PROTO-AGENT and PROTO-PATIENT are generalized roles that express roughly agent-like and roughly patient-like meanings. These roles are defined, not by necessary and sufficient conditions, but rather by a set of heuristic features that accompany more agent-like or more patient-like meanings. Thus, the more an argument displays agent-like properties (being volitionally involved in the event, causing an event or a change of state in another participant, being sentient or intentionally involved, moving) the greater the likelihood that the argument can be labeled a PROTO-AGENT. The more patient-like the properties (undergoing change of state, causally affected by another participant, stationary relative to other participants, etc.), the greater the likelihood that the argument can be labeled a PROTO-PATIENT.

The second direction is instead to define semantic roles that are specific to a particular verb or a particular group of semantically related verbs or nouns.

In the next two sections we describe two commonly used lexical resources that make use of these alternative versions of semantic roles. **PropBank** uses both proto-roles and verb-specific semantic roles. **FrameNet** uses semantic roles that are specific to a general semantic idea called a *frame*.

21.4 The Proposition Bank

PropBank

The **Proposition Bank**, generally referred to as **PropBank**, is a resource of sentences annotated with semantic roles. The English PropBank labels all the sentences in the Penn TreeBank; the Chinese PropBank labels sentences in the Penn Chinese TreeBank. Because of the difficulty of defining a universal set of thematic roles, the semantic roles in PropBank are defined with respect to an individual verb sense. Each sense of each verb thus has a specific set of roles, which are given only numbers rather than names: **Arg0**, **Arg1**, **Arg2**, and so on. In general, **Arg0** represents the PROTO-AGENT, and **Arg1**, the PROTO-PATIENT. The semantics of the other roles are less consistent, often being defined specifically for each verb. Nonetheless there are some generalization; the **Arg2** is often the benefactive, instrument, attribute, or end state, the **Arg3** the start point, benefactive, instrument, or attribute, and the **Arg4** the end point.

Here are some slightly simplified PropBank entries for one sense each of the verbs *agree* and *fall*. Such PropBank entries are called **frame files**; note that the definitions in the frame file for each role ("Other entity agreeing", "Extent, amount fallen") are informal glosses intended to be read by humans, rather than being formal definitions.

(21.11) **agree.01**
Arg0: Agreer
Arg1: Proposition
Arg2: Other entity agreeing

Ex1: [$_{\text{Arg0}}$ The group] *agreed* [$_{\text{Arg1}}$ it wouldn't make an offer].
Ex2: [$_{\text{ArgM-TMP}}$ Usually] [$_{\text{Arg0}}$ John] *agrees* [$_{\text{Arg2}}$ with Mary] [$_{\text{Arg1}}$ on everything].

(21.12) **fall.01**
Arg1: Logical subject, patient, thing falling
Arg2: Extent, amount fallen
Arg3: start point
Arg4: end point, end state of arg1
Ex1: [$_{\text{Arg1}}$ Sales] *fell* [$_{\text{Arg4}}$ to $25 million] [$_{\text{Arg3}}$ from $27 million].
Ex2: [$_{\text{Arg1}}$ The average junk bond] *fell* [$_{\text{Arg2}}$ by 4.2%].

Note that there is no Arg0 role for *fall*, because the normal subject of *fall* is a PROTO-PATIENT.

The PropBank semantic roles can be useful in recovering shallow semantic information about verbal arguments. Consider the verb *increase*:

(21.13) **increase.01** "go up incrementally"
Arg0: causer of increase
Arg1: thing increasing
Arg2: amount increased by, EXT, or MNR
Arg3: start point
Arg4: end point

A PropBank semantic role labeling would allow us to infer the commonality in the event structures of the following three examples, that is, that in each case *Big Fruit Co.* is the AGENT and *the price of bananas* is the THEME, despite the differing surface forms.

(21.14) [$_{\text{Arg0}}$ Big Fruit Co.] increased [$_{\text{Arg1}}$ the price of bananas].
(21.15) [$_{\text{Arg1}}$ The price of bananas] was increased again [$_{\text{Arg0}}$ by Big Fruit Co.]
(21.16) [$_{\text{Arg1}}$ The price of bananas] increased [$_{\text{Arg2}}$ 5%].

PropBank also has a number of non-numbered arguments called **ArgMs**, (ArgM-TMP, ArgM-LOC, etc.) which represent modification or adjunct meanings. These are relatively stable across predicates, so aren't listed with each frame file. Data labeled with these modifiers can be helpful in training systems to detect temporal, location, or directional modification across predicates. Some of the ArgMs include:

TMP	when?	yesterday evening, now
LOC	where?	at the museum, in San Francisco
DIR	where to/from?	down, to Bangkok
MNR	how?	clearly, with much enthusiasm
PRP/CAU	why?	because ... , in response to the ruling
REC		themselves, each other
ADV	miscellaneous	
PRD	secondary predication	...ate the meat raw

NomBank While PropBank focuses on verbs, a related project, **NomBank** (Meyers et al., 2004) adds annotations to noun predicates. For example the noun *agreement* in *Apple's agreement with IBM* would be labeled with Apple as the Arg0 and IBM as the Arg2. This allows semantic role labelers to assign labels to arguments of both verbal and nominal predicates.

21.5 FrameNet

While making inferences about the semantic commonalities across different sentences with *increase* is useful, it would be even more useful if we could make such inferences in many more situations, across different verbs, and also between verbs and nouns. For example, we'd like to extract the similarity among these three sentences:

(21.17) [$_{\text{Arg1}}$ The price of bananas] increased [$_{\text{Arg2}}$ 5%].
(21.18) [$_{\text{Arg1}}$ The price of bananas] rose [$_{\text{Arg2}}$ 5%].
(21.19) There has been a [$_{\text{Arg2}}$ 5%] rise [$_{\text{Arg1}}$ in the price of bananas].

Note that the second example uses the different verb *rise*, and the third example uses the noun rather than the verb *rise*. We'd like a system to recognize that *the price of bananas* is what went up, and that *5%* is the amount it went up, no matter whether the *5%* appears as the object of the verb *increased* or as a nominal modifier of the noun *rise*.

FrameNet The **FrameNet** project is another semantic-role-labeling project that attempts to address just these kinds of problems (Baker et al. 1998, Fillmore et al. 2003, Fillmore and Baker 2009, Ruppenhofer et al. 2016). Whereas roles in the PropBank project are specific to an individual verb, roles in the FrameNet project are specific to a **frame**.

What is a frame? Consider the following set of words:

reservation, flight, travel, buy, price, cost, fare, rates, meal, plane

There are many individual lexical relations of hyponymy, synonymy, and so on between many of the words in this list. The resulting set of relations does not,

however, add up to a complete account of how these words are related. They are clearly all defined with respect to a coherent chunk of common-sense background information concerning air travel.

frame

We call the holistic background knowledge that unites these words a **frame** (Fillmore, 1985). The idea that groups of words are defined with respect to some background information is widespread in artificial intelligence and cognitive science, where besides **frame** we see related works like a **model** (Johnson-Laird, 1983), or even **script** (Schank and Abelson, 1977).

model

script

A frame in FrameNet is a background knowledge structure that defines a set of frame-specific semantic roles, called **frame elements**, and includes a set of predicates that use these roles. Each word evokes a frame and profiles some aspect of the frame and its elements. The FrameNet dataset includes a set of frames and frame elements, the lexical units associated with each frame, and a set of labeled example sentences. For example, the **change_position_on_a_scale** frame is defined as follows:

frame elements

> This frame consists of words that indicate the change of an Item's position on a scale (the Attribute) from a starting point (Initial_value) to an end point (Final_value).

Some of the semantic roles (frame elements) in the frame are defined as in Fig. 21.3. Note that these are separated into **core roles**, which are frame specific, and **non-core roles**, which are more like the Arg-M arguments in PropBank, expressing more general properties of time, location, and so on.

core roles

non-core roles

Core Roles	
ATTRIBUTE	The ATTRIBUTE is a scalar property that the ITEM possesses.
DIFFERENCE	The distance by which an ITEM changes its position on the scale.
FINAL_STATE	A description that presents the ITEM's state after the change in the ATTRIBUTE's value as an independent predication.
FINAL_VALUE	The position on the scale where the ITEM ends up.
INITIAL_STATE	A description that presents the ITEM's state before the change in the ATTRIBUTE's value as an independent predication.
INITIAL_VALUE	The initial position on the scale from which the ITEM moves away.
ITEM	The entity that has a position on the scale.
VALUE_RANGE	A portion of the scale, typically identified by its end points, along which the values of the ATTRIBUTE fluctuate.
Some Non-Core Roles	
DURATION	The length of time over which the change takes place.
SPEED	The rate of change of the VALUE.
GROUP	The GROUP in which an ITEM changes the value of an ATTRIBUTE in a specified way.

Figure 21.3 The frame elements in the **change_position_on_a_scale** frame from the FrameNet Labelers Guide (Ruppenhofer et al., 2016).

Here are some example sentences:

(21.20) [$_{\text{ITEM}}$ Oil] *rose* [$_{\text{ATTRIBUTE}}$ in price] [$_{\text{DIFFERENCE}}$ by 2%].

(21.21) [$_{\text{ITEM}}$ It] has *increased* [$_{\text{FINAL\_STATE}}$ to having them 1 day a month].

(21.22) [$_{\text{ITEM}}$ Microsoft shares] *fell* [$_{\text{FINAL\_VALUE}}$ to 7 5/8].

(21.23) [$_{\text{ITEM}}$ Colon cancer incidence] *fell* [$_{\text{DIFFERENCE}}$ by 50%] [$_{\text{GROUP}}$ among men].

(21.24) a steady *increase* [INITIAL_VALUE from 9.5] [FINAL_VALUE to 14.3] [ITEM in dividends]

(21.25) a [DIFFERENCE 5%] [ITEM dividend] *increase*...

Note from these example sentences that the frame includes target words like *rise*, *fall*, and *increase*. In fact, the complete frame consists of the following words:

VERBS:	dwindle	move	soar	escalation	shift
advance	edge	mushroom	swell	explosion	tumble
climb	explode	plummet	swing	fall	
decline	fall	reach	triple	fluctuation	**ADVERBS:**
decrease	fluctuate	rise	tumble	gain	increasingly
diminish	gain	rocket		growth	
dip	grow	shift	**NOUNS:**	hike	
double	increase	skyrocket	decline	increase	
drop	jump	slide	decrease	rise	

FrameNet also codes relationships between frames, allowing frames to inherit from each other, or representing relations between frames like causation (and generalizations among frame elements in different frames can be represented by inheritance as well). Thus, there is a **Cause_change_of_position_on_a_scale** frame that is linked to the **Change_of_position_on_a_scale** frame by the **cause** relation, but that adds an AGENT role and is used for causative examples such as the following:

(21.26) [AGENT They] *raised* [ITEM the price of their soda] [DIFFERENCE by 2%].

Together, these two frames would allow an understanding system to extract the common event semantics of all the verbal and nominal causative and non-causative usages.

FrameNets have also been developed for many other languages including Spanish, German, Japanese, Portuguese, Italian, and Chinese.

21.6 Semantic Role Labeling

semantic role labeling

Semantic role labeling (sometimes shortened as **SRL**) is the task of automatically finding the **semantic roles** of each argument of each predicate in a sentence. Current approaches to semantic role labeling are based on supervised machine learning, often using the FrameNet and PropBank resources to specify what counts as a predicate, define the set of roles used in the task, and provide training and test sets.

Recall that the difference between these two models of semantic roles is that FrameNet (21.27) employs many frame-specific frame elements as roles, while PropBank (21.28) uses a smaller number of numbered argument labels that can be interpreted as verb-specific labels, along with the more general ARGM labels. Some examples:

(21.27)

[You]	can't	[blame]	[the program]	[for being unable to identify it]
COGNIZER		TARGET	EVALUEE	REASON

(21.28)

[The San Francisco Examiner]	issued	[a special edition]	[yesterday]
ARG0	TARGET	ARG1	ARGM-TMP

21.6.1 A Feature-based Algorithm for Semantic Role Labeling

A simplified feature-based semantic role labeling algorithm is sketched in Fig. 21.4. Feature-based algorithms—from the very earliest systems like (Simmons, 1973)—begin by parsing, using broad-coverage parsers to assign a parse to the input string.

Figure 21.5 shows a parse of (21.28) above. The parse is then traversed to find all words that are predicates.

For each of these predicates, the algorithm examines each node in the parse tree and uses supervised classification to decide the semantic role (if any) it plays for this predicate. Given a labeled training set such as PropBank or FrameNet, a feature vector is extracted for each node, using feature templates described in the next subsection. A 1-of-N classifier is then trained to predict a semantic role for each constituent given these features, where N is the number of potential semantic roles plus an extra NONE role for non-role constituents. Any standard classification algorithms can be used. Finally, for each test sentence to be labeled, the classifier is run on each relevant constituent.

```
function SEMANTICROLELABEL(words) returns labeled tree

    parse ← PARSE(words)
    for each predicate in parse do
        for each node in parse do
            featurevector ← EXTRACTFEATURES(node, predicate, parse)
            CLASSIFYNODE(node, featurevector, parse)
```

Figure 21.4 A generic semantic-role-labeling algorithm. CLASSIFYNODE is a 1-of-*N* classifier that assigns a semantic role (or NONE for non-role constituents), trained on labeled data such as FrameNet or PropBank.

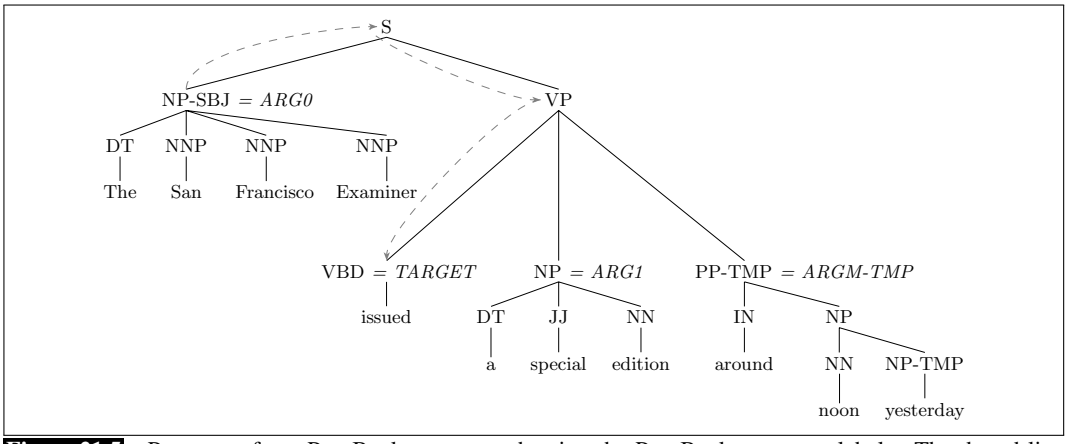


Figure 21.5 Parse tree for a PropBank sentence, showing the PropBank argument labels. The dotted line shows the **path** feature NP↑S↓VP↓VBD for ARG0, the NP-SBJ constituent *The San Francisco Examiner.*

Instead of training a single-stage classifier as in Fig. 21.5, the node-level classification task can be broken down into multiple steps:

1. **Pruning:** Since only a small number of the constituents in a sentence are arguments of any given predicate, many systems use simple heuristics to prune unlikely constituents.
2. **Identification:** a binary classification of each node as an argument to be labeled or a NONE.
3. **Classification:** a 1-of-*N* classification of all the constituents that were labeled as arguments by the previous stage

The separation of identification and classification may lead to better use of features (different features may be useful for the two tasks) or to computational efficiency.

Global Optimization

The classification algorithm of Fig. 21.5 classifies each argument separately ('locally'), making the simplifying assumption that each argument of a predicate can be labeled independently. This assumption is false; there are interactions between arguments that require a more 'global' assignment of labels to constituents. For example, constituents in FrameNet and PropBank are required to be non-overlapping. More significantly, the semantic roles of constituents are not independent. For example PropBank does not allow multiple identical arguments; two constituents of the same verb cannot both be labeled ARG0 .

Role labeling systems thus often add a fourth step to deal with global consistency across the labels in a sentence. For example, the local classifiers can return a list of possible labels associated with probabilities for each constituent, and a second-pass Viterbi decoding or re-ranking approach can be used to choose the best consensus label. Integer linear programming (ILP) is another common way to choose a solution that conforms best to multiple constraints.

Features for Semantic Role Labeling

Most systems use some generalization of the core set of features introduced by Gildea and Jurafsky (2000). Common basic features templates (demonstrated on the *NP-SBJ* constituent *The San Francisco Examiner* in Fig. 21.5) include:

- The governing **predicate**, in this case the verb *issued*. The predicate is a crucial feature since labels are defined only with respect to a particular predicate.
- The **phrase type** of the constituent, in this case, *NP* (or *NP-SBJ*). Some semantic roles tend to appear as *NP*s, others as *S* or *PP*, and so on.
- The **headword** of the constituent, *Examiner*. The headword of a constituent can be computed with standard head rules, such as those given in Appendix D in Fig. 18.17. Certain headwords (e.g., pronouns) place strong constraints on the possible semantic roles they are likely to fill.
- The **headword part of speech** of the constituent, *NNP*.
- The **path** in the parse tree from the constituent to the predicate. This path is marked by the dotted line in Fig. 21.5. Following Gildea and Jurafsky (2000), we can use a simple linear representation of the path, NP↑S↓VP↓VBD. ↑ and ↓ represent upward and downward movement in the tree, respectively. The path is very useful as a compact representation of many kinds of grammatical function relationships between the constituent and the predicate.
- The **voice** of the clause in which the constituent appears, in this case, **active** (as contrasted with **passive**). Passive sentences tend to have strongly different linkings of semantic roles to surface form than do active ones.
- The binary **linear position** of the constituent with respect to the predicate, either **before** or **after**.
- The **subcategorization** of the predicate, the set of expected arguments that appear in the verb phrase. We can extract this information by using the phrase-structure rule that expands the immediate parent of the predicate; VP → VBD NP PP for the predicate in Fig. 21.5.
- The named entity type of the constituent.

- The first words and the last word of the constituent.

The following feature vector thus represents the first NP in our example (recall that most observations will have the value NONE rather than, for example, ARG0, since most constituents in the parse tree will not bear a semantic role):

ARG0: [issued, NP, Examiner, NNP, NP↑S↓VP↓VBD, active, before, VP → NP PP, ORG, The, Examiner]

Other features are often used in addition, such as sets of n-grams inside the constituent, or more complex versions of the path features (the upward or downward halves, or whether particular nodes occur in the path).

It's also possible to use dependency parses instead of constituency parses as the basis of features, for example using dependency parse paths instead of constituency paths.

21.6.2 A Neural Algorithm for Semantic Role Labeling

A simple neural approach to SRL is to treat it as a sequence labeling task like named-entity recognition, using the BIO approach. Let's assume that we are given the predicate and the task is just detecting and labeling spans. Recall that with BIO tagging, we have a begin and end tag for each possible role (B-ARG0, I-ARG0; B-ARG1, I-ARG1, and so on), plus an outside tag O.

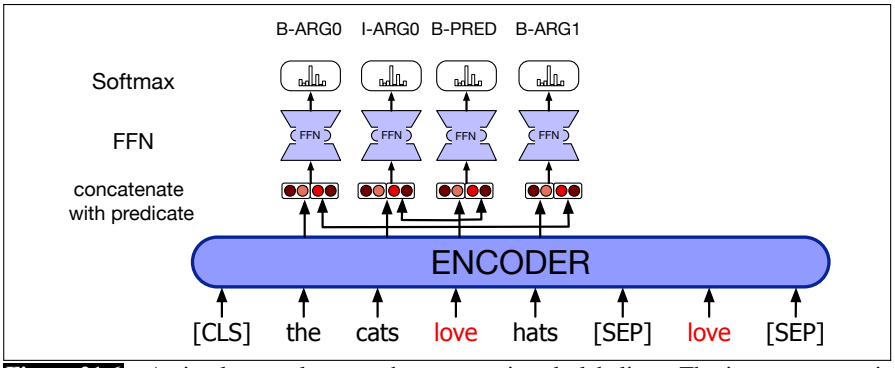


Figure 21.6 A simple neural approach to semantic role labeling. The input sentence is followed by [SEP] and an extra input for the predicate, in this case *love*. The encoder outputs are concatenated to an indicator variable which is 1 for the predicate and 0 for all other words After He et al. (2017) and Shi and Lin (2019).

As with all the taggers, the goal is to compute the highest probability tag sequence $\hat{y}$, given the input sequence of words w:

$$\hat{y} = \underset{y \in T}{\operatorname{argmax}} P(\mathbf{y}|\mathbf{w})$$

Fig. 21.6 shows a sketch of a standard algorithm from He et al. (2017). Here each input word is mapped to pretrained embeddings, and then each token is concatenated with the predicate embedding and then passed through a feedforward network with a softmax which outputs a distribution over each SRL label. For decoding, a CRF layer can be used instead of the MLP layer on top of the biLSTM output to do global inference, but in practice this doesn't seem to provide much benefit.

21.6.3 Evaluation of Semantic Role Labeling

The standard evaluation for semantic role labeling is to require that each argument label must be assigned to the exactly correct word sequence or parse constituent, and then compute precision, recall, and *F*-measure. Identification and classification can also be evaluated separately. Two common datasets used for evaluation are CoNLL-2005 (Carreras and Màrquez, 2005) and CoNLL-2012 (Pradhan et al., 2013).

21.7 Selectional Restrictions

selectional restriction

We turn in this section to another way to represent facts about the relationship between predicates and arguments. A **selectional restriction** is a semantic type constraint that a verb imposes on the kind of concepts that are allowed to fill its argument roles. Consider the two meanings associated with the following example:

(21.29) I want to eat someplace nearby.

There are two possible parses and semantic interpretations for this sentence. In the sensible interpretation, *eat* is intransitive and the phrase *someplace nearby* is an adjunct that gives the location of the eating event. In the nonsensical *speaker-as-Godzilla* interpretation, *eat* is transitive and the phrase *someplace nearby* is the direct object and the THEME of the eating, like the NP *Malaysian food* in the following sentences:

(21.30) I want to eat Malaysian food.

How do we know that *someplace nearby* isn't the direct object in this sentence? One useful cue is the semantic fact that the THEME of EATING events tends to be something that is *edible*. This restriction placed by the verb *eat* on the filler of its THEME argument is a selectional restriction.

Selectional restrictions are associated with senses, not entire lexemes. We can see this in the following examples of the lexeme *serve*:

(21.31) The restaurant serves green-lipped mussels.

(21.32) Which airlines serve Denver?

Example (21.31) illustrates the offering-food sense of *serve*, which ordinarily restricts its THEME to be some kind of food Example (21.32) illustrates the *provides a commercial service to* sense of *serve*, which constrains its THEME to be some type of appropriate location.

Selectional restrictions vary widely in their specificity. The verb *imagine*, for example, imposes strict requirements on its AGENT role (restricting it to humans and other animate entities) but places very few semantic requirements on its THEME role. A verb like *diagonalize*, on the other hand, places a very specific constraint on the filler of its THEME role: it has to be a matrix, while the arguments of the adjective *odorless* are restricted to concepts that could possess an odor:

(21.33) In rehearsal, I often ask the musicians to *imagine* a tennis game.

(21.34) Radon is an *odorless* gas that can't be detected by human senses.

(21.35) To *diagonalize* a matrix is to find its eigenvalues.

These examples illustrate that the set of concepts we need to represent selectional restrictions (being a matrix, being able to possess an odor, etc) is quite open ended. This distinguishes selectional restrictions from other features for representing lexical knowledge, like parts-of-speech, which are quite limited in number.

21.7.1 Representing Selectional Restrictions

One way to capture the semantics of selectional restrictions is to use and extend the event representation of Appendix F. Recall that the neo-Davidsonian representation of an event consists of a single variable that stands for the event, a predicate denoting the kind of event, and variables and relations for the event roles. Ignoring the issue of the λ-structures and using thematic roles rather than deep event roles, the semantic contribution of a verb like *eat* might look like the following:

$$\exists e,x,y\ Eating(e) \wedge Agent(e,x) \wedge Theme(e,y)$$

With this representation, all we know about y, the filler of the THEME role, is that it is associated with an *Eating* event through the *Theme* relation. To stipulate the selectional restriction that y must be something edible, we simply add a new term to that effect:

$$\exists e,x,y\ Eating(e) \wedge Agent(e,x) \wedge Theme(e,y) \wedge EdibleThing(y)$$

When a phrase like *ate a hamburger* is encountered, a semantic analyzer can form the following kind of representation:

$$\exists e,x,y\ Eating(e) \wedge Eater(e,x) \wedge Theme(e,y) \wedge EdibleThing(y) \wedge Hamburger(y)$$

This representation is perfectly reasonable since the membership of y in the category *Hamburger* is consistent with its membership in the category *EdibleThing*, assuming a reasonable set of facts in the knowledge base. Correspondingly, the representation for a phrase such as *ate a takeoff* would be ill-formed because membership in an event-like category such as *Takeoff* would be inconsistent with membership in the category *EdibleThing*.

While this approach adequately captures the semantics of selectional restrictions, there are two problems with its direct use. First, using FOL to perform the simple task of enforcing selectional restrictions is overkill. Other, far simpler, formalisms can do the job with far less computational cost. The second problem is that this approach presupposes a large, logical knowledge base of facts about the concepts that make up selectional restrictions. Unfortunately, although such common-sense knowledge bases are being developed, none currently have the kind of coverage necessary to the task.

A more practical approach is to state selectional restrictions in terms of WordNet synsets rather than as logical concepts. Each predicate simply specifies a WordNet synset as the selectional restriction on each of its arguments. A meaning representation is well-formed if the role filler word is a hyponym (subordinate) of this synset.

For our *ate a hamburger* example, for instance, we could set the selectional restriction on the THEME role of the verb *eat* to the synset **{food, nutrient}**, glossed as *any substance that can be metabolized by an animal to give energy and build tissue*. Luckily, the chain of hypernyms for *hamburger* shown in Fig. 21.7 reveals that hamburgers are indeed food. Again, the filler of a role need not match the restriction synset exactly; it just needs to have the synset as one of its superordinates.

We can apply this approach to the THEME roles of the verbs *imagine*, *lift*, and *diagonalize*, discussed earlier. Let us restrict *imagine*'s THEME to the synset {entity}, *lift*'s THEME to {physical entity}, and *diagonalize* to {matrix}. This arrangement correctly permits *imagine a hamburger* and *lift a hamburger*, while also correctly ruling out *diagonalize a hamburger*.

```
Sense 1
hamburger, beefburger --
(a fried cake of minced beef served on a bun)
=> sandwich
   => snack food
      => dish
         => nutriment, nourishment, nutrition...
            => food, nutrient
               => substance
                  => matter
                     => physical entity
                        => entity
```

Figure 21.7 Evidence from WordNet that hamburgers are edible.

21.7.2 Selectional Preferences

In the earliest implementations, selectional restrictions were considered strict constraints on the kind of arguments a predicate could take (Katz and Fodor 1963, Hirst 1987). For example, the verb *eat* might require that its THEME argument be [+FOOD]. Early word sense disambiguation systems used this idea to rule out senses that violated the selectional restrictions of their governing predicates.

Very quickly, however, it became clear that these selectional restrictions were better represented as preferences rather than strict constraints (Wilks 1975b, Wilks 1975a). For example, selectional restriction violations (like inedible arguments of eat) often occur in well-formed sentences, for example because they are negated (21.36), or because selectional restrictions are overstated (21.37):

(21.36) But it fell apart in 1931, perhaps because people realized you can't **eat** gold for lunch if you're hungry.

(21.37) In his two championship trials, Mr. Kulkarni **ate** glass on an empty stomach, accompanied only by water and tea.

Modern systems for selectional preferences therefore specify the relation between a predicate and its possible arguments with soft constraints of some kind.

Selectional Association

selectional preference strength

One of the most influential has been the **selectional association** model of Resnik (1993). Resnik defines the idea of **selectional preference strength** as the general amount of information that a predicate tells us about the semantic class of its arguments. For example, the verb *eat* tells us a lot about the semantic class of its direct objects, since they tend to be edible. The verb *be*, by contrast, tells us less about its direct objects. The selectional preference strength can be defined by the difference in information between two distributions: the distribution of expected semantic classes $P(c)$ (how likely is it that a direct object will fall into class c) and the distribution of expected semantic classes for the particular verb $P(c|v)$ (how likely is it that the direct object of the specific verb v will fall into semantic class c). The greater the difference between these distributions, the more information the verb is giving us about possible objects. The difference between these two distributions can be quantified by **relative entropy**, or the Kullback-Leibler divergence (Kullback and Leibler, 1951). The Kullback-Leibler or **KL divergence** $D(P||Q)$ expresses the

relative entropy

KL divergence

difference between two probability distributions P and Q

$$D(P||Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)} \tag{21.38}$$

The selectional preference $S_R(v)$ uses the KL divergence to express how much information, in bits, the verb v expresses about the possible semantic class of its argument.

$$\begin{aligned} S_R(v) &= D(P(c|v)||P(c)) \\ &= \sum_c P(c|v) \log \frac{P(c|v)}{P(c)} \end{aligned} \tag{21.39}$$

selectional association

Resnik then defines the **selectional association** of a particular class and verb as the relative contribution of that class to the general selectional preference of the verb:

$$A_R(v,c) = \frac{1}{S_R(v)} P(c|v) \log \frac{P(c|v)}{P(c)} \tag{21.40}$$

The selectional association is thus a probabilistic measure of the strength of association between a predicate and a class dominating the argument to the predicate. Resnik estimates the probabilities for these associations by parsing a corpus, counting all the times each predicate occurs with each argument word, and assuming that each word is a partial observation of all the WordNet concepts containing the word. The following table from Resnik (1996) shows some sample high and low selectional associations for verbs and some WordNet semantic classes of their direct objects.

Verb	Direct Object Semantic Class	Assoc	Direct Object Semantic Class	Assoc
read	WRITING	6.80	ACTIVITY	-.20
write	WRITING	7.26	COMMERCE	0
see	ENTITY	5.79	METHOD	-0.01

Selectional Preference via Conditional Probability

An alternative to using selectional association between a verb and the WordNet class of its arguments is to use the conditional probability of an argument word given a predicate verb, directly modeling the strength of association of one verb (predicate) with one noun (argument).

The conditional probability model can be computed by parsing a very large corpus (billions of words), and computing co-occurrence counts: how often a given verb occurs with a given noun in a given relation. The conditional probability of an argument noun given a verb for a particular relation $P(n|v,r)$ can then be used as a selectional preference metric for that pair of words (Brockmann and Lapata 2003, Keller and Lapata 2003):

$$P(n|v,r) = \begin{cases} \frac{C(n,v,r)}{C(v,r)} & \text{if } C(n,v,r) > 0 \\ 0 & \text{otherwise} \end{cases}$$

The inverse probability $P(v|n,r)$ was found to have better performance in some cases (Brockmann and Lapata, 2003):

$$P(v|n,r) = \begin{cases} \frac{C(n,v,r)}{C(n,r)} & \text{if } C(n,v,r) > 0 \\ 0 & \text{otherwise} \end{cases}$$

An even simpler approach is to use the simple log co-occurrence frequency of the predicate with the argument $\log count(v,n,r)$ instead of conditional probability; this seems to do better for extracting preferences for syntactic subjects rather than objects (Brockmann and Lapata, 2003).

Evaluating Selectional Preferences

pseudowords

One way to evaluate models of selectional preferences is to use **pseudowords** (Gale et al. 1992b, Schütze 1992a). A pseudoword is an artificial word created by concatenating a test word in some context (say *banana*) with a confounder word (say *door*) to create *banana-door*). The task of the system is to identify which of the two words is the original word. To evaluate a selectional preference model (for example on the relationship between a verb and a direct object) we take a test corpus and select all verb tokens. For each verb token (say *drive*) we select the direct object (e.g., *car*), concatenated with a confounder word that is its *nearest neighbor*, the noun with the frequency closest to the original (say *house*), to make *car/house*). We then use the selectional preference model to choose which of *car* and *house* are more preferred objects of *drive*, and compute how often the model chooses the correct original object (e.g., *car*) (Chambers and Jurafsky, 2010).

Another evaluation metric is to get human preferences for a test set of verb-argument pairs, and have them rate their degree of plausibility. This is usually done by using magnitude estimation, a technique from psychophysics, in which subjects rate the plausibility of an argument proportional to a modulus item. A selectional preference model can then be evaluated by its correlation with the human preferences (Keller and Lapata, 2003).

21.8 Primitive Decomposition of Predicates

componential analysis

One way of thinking about the semantic roles we have discussed through the chapter is that they help us define the roles that arguments play in a decompositional way, based on finite lists of thematic roles (agent, patient, instrument, proto-agent, proto-patient, etc.). This idea of decomposing meaning into sets of primitive semantic elements or features, called **primitive decomposition** or **componential analysis**, has been taken even further, and focused particularly on predicates.

Consider these examples of the verb *kill*:

(21.41) Jim killed his philodendron.

(21.42) Jim did something to cause his philodendron to become not alive.

There is a truth-conditional ('propositional semantics') perspective from which these two sentences have the same meaning. Assuming this equivalence, we could represent the meaning of *kill* as:

(21.43) KILL(x,y) $\Leftrightarrow$ CAUSE(x, BECOME(NOT(ALIVE(y))))

thus using semantic primitives like *do*, *cause*, *become not*, and *alive*.

Indeed, one such set of potential semantic primitives has been used to account for some of the verbal alternations discussed in Section 21.2 (Lakoff 1965, Dowty 1979). Consider the following examples.

(21.44) John opened the door. $\Rightarrow$ CAUSE(John, BECOME(OPEN(door)))

(21.45) The door opened. $\Rightarrow$ BECOME(OPEN(door))

(21.46) The door is open. ⇒ OPEN(door)

The decompositional approach asserts that a single state-like predicate associated with *open* underlies all of these examples. The differences among the meanings of these examples arises from the combination of this single predicate with the primitives CAUSE and BECOME.

While this approach to primitive decomposition can explain the similarity between states and actions or causative and non-causative predicates, it still relies on having a large number of predicates like *open*. More radical approaches choose to break down these predicates as well. One such approach to verbal predicate decomposition that played a role in early natural language systems is **conceptual dependency** (CD), a set of ten primitive predicates, shown in Fig. 21.8.

conceptual dependency

Primitive	Definition
ATRANS	The abstract transfer of possession or control from one entity to another
PTRANS	The physical transfer of an object from one location to another
MTRANS	The transfer of mental concepts between entities or within an entity
MBUILD	The creation of new information within an entity
PROPEL	The application of physical force to move an object
MOVE	The integral movement of a body part by an animal
INGEST	The taking in of a substance by an animal
EXPEL	The expulsion of something from an animal
SPEAK	The action of producing a sound
ATTEND	The action of focusing a sense organ

Figure 21.8 A set of conceptual dependency primitives.

Below is an example sentence along with its CD representation. The verb *brought* is translated into the two primitives ATRANS and PTRANS to indicate that the waiter both physically conveyed the check to Mary and passed control of it to her. Note that CD also associates a fixed set of thematic roles with each primitive to represent the various participants in the action.

(21.47) The waiter brought Mary the check.

$$\exists x, y\ Atrans(x) \wedge Actor(x, Waiter) \wedge Object(x, Check) \wedge To(x, Mary)$$
$$\wedge Ptrans(y) \wedge Actor(y, Waiter) \wedge Object(y, Check) \wedge To(y, Mary)$$

21.9 Summary

- **Semantic roles** are abstract models of the role an argument plays in the event described by the predicate.
- **Thematic roles** are a model of semantic roles based on a single finite list of roles. Other semantic role models include per-verb semantic role lists and **proto-agent/proto-patient**, both of which are implemented in **PropBank**, and per-frame role lists, implemented in **FrameNet**.

- **Semantic role labeling** is the task of assigning semantic role labels to the constituents of a sentence. The task is generally treated as a supervised machine learning task, with models trained on PropBank or FrameNet. Algorithms generally start by parsing a sentence and then automatically tag each parse tree node with a semantic role. Neural models map straight from words end-to-end.
- Semantic **selectional restrictions** allow words (particularly predicates) to post constraints on the semantic properties of their argument words. **Selectional preference** models (like **selectional association** or simple conditional probability) allow a weight or probability to be assigned to the association between a predicate and an argument word or class.

Historical Notes

Although the idea of semantic roles dates back to Pāṇini, they were re-introduced into modern linguistics by Gruber (1965), Fillmore (1966) and Fillmore (1968). Fillmore had become interested in argument structure by studying Lucien Tesnière's groundbreaking *Éléments de Syntaxe Structurale* (Tesnière, 1959) in which the term 'dependency' was introduced and the foundations were laid for dependency grammar. Following Tesnière's terminology, Fillmore first referred to argument roles as *actants* (Fillmore, 1966) but quickly switched to the term *case*, (see Fillmore (2003)) and proposed a universal list of semantic roles or cases (Agent, Patient, Instrument, etc.), that could be taken on by the arguments of predicates. Verbs would be listed in the lexicon with their **case frame**, the list of obligatory (or optional) case arguments.

The idea that semantic roles could provide an intermediate level of semantic representation that could help map from syntactic parse structures to deeper, more fully-specified representations of meaning was quickly adopted in natural language processing, and systems for extracting case frames were created for machine translation (Wilks, 1973), question-answering (Hendrix et al., 1973), spoken-language processing (Nash-Webber, 1975), and dialogue systems (Bobrow et al., 1977). General-purpose semantic role labelers were developed. The earliest ones (Simmons, 1973) first parsed a sentence by means of an ATN (Augmented Transition Network) parser. Each verb then had a set of rules specifying how the parse should be mapped to semantic roles. These rules mainly made reference to grammatical functions (subject, object, complement of specific prepositions) but also checked constituent internal features such as the animacy of head nouns. Later systems assigned roles from pre-built parse trees, again by using dictionaries with verb-specific case frames (Levin 1977, Marcus 1980).

By 1977 case representation was widely used and taught in AI and NLP courses, and was described as a standard of natural language processing in the first edition of Winston's 1977 textbook *Artificial Intelligence*.

In the 1980s Fillmore proposed his model of *frame semantics*, later describing the intuition as follows:

> "The idea behind frame semantics is that speakers are aware of possibly quite complex situation types, packages of connected expectations, that go by various names—frames, schemas, scenarios, scripts, cultural narratives, memes—and the words in our language are understood with such frames as their presupposed background." (Fillmore, 2012, p. 712)

The word *frame* seemed to be in the air for a suite of related notions proposed at about the same time by Minsky (1974), Hymes (1974), and Goffman (1974), as well as related notions with other names like *scripts* (Schank and Abelson, 1975) and *schemata* (Bobrow and Norman, 1975) (see Tannen (1979) for a comparison). Fillmore was also influenced by the semantic field theorists and by a visit to the Yale AI lab where he took notice of the lists of slots and fillers used by early information extraction systems like DeJong (1982) and Schank and Abelson (1977). In the 1990s Fillmore drew on these insights to begin the FrameNet corpus annotation project.

At the same time, Beth Levin drew on her early case frame dictionaries (Levin, 1977) to develop her book which summarized sets of verb classes defined by shared argument realizations (Levin, 1993). The VerbNet project built on this work (Kipper et al., 2000), leading soon afterwards to the PropBank semantic-role-labeled corpus created by Martha Palmer and colleagues (Palmer et al., 2005).

The combination of rich linguistic annotation and corpus-based approach instantiated in FrameNet and PropBank led to a revival of automatic approaches to semantic role labeling, first on FrameNet (Gildea and Jurafsky, 2000) and then on PropBank data (Gildea and Palmer, 2002, inter alia). The problem first addressed in the 1970s by handwritten rules was thus now generally recast as one of supervised machine learning enabled by large and consistent databases. Many popular features used for role labeling are defined in Gildea and Jurafsky (2002), Surdeanu et al. (2003), Xue and Palmer (2004), Pradhan et al. (2005), Che et al. (2009), and Zhao et al. (2009). The use of dependency rather than constituency parses was introduced in the CoNLL-2008 shared task (Surdeanu et al., 2008). For surveys see Palmer et al. (2010) and Màrquez et al. (2008).

The use of neural approaches to semantic role labeling was pioneered by Collobert et al. (2011), who applied a CRF on top of a convolutional net. Early work like Foland, Jr. and Martin (2015) focused on using dependency features. Later work eschewed syntactic features altogether; Zhou and Xu (2015b) introduced the use of a stacked (6-8 layer) biLSTM architecture, and (He et al., 2017) showed how to augment the biLSTM architecture with highway networks and also replace the CRF with A* decoding that make it possible to apply a wide variety of global constraints in SRL decoding.

Most semantic role labeling schemes only work within a single sentence, focusing on the object of the verbal (or nominal, in the case of NomBank) predicate. However, in many cases, a verbal or nominal predicate may have an **implicit argument**: one that appears only in a contextual sentence, or perhaps not at all and must be inferred. In the two sentences *This house has a new owner. The sale was finalized 10 days ago.* the *sale* in the second sentence has no ARG1, but a reasonable reader would infer that the Arg1 should be the *house* mentioned in the prior sentence. Finding these arguments, **implicit argument detection** (sometimes shortened as **iSRL**) was introduced by Gerber and Chai (2010) and Ruppenhofer et al. (2010). See Do et al. (2017) for more recent neural models.

implicit argument

iSRL

To avoid the need for huge labeled training sets, unsupervised approaches for semantic role labeling attempt to induce the set of semantic roles by clustering over arguments. The task was pioneered by Riloff and Schmelzenbach (1998) and Swier and Stevenson (2004); see Grenager and Manning (2006), Titov and Klementiev (2012), Lang and Lapata (2014), Woodsend and Lapata (2015), and Titov and Khoddam (2014).

Recent innovations in frame labeling include **connotation frames**, which mark richer information about the argument of predicates. Connotation frames mark the

sentiment of the writer or reader toward the arguments (for example using the verb *survive* in *he survived a bombing* expresses the writer's sympathy toward the subject *he* and negative sentiment toward the bombing. See Chapter 22 for more details.

Selectional preference has been widely studied beyond the selectional association models of Resnik (1993) and Resnik (1996). Methods have included clustering (Rooth et al., 1999), discriminative learning (Bergsma et al., 2008a), and topic models (Séaghdha 2010, Ritter et al. 2010), and constraints can be expressed at the level of words or classes (Agirre and Martinez, 2001). Selectional preferences have also been successfully integrated into semantic role labeling (Erk 2007, Zapirain et al. 2013, Do et al. 2017).

Exercises

CHAPTER

22 Lexicons for Sentiment, Affect, and Connotation

Some day we'll be able to measure the power of words
Maya Angelou

affective

In this chapter we turn to tools for interpreting **affective** meaning, extending our study of sentiment analysis in Appendix K. We use the word 'affective', following the tradition in **affective computing** (Picard, 1995) to mean emotion, sentiment, personality, mood, and attitudes. Affective meaning is closely related to **subjectivity**, the study of a speaker or writer's evaluations, opinions, emotions, and speculations (Wiebe et al., 1999).

subjectivity

How should affective meaning be defined? One influential typology of affective states comes from Scherer (2000), who defines each class of affective states by factors like its cognitive realization and time course (Fig. 22.1).

Emotion: Relatively brief episode of response to the evaluation of an external or internal event as being of major significance.
(*angry, sad, joyful, fearful, ashamed, proud, elated, desperate*)

Mood: Diffuse affect state, most pronounced as change in subjective feeling, of low intensity but relatively long duration, often without apparent cause.
(*cheerful, gloomy, irritable, listless, depressed, buoyant*)

Interpersonal stance: Affective stance taken toward another person in a specific interaction, coloring the interpersonal exchange in that situation.
(*distant, cold, warm, supportive, contemptuous, friendly*)

Attitude: Relatively enduring, affectively colored beliefs, preferences, and predispositions towards objects or persons.
(*liking, loving, hating, valuing, desiring*)

Personality traits: Emotionally laden, stable personality dispositions and behavior tendencies, typical for a person.
(*nervous, anxious, reckless, morose, hostile, jealous*)

Figure 22.1 The Scherer typology of affective states (Scherer, 2000).

We can design extractors for each of these kinds of affective states. Appendix K already introduced *sentiment analysis*, the task of extracting the positive or negative orientation that a writer expresses in a text. This corresponds in Scherer's typology to the extraction of **attitudes**: figuring out what people like or dislike, from affect-rich texts like consumer reviews of books or movies, newspaper editorials, or public sentiment in blogs or tweets.

Detecting **emotion** and **moods** is useful for detecting whether a student is confused, engaged, or certain when interacting with a tutorial system, whether a caller to a help line is frustrated, whether someone's blog posts or tweets indicated depression. Detecting emotions like fear in novels, for example, could help us trace what groups or situations are feared and how that changes over time.

Detecting different **interpersonal stances** can be useful when extracting information from human-human conversations. The goal here is to detect stances like friendliness or awkwardness in interviews or friendly conversations, for example for summarizing meetings or finding parts of a conversation where people are especially excited or engaged, conversational **hot spots** that can help in meeting summarization. Detecting the **personality** of a user—such as whether the user is an **extrovert** or the extent to which they are **open to experience**— can help improve conversational agents, which seem to work better if they match users' personality expectations (Mairesse and Walker, 2008). And affect is important for generation as well as recognition; synthesizing affect is important for conversational agents in various domains, including literacy tutors such as children's storybooks, or computer games.

In Appendix K we introduced the use of naive Bayes classification to classify a document's sentiment. Various classifiers have been successfully applied to many of these tasks, using all the words in the training set as input to a classifier which then determines the affect status of the text.

In this chapter we focus on an alternative model, in which instead of using every word as a feature, we focus only on certain words, ones that carry particularly strong cues to affect or sentiment. We call these lists of words **affective lexicons** or **sentiment lexicons**. These lexicons presuppose a fact about semantics: that words have *affective meanings* or **connotations**. The word *connotation* has different meanings in different fields, but here we use it to mean the aspects of a word's meaning that are related to a writer or reader's emotions, sentiment, opinions, or evaluations. In addition to their ability to help determine the affective status of a text, connotation lexicons can be useful features for other kinds of affective tasks, and for computational social science analysis.

connotations

In the next sections we introduce basic theories of emotion, show how sentiment lexicons are a special case of emotion lexicons, and mention some useful lexicons. We then survey three ways for building lexicons: human labeling, semi-supervised, and supervised. Finally, we talk about how to detect affect toward a particular entity, and introduce connotation frames.

22.1 Defining Emotion

One of the most important affective classes is **emotion**, which Scherer (2000) defines as a "relatively brief episode of response to the evaluation of an external or internal event as being of major significance".

emotion

Detecting emotion has the potential to improve a number of language processing tasks. Emotion recognition could help dialogue systems like tutoring systems detect that a student was unhappy, bored, hesitant, confident, and so on. Automatically detecting emotions in reviews or customer responses (anger, dissatisfaction, trust) could help businesses recognize specific problem areas or ones that are going well. Emotion can play a role in medical NLP tasks like helping diagnose depression or suicidal intent. Detecting emotions expressed toward characters in novels might play a role in understanding how different social groups were viewed by society at different times.

Computational models of emotion in NLP have mainly been based on two families of theories of emotion (out of the many studied in the field of affective science). In one of these families, emotions are viewed as fixed atomic units, limited in number, and from which others are generated, often called **basic emotions** (Tomkins

basic emotions

1962, Plutchik 1962), a model dating back to Darwin. Perhaps the most well-known of this family of theories are the 6 emotions proposed by Ekman (e.g., Ekman 1999) to be universally present in all cultures: *surprise, happiness, anger, fear, disgust, sadness*. Another atomic theory is the Plutchik (1980) wheel of emotion, consisting of 8 basic emotions in four opposing pairs: *joy–sadness*, *anger–fear*, *trust–disgust*, and *anticipation–surprise*, together with the emotions derived from them, shown in Fig. 22.2.

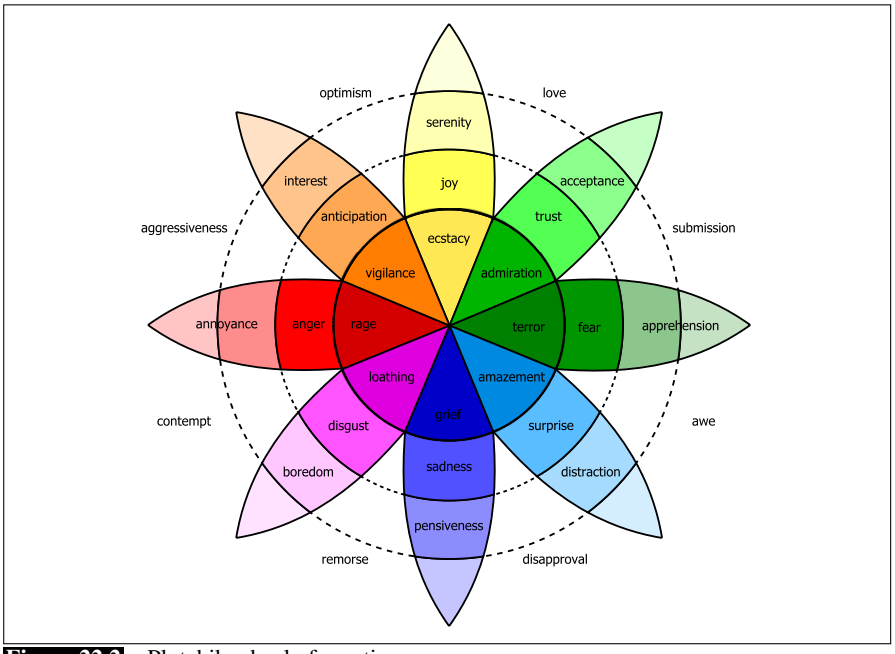


Figure 22.2 Plutchik wheel of emotion.

The second class of emotion theories widely used in NLP views emotion as a space in 2 or 3 dimensions (Russell, 1980). Most models include the two dimensions **valence** and **arousal**, and many add a third, **dominance**. These can be defined as:

valence: the pleasantness of the stimulus

arousal: the level of alertness, activeness, or energy provoked by the stimulus

dominance: the degree of control or dominance exerted by the stimulus or the emotion

Sentiment can be viewed as a special case of this second view of emotions as points in space. In particular, the **valence** dimension, measuring how pleasant or unpleasant a word is, is often used directly as a measure of sentiment.

In these lexicon-based models of affect, the affective meaning of a word is generally fixed, irrespective of the linguistic context in which a word is used, or the dialect or culture of the speaker. By contrast, other models in affective science represent emotions as much richer processes involving cognition (Barrett et al., 2007). In **appraisal theory**, for example, emotions are complex processes, in which a person considers how an event is congruent with their goals, taking into account variables like the agency, certainty, urgency, novelty and control associated with the event (Moors et al., 2013). Computational models in NLP taking into account these richer theories of emotion will likely play an important role in future work.

22.2 Available Sentiment and Affect Lexicons

A wide variety of affect lexicons have been created and released. The most basic lexicons label words along one dimension of semantic variability, generally called "sentiment" or "valence".

General Inquirer

In the simplest lexicons this dimension is represented in a binary fashion, with a wordlist for positive words and a wordlist for negative words. The oldest is the **General Inquirer** (Stone et al., 1966), which drew on content analysis and on early work in the cognitive psychology of word meaning (Osgood et al., 1957). The General Inquirer has a lexicon of 1915 positive words and a lexicon of 2291 negative words (as well as other lexicons discussed below). The MPQA Subjectivity lexicon (Wilson et al., 2005) has 2718 positive and 4912 negative words drawn from prior lexicons plus a bootstrapped list of subjective words and phrases (Riloff and Wiebe, 2003). Each entry in the lexicon is hand-labeled for sentiment and also labeled for reliability (strongly subjective or weakly subjective). The polarity lexicon of Hu and Liu (2004) gives 2006 positive and 4783 negative words, drawn from product reviews, labeled using a bootstrapping method from WordNet.

Positive	admire, amazing, assure, celebration, charm, eager, enthusiastic, excellent, fancy, fantastic, frolic, graceful, happy, joy, luck, majesty, mercy, nice, patience, perfect, proud, rejoice, relief, respect, satisfactorily, sensational, super, terrific, thank, vivid, wise, wonderful, zest
Negative	abominable, anger, anxious, bad, catastrophe, cheap, complaint, condescending, deceit, defective, disappointment, embarrass, fake, fear, filthy, fool, guilt, hate, idiot, inflict, lazy, miserable, mourn, nervous, objection, pest, plot, reject, scream, silly, terrible, unfriendly, vile, wicked

Figure 22.3 Some words with consistent sentiment across the General Inquirer (Stone et al., 1966), the MPQA Subjectivity lexicon (Wilson et al., 2005), and the polarity lexicon of Hu and Liu (2004).

Slightly more general than these sentiment lexicons are lexicons that assign each word a value on all three affective dimensions. The NRC Valence, Arousal, and Dominance (VAD) lexicon (Mohammad, 2018a) assigns valence, arousal, and dominance scores to 20,000 words. Some examples are shown in Fig. 22.4.

Valence		Arousal		Dominance	
vacation	.840	enraged	.962	powerful	.991
delightful	.918	party	.840	authority	.935
whistle	.653	organized	.337	saxophone	.482
consolation	.408	effortless	.120	discouraged	.0090
torture	.115	napping	.046	weak	.045

Figure 22.4 Values of sample words on the emotional dimensions of Mohammad (2018a).

EmoLex

The NRC Word-Emotion Association Lexicon, also called **EmoLex** (Mohammad and Turney, 2013), uses the Plutchik (1980) 8 basic emotions defined above. The lexicon includes around 14,000 words including words from prior lexicons as well as frequent nouns, verbs, adverbs and adjectives. Values from the lexicon for some sample words:

Word	anger	anticipation	disgust	fear	joy	sadness	surprise	trust	positive	negative
reward	0	1	0	0	1	0	1	1	1	0
worry	0	1	0	1	0	1	0	0	0	1
tenderness	0	0	0	0	1	0	0	0	1	0
sweetheart	0	1	0	0	1	1	0	1	1	0
suddenly	0	0	0	0	0	0	1	0	0	0
thirst	0	1	0	0	0	1	1	0	0	0
garbage	0	0	1	0	0	0	0	0	0	1

For a smaller set of 5,814 words, the NRC Emotion/Affect Intensity Lexicon (Mohammad, 2018b) contains real-valued scores of association for anger, fear, joy, and sadness; Fig. 22.5 shows examples.

Anger		Fear		Joy		Sadness	
outraged	0.964	horror	0.923	superb	0.864	sad	0.844
violence	0.742	anguish	0.703	cheered	0.773	guilt	0.750
coup	0.578	pestilence	0.625	rainbow	0.531	unkind	0.547
oust	0.484	stressed	0.531	gesture	0.387	difficulties	0.421
suspicious	0.484	failing	0.531	warms	0.391	beggar	0.422
nurture	0.059	confident	0.094	hardship	.031	sing	0.017

Figure 22.5 Sample emotional intensities for words for anger, fear, joy, and sadness from Mohammad (2018b).

LIWC

LIWC, **Linguistic Inquiry and Word Count**, is a widely used set of 73 lexicons containing over 2300 words (Pennebaker et al., 2007), designed to capture aspects of lexical meaning relevant for social psychological tasks. In addition to sentiment-related lexicons like ones for negative emotion (*bad, weird, hate, problem, tough*) and positive emotion (*love, nice, sweet*), LIWC includes lexicons for categories like anger, sadness, cognitive mechanisms, perception, tentative, and inhibition, shown in Fig. 22.6.

There are various other hand-built affective lexicons. The General Inquirer includes additional lexicons for dimensions like strong vs. weak, active vs. passive, overstated vs. understated, as well as lexicons for categories like pleasure, pain, virtue, vice, motivation, and cognitive orientation.

concrete
abstract

Another useful feature for various tasks is the distinction between **concrete** words like *banana* or *bathrobe* and **abstract** words like *belief* and *although*. The lexicon in Brysbaert et al. (2014) used crowdsourcing to assign a rating from 1 to 5 of the concreteness of 40,000 words, thus assigning *banana*, *bathrobe*, and *bagel* 5, *belief* 1.19, *although* 1.07, and in between words like *brisk* a 2.5.

22.3 Creating Affect Lexicons by Human Labeling

crowdsourcing

The earliest method used to build affect lexicons, and still in common use, is to have humans label each word. This is now most commonly done via **crowdsourcing**: breaking the task into small pieces and distributing them to a large number of anno-

Positive Emotion	Negative Emotion	Insight	Inhibition	Family	Negate
appreciat*	anger*	aware*	avoid*	brother*	aren't
comfort*	bore*	believe	careful*	cousin*	cannot
great	cry	decid*	hesitat*	daughter*	didn't
happy	despair*	feel	limit*	family	neither
interest	fail*	figur*	oppos*	father*	never
joy*	fear	know	prevent*	grandf*	no
perfect*	griev*	knew	reluctan*	grandm*	nobod*
please*	hate*	means	safe*	husband	none
safe*	panic*	notice*	stop	mom	nor
terrific	suffers	recogni*	stubborn*	mother	nothing
value	terrify	sense	wait	niece*	nowhere
wow*	violent*	think	wary	wife	without

Figure 22.6 Samples from 5 of the 73 lexical categories in LIWC (Pennebaker et al., 2007). The * means the previous letters are a word prefix and all words with that prefix are included in the category.

tators. Let's take a look at some of the methodological choices for two crowdsourced emotion lexicons.

The NRC Emotion Lexicon (EmoLex) (Mohammad and Turney, 2013), labeled emotions in two steps. To ensure that the annotators were judging the correct sense of the word, they first answered a multiple-choice synonym question that primed the correct sense of the word (without requiring the annotator to read a potentially confusing sense definition). These were created automatically using the headwords associated with the thesaurus category of the sense in question in the Macquarie dictionary and the headwords of 3 random distractor categories. An example:

```
Which word is closest in meaning (most related) to startle?
```

- `automobile`
- `shake`
- `honesty`
- `entertain`

For each word (e.g. *startle*), the annotator was then asked to rate how associated that word is with each of the 8 emotions (*joy*, *fear*, *anger*, etc.). The associations were rated on a scale of *not*, *weakly*, *moderately*, and *strongly* associated. Outlier ratings were removed, and then each term was assigned the class chosen by the majority of the annotators, with ties broken by choosing the stronger intensity, and then the 4 levels were mapped into a binary label for each word (no and weak mapped to 0, moderate and strong mapped to 1).

best-worst scaling

The NRC VAD Lexicon (Mohammad, 2018a) was built by selecting words and emoticons from prior lexicons and annotating them with crowd-sourcing using **best-worst scaling** (Louviere et al. 2015, Kiritchenko and Mohammad 2017). In best-worst scaling, annotators are given N items (usually 4) and are asked which item is the **best** (highest) and which is the **worst** (lowest) in terms of some property. The set of words used to describe the ends of the scales are taken from prior literature. For valence, for example, the raters were asked:

> Q1. Which of the four words below is associated with the MOST happiness / pleasure / positiveness / satisfaction / contentedness / hopefulness OR LEAST unhappiness / annoyance / negativeness / dissatisfaction /

melancholy / despair? (Four words listed as options.)
Q2. Which of the four words below is associated with the LEAST happiness / pleasure / positiveness / satisfaction / contentedness / hopefulness OR MOST unhappiness / annoyance / negativeness / dissatisfaction / melancholy / despair? (Four words listed as options.)

split-half reliability

The score for each word in the lexicon is the proportion of times the item was chosen as the best (highest V/A/D) minus the proportion of times the item was chosen as the worst (lowest V/A/D). The agreement between annotations are evaluated by **split-half reliability**: split the corpus in half and compute the correlations between the annotations in the two halves.

22.4 Semi-supervised Induction of Affect Lexicons

Another common way to learn sentiment lexicons is to start from a set of seed words that define two poles of a semantic axis (words like *good* or *bad*), and then find ways to label each word w by its similarity to the two seed sets. Here we summarize two families of seed-based semi-supervised lexicon induction algorithms, axis-based and graph-based.

22.4.1 Semantic Axis Methods

One of the most well-known lexicon induction methods, the Turney and Littman (2003) algorithm, is given seed words like *good* or *bad*, and then for each word w to be labeled, measures both how similar it is to *good* and how different it is from *bad*. Here we describe a slight extension of the algorithm due to An et al. (2018), which is based on computing a **semantic axis**.

In the first step, we choose seed words by hand. There are two methods for dealing with the fact that the affect of a word is different in different contexts: (1) start with a single large seed lexicon and rely on the induction algorithm to finetune it to the domain, or (2) choose different seed words for different genres. Hellrich et al. (2019) suggests that for modeling affect across different historical time periods, starting with a large modern affect dictionary is better than small seedsets tuned to be stable across time. As an example of the second approach, Hamilton et al. (2016a) define one set of seed words for general sentiment analysis, a different set for Twitter, and yet another set for sentiment in financial text:

Domain	Positive seeds	Negative seeds
General	good, lovely, excellent, fortunate, pleasant, delightful, perfect, loved, love, happy	bad, horrible, poor, unfortunate, unpleasant, disgusting, evil, hated, hate, unhappy
Twitter	love, loved, loves, awesome, nice, amazing, best, fantastic, correct, happy	hate, hated, hates, terrible, nasty, awful, worst, horrible, wrong, sad
Finance	successful, excellent, profit, beneficial, improving, improved, success, gains, positive	negligent, loss, volatile, wrong, losses, damages, bad, litigation, failure, down, negative

In the second step, we compute embeddings for each of the pole words. These embeddings can be off-the-shelf word2vec embeddings, or can be computed directly

on a specific corpus (for example using a financial corpus if a finance lexicon is the goal), or we can finetune off-the-shelf embeddings to a corpus. Fine-tuning is especially important if we have a very specific genre of text but don't have enough data to train good embeddings. In finetuning, we begin with off-the-shelf embeddings like word2vec, and continue training them on the small target corpus.

Once we have embeddings for each pole word, we create an embedding that represents each pole by taking the centroid of the embeddings of each of the seed words; recall that the centroid is the multidimensional version of the mean. Given a set of embeddings for the positive seed words $S^+ = \{E(w_1^+), E(w_2^+), ..., E(w_n^+)\}$, and embeddings for the negative seed words $S^- = \{E(w_1^-), E(w_2^-), ..., E(w_m^-)\}$, the pole centroids are:

$$\mathbf{V}^+ = \frac{1}{n}\sum_{1}^{n} E(w_i^+)$$
$$\mathbf{V}^- = \frac{1}{m}\sum_{1}^{m} E(w_i^-) \tag{22.1}$$

The semantic axis defined by the poles is computed just by subtracting the two vectors:

$$\mathbf{V}_{axis} = \mathbf{V}^+ - \mathbf{V}^- \tag{22.2}$$

$\mathbf{V}_{axis}$, the semantic axis, is a vector in the direction of positive sentiment. Finally, we compute (via cosine similarity) the angle between the vector in the direction of positive sentiment and the direction of w's embedding. A higher cosine means that w is more aligned with S^+ than S^-.

$$\begin{aligned} \text{score}(w) &= \cos\left(E(w), \mathbf{V}_{\text{axis}}\right) \\ &= \frac{E(w)\cdot \mathbf{V}_{\text{axis}}}{\|E(w)\|\|\mathbf{V}_{\text{axis}}\|} \end{aligned} \tag{22.3}$$

If a dictionary of words with sentiment scores is sufficient, we're done! Or if we need to group words into a positive and a negative lexicon, we can use a threshold or other method to give us discrete lexicons.

22.4.2 Label Propagation

An alternative family of methods defines lexicons by propagating sentiment labels on graphs, an idea suggested in early work by Hatzivassiloglou and McKeown (1997). We'll describe the simple SentProp (Sentiment Propagation) algorithm of Hamilton et al. (2016a), which has four steps:

1. **Define a graph**: Given word embeddings, build a weighted lexical graph by connecting each word with its k nearest neighbors (according to cosine similarity). The weights of the edge between words w_i and w_j are set as:

$$\mathbf{E}_{i,j} = \arccos\left(-\frac{\mathbf{w_i}^\top \mathbf{w_j}}{\|\mathbf{w_i}\|\|\mathbf{w_j}\|}\right). \tag{22.4}$$

2. **Define a seed set:** Choose positive and negative seed words.
3. **Propagate polarities from the seed set:** Now we perform a random walk on this graph, starting at the seed set. In a random walk, we start at a node and

then choose a node to move to with probability proportional to the edge probability. A word's polarity score for a seed set is proportional to the probability of a random walk from the seed set landing on that word (Fig. 22.7).

4. **Create word scores**: We walk from both positive and negative seed sets, resulting in positive ($\text{rawscore}^+(w_i)$) and negative ($\text{rawscore}^-(w_i)$) raw label scores. We then combine these values into a positive-polarity score as:

$$\text{score}^+(w_i) = \frac{\text{rawscore}^+(w_i)}{\text{rawscore}^+(w_i) + \text{rawscore}^-(w_i)} \tag{22.5}$$

It's often helpful to standardize the scores to have zero mean and unit variance within a corpus.

5. **Assign confidence to each score:** Because sentiment scores are influenced by the seed set, we'd like to know how much the score of a word would change if a different seed set is used. We can use bootstrap sampling to get confidence regions, by computing the propagation B times over random subsets of the positive and negative seed sets (for example using $B = 50$ and choosing 7 of the 10 seed words each time). The standard deviation of the bootstrap sampled polarity scores gives a confidence measure.

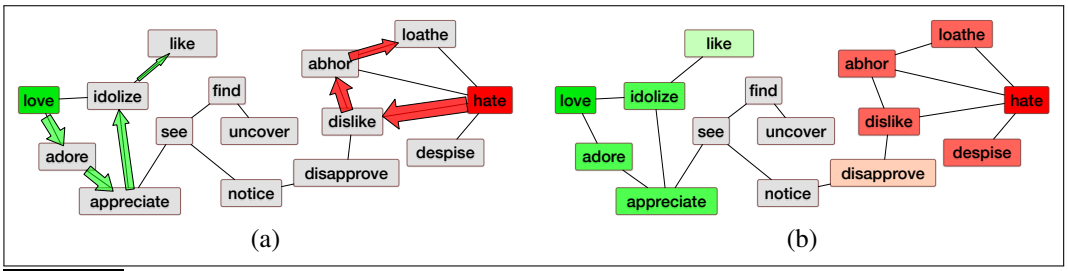


Figure 22.7 Intuition of the SENTPROP algorithm. (a) Run random walks from the seed words. (b) Assign polarity scores (shown here as colors green or red) based on the frequency of random walk visits.

22.4.3 Other Methods

The core of semisupervised algorithms is the metric for measuring similarity with the seed words. The Turney and Littman (2003) and Hamilton et al. (2016a) approaches above used embedding cosine as the distance metric: words were labeled as positive basically if their embeddings had high cosines with positive seeds and low cosines with negative seeds. Other methods have chosen other kinds of distance metrics besides embedding cosine.

For example the Hatzivassiloglou and McKeown (1997) algorithm uses syntactic cues; two adjectives are considered similar if they were frequently conjoined by *and* and rarely conjoined by *but*. This is based on the intuition that adjectives conjoined by the words *and* tend to have the same polarity; positive adjectives are generally coordinated with positive, negative with negative:

fair and legitimate, corrupt and brutal

but less often positive adjectives coordinated with negative:

*fair and brutal, *corrupt and legitimate

By contrast, adjectives conjoined by *but* are likely to be of opposite polarity:

fair but brutal

Another cue to opposite polarity comes from morphological negation (*un-*, *im-*, *-less*). Adjectives with the same root but differing in a morphological negative (*adequate/inadequate*, *thoughtful/thoughtless*) tend to be of opposite polarity.

Yet another method for finding words that have a similar polarity to seed words is to make use of a thesaurus like WordNet (Kim and Hovy 2004, Hu and Liu 2004). A word's synonyms presumably share its polarity while a word's antonyms probably have the opposite polarity. After a seed lexicon is built, each lexicon is updated as follows, possibly iterated.

Lex$^+$: Add synonyms of positive words (*well*) and antonyms (like *fine*) of negative words

Lex$^-$: Add synonyms of negative words (*awful*) and antonyms (like *evil*) of positive words

SentiWordNet

An extension of this algorithm assigns polarity to WordNet senses, called **SentiWordNet** (Baccianella et al., 2010). Fig. 22.8 shows some examples.

Synset	Pos	Neg	Obj
good#6 'agreeable or pleasing'	1	0	0
respectable#2 honorable#4 good#4 estimable#2 'deserving of esteem'	0.75	0	0.25
estimable#3 computable#1 'may be computed or estimated'	0	0	1
sting#1 burn#4 bite#2 'cause a sharp or stinging pain'	0	0.875	.125
acute#6 'of critical importance and consequence'	0.625	0.125	.250
acute#4 'of an angle; less than 90 degrees'	0	0	1
acute#1 'having or experiencing a rapid onset and short but severe course'	0	0.5	0.5

Figure 22.8 Examples from SentiWordNet 3.0 (Baccianella et al., 2010). Note the differences between senses of homonymous words: *estimable#3* is purely objective, while *estimable#2* is positive; *acute* can be positive (*acute#6*), negative (*acute#1*), or neutral (*acute #4*).

In this algorithm, polarity is assigned to entire synsets rather than words. A positive lexicon is built from all the synsets associated with 7 positive words, and a negative lexicon from synsets associated with 7 negative words. A classifier is then trained from this data to take a WordNet gloss and decide if the sense being defined is positive, negative or neutral. A further step (involving a random-walk algorithm) assigns a score to each WordNet synset for its degree of positivity, negativity, and neutrality.

In summary, semisupervised algorithms use a human-defined set of seed words for the two poles of a dimension, and use similarity metrics like embedding cosine, coordination, morphology, or thesaurus structure to score words by how similar they are to the positive seeds and how dissimilar to the negative seeds.

22.5 Supervised Learning of Word Sentiment

Semi-supervised methods require only minimal human supervision (in the form of seed sets). But sometimes a supervision signal exists in the world and can be made use of. One such signal is the scores associated with *online reviews*.

The web contains an enormous number of online reviews for restaurants, movies, books, or other products, each of which have the text of the review along with an

	Movie review excerpts (IMDb)
10	A great movie. This film is just a wonderful experience. It's surreal, zany, witty and slapstick all at the same time. And terrific performances too.
1	This was probably the worst movie I have ever seen. The story went nowhere even though they could have done some interesting stuff with it.
	Restaurant review excerpts (Yelp)
5	The service was impeccable. The food was cooked and seasoned perfectly... The watermelon was perfectly square ... The grilled octopus was ... mouthwatering...
2	...it took a while to get our waters, we got our entree before our starter, and we never received silverware or napkins until we requested them...
	Book review excerpts (GoodReads)
1	I am going to try and stop being deceived by eye-catching titles. I so wanted to like this book and was so disappointed by it.
5	This book is hilarious. I would recommend it to anyone looking for a satirical read with a romantic twist and a narrator that keeps butting in
	Product review excerpts (Amazon)
5	The lid on this blender though is probably what I like the best about it... enables you to pour into something without even taking the lid off! ... the perfect pitcher! ... works fantastic.
1	I hate this blender... It is nearly impossible to get frozen fruit and ice to turn into a smoothie... You have to add a TON of liquid. I also wish it had a spout ...

Figure 22.9 Excerpts from some reviews from various review websites, all on a scale of 1 to 5 stars except IMDb, which is on a scale of 1 to 10 stars.

associated review score: a value that may range from 1 star to 5 stars, or scoring 1 to 10. Fig. 22.9 shows samples extracted from restaurant, book, and movie reviews.

We can use this review score as supervision: positive words are more likely to appear in 5-star reviews; negative words in 1-star reviews. And instead of just a binary polarity, this kind of supervision allows us to assign a word a more complex representation of its polarity: its distribution over stars (or other scores).

Thus in a ten-star system we could represent the sentiment of each word as a 10-tuple, each number a score representing the word's association with that polarity level. This association can be a raw count, or a likelihood $P(w|c)$, or some other function of the count, for each class c from 1 to 10.

For example, we could compute the IMDb likelihood of a word like *disappoint(ed/ing)* occurring in a 1 star review by dividing the number of times *disappoint(ed/ing)* occurs in 1-star reviews in the IMDb dataset (8,557) by the total number of words occurring in 1-star reviews (25,395,214), so the IMDb estimate of $P(\textit{disappointing}|1)$ is .0003.

A slight modification of this weighting, the normalized likelihood, can be used as an illuminating visualization (Potts, 2011)[1]

$$\begin{aligned} P(w|c) &= \frac{count(w,c)}{\sum_{w \in C} count(w,c)} \\ PottsScore(w) &= \frac{P(w|c)}{\sum_c P(w|c)} \end{aligned} \tag{22.6}$$

Dividing the IMDb estimate $P(\textit{disappointing}|1)$ of .0003 by the sum of the likelihood $P(w|c)$ over all categories gives a Potts score of 0.10. The word *disappointing* thus is associated with the vector [.10, .12, .14, .14, .13, .11, .08, .06, .06, .05]. The

[1] Each element of the Potts score of a word w and category c can be shown to be a variant of the pointwise mutual information $\text{pmi}(w,c)$ without the log term; see Exercise 22.1.

Potts diagram

Potts diagram (Potts, 2011) is a visualization of these word scores, representing the prior sentiment of a word as a distribution over the rating categories.

Fig. 22.10 shows the Potts diagrams for 3 positive and 3 negative scalar adjectives. Note that the curve for strongly positive scalars have the shape of the letter J, while strongly negative scalars look like a reverse J. By contrast, weakly positive and negative scalars have a hump-shape, with the maximum either below the mean (weakly negative words like *disappointing*) or above the mean (weakly positive words like *good*). These shapes offer an illuminating typology of affective meaning.

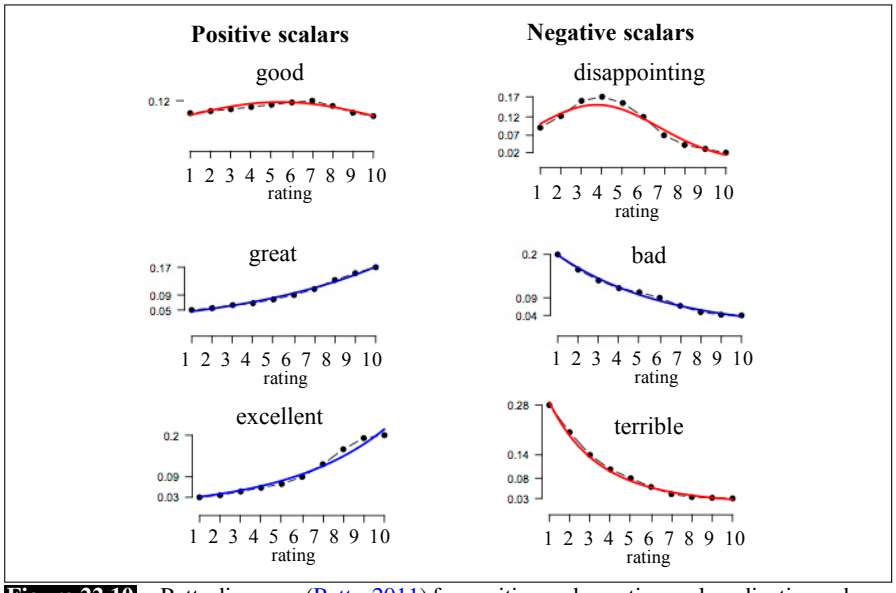


Figure 22.10 Potts diagrams (Potts, 2011) for positive and negative scalar adjectives, showing the J-shape and reverse J-shape for strongly positive and negative adjectives, and the hump-shape for more weakly polarized adjectives.

Fig. 22.11 shows the Potts diagrams for emphasizing and attenuating adverbs. Note that emphatics tend to have a J-shape (most likely to occur in the most positive reviews) or a U-shape (most likely to occur in the strongly positive and negative). Attenuators all have the hump-shape, emphasizing the middle of the scale and downplaying both extremes. The diagrams can be used both as a typology of lexical sentiment, and also play a role in modeling sentiment compositionality.

In addition to functions like posterior $P(c|w)$, likelihood $P(w|c)$, or normalized likelihood (Eq. 22.6) many other functions of the count of a word occurring with a sentiment label have been used. We'll introduce some of these on page 516, including ideas like normalizing the counts per writer in Eq. 22.14.

22.5.1 Log Odds Ratio Informative Dirichlet Prior

One thing we often want to do with word polarity is to distinguish between words that are more likely to be used in one category of texts than in another. We may, for example, want to know the words most associated with 1 star reviews versus those associated with 5 star reviews. These differences may not be just related to sentiment. We might want to find words used more often by Democratic than Republican members of Congress, or words used more often in menus of expensive restaurants

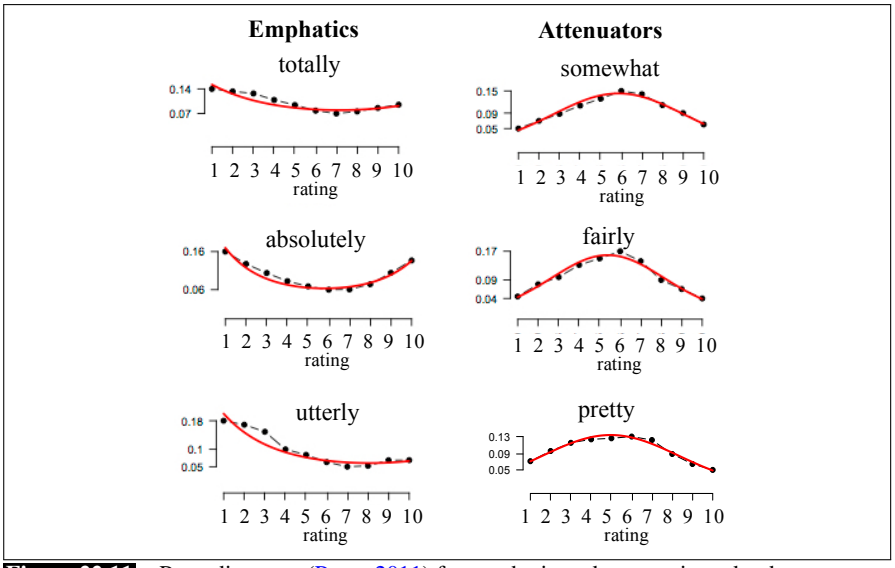


Figure 22.11 Potts diagrams (Potts, 2011) for emphatic and attenuating adverbs.

than cheap restaurants.

Given two classes of documents, to find words more associated with one category than another, we could measure the difference in frequencies (is a word w more frequent in class A or class B?). Or instead of the difference in frequencies we could compute the ratio of frequencies, or compute the log odds ratio (the log of the ratio between the odds of the two words). We could then sort words by whichever association measure we pick, ranging from words overrepresented in category A to words overrepresented in category B.

The problem with simple log-likelihood or log odds methods is that they overemphasize differences in very rare words, and often also in very frequent words. Very rare words will seem to occur very differently in the two corpora since with tiny counts there may be statistical fluctuations, or even zero occurrences in one corpus compared to non-zero occurrences in the other. Very frequent words will also seem different since all counts are large.

In this section we walk through the details of one solution to this problem: the "log odds ratio informative Dirichlet prior" method of Monroe et al. (2008) that is a particularly useful method for finding words that are statistically overrepresented in one particular category of texts compared to another. It's based on the idea of using another large corpus to get a prior estimate of what we expect the frequency of each word to be.

log likelihood ratio

Let's start with the goal: assume we want to know whether the word *horrible* occurs more in corpus i or corpus j. We could compute the **log likelihood ratio**, using $f^i(w)$ to mean the frequency of word w in corpus i, and n^i to mean the total number of words in corpus i:

$$\begin{aligned}\text{llr}(horrible) &= \log\frac{P^i(horrible)}{P^j(horrible)}\\ &= \log P^i(horrible) - \log P^j(horrible)\\ &= \log\frac{\text{f}^i(horrible)}{n^i} - \log\frac{\text{f}^j(horrible)}{n^j}\end{aligned} \tag{22.7}$$

log odds ratio

Instead, let's compute the **log odds ratio**: does *horrible* have higher odds in i or in

j:

$$\begin{aligned}\text{lor}(horrible) &= \log\left(\frac{P^i(horrible)}{1-P^i(horrible)}\right) - \log\left(\frac{P^j(horrible)}{1-P^j(horrible)}\right)\\ &= \log\left(\frac{\frac{\text{f}^i(horrible)}{n^i}}{1-\frac{\text{f}^i(horrible)}{n^i}}\right) - \log\left(\frac{\frac{\text{f}^j(horrible)}{n^j}}{1-\frac{\text{f}^j(horrible)}{n^j}}\right)\\ &= \log\left(\frac{\text{f}^i(horrible)}{n^i-\text{f}^i(horrible)}\right) - \log\left(\frac{\text{f}^j(horrible)}{n^j-\text{f}^j(horrible)}\right) \qquad (22.8)\end{aligned}$$

The Dirichlet intuition is to use a large background corpus to get a prior estimate of what we expect the frequency of each word w to be. We'll do this very simply by adding the counts from that corpus to the numerator and denominator, so that we're essentially shrinking the counts toward that prior. It's like asking how large are the differences between i and j given what we would expect given their frequencies in a well-estimated large background corpus.

The method estimates the difference between the frequency of word w in two corpora i and j via the prior-modified log odds ratio for w, $\delta_w^{(i-j)}$, which is estimated as:

$$\delta_w^{(i-j)} = \log\left(\frac{f_w^i+\alpha_w}{n^i+\alpha_0-(f_w^i+\alpha_w)}\right) - \log\left(\frac{f_w^j+\alpha_w}{n^j+\alpha_0-(f_w^j+\alpha_w)}\right) \qquad (22.9)$$

(where n^i is the size of corpus i, n^j is the size of corpus j, f_w^i is the count of word w in corpus i, f_w^j is the count of word w in corpus j, α_0 is the scaled size of the background corpus, and α_w is the scaled count of word w in the background corpus.)

In addition, Monroe et al. (2008) make use of an estimate for the variance of the log–odds–ratio:

$$\sigma^2\left(\hat{\delta}_w^{(i-j)}\right) \approx \frac{1}{f_w^i+\alpha_w} + \frac{1}{f_w^j+\alpha_w} \qquad (22.10)$$

The final statistic for a word is then the z–score of its log–odds–ratio:

$$\frac{\hat{\delta}_w^{(i-j)}}{\sqrt{\sigma^2\left(\hat{\delta}_w^{(i-j)}\right)}} \qquad (22.11)$$

The Monroe et al. (2008) method thus modifies the commonly used log odds ratio in two ways: it uses the z-scores of the log odds ratio, which controls for the amount of variance in a word's frequency, and it uses counts from a background corpus to provide a prior count for words.

Fig. 22.12 shows the method applied to a dataset of restaurant reviews from Yelp, comparing the words used in 1-star reviews to the words used in 5-star reviews (Jurafsky et al., 2014). The largest difference is in obvious sentiment words, with the 1-star reviews using negative sentiment words like *worse, bad, awful* and the 5-star reviews using positive sentiment words like *great, best, amazing*. But there are other illuminating differences. 1-star reviews use logical negation (*no, not*), while 5-star reviews use emphatics and emphasize universality (*very, highly, every, always*). 1-star reviews use first person plurals (*we, us, our*) while 5 star reviews use the second person. 1-star reviews talk about people (*manager, waiter, customer*) while 5-star reviews talk about dessert and properties of expensive restaurants like courses and atmosphere. See Jurafsky et al. (2014) for more details.

Class	Words in 1-star reviews	Class	Words in 5-star reviews
Negative	*worst, rude, terrible, horrible, bad, awful, disgusting, bland, tasteless, gross, mediocre, overpriced, worse, poor*	**Positive**	*great, best, love(d), delicious, amazing, favorite, perfect, excellent, awesome, friendly, fantastic, fresh, wonderful, incredible, sweet, yum(my)*
Negation	*no, not*	**Emphatics/ universals**	*very, highly, perfectly, definitely, absolutely, everything, every, always*
1Pl pro	*we, us, our*	**2 pro**	*you*
3 pro	*she, he, her, him*	**Articles**	*a, the*
Past verb	*was, were, asked, told, said, did, charged, waited, left, took*	**Advice**	*try, recommend*
Sequencers	*after, then*	**Conjunct**	*also, as, well, with, and*
Nouns	*manager, waitress, waiter, customer, customers, attitude, waste, poisoning, money, bill, minutes*	**Nouns**	*atmosphere, dessert, chocolate, wine, course, menu*
Irrealis modals	*would, should*	**Auxiliaries**	*is/'s, can, 've, are*
Comp	*to, that*	**Prep, other**	*in, of, die, city, mouth*

Figure 22.12 The top 50 words associated with one–star and five-star restaurant reviews in a Yelp dataset of 900,000 reviews, using the Monroe et al. (2008) method (Jurafsky et al., 2014).

22.6 Using Lexicons for Sentiment Recognition

In Appendix K we introduced the naive Bayes algorithm for sentiment analysis. The lexicons we have focused on throughout the chapter so far can be used in a number of ways to improve sentiment detection.

In the simplest case, lexicons can be used when we don't have sufficient training data to build a supervised sentiment analyzer; it can often be expensive to have a human assign sentiment to each document to train the supervised classifier.

In such situations, lexicons can be used in a rule-based algorithm for classification. The simplest version is just to use the ratio of positive to negative words: if a document has more positive than negative words (using the lexicon to decide the polarity of each word in the document), it is classified as positive. Often a threshold λ is used, in which a document is classified as positive only if the ratio is greater than λ. If the sentiment lexicon includes positive and negative weights for each word, θ_w^+ and θ_w^-, these can be used as well. Here's a simple such sentiment algorithm:

$$
\begin{aligned}
f^+ &= \sum_{w \text{ s.t. } w \in positivelexicon} \theta_w^+ count(w) \\
f^- &= \sum_{w \text{ s.t. } w \in negativelexicon} \theta_w^- count(w) \\
sentiment &= \begin{cases} + & \text{if } \frac{f^+}{f^-} > \lambda \\ - & \text{if } \frac{f^-}{f^+} > \lambda \\ 0 & \text{otherwise.} \end{cases}
\end{aligned} \tag{22.12}
$$

If supervised training data is available, these counts computed from sentiment lexicons, sometimes weighted or normalized in various ways, can also be used as features in a classifier along with other lexical or non-lexical features. We return to such algorithms in Section 22.7.

22.7 Using Lexicons for Affect Recognition

Detection of emotion (and the other kinds of affective meaning described by Scherer (2000)) can be done by generalizing the algorithms described above for detecting sentiment.

The most common algorithms involve supervised classification: a training set is labeled for the affective meaning to be detected, and a classifier is built using features extracted from the training set. As with sentiment analysis, if the training set is large enough, and the test set is sufficiently similar to the training set, simply using all the words or all the bigrams as features in a powerful classifier like logistic regression or SVM is an excellent algorithm whose performance is hard to beat. Thus we can treat affective meaning classification of a text sample as simple document classification.

Some modifications are nonetheless often necessary for very large datasets. For example, the Schwartz et al. (2013) study of personality, gender, and age using 700 million words of Facebook posts used only a subset of the n-grams of lengths 1-3. Only words and phrases used by at least 1% of the subjects were included as features, and 2-grams and 3-grams were only kept if they had sufficiently high PMI (PMI greater than $2 * \mathit{length}$, where *length* is the number of words):

$$\text{pmi}(\mathit{phrase}) = \log \frac{p(\mathit{phrase})}{\prod_{w \in \mathit{phrase}} p(w)} \tag{22.13}$$

Various weights can be used for the features, including the raw count in the training set, or some normalized probability or log probability. Schwartz et al. (2013), for example, turn feature counts into phrase likelihoods by normalizing them by each subject's total word use.

$$p(\mathit{phrase}|\mathit{subject}) = \frac{\text{freq}(\mathit{phrase}, \mathit{subject})}{\sum_{\mathit{phrase}' \in \text{vocab}(\mathit{subject})} \text{freq}(\mathit{phrase}', \mathit{subject})} \tag{22.14}$$

If the training data is sparser, or not as similar to the test set, any of the lexicons we've discussed can play a helpful role, either alone or in combination with all the words and n-grams.

Many possible values can be used for lexicon features. The simplest is just an indicator function, in which the value of a feature f_L takes the value 1 if a particular text has any word from the relevant lexicon L. Using the notation of Appendix K, in which a feature value is defined for a particular output class c and document x.

$$f_L(c,x) = \begin{cases} 1 & \text{if } \exists w : w \in L \ \& \ w \in x \ \& \ \mathit{class} = c \\ 0 & \text{otherwise} \end{cases}$$

Alternatively the value of a feature f_L for a particular lexicon L can be the total number of word *tokens* in the document that occur in L:

$$f_L = \sum_{w \in L} \mathit{count}(w)$$

For lexica in which each word is associated with a score or weight, the count can be multiplied by a weight θ_w^L:

$$f_L = \sum_{w \in L} \theta_w^L \mathit{count}(w)$$

Counts can alternatively be logged or normalized per writer as in Eq. 22.14.

However they are defined, these lexicon features are then used in a supervised classifier to predict the desired affective category for the text or document. Once a classifier is trained, we can examine which lexicon features are associated with which classes. For a classifier like logistic regression the feature weight gives an indication of how associated the feature is with the class.

22.8 Lexicon-based methods for Entity-Centric Affect

What if we want to get an affect score not for an entire document, but for a particular entity in the text? The entity-centric method of Field and Tsvetkov (2019) combines affect lexicons with contextual embeddings to assign an affect score to an entity in text. In the context of affect about people, they relabel the Valence/Arousal/Dominance dimension as Sentiment/Agency/Power. The algorithm first trains classifiers to map embeddings to scores:

1. For each word w in the training corpus:
   (a) Use off-the-shelf pretrained encoders (like BERT) to extract a contextual embedding $\mathbf{e}$ for each instance of the word. No additional finetuning is done.
   (b) Average over the $\mathbf{e}$ embeddings of each instance of w to obtain a single embedding vector for one training point w.
   (c) Use the NRC VAD Lexicon to get S, A, and P scores for w.
2. Train (three) regression models on all words w to predict V, A, D scores from a word's average embedding.

Now given an entity mention m in a text, we assign affect scores as follows:

1. Use the same pretrained LM to get contextual embeddings for m in context.
2. Feed this embedding through the 3 regression models to get S, A, P scores for the entity.

This results in a (S,A,P) tuple for a given entity mention; To get scores for the representation of an entity in a complete document, we can run coreference resolution and average the (S,A,P) scores for all the mentions. Fig. 22.13 shows the scores from their algorithm for characters from the movie *The Dark Knight* when run on Wikipedia plot summary texts with gold coreference.

22.9 Connotation Frames

connotation frame

The lexicons we've described so far define a word as a point in affective space. A **connotation frame**, by contrast, is a lexicon that incorporates a richer kind of grammatical structure, by combining affective lexicons with the frame semantic lexicons of Chapter 21. The basic insight of connotation frame lexicons is that a predicate like a verb expresses connotations about the verb's arguments (Rashkin et al. 2016, Rashkin et al. 2017).

Consider sentences like:

(22.15) Country A violated the sovereignty of Country B

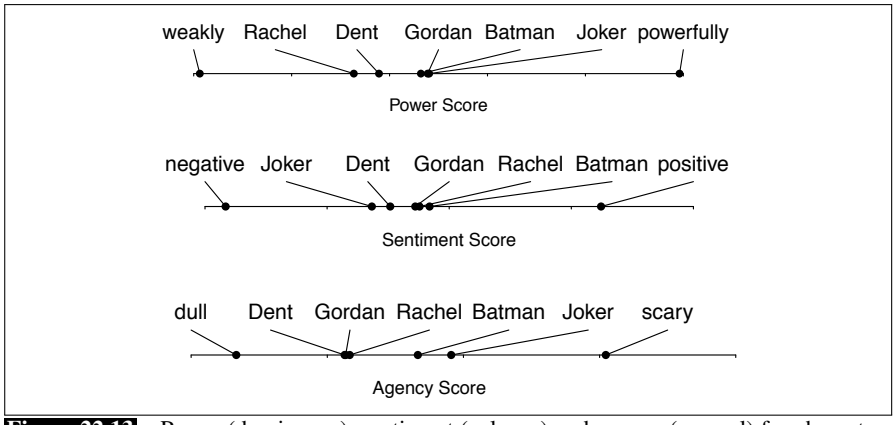


Figure 22.13 Power (dominance), sentiment (valence) and agency (arousal) for characters in the movie *The Dark Knight* computed from embeddings trained on the NRC VAD Lexicon. Note the protagonist (Batman) and the antagonist (the Joker) have high power and agency scores but differ in sentiment, while the love interest Rachel has low power and agency but high sentiment.

(22.16) the teenager ... survived the Boston Marathon bombing"

By using the verb *violate* in (22.15), the author is expressing their sympathies with Country B, portraying Country B as a victim, and expressing antagonism toward the agent Country A. By contrast, in using the verb *survive*, the author of (22.16) is expressing that the bombing is a negative experience, and the subject of the sentence, the teenager, is a sympathetic character. These aspects of connotation are inherent in the meaning of the verbs *violate* and *survive*, as shown in Fig. 22.14.

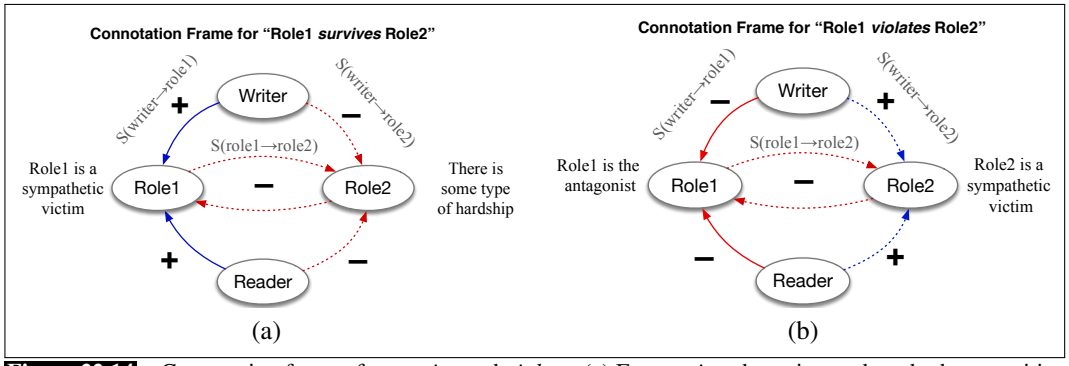


Figure 22.14 Connotation frames for *survive* and *violate*. (a) For *survive*, the writer and reader have positive sentiment toward Role1, the subject, and negative sentiment toward Role2, the direct object. (b) For *violate*, the writer and reader have positive sentiment instead toward Role2, the direct object.

The connotation frame lexicons of Rashkin et al. (2016) and Rashkin et al. (2017) also express other connotative aspects of the predicate toward each argument, including the *effect* (something bad happened to x) *value*: (x is valuable), and *mental state*: (x is distressed by the event). Connotation frames can also mark the *power differential* between the arguments (using the verb *implore* means that the theme argument has greater power than the agent), and the *agency* of each argument (*waited* is low agency). Fig. 22.15 shows a visualization from Sap et al. (2017).

Connotation frames can be built by hand (Sap et al., 2017), or they can be learned by supervised learning (Rashkin et al., 2016), for example using hand-labeled train-

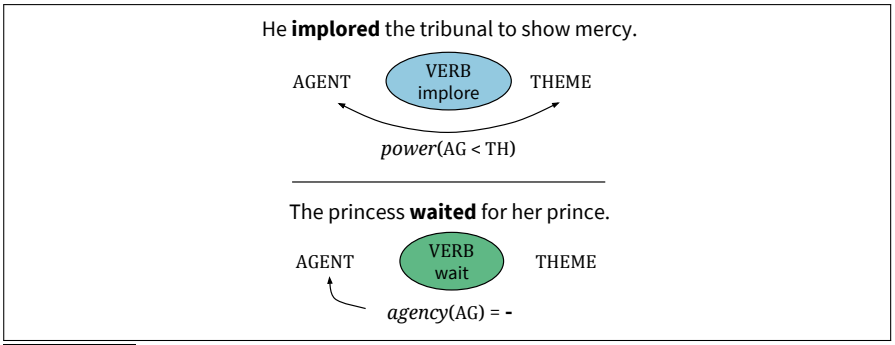


Figure 22.15 The connotation frames of Sap et al. (2017), showing that the verb *implore* implies the agent has lower power than the theme (in contrast, say, with a verb like *demanded*), and showing the low level of agency of the subject of *waited*. Figure from Sap et al. (2017).

ing data to supervise classifiers for each of the individual relations, e.g., whether S(writer → Role1) is + or -, and then improving accuracy via global constraints across all relations.

22.10 Summary

- Many kinds of affective states can be distinguished, including *emotions*, *moods*, *attitudes* (which include *sentiment*), *interpersonal stance*, and *personality*.
- **Emotion** can be represented by fixed atomic units often called **basic emotions**, or as points in space defined by dimensions like **valence** and **arousal**.
- Words have **connotational** aspects related to these affective states, and this connotational aspect of word meaning can be represented in lexicons.
- Affective lexicons can be built by hand, using **crowd sourcing** to label the affective content of each word.
- Lexicons can be built with **semi-supervised**, bootstrapping from seed words using similarity metrics like embedding cosine.
- Lexicons can be learned in a **fully supervised** manner, when a convenient training signal can be found in the world, such as ratings assigned by users on a review site.
- Words can be assigned weights in a lexicon by using various functions of word counts in training texts, and ratio metrics like **log odds ratio informative Dirichlet prior**.
- Affect can be detected, just like sentiment, by using standard supervised **text classification** techniques, using all the words or bigrams in a text as features. Additional features can be drawn from counts of words in lexicons.
- Lexicons can also be used to detect affect in a **rule-based classifier** by picking the simple majority sentiment based on counts of words in each lexicon.
- **Connotation frames** express richer relations of affective meaning that a predicate encodes about its arguments.

Historical Notes

The idea of formally representing the subjective meaning of words began with Osgood et al. (1957), the same pioneering study that first proposed the vector space model of meaning described in Chapter 5. Osgood et al. (1957) had participants rate words on various scales, and ran factor analysis on the ratings. The most significant factor they uncovered was the evaluative dimension, which distinguished between pairs like *good/bad*, *valuable/worthless*, *pleasant/unpleasant*. This work influenced the development of early dictionaries of sentiment and affective meaning in the field of **content analysis** (Stone et al., 1966).

subjectivity

Wiebe (1994) began an influential line of work on detecting **subjectivity** in text, beginning with the task of identifying subjective sentences and the subjective characters who are described in the text as holding private states, beliefs or attitudes. Learned sentiment lexicons such as the polarity lexicons of Hatzivassiloglou and McKeown (1997) were shown to be a useful feature in subjectivity detection (Hatzivassiloglou and Wiebe 2000, Wiebe 2000).

The term **sentiment** seems to have been introduced in 2001 by Das and Chen (2001), to describe the task of measuring market sentiment by looking at the words in stock trading message boards. In the same paper Das and Chen (2001) also proposed the use of a sentiment lexicon. The list of words in the lexicon was created by hand, but each word was assigned weights according to how much it discriminated a particular class (say buy versus sell) by maximizing across-class variation and minimizing within-class variation. The term *sentiment*, and the use of lexicons, caught on quite quickly (e.g., inter alia, Turney 2002). Pang et al. (2002) first showed the power of using all the words without a sentiment lexicon; see also Wang and Manning (2012).

Most of the semi-supervised methods we describe for extending sentiment dictionaries drew on the early idea that synonyms and antonyms tend to co-occur in the same sentence (Miller and Charles 1991, Justeson and Katz 1991, Riloff and Shepherd 1997). Other semi-supervised methods for learning cues to affective meaning rely on information extraction techniques, like the AutoSlog pattern extractors (Riloff and Wiebe, 2003). Graph based algorithms for sentiment were first suggested by Hatzivassiloglou and McKeown (1997), and graph propagation became a standard method (Zhu and Ghahramani 2002, Zhu et al. 2003, Zhou et al. 2004a, Velikovich et al. 2010). Crowdsourcing can also be used to improve precision by filtering the result of semi-supervised lexicon learning (Riloff and Shepherd 1997, Fast et al. 2016).

Much recent work focuses on ways to learn embeddings that directly encode sentiment or other properties, such as the DENSIFIER algorithm of Rothe et al. (2016) that learns to transform the embedding space to focus on sentiment (or other) information.

Exercises

22.1 Show that the relationship between a word w and a category c in the Potts Score in Eq. 22.6 is a variant of the pointwise mutual information $\text{pmi}(w,c)$ without the log term.

CHAPTER

23 Coreference Resolution and Entity Linking

and even Stigand, the patriotic archbishop of Canterbury, found it advisable–"'
'Found WHAT?' said the Duck.
'Found IT,' the Mouse replied rather crossly: 'of course you know what "it"means.'
'I know what "it"means well enough, when I find a thing,' said the Duck: 'it's generally a frog or a worm. The question is, what did the archbishop find?'

Lewis Carroll, *Alice in Wonderland*

An important component of language processing is knowing *who* is being talked about in a text. Consider the following passage:

(23.1) Victoria Chen, CFO of Megabucks Banking, saw her pay jump to $2.3 million, as the 38-year-old became the company's president. It is widely known that she came to Megabucks from rival Lotsabucks.

Each of the underlined phrases in this passage is used by the writer to refer to a person named Victoria Chen. We call linguistic expressions like *her* or *Victoria Chen* **mentions** or **referring expressions**, and the discourse entity that is referred to (Victoria Chen) the **referent**. (To distinguish between referring expressions and their referents, we italicize the former.)[1] Two or more referring expressions that are used to refer to the same discourse entity are said to **corefer**; thus, *Victoria Chen* and *she* corefer in (23.1).

mention
referent
corefer

Coreference is an important component of natural language processing. A dialogue system that has just told the user *"There is a 2pm flight on United and a 4pm one on Cathay Pacific"* must know which flight the user means by *"I'll take the second one"*. A question answering system that uses Wikipedia to answer a question about Marie Curie must know who *she* was in the sentence *"She was born in Warsaw"*. And a machine translation system translating from a language like Spanish, in which pronouns can be dropped, must use coreference from the previous sentence to decide whether the Spanish sentence '*"Me encanta el conocimiento", dice.*' should be translated as '*"I love knowledge", he says*', or '*"I love knowledge", she says*'. Indeed, this example comes from an actual news article in *El País* about a female professor and was mistranslated as "he" in machine translation because of inaccurate coreference resolution (Schiebinger, 2013).

Natural language processing systems (and humans) interpret linguistic expressions with respect to a **discourse model** (Karttunen, 1969). A discourse model (Fig. 23.1) is a mental model that the understander builds incrementally when interpreting a text, containing representations of the entities referred to in the text, as well as properties of the entities and relations among them. When a referent is first mentioned in a discourse, we say that a representation for it is **evoked** into the model. Upon subsequent mention, this representation is **accessed** from the model.

discourse model
evoked
accessed

[1] As a convenient shorthand, we sometimes speak of a referring expression referring to a referent, e.g., saying that *she* refers to Victoria Chen. However, the reader should keep in mind that what we really mean is that the speaker is performing the act of referring to Victoria Chen by uttering *she*.

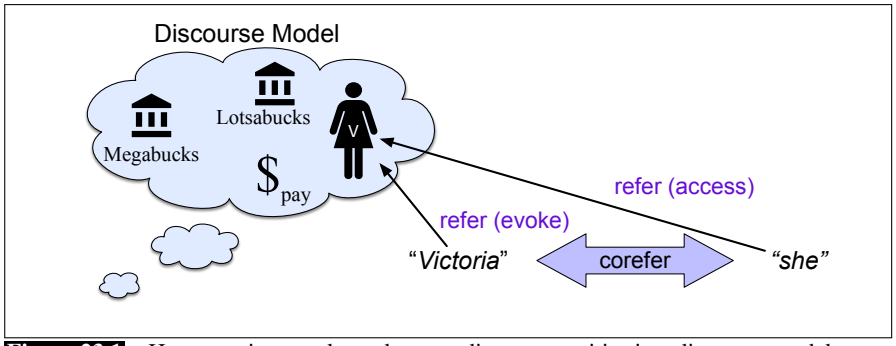


Figure 23.1 How mentions evoke and access discourse entities in a discourse model.

Reference in a text to an entity that has been previously introduced into the discourse is called **anaphora**, and the referring expression used is said to be an **anaphor**, or anaphoric.[2] In passage (23.1), the pronouns *she* and *her* and the definite NP *the 38-year-old* are therefore anaphoric. The anaphor corefers with a prior mention (in this case *Victoria Chen*) that is called the **antecedent**. Not every referring expression is an antecedent. An entity that has only a single mention in a text (like Lotsabucks in (23.1)) is called a **singleton**.

anaphora
anaphor
antecedent
singleton

In this chapter we focus on the task of **coreference resolution**. Coreference resolution is the task of determining whether two mentions *corefer*, by which we mean they refer to the same entity in the discourse model (the same *discourse entity*). The set of coreferring expressions is often called a **coreference chain** or a **cluster**. For example, in processing (23.1), a coreference resolution algorithm would need to find at least four coreference chains, corresponding to the four entities in the discourse model in Fig. 23.1.

coreference resolution
coreference chain
cluster

1. {*Victoria Chen*, *her*, *the 38-year-old*, *She*}
2. {*Megabucks Banking*, *the company*, *Megabucks*}
3. {*her pay*}
4. {*Lotsabucks*}

Note that mentions can be nested; for example the mention *her* is syntactically part of another mention, *her pay*, referring to a completely different discourse entity.

Coreference resolution thus comprises two tasks (although they are often performed jointly): (1) identifying the mentions, and (2) clustering them into coreference chains/discourse entities.

We said that two mentions corefered if they are associated with the same *discourse entity*. But often we'd like to go further, deciding which real world entity is associated with this discourse entity. For example, the mention *Washington* might refer to the US state, or the capital city, or the person George Washington; the interpretation of the sentence will of course be very different for each of these. The task of **entity linking** (Ji and Grishman, 2011) or *entity resolution* is the task of mapping a discourse entity to some real-world individual.[3] We usually operationalize entity

entity linking

[2] We will follow the common NLP usage of *anaphor* to mean any mention that has an antecedent, rather than the more narrow usage to mean only mentions (like pronouns) whose interpretation depends on the antecedent (under the narrower interpretation, repeated names are not anaphors).

[3] Computational linguistics/NLP thus differs in its use of the term *reference* from the field of formal semantics, which uses the words *reference* and *coreference* to describe the relation between a mention and a real-world entity. By contrast, we follow the functional linguistics tradition in which a mention *refers* to a *discourse entity* (Webber, 1978) and the relation between a discourse entity and the real world individual requires an additional step of *linking*.

linking or resolution by mapping to an *ontology*: a list of entities in the world, like a gazeteer (Appendix F). Perhaps the most common ontology used for this task is Wikipedia; each Wikipedia page acts as the unique id for a particular entity. Thus the entity linking task of **wikification** (Mihalcea and Csomai, 2007) is the task of deciding which Wikipedia page corresponding to an individual is being referred to by a mention. But entity linking can be done with any ontology; for example if we have an ontology of genes, we can link mentions of genes in text to the disambiguated gene name in the ontology.

In the next sections we introduce the task of coreference resolution in more detail, and survey a variety of architectures for resolution. We also introduce two architectures for the task of entity linking.

Before turning to algorithms, however, we mention some important tasks we will only touch on briefly at the end of this chapter. First are the famous Winograd Schema problems (so-called because they were first pointed out by Terry Winograd in his dissertation). These entity coreference resolution problems are designed to be too difficult to be solved by the resolution methods we describe in this chapter, and the kind of real-world knowledge they require has made them a kind of challenge task for natural language processing. For example, consider the task of determining the correct antecedent of the pronoun *they* in the following example:

(23.2) The city council denied the demonstrators a permit because
  a. they feared violence.
  b. they advocated violence.

Determining the correct antecedent for the pronoun *they* requires understanding that the second clause is intended as an explanation of the first clause, and also that city councils are perhaps more likely than demonstrators to fear violence and that demonstrators might be more likely to advocate violence. Solving Winograd Schema problems requires finding way to represent or discover the necessary real world knowledge.

event coreference

A problem we won't discuss in this chapter is the related task of **event coreference**, deciding whether two event mentions (such as the *buy* and the *acquisition* in these two sentences from the ECB+ corpus) refer to the same event:

(23.3) AMD agreed to [**buy**] Markham, Ontario-based ATI for around $5.4 billion in cash and stock, the companies announced Monday.

(23.4) The [**acquisition**] would turn AMD into one of the world's largest providers of graphics chips.

Event mentions are much harder to detect than entity mentions, since they can be verbal as well as nominal. Once detected, the same mention-pair and mention-ranking models used for entities are often applied to events.

discourse deixis

An even more complex kind of coreference is **discourse deixis** (Webber, 1988), in which an anaphor refers back to a discourse segment, which can be quite hard to delimit or categorize, like the examples in (23.5) adapted from Webber (1991):

(23.5) According to Soleil, Beau just opened a restaurant
  a. But *that* turned out to be a lie.
  b. But *that* was false.
  c. *That* struck me as a funny way to describe the situation.

The referent of *that* is a speech act (see Chapter 25) in (23.5a), a proposition in (23.5b), and a manner of description in (23.5c). We don't give algorithms in this chapter for these difficult types of **non-nominal antecedents**, but see Kolhatkar et al. (2018) for a survey.

23.1 Coreference Phenomena: Linguistic Background

We now offer some linguistic background on reference phenomena. We introduce the four types of referring expressions (definite and indefinite NPs, pronouns, and names), describe how these are used to evoke and access entities in the discourse model, and talk about linguistic features of the anaphor/antecedent relation (like number/gender agreement, or properties of verb semantics).

23.1.1 Types of Referring Expressions

Indefinite Noun Phrases: The most common form of indefinite reference in English is marked with the determiner *a* (or *an*), but it can also be marked by a quantifier such as *some* or even the determiner *this*. Indefinite reference generally introduces into the discourse context entities that are new to the hearer.

(23.6) a. Mrs. Martin was so very kind as to send Mrs. Goddard *a beautiful goose*.
b. He had gone round one day to bring her *some walnuts*.
c. I saw *this beautiful cauliflower* today.

Definite Noun Phrases: Definite reference, such as via NPs that use the English article *the*, refers to an entity that is identifiable to the hearer. An entity can be identifiable to the hearer because it has been mentioned previously in the text and thus is already represented in the discourse model:

(23.7) It concerns a white stallion which I have sold to an officer. But the pedigree of *the white stallion* was not fully established.

Alternatively, an entity can be identifiable because it is contained in the hearer's set of beliefs about the world, or the uniqueness of the object is implied by the description itself, in which case it evokes a representation of the referent into the discourse model, as in (23.9):

(23.8) I read about it in the *New York Times*.

(23.9) Have you seen the car keys?

These last uses are quite common; more than half of definite NPs in newswire texts are non-anaphoric, often because they are the first time an entity is mentioned (Poesio and Vieira 1998, Bean and Riloff 1999).

Pronouns: Another form of definite reference is pronominalization, used for entities that are extremely salient in the discourse, (as we discuss below):

(23.10) Emma smiled and chatted as cheerfully as *she* could,

cataphora

Pronouns can also participate in **cataphora**, in which they are mentioned before their referents are, as in (23.11).

(23.11) Even before *she* saw *it*, Dorothy had been thinking about the Emerald City every day.

Here, the pronouns *she* and *it* both occur *before* their referents are introduced.

bound

Pronouns also appear in quantified contexts in which they are considered to be **bound**, as in (23.12).

(23.12) Every dancer brought *her* left arm forward.

Under the relevant reading, *her* does not refer to some woman in context, but instead behaves like a variable bound to the quantified expression *every dancer*. We are not concerned with the bound interpretation of pronouns in this chapter.

In some languages, pronouns can appear as clitics attached to a word, like *lo* ('it') in this Spanish example from AnCora (Recasens and Martí, 2010):

(23.13) La intención es reconocer el gran prestigio que tiene la maratón y unir**lo** con esta gran carrera.
'The aim is to recognize the great prestige that the Marathon has and join|**it** with this great race."

Demonstrative Pronouns: Demonstrative pronouns *this* and *that* can appear either alone or as determiners, for instance, *this ingredient*, *that spice*:

(23.14) I just bought a copy of Thoreau's *Walden*. I had bought one five years ago. *That one* had been very tattered; *this one* was in much better condition.

Note that *this NP* is ambiguous; in colloquial spoken English, it can be indefinite, as in (23.6), or definite, as in (23.14).

Zero Anaphora: Instead of using a pronoun, in some languages (including Chinese, Japanese, and Italian) it is possible to have an anaphor that has no lexical realization at all, called a **zero anaphor** or zero pronoun, as in the following Italian and Japanese examples from Poesio et al. (2016):

zero anaphor

(23.15) EN [John]$_i$ went to visit some friends. On the way [he]$_i$ bought some wine.
IT [Giovanni]$_i$ andò a far visita a degli amici. Per via ϕ_i comprò del vino.
JA [John]$_i$-wa yujin-o houmon-sita. Tochu-de ϕ_i wain-o ka-tta.

or this Chinese example:

(23.16) [我] 前一会精神上太紧张。[0] 现在比较平静了
[I] was too nervous a while ago. ... [0] am now calmer.

Zero anaphors complicate the task of mention detection in these languages.

Names: Names (such as of people, locations, or organizations) can be used to refer to both new and old entities in the discourse:

(23.17) a. **Miss Woodhouse** certainly had not done him justice.
b. **International Business Machines** sought patent compensation from Amazon; **IBM** had previously sued other companies.

23.1.2 Information Status

The way referring expressions are used to evoke new referents into the discourse (introducing new information), or access old entities from the model (old information), is called their **information status** or **information structure**. Entities can be **discourse-new** or **discourse-old**, and indeed it is common to distinguish at least three kinds of entities informationally (Prince, 1981):

information status
discourse-new
discourse-old

new NPs:

brand new NPs: these introduce entities that are discourse-new and hearer-new like *a fruit* or *some walnuts*.

unused NPs: these introduce entities that are discourse-new but hearer-old (like *Hong Kong*, *Marie Curie*, or *the New York Times*.

old NPs: also called **evoked NPs**, these introduce entities that already in the discourse model, hence are both discourse-old and hearer-old, like *it* in "*I went to a new restaurant. It was...*".

inferrables: these introduce entities that are neither hearer-old nor discourse-old, but the hearer can infer their existence by reasoning based on other entities that are in the discourse. Consider the following examples:

(23.18) I went to a superb restaurant yesterday. *The chef* had just opened it.

(23.19) Mix flour, butter and water. Knead *the dough* until shiny.

bridging inference

Neither *the chef* nor *the dough* were in the discourse model based on the first sentence of either example, but the reader can make a **bridging inference** that these entities should be added to the discourse model and associated with the restaurant and the ingredients, based on world knowledge that restaurants have chefs and dough is the result of mixing flour and liquid (Haviland and Clark 1974, Webber and Baldwin 1992, Nissim et al. 2004, Hou et al. 2018).

given-new

The form of an NP gives strong clues to its information status. We often talk about an entity's position on the **given-new** dimension, the extent to which the referent is **given** (salient in the discourse, easier for the hearer to call to mind, predictable by the hearer), versus **new** (non-salient in the discourse, unpredictable) (Chafe 1976, Prince 1981, Gundel et al. 1993). A referent that is very **accessible** (Ariel, 2001) i.e., very salient in the hearer's mind or easy to call to mind, can be referred to with less linguistic material. For example pronouns are used only when the referent has a high degree of activation or **salience** in the discourse model.[4] By contrast, less salient entities, like a new referent being introduced to the discourse, will need to be introduced with a longer and more explicit referring expression to help the hearer recover the referent.

accessible

salience

Thus when an entity is first introduced into a discourse its mentions are likely to have full names, titles or roles, or appositive or restrictive relative clauses, as in the introduction of our protagonist in (23.1): *Victoria Chen, CFO of Megabucks Banking*. As an entity is discussed over a discourse, it becomes more salient to the hearer and its mentions on average typically becomes shorter and less informative, for example with a shortened name (for example *Ms. Chen*), a definite description (*the 38-year-old*), or a pronoun (*she* or *her*) (Hawkins 1978). However, this change in length is not monotonic, and is sensitive to discourse structure (Grosz 1977b, Reichman 1985, Fox 1993).

23.1.3 Complications: Non-Referring Expressions

Many noun phrases or other nominals are not referring expressions, although they may bear a confusing superficial resemblance. For example in some of the earliest computational work on reference resolution, Karttunen (1969) pointed out that the NP *a car* in the following example does not create a discourse referent:

(23.20) Janet doesn't have *a car*.

and cannot be referred back to by anaphoric *it* or *the car*:

(23.21) **It* is a Toyota.

(23.22) **The car* is red.

We summarize here four common types of structures that are not counted as mentions in coreference tasks and hence complicate the task of mention-detection:

[4] Pronouns also usually (but not always) refer to entities that were introduced no further than one or two sentences back in the ongoing discourse, whereas definite noun phrases can often refer further back.

Appositives: An appositional structure is a noun phrase that appears next to a head noun phrase, describing the head. In English they often appear in commas, like "a unit of UAL" appearing in apposition to the NP *United*, or *CFO of Megabucks Banking* in apposition to *Victoria Chen*.

(23.23) Victoria Chen, CFO of Megabucks Banking, saw ...

(23.24) United, a unit of UAL, matched the fares.

Appositional NPs are not referring expressions, instead functioning as a kind of supplementary parenthetical description of the head NP. Nonetheless, sometimes it is useful to link these phrases to an entity they describe, and so some datasets like OntoNotes mark appositional relationships.

Predicative and Prenominal NPs: Predicative or attributive NPs describe properties of the head noun. In *United is a unit of UAL*, the NP *a unit of UAL* describes a property of United, rather than referring to a distinct entity. Thus they are not marked as mentions in coreference tasks; in our example the NPs *$2.3 million* and *the company's president*, are attributive, describing properties of *her pay* and *the 38-year-old*; Example (23.27) shows a Chinese example in which the predicate NP (中国最大的城市*; China's biggest city*) is not a mention.

(23.25) her pay jumped to *$2.3 million*

(23.26) the 38-year-old became *the company's president*

(23.27) 上海是[中国最大的城市] [Shanghai is *China's biggest city*]

Expletives: Many uses of pronouns like *it* in English and corresponding pronouns in other languages are not referential. Such **expletive** or **pleonastic** cases include *it is raining*, in idioms like *hit it off*, or in particular syntactic situations like **clefts** (23.28a) or **extraposition** (23.28b):

expletive

clefts

(23.28) a. *It* was Emma Goldman who founded *Mother Earth*

b. *It* surprised me that there was a herring hanging on her wall.

Generics: Another kind of expression that does not refer back to an entity explicitly evoked in the text is *generic* reference. Consider (23.29).

(23.29) I love mangos. *They* are very tasty.

Here, *they* refers, not to a particular mango or set of mangos, but instead to the class of mangos in general. The pronoun *you* can also be used generically:

(23.30) In July in San Francisco *you* have to wear a jacket.

23.1.4 Linguistic Properties of the Coreference Relation

Now that we have seen the linguistic properties of individual referring expressions we turn to properties of the antecedent/anaphor pair. Understanding these properties is helpful both in designing novel features and performing error analyses.

Number Agreement: Referring expressions and their referents must generally agree in number; English *she/her/he/him/his/it* are singular, *we/us/they/them* are plural, and *you* is unspecified for number. So a plural antecedent like *the chefs* cannot generally corefer with a singular anaphor like *she*. However, algorithms cannot enforce number agreement too strictly. First, semantically plural entities can be referred to by either *it* or *they*:

(23.31) IBM announced a new machine translation product yesterday. *They* have been working on it for 20 years.

singular they

Second, **singular they** has become much more common, in which *they* is used to describe singular individuals, often useful because *they* is gender neutral. Although recently increasing, singular they is quite old, part of English for many centuries.[5]

Person Agreement: English distinguishes between first, second, and third person, and a pronoun's antecedent must agree with the pronoun in person. Thus a third person pronoun (*he, she, they, him, her, them, his, her, their*) must have a third person antecedent (one of the above or any other noun phrase). However, phenomena like quotation can cause exceptions; in this example *I*, *my*, and *she* are coreferent:

(23.32) "I voted for Nader because he was most aligned with my values," she said.

Gender or Noun Class Agreement: In many languages, all nouns have grammatical gender or noun class[6] and pronouns generally agree with the grammatical gender of their antecedent. In English this occurs only with third-person singular pronouns, which distinguish between *male* (*he, him, his*), *female* (*she, her*), and *nonpersonal* (*it*) grammatical genders. Non-binary pronouns like *ze* or *hir* may also occur in more recent texts. Knowing which gender to associate with a name in text can be complex, and may require world knowledge about the individual. Some examples:

(23.33) Maryam has a theorem. She is exciting. (she=Maryam, not the theorem)

(23.34) Maryam has a theorem. It is exciting. (it=the theorem, not Maryam)

Binding Theory Constraints: The **binding theory** is a name for syntactic constraints on the relations between a mention and an antecedent in the same sentence (Chomsky, 1981). Oversimplifying a bit, **reflexive** pronouns like *himself* and *herself* corefer with the subject of the most immediate clause that contains them (23.35), whereas nonreflexives cannot corefer with this subject (23.36).

reflexive

(23.35) Janet bought herself a bottle of fish sauce. [herself=Janet]

(23.36) Janet bought her a bottle of fish sauce. [her≠Janet]

Recency: Entities introduced in recent utterances tend to be more salient than those introduced from utterances further back. Thus, in (23.37), the pronoun *it* is more likely to refer to Jim's map than the doctor's map.

(23.37) The doctor found an old map in the captain's chest. Jim found an even older map hidden on the shelf. It described an island.

Grammatical Role: Entities mentioned in subject position are more salient than those in object position, which are in turn more salient than those mentioned in oblique positions. Thus although the first sentence in (23.38) and (23.39) expresses roughly the same propositional content, the preferred referent for the pronoun *he* varies with the subject—Billy Bones in (23.38) and Jim Hawkins in (23.39).

(23.38) Billy Bones went to the bar with Jim Hawkins. He called for a glass of rum. [he = Billy]

(23.39) Jim Hawkins went to the bar with Billy Bones. He called for a glass of rum. [he = Jim]

---

[5] Here's a bound pronoun example from Shakespeare's *Comedy of Errors*: *There's not a man I meet but doth salute me As if I were their well-acquainted friend*

[6] The word "gender" is generally only used for languages with 2 or 3 noun classes, like most Indo-European languages; many languages, like the Bantu languages or Chinese, have a much larger number of noun classes.

Verb Semantics: Some verbs semantically emphasize one of their arguments, biasing the interpretation of subsequent pronouns. Compare (23.40) and (23.41).

(23.40) John telephoned Bill. He lost the laptop.

(23.41) John criticized Bill. He lost the laptop.

These examples differ only in the verb used in the first sentence, yet "he" in (23.40) is typically resolved to John, whereas "he" in (23.41) is resolved to Bill. This may be partly due to the link between implicit causality and saliency: the implicit cause of a "criticizing" event is its object, whereas the implicit cause of a "telephoning" event is its subject. In such verbs, the entity which is the implicit cause may be more salient.

Selectional Restrictions: Many other kinds of semantic knowledge can play a role in referent preference. For example, the selectional restrictions that a verb places on its arguments (Chapter 21) can help eliminate referents, as in (23.42).

(23.42) I ate the soup in my new bowl after cooking it for hours

There are two possible referents for *it*, the soup and the bowl. The verb *eat*, however, requires that its direct object denote something edible, and this constraint can rule out *bowl* as a possible referent.

23.2 Coreference Tasks and Datasets

We can formulate the task of coreference resolution as follows: Given a text T, find all entities and the coreference links between them. We evaluate our task by comparing the links our system creates with those in human-created gold coreference annotations on T.

Let's return to our coreference example, now using superscript numbers for each coreference chain (cluster), and subscript letters for individual mentions in the cluster:

(23.43) [Victoria Chen]^1_a, CFO of [Megabucks Banking]^2_a, saw [[her]^1_b pay]^3_a jump to $2.3 million, as [the 38-year-old]^1_c also became [[the company]^2_b's president. It is widely known that [she]^1_d came to [Megabucks]^2_c from rival [Lotsabucks]^4_a.

Assuming example (23.43) was the entirety of the article, the chains for *her pay* and *Lotsabucks* are singleton mentions:

1. {*Victoria Chen*, *her*, *the 38-year-old*, *She*}
2. {*Megabucks Banking*, *the company*, *Megabucks*}
3. { *her pay*}
4. { *Lotsabucks*}

For most coreference evaluation campaigns, the input to the system is the raw text of articles, and systems must detect mentions and then link them into clusters. Solving this task requires dealing with pronominal anaphora (figuring out that *her* refers to *Victoria Chen*), filtering out non-referential pronouns like the pleonastic *It* in *It has been ten years*), dealing with definite noun phrases to figure out that *the 38-year-old* is coreferent with *Victoria Chen*, and that *the company* is the same as *Megabucks*. And we need to deal with names, to realize that *Megabucks* is the same as *Megabucks Banking*.

Exactly what counts as a mention and what links are annotated differs from task to task and dataset to dataset. For example some coreference datasets do not label singletons, making the task much simpler. Resolvers can achieve much higher scores on corpora without singletons, since singletons constitute the majority of mentions in running text, and they are often hard to distinguish from non-referential NPs. Some tasks use gold mention-detection (i.e. the system is given human-labeled mention boundaries and the task is just to cluster these gold mentions), which eliminates the need to detect and segment mentions from running text.

Coreference is usually evaluated by the **CoNLL F1** score, which combines three metrics: MUC, B^3, and $CEAF_e$; Section 23.8 gives the details.

Let's mention a few characteristics of one popular coreference dataset, OntoNotes (Pradhan et al. 2007c, Pradhan et al. 2007a), and the CoNLL 2012 Shared Task based on it (Pradhan et al., 2012a). OntoNotes contains hand-annotated Chinese and English coreference datasets of roughly one million words each, consisting of newswire, magazine articles, broadcast news, broadcast conversations, web data and conversational speech data, as well as about 300,000 words of annotated Arabic newswire. The most important distinguishing characteristic of OntoNotes is that it does not label singletons, simplifying the coreference task, since singletons represent 60%-70% of all entities. In other ways, it is similar to other coreference datasets. Referring expression NPs that are coreferent are marked as mentions, but generics and pleonastic pronouns are not marked. Appositive clauses are not marked as separate mentions, but they are included in the mention. Thus in the NP, "Richard Godown, president of the Industrial Biotechnology Association" the mention is the entire phrase. Prenominal modifiers are annotated as separate entities only if they are proper nouns. Thus *wheat* is not an entity in *wheat fields*, but *UN* is an entity in *UN policy* (but not adjectives like *American* in *American policy*).

A number of corpora mark richer discourse phenomena. The ISNotes corpus annotates a portion of OntoNotes for information status, include bridging examples (Hou et al., 2018). The LitBank coreference corpus (Bamman et al., 2020) contains coreference annotations for 210,532 tokens from 100 different literary novels, including singletons and quantified and negated noun phrases. The AnCora-CO coreference corpus (Recasens and Martí, 2010) contains 400,000 words each of Spanish (AnCora-CO-Es) and Catalan (AnCora-CO-Ca) news data, and includes labels for complex phenomena like discourse deixis in both languages. The ARRAU corpus (Uryupina et al., 2020) contains 350,000 words of English marking all NPs, which means singleton clusters are available. ARRAU includes diverse genres like dialog (the TRAINS data) and fiction (the Pear Stories), and has labels for bridging references, discourse deixis, generics, and ambiguous anaphoric relations.

23.3 Mention Detection

mention detection

The first stage of coreference is **mention detection**: finding the spans of text that constitute each mention. Mention detection algorithms are usually very liberal in proposing candidate mentions (i.e., emphasizing recall), and only filtering later. For example many systems run parsers and named entity taggers on the text and extract every span that is either an **NP**, a **possessive pronoun**, or a **named entity**.

Doing so from our sample text repeated in (23.44):

(23.44) Victoria Chen, CFO of Megabucks Banking, saw her pay jump to $2.3

million, as the 38-year-old also became the company's president. It is widely known that she came to Megabucks from rival Lotsabucks.

might result in the following list of 13 potential mentions:

Victoria Chen	$2.3 million	she
CFO of Megabucks Banking	the 38-year-old	Megabucks
Megabucks Banking	the company	Lotsabucks
her	the company's president	
her pay	It	

More recent mention detection systems are even more generous; the span-based algorithm we will describe in Section 23.6 first extracts literally all n-gram spans of words up to N=10. Of course recall from Section 23.1.3 that many NPs—and the overwhelming majority of random n-gram spans—are not referring expressions. Therefore all such mention detection systems need to eventually filter out pleonastic/expletive pronouns like *It* above, appositives like *CFO of Megabucks Banking Inc*, or predicate nominals like *the company's president* or *$2.3 million*.

Some of this filtering can be done by rules. Early rule-based systems designed regular expressions to deal with pleonastic *it*, like the following rules from Lappin and Leass (1994) that use dictionaries of cognitive verbs (e.g., *believe*, *know*, *anticipate*) to capture pleonastic *it* in "It is *thought* that ketchup...", or modal adjectives (e.g., *necessary*, *possible*, *certain*, *important*), for, e.g., "It is *likely* that I...". Such rules are sometimes used as part of modern systems:

```
It is Modaladjective that S
It is Modaladjective (for NP) to VP
It is Cogv-ed that S
It seems/appears/means/follows (that) S
```

Mention-detection rules are sometimes designed specifically for particular evaluation campaigns. For OntoNotes, for example, mentions are not embedded within larger mentions, and while numeric quantities are annotated, they are rarely coreferential. Thus for OntoNotes tasks like CoNLL 2012 (Pradhan et al., 2012a), a common first pass rule-based mention detection algorithm (Lee et al., 2013) is:

1. Take all NPs, possessive pronouns, and named entities.
2. Remove numeric quantities (100 dollars, 8%), mentions embedded in larger mentions, adjectival forms of nations, and stop words (like *there*).
3. Remove pleonastic *it* based on regular expression patterns.

Rule-based systems, however, are generally insufficient to deal with mention-detection, and so modern systems incorporate some sort of learned mention detection component, such as a **referentiality** classifier, an **anaphoricity classifier**—detecting whether an NP is an anaphor—or a **discourse-new** classifier— detecting whether a mention is discourse-new and a potential antecedent for a future anaphor.

anaphoricity detector

An **anaphoricity detector**, for example, can draw its positive training examples from any span that is labeled as an anaphoric referring expression in hand-labeled datasets like OntoNotes, ARRAU, or AnCora. Any other NP or named entity can be marked as a negative training example. Anaphoricity classifiers use features of the candidate mention such as its head word, surrounding words, definiteness, animacy, length, position in the sentence/discourse, many of which were first proposed in early work by Ng and Cardie (2002a); see Section 23.5 for more on features.

Referentiality or anaphoricity detectors can be run as filters, in which only mentions that are classified as anaphoric or referential are passed on to the coreference system. The end result of such a filtering mention detection system on our example above might be the following filtered set of 9 potential mentions:

Victoria Chen	her pay	she
Megabucks Bank	the 38-year-old	Megabucks
her	the company	Lotsabucks

It turns out, however, that hard filtering of mentions based on an anaphoricity or referentiality classifier leads to poor performance. If the anaphoricity classifier threshold is set too high, too many mentions are filtered out and recall suffers. If the classifier threshold is set too low, too many pleonastic or non-referential mentions are included and precision suffers.

The modern approach is instead to perform mention detection, anaphoricity, and coreference jointly in a single end-to-end model (Ng 2005b, Denis and Baldridge 2007, Rahman and Ng 2009). For example mention detection in the Lee et al. (2017b),2018 system is based on a single end-to-end neural network that computes a score for each mention being referential, a score for two mentions being coreference, and combines them to make a decision, training all these scores with a single end-to-end loss. We'll describe this method in detail in Section 23.6. [7]

Despite these advances, correctly detecting referential mentions seems to still be an unsolved problem, since systems incorrectly marking pleonastic pronouns like *it* and other non-referential NPs as coreferent is a large source of errors of modern coreference resolution systems (Kummerfeld and Klein 2013, Martschat and Strube 2014, Martschat and Strube 2015, Wiseman et al. 2015, Lee et al. 2017a).

Mention, referentiality, or anaphoricity detection is thus an important open area of investigation. Other sources of knowledge may turn out to be helpful, especially in combination with unsupervised and semisupervised algorithms, which also mitigate the expense of labeled datasets. In early work, for example Bean and Riloff (1999) learned patterns for characterizing anaphoric or non-anaphoric NPs; (by extracting and generalizing over the first NPs in a text, which are guaranteed to be non-anaphoric). Chang et al. (2012) look for head nouns that appear frequently in the training data but never appear as gold mentions to help find non-referential NPs. Bergsma et al. (2008b) use web counts as a semisupervised way to augment standard features for anaphoricity detection for English *it*, an important task because *it* is both common and ambiguous; between a quarter and half *it* examples are non-anaphoric. Consider the following two examples:

(23.45) You can make [it] in advance. [anaphoric]

(23.46) You can make [it] in Hollywood. [non-anaphoric]

The *it* in *make it* is non-anaphoric, part of the idiom *make it*. Bergsma et al. (2008b) turn the context around each example into patterns, like "make * in advance" from (23.45), and "make * in Hollywood" from (23.46). They then use Google n-grams to enumerate all the words that can replace *it* in the patterns. Non-anaphoric contexts tend to only have *it* in the wildcard positions, while anaphoric contexts occur with many other NPs (for example *make them in advance* is just as frequent in their data

[7] Some systems try to avoid mention detection or anaphoricity detection altogether. For datasets like OntoNotes which don't label singletons, an alternative to filtering out non-referential mentions is to run coreference resolution, and then simply delete any candidate mentions which were not corefered with another mention. This likely doesn't work as well as explicitly modeling referentiality, and cannot solve the problem of detecting singletons, which is important for tasks like entity linking.

as *make it in advance*, but *make them in Hollywood* did not occur at all). These n-gram contexts can be used as features in a supervised anaphoricity classifier.

23.4 Architectures for Coreference Algorithms

Modern systems for coreference are based on supervised neural machine learning, supervised from hand-labeled datasets like OntoNotes. In this section we overview the various architecture of modern systems, using the categorization of Ng (2010), which distinguishes algorithms based on whether they make each coreference decision in a way that is *entity-based*—representing each entity in the discourse model—or only *mention-based*—considering each mention independently, and whether they use *ranking models* to directly compare potential antecedents. Afterwards, we go into more detail on one state-of-the-art algorithm in Section 23.6.

23.4.1 The Mention-Pair Architecture

mention-pair

We begin with the **mention-pair** architecture, the simplest and most influential coreference architecture, which introduces many of the features of more complex algorithms, even though other architectures perform better. The **mention-pair** architecture is based around a classifier that— as its name suggests—is given a pair of mentions, a candidate anaphor and a candidate antecedent, and makes a binary classification decision: coreferring or not.

Let's consider the task of this classifier for the pronoun *she* in our example, and assume the slightly simplified set of potential antecedents in Fig. 23.2.

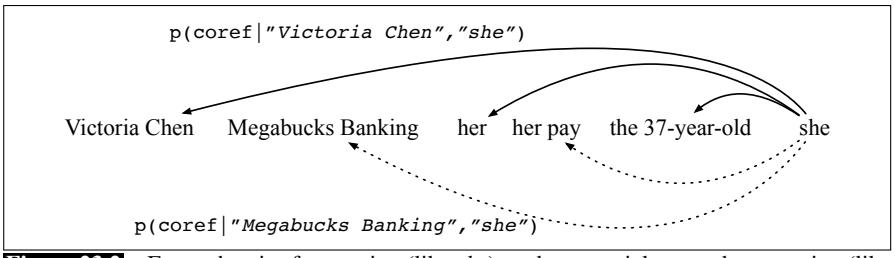


Figure 23.2 For each pair of a mention (like *she*), and a potential antecedent mention (like *Victoria Chen* or *her*), the mention-pair classifier assigns a probability of a coreference link.

For each prior mention (*Victoria Chen*, *Megabucks Banking*, *her*, etc.), the binary classifier computes a probability: whether or not the mention is the antecedent of *she*. We want this probability to be high for actual antecedents (*Victoria Chen*, *her*, *the 38-year-old*) and low for non-antecedents (*Megabucks Banking*, *her pay*).

Early classifiers used hand-built features (Section 23.5); more recent classifiers use neural representation learning (Section 23.6)

For training, we need a heuristic for selecting training samples; since most pairs of mentions in a document are not coreferent, selecting every pair would lead to a massive overabundance of negative samples. The most common heuristic, from (Soon et al., 2001), is to choose the closest antecedent as a positive example, and all pairs in between as the negative examples. More formally, for each anaphor mention m_i we create

- one positive instance (m_i, m_j) where m_j is the closest antecedent to m_i, and

- a negative instance (m_i, m_k) for each m_k between m_j and m_i

Thus for the anaphor *she*, we would choose (*she*, *her*) as the positive example and no negative examples. Similarly, for the anaphor *the company* we would choose (*the company*, *Megabucks*) as the positive example and (*the company*, *she*) (*the company*, *the 38-year-old*) (*the company*, *her pay*) and (*the company*, *her*) as negative examples.

Once the classifier is trained, it is applied to each test sentence in a clustering step. For each mention i in a document, the classifier considers each of the prior $i-1$ mentions. In **closest-first** clustering (Soon et al., 2001), the classifier is run right to left (from mention $i-1$ down to mention 1) and the first antecedent with probability $> .5$ is linked to i. If no antecedent has probably > 0.5, no antecedent is selected for i. In **best-first** clustering, the classifier is run on all $i-1$ antecedents and the most probable preceding mention is chosen as the antecedent for i. The transitive closure of the pairwise relation is taken as the cluster.

While the mention-pair model has the advantage of simplicity, it has two main problems. First, the classifier doesn't directly compare candidate antecedents to each other, so it's not trained to decide, between two likely antecedents, which one is in fact better. Second, it ignores the discourse model, looking only at mentions, not entities. Each classifier decision is made completely locally to the pair, without being able to take into account other mentions of the same entity. The next two models each address one of these two flaws.

23.4.2 The Mention-Rank Architecture

The mention ranking model directly compares candidate antecedents to each other, choosing the highest-scoring antecedent for each anaphor.

In early formulations, for mention i, the classifier decides which of the $\{1,...,i-1\}$ prior mentions is the antecedent (Denis and Baldridge, 2008). But suppose i is in fact not anaphoric, and none of the antecedents should be chosen? Such a model would need to run a separate anaphoricity classifier on i. Instead, it turns out to be better to jointly learn anaphoricity detection and coreference together with a single loss (Rahman and Ng, 2009).

So in modern mention-ranking systems, for the ith mention (anaphor), we have an associated random variable y_i ranging over the values $Y(i) = \{1,...,i-1,\epsilon\}$. The value ϵ is a special dummy mention meaning that i does not have an antecedent (i.e., is either discourse-new and starts a new coref chain, or is non-anaphoric).

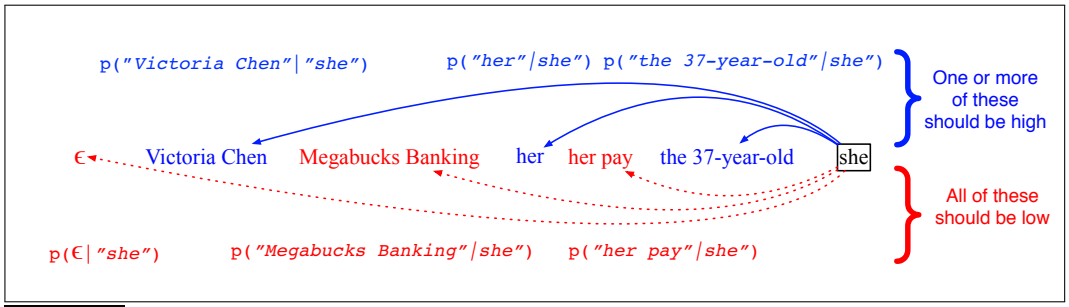


Figure 23.3 For each candidate anaphoric mention (like *she*), the mention-ranking system assigns a probability distribution over all previous mentions plus the special dummy mention ϵ.

At test time, for a given mention i the model computes one softmax over all the antecedents (plus ϵ) giving a probability for each candidate antecedent (or none).

Fig. 23.3 shows an example of the computation for the single candidate anaphor *she*.

Once the antecedent is classified for each anaphor, transitive closure can be run over the pairwise decisions to get a complete clustering.

Training is trickier in the mention-ranking model than the mention-pair model, because for each anaphor we don't know which of all the possible gold antecedents to use for training. Instead, the best antecedent for each mention is *latent*; that is, for each mention we have a whole cluster of legal gold antecedents to choose from. Early work used heuristics to choose an antecedent, for example choosing the closest antecedent as the gold antecedent and all non-antecedents in a window of two sentences as the negative examples (Denis and Baldridge, 2008). Various kinds of ways to model latent antecedents exist (Fernandes et al. 2012, Chang et al. 2013, Durrett and Klein 2013). The simplest way is to give credit to any legal antecedent by summing over all of them, with a loss function that optimizes the likelihood of all correct antecedents from the gold clustering (Lee et al., 2017b). We'll see the details in Section 23.6.

Mention-ranking models can be implemented with hand-build features or with neural representation learning (which might also incorporate some hand-built features). we'll explore both directions in Section 23.5 and Section 23.6.

23.4.3 Entity-based Models

Both the mention-pair and mention-ranking models make their decisions about *mentions*. By contrast, entity-based models link each mention not to a previous mention but to a previous discourse *entity* (cluster of mentions).

A mention-ranking model can be turned into an entity-ranking model simply by having the classifier make its decisions over clusters of mentions rather than individual mentions (Rahman and Ng, 2009).

For traditional feature-based models, this can be done by extracting features over clusters. The size of a cluster is a useful feature, as is its 'shape', which is the list of types of the mentions in the cluster i.e., sequences of the tokens (P)roper, (D)efinite, (I)ndefinite, (Pr)onoun, so that a cluster composed of {*Victoria*, *her*, *the 38-year-old*} would have the shape *P-Pr-D* (Björkelund and Kuhn, 2014). An entity-based model that includes a mention-pair classifier can use as features aggregates of mention-pair probabilities, for example computing the average probability of coreference over all mention-pairs in the two clusters (Clark and Manning 2015).

Neural models can learn representations of clusters automatically, for example by using an RNN over the sequence of cluster mentions to encode a state corresponding to a cluster representation (Wiseman et al., 2016), or by learning distributed representations for pairs of clusters by pooling over learned representations of mention pairs (Clark and Manning, 2016b).

However, although entity-based models are more expressive, the use of cluster-level information in practice has not led to large gains in performance, so mention-ranking models are still more commonly used.

23.5 Classifiers using hand-built features

Feature-based classifiers, use hand-designed features in logistic regression, SVM, or random forest classifiers for coreference resolution. These classifiers don't per-

form as well as neural ones. Nonetheless, they are still sometimes useful to build lightweight systems when compute or data are sparse, and the features themselves are useful for error analysis even in neural systems.

Given an anaphor mention and a potential antecedent mention, feature based classifiers make use of three types of features: (i) features of the anaphor, (ii) features of the candidate antecedent, and (iii) features of the relationship between the pair. Entity-based models can make additional use of two additional classes: (iv) feature of all mentions from the antecedent's entity cluster, and (v) features of the relation between the anaphor and the mentions in the antecedent entity cluster.

Features of the Anaphor or Antecedent Mention		
First (last) word	Victoria/she	First or last word (or embedding) of antecedent/anaphor
Head word	Victoria/she	Head word (or head embedding) of antecedent/anaphor
Attributes	Sg-F-A-3-PER/ Sg-F-A-3-PER	The number, gender, animacy, person, named entity type attributes of (antecedent/anaphor)
Length	2/1	length in words of (antecedent/anaphor)
Mention type	P/Pr	Type: (P)roper, (D)efinite, (I)ndefinite, (Pr)onoun) of antecedent/anaphor
Features of the Antecedent Entity		
Entity shape	P-Pr-D	The 'shape' or list of types of the mentions in the antecedent entity (cluster), i.e., sequences of (P)roper, (D)efinite, (I)ndefinite, (Pr)onoun.
Entity attributes	Sg-F-A-3-PER	The number, gender, animacy, person, named entity type attributes of the antecedent entity
Ant. cluster size	3	Number of mentions in the antecedent cluster
Features of the Pair of Mentions		
Sentence distance	1	The number of sentences between antecedent and anaphor
Mention distance	4	The number of mentions between antecedent and anaphor
i-within-i	F	Anaphor has i-within-i relation with antecedent
Cosine		Cosine between antecedent and anaphor embeddings
Features of the Pair of Entities		
Exact String Match	F	True if the strings of any two mentions from the antecedent and anaphor clusters are identical.
Head Word Match	F	True if any mentions from antecedent cluster has same headword as any mention in anaphor cluster
Word Inclusion	F	All words in anaphor cluster included in antecedent cluster

Figure 23.4 Feature-based coreference: sample feature values for anaphor "she" and potential antecedent "Victoria Chen".

Figure 23.4 shows a selection of commonly used features, and shows the value that would be computed for the potential anaphor "she" and potential antecedent "Victoria Chen" in our example sentence, repeated below:

(23.47) **Victoria Chen**, CFO of Megabucks Banking, saw her pay jump to $2.3 million, as the 38-year-old also became the company's president. It is widely known that **she** came to Megabucks from rival Lotsabucks.

Features that prior work has found to be particularly useful are exact string match, entity headword agreement, mention distance, as well as (for pronouns) exact attribute match and i-within-i, and (for nominals and proper names) word inclusion and cosine. For lexical features (like head words) it is common to only use words that appear enough times (>20 times).

It is crucial in feature-based systems to use conjunctions of features; one experiment suggested that moving from individual features in a classifier to conjunctions of multiple features increased F1 by 4 points (Lee et al., 2017a). Specific conjunctions can be designed by hand (Durrett and Klein, 2013), all pairs of features can be conjoined (Bengtson and Roth, 2008), or feature conjunctions can be learned using decision tree or random forest classifiers (Ng and Cardie 2002a, Lee et al. 2017a).

Features can also be used in neural models as well. Neural systems use contextual word embeddings so don't benefit from shallow features like string match or or mention types. However features like mention length, distance between mentions, or genre can complement neural contextual embedding models.

23.6 A neural mention-ranking algorithm

In this section we describe the neural **e2e-coref** algorithms of Lee et al. (2017b) (simplified and extended a bit, drawing on Joshi et al. (2019) and others). This is a **mention-ranking** algorithm that considers all possible spans of text in the document, assigns a mention-score to each span, prunes the mentions based on this score, then assigns coreference links to the remaining mentions.

More formally, given a document D with T words, the model considers all of the $\frac{T(T+1)}{2}$ text spans in D (unigrams, bigrams, trigrams, 4-grams, etc; in practice we only consider spans up a maximum length around 10). The task is to assign to each span i an antecedent y_i, a random variable ranging over the values $Y(i) = \{1, ..., i-1, \epsilon\}$; each previous span and a special dummy token ϵ. Choosing the dummy token means that i does not have an antecedent, either because i is discourse-new and starts a new coreference chain, or because i is non-anaphoric.

For each pair of spans i and j, the system assigns a score $s(i,j)$ for the coreference link between span i and span j. The system then learns a distribution $P(y_i)$ over the antecedents for span i:

$$P(y_i) = \frac{\exp(s(i,y_i))}{\sum_{y' \in Y(i)} \exp(s(i,y'))} \tag{23.48}$$

This score $s(i,j)$ includes three factors that we'll define below: $m(i)$; whether span i is a mention; $m(j)$; whether span j is a mention; and $c(i,j)$; whether j is the antecedent of i:

$$s(i,j) = m(i) + m(j) + c(i,j) \tag{23.49}$$

For the dummy antecedent ϵ, the score $s(i,\epsilon)$ is fixed to 0. This way if any non-dummy scores are positive, the model predicts the highest-scoring antecedent, but if all the scores are negative it abstains.

23.6.1 Computing span representations

To compute the two functions $m(i)$ and $c(i,j)$ which score a span i or a pair of spans (i,j), we'll need a way to represent a span. The e2e-coref family of algorithms represents each span by trying to capture 3 words/tokens: the first word, the last word, and the most important word. We first run each paragraph or subdocument through an encoder (like BERT) to generate embeddings h_i for each token i. The span i is then represented by a vector $\mathbf{g}_i$ that is a concatenation of the encoder output

embedding for the first (start) token of the span, the encoder output for the last (end) token of the span, and a third vector which is an attention-based representation:

$$\mathbf{g}_i = [\mathbf{h}_{\text{START}(i)}, \mathbf{h}_{\text{END}(i)}, \mathbf{h}_{\text{ATT}(i)}] \tag{23.50}$$

The goal of the attention vector is to represent which word/token is the likely syntactic head-word of the span; we saw in the prior section that head-words are a useful feature; a matching head-word is a good indicator of coreference. The attention representation is computed as usual; the system learns a weight vector $\mathbf{w}_\alpha$, and computes its dot product with the hidden state $\mathbf{h}_t$ transformed by a FFN:

$$\alpha_t = \mathbf{w}_\alpha \cdot \text{FFN}_\alpha(\mathbf{h}_t) \tag{23.51}$$

The attention score is normalized into a distribution via a softmax:

$$a_{i,t} = \frac{\exp(\alpha_t)}{\sum_{k=\text{START}(i)}^{\text{END}(i)} \exp(\alpha_k)} \tag{23.52}$$

And then the attention distribution is used to create a vector $\mathbf{h}_{\text{ATT}(i)}$ which is an attention-weighted sum of the embeddings $\mathbf{e}_t$ of each of the words in span i:

$$\mathbf{h}_{\text{ATT}(i)} = \sum_{t=\text{START}(i)}^{\text{END}(i)} a_{i,t} \cdot \mathbf{e}_t \tag{23.53}$$

Fig. 23.5 shows the computation of the span representation and the mention score.

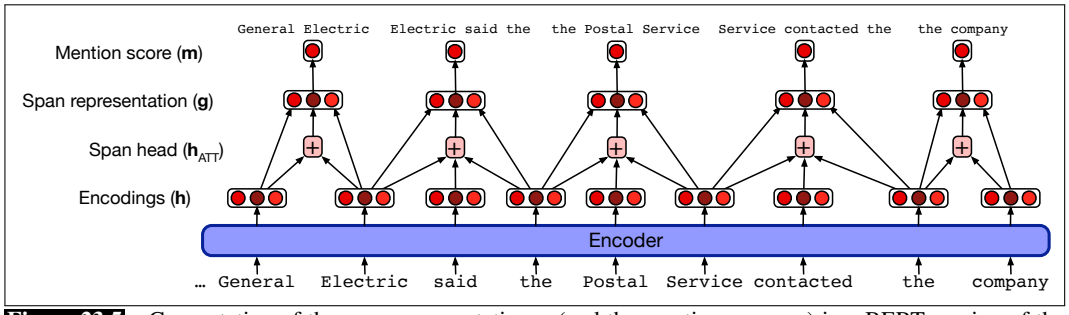


Figure 23.5 Computation of the span representation $\mathbf{g}$ (and the mention score $\mathbf{m}$) in a BERT version of the e2e-coref model (Lee et al. 2017b, Joshi et al. 2019). The model considers all spans up to a maximum width of say 10; the figure shows a small subset of the bigram and trigram spans.

23.6.2 Computing the mention and antecedent scores m and c

Now that we know how to compute the vector $\mathbf{g}_i$ for representing span i, we can see the details of the two scoring functions $m(i)$ and $c(i,j)$. Both are computed by feedforward networks:

$$m(i) = w_m \cdot \text{FFN}_m(\mathbf{g}_i) \tag{23.54}$$

$$c(i,j) = w_c \cdot \text{FFN}_c([\mathbf{g}_i, \mathbf{g}_j, \mathbf{g}_i \circ \mathbf{g}_j,]) \tag{23.55}$$

At inference time, this mention score m is used as a filter to keep only the best few mentions.

We then compute the antecedent score for high-scoring mentions. The antecedent score $c(i, j)$ takes as input a representation of the spans i and j, but also the element-wise similarity of the two spans to each other $\mathbf{g}_i \circ \mathbf{g}_j$ (here $\circ$ is element-wise multiplication). Fig. 23.6 shows the computation of the score s for the three possible antecedents of *the company* in the example sentence from Fig. 23.5.

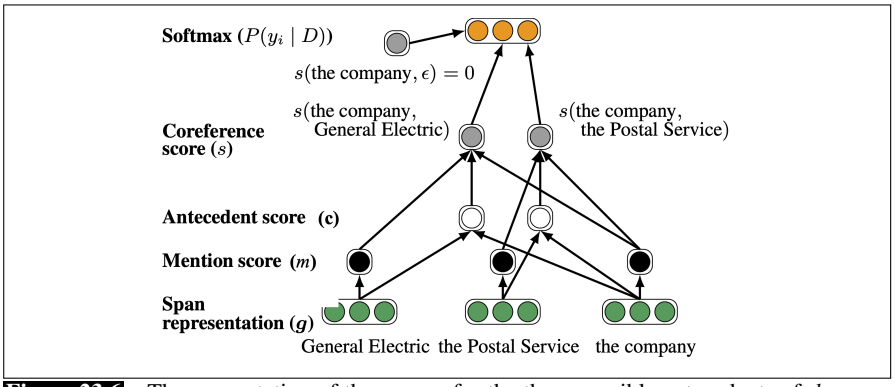


Figure 23.6 The computation of the score s for the three possible antecedents of *the company* in the example sentence from Fig. 23.5. Figure after Lee et al. (2017b).

Given the set of mentions, the joint distribution of antecedents for each document is computed in a forward pass, and we can then do transitive closure on the antecedents to create a final clustering for the document.

Fig. 23.7 shows example predictions from the model, showing the attention weights, which Lee et al. (2017b) find correlate with traditional semantic heads. Note that the model gets the second example wrong, presumably because *attendants* and *pilot* likely have nearby word embeddings.

We are looking for **(a region of central Italy bordering the Adriatic Sea)**. **(The area)** is mostly mountainous and includes Mt. Corno, the highest peak of the Apennines. **(It)** also includes a lot of sheep, good clean-living, healthy sheep, and an Italian entrepreneur has an idea about how to make a little money of them.

(The flight attendants) have until 6:00 today to ratify labor concessions. **(The pilots')** union and ground crew did so yesterday.

Figure 23.7 Sample predictions from the Lee et al. (2017b) model, with one cluster per example, showing one correct example and one mistake. Bold, parenthesized spans are mentions in the predicted cluster. The amount of red color on a word indicates the head-finding attention weight $a_{i,t}$ in Eq. 23.52. Figure adapted from Lee et al. (2017b).

23.6.3 Learning

For training, we don't have a single gold antecedent for each mention; instead the coreference labeling only gives us each entire cluster of coreferent mentions; so a mention only has a latent antecedent. We therefore use a loss function that maximizes the sum of the coreference probability of any of the legal antecedents. For a given mention i with possible antecedents $Y(i)$, let $\text{GOLD}(i)$ be the set of mentions in the gold cluster containing i. Since the set of mentions occurring before i is $Y(i)$, the set of mentions in that gold cluster that also occur before i is $Y(i) \cap \text{GOLD}(i)$. We

therefore want to maximize:

$$\sum_{\hat{y} \in Y(i) \cap \text{GOLD}(i)} P(\hat{y}) \tag{23.56}$$

If a mention i is not in a gold cluster $\text{GOLD}(i) = \epsilon$.

To turn this probability into a loss function, we'll use the cross-entropy loss function we defined in Eq. 4.19 in Chapter 4, by taking the $-\log$ of the probability. If we then sum over all mentions, we get the final loss function for training:

$$L = \sum_{i=2}^{N} -\log \sum_{\hat{y} \in Y(i) \cap \text{GOLD}(i)} P(\hat{y}) \tag{23.57}$$

23.7 Entity Linking

entity linking

Entity linking is the task of associating a mention in text with the representation of some real-world entity in an ontology or knowledge base (Ji and Grishman, 2011). It is the natural follow-on to coreference resolution; coreference resolution is the task of associating textual mentions that corefer to the same entity. Entity linking takes the further step of identifying who that entity is. It is especially important for any NLP task that links to a knowledge base.

While there are all sorts of potential knowledge-bases, we'll focus in this section on Wikipedia, since it's widely used as an ontology for NLP tasks. In this usage, each unique Wikipedia page acts as the unique id for a particular entity. This task of deciding which Wikipedia page corresponding to an individual is being referred to by a text mention has its own name: **wikification** (Mihalcea and Csomai, 2007).

wikification

Since the earliest systems (Mihalcea and Csomai 2007, Cucerzan 2007, Milne and Witten 2008), entity linking is done in (roughly) two stages: **mention detection** and **mention disambiguation**. We'll give two algorithms, one simple classic baseline that uses **anchor dictionaries** and information from the Wikipedia graph structure (Ferragina and Scaiella, 2011) and one modern neural algorithm (Li et al., 2020). We'll focus here mainly on the application of entity linking to questions, since a lot of the literature has been in that context.

23.7.1 Linking based on Anchor Dictionaries and Web Graph

As a simple baseline we introduce the TAGME linker (Ferragina and Scaiella, 2011) for Wikipedia, which itself draws on earlier algorithms (Mihalcea and Csomai 2007, Cucerzan 2007, Milne and Witten 2008). Wikification algorithms define the set of entities as the set of Wikipedia pages, so we'll refer to each Wikipedia page as a unique entity e. TAGME first creates a catalog of all entities (i.e. all Wikipedia pages, removing some disambiguation and other meta-pages) and indexes them in a standard IR engine like Lucene. For each page e, the algorithm computes an **in-link** count $\text{in}(e)$: the total number of in-links from other Wikipedia pages that point to e. These counts can be derived from Wikipedia dumps.

anchor texts

Finally, the algorithm requires an **anchor dictionary**. An anchor dictionary lists for each Wikipedia page, its **anchor texts**: the hyperlinked spans of text on other pages that point to it. For example, the web page for Stanford University, `http://www.stanford.edu`, might be pointed to from another page using anchor texts like *Stanford* or *Stanford University*:

```
<a href="http://www.stanford.edu">Stanford University</a>
```

We compute a Wikipedia anchor dictionary by including, for each Wikipedia page e, e's title as well as all the anchor texts from all Wikipedia pages that point to e. For each anchor string a we'll also compute its total frequency $\text{freq}(a)$ in Wikipedia (including non-anchor uses), the number of times a occurs as a link (which we'll call $link(a)$), and its link probability $\text{linkprob}(a) = \text{link}(a)/\text{freq}(a)$. Some cleanup of the final anchor dictionary is required, for example removing anchor strings composed only of numbers or single characters, that are very rare, or that are very unlikely to be useful entities because they have a very low linkprob.

Mention Detection Given a question (or other text we are trying to link), TAGME detects mentions by querying the anchor dictionary for each token sequence up to 6 words. This large set of sequences is pruned with some simple heuristics (for example pruning substrings if they have small linkprobs). The question:

When was Ada Lovelace born?

might give rise to the anchor *Ada Lovelace* and possibly *Ada*, but substrings spans like *Lovelace* might be pruned as having too low a linkprob, and but spans like *born* have such a low linkprob that they would not be in the anchor dictionary at all.

Mention Disambiguation If a mention span is unambiguous (points to only one entity/Wikipedia page), we are done with entity linking! However, many spans are ambiguous, matching anchors for multiple Wikipedia entities/pages. The TAGME algorithm uses two factors for disambiguating ambiguous spans, which have been referred to as *prior probability* and *relatedness/coherence*. The first factor is $p(e|a)$, the probability with which the span refers to a particular entity. For each page $e \in \mathcal{E}(a)$, the probability $p(e|a)$ that anchor a points to e, is the ratio of the number of links into e with anchor text a to the total number of occurrences of a as an anchor:

$$\text{prior}(a \rightarrow e) \;=\; p(e|a) = \frac{\text{count}(a \rightarrow e)}{\text{link}(a)} \tag{23.58}$$

Let's see how that factor works in linking entities in the following question:

What Chinese Dynasty came before the Yuan?

The most common association for the span *Yuan* in the anchor dictionary is the name of the Chinese currency, i.e., the probability $p(\texttt{Yuan\_currency}|\ yuan)$ is very high. Rarer Wikipedia associations for *Yuan* include the common Chinese last name, a language spoken in Thailand, and the correct entity in this case, the name of the Chinese dynasty. So if we chose based only on $p(e|a)$, we would make the wrong disambiguation and miss the correct link, `Yuan_dynasty`.

To help in just this sort of case, TAGME uses a second factor, the **relatedness** of this entity to other entities in the input question. In our example, the fact that the question also contains the span *Chinese Dynasty*, which has a high probability link to the page `Dynasties_in_Chinese_history`, ought to help match `Yuan_dynasty`.

Let's see how this works. Given a question q, for each candidate anchors span a detected in q, we assign a relatedness score to each possible entity $e \in \mathcal{E}(a)$ of a. The relatedness score of the link $a \rightarrow e$ is the weighted average relatedness between e and all other entities in q. Two entities are considered related to the extent their Wikipedia pages share many in-links. More formally, the relatedness between two entities A and B is computed as

$$\text{rel}(A,B) \;=\; \frac{\log(\max(|\text{in}(A)|,|\text{in}(B)|)) - \log(|\text{in}(A) \cap \text{in}(B)|)}{\log(|W|) - \log(\min(|\text{in}(A)|,|\text{in}(B)|))} \tag{23.59}$$

where $\text{in}(x)$ is the set of Wikipedia pages pointing to x and W is the set of all Wikipedia pages in the collection.

The vote given by anchor b to the candidate annotation $a \rightarrow X$ is the average, over all the possible entities of b, of their relatedness to X, weighted by their prior probability:

$$\text{vote}(b,X) = \frac{1}{|\mathcal{E}(b)|} \sum_{Y \in \mathcal{E}(b)} \text{rel}(X,Y) p(Y|b) \tag{23.60}$$

The total relatedness score for $a \rightarrow X$ is the sum of the votes of all the other anchors detected in q:

$$\text{relatedness}(a \rightarrow X) = \sum_{b \in \mathcal{X}_q \setminus a} \text{vote}(b,X) \tag{23.61}$$

To score $a \rightarrow X$, we combine relatedness and prior by choosing the entity X that has the highest $\text{relatedness}(a \rightarrow X)$, finding other entities within a small ϵ of this value, and from this set, choosing the entity with the highest prior $P(X|a)$. The result of this step is a single entity assigned to each span in q.

The TAGME algorithm has one further step of pruning spurious anchor/entity pairs, assigning a score averaging link probability with the coherence.

$$\begin{aligned} \text{coherence}(a \rightarrow X) &= \frac{1}{|S|-1} \sum_{B \in \mathcal{S} \setminus X} \text{rel}(B,X) \\ \text{score}(a \rightarrow X) &= \frac{\text{coherence}(a \rightarrow X) + \text{linkprob}(a)}{2} \end{aligned} \tag{23.62}$$

Finally, pairs are pruned if $\text{score}(a \rightarrow X) < \lambda$, where the threshold λ is set on a held-out set.

23.7.2 Neural Graph-based linking

More recent entity linking models are based on **bi-encoders**, encoding a candidate mention span, encoding an entity, and computing the dot product between the encodings. This allows embeddings for all the entities in the knowledge base to be precomputed and cached (Wu et al., 2020). Let's sketch the ELQ linking algorithm of Li et al. (2020), which is given a question q and a set of candidate entities from Wikipedia with associated Wikipedia text, and outputs tuples (e, m_s, m_e) of entity id, mention start, and mention end. As Fig. 23.8 shows, it does this by encoding each Wikipedia entity using text from Wikipedia, encoding each mention span using text from the question, and computing their similarity, as we describe below.

Entity Mention Detection To get an h-dimensional embedding for each question token, the algorithm runs the question through BERT in the normal way:

$$[\mathbf{q}_1 \cdots \mathbf{q}_n] = \text{BERT}([\text{CLS}] q_1 \cdots q_n [\text{SEP}]) \tag{23.63}$$

It then computes the likelihood of each span $[i,j]$ in q being an entity mention, in a way similar to the span-based algorithm we saw for the reader above. First we compute the score for i/j being the start/end of a mention:

$$s_{\text{start}}(i) = \mathbf{w}_{\text{start}} \cdot \mathbf{q}_i, \quad s_{\text{end}}(j) = \mathbf{w}_{\text{end}} \cdot \mathbf{q}_j, \tag{23.64}$$

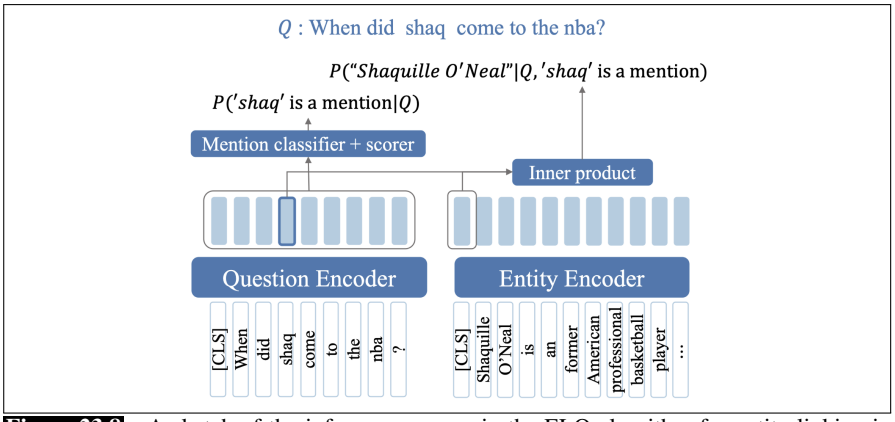


Figure 23.8 A sketch of the inference process in the ELQ algorithm for entity linking in questions (Li et al., 2020). Each candidate question mention span and candidate entity are separately encoded, and then scored by the entity/span dot product.

where $\mathbf{w}_{\text{start}}$ and $\mathbf{w}_{\text{end}}$ are vectors learned during training. Next, another trainable embedding, $\mathbf{w}_{\text{mention}}$ is used to compute a score for each token being part of a mention:

$$s_{\text{mention}}(t) = \mathbf{w}_{\text{mention}} \cdot \mathbf{q}_t \tag{23.65}$$

Mention probabilities are then computed by combining these three scores:

$$p([i,j]) = \sigma\left(s_{\text{start}}(i) + s_{\text{end}}(j) + \sum_{t=i}^{j} s_{\text{mention}}(t)\right) \tag{23.66}$$

Entity Linking To link mentions to entities, we next compute embeddings for each entity in the set $\mathcal{E} = e_1, \cdots, e_i, \cdots, e_w$ of all Wikipedia entities. For each entity e_i we'll get text from the entity's Wikipedia page, the title $t(e_i)$ and the first 128 tokens of the Wikipedia page which we'll call the description $d(e_i)$. This is again run through BERT, taking the output of the CLS token $\text{BERT}_{[\text{CLS}]}$ as the entity representation:

$$\mathbf{x}_{e_i} = \text{BERT}_{[\text{CLS}]}([\texttt{CLS}]t(e_i)[\texttt{ENT}]d(e_i)[\texttt{SEP}]) \tag{23.67}$$

Mention spans can be linked to entities by computing, for each entity e and span $[i,j]$, the dot product similarity between the span encoding (the average of the token embeddings) and the entity encoding.

$$\begin{aligned} \mathbf{y}_{i,j} &= \frac{1}{(j-i+1)} \sum_{t=i}^{j} \mathbf{q}_t \\ s(e,[i,j]) &= \mathbf{x}_e \mathbf{y}_{i,j} \end{aligned} \tag{23.68}$$

Finally, we take a softmax to get a distribution over entities for each span:

$$p(e|[i,j]) = \frac{\exp(s(e,[i,j]))}{\sum_{e' \in \mathcal{E}} \exp(s(e',[i,j]))} \tag{23.69}$$

Training The ELQ mention detection and entity linking algorithm is fully supervised. This means, unlike the anchor dictionary algorithms from Section 23.7.1,

it requires datasets with entity boundaries marked and linked. Two such labeled datasets are WebQuestionsSP (Yih et al., 2016), an extension of the WebQuestions (Berant et al., 2013) dataset derived from Google search questions, and GraphQuestions (Su et al., 2016). Both have had entity spans in the questions marked and linked (Sorokin and Gurevych 2018, Li et al. 2020) resulting in entity-labeled versions $\text{WebQSP}_{\text{EL}}$ and $\text{GraphQ}_{\text{EL}}$ (Li et al., 2020).

Given a training set, the ELQ mention detection and entity linking phases are trained jointly, optimizing the sum of their losses. The mention detection loss is a binary cross-entropy loss, with L the length of the passage and N the number of candidates:

$$\mathcal{L}_{\text{MD}} = -\frac{1}{N} \sum_{1 \le i \le j \le min(i+L-1,n)} \left(y_{[i,j]} \log p([i,j]) + (1 - y_{[i,j]}) \log(1 - p([i,j])) \right) \tag{23.70}$$

with $y_{[i,j]} = 1$ if $[i,j]$ is a gold mention span, else 0. The entity linking loss is:

$$\mathcal{L}_{\text{ED}} = -log p(e_g | [i,j]) \tag{23.71}$$

where e_g is the gold entity for mention $[i,j]$.

23.8 Evaluation of Coreference Resolution

We evaluate coreference algorithms model-theoretically, comparing a set of **hypothesis** chains or clusters H produced by the system against a set of gold or **reference** chains or clusters R from a human labeling, and reporting precision and recall.

However, there are a wide variety of methods for doing this comparison. In fact, there are 5 common metrics used to evaluate coreference algorithms: the **link** based MUC (Vilain et al., 1995) and BLANC (Recasens and Hovy 2011, Luo et al. 2014) metrics, the **mention** based B^3 metric (Bagga and Baldwin, 1998), the **entity** based CEAF metric (Luo, 2005), and the **link** based **entity** aware LEA metric (Moosavi and Strube, 2016).

MUC F-measure

Let's just explore two of the metrics. The **MUC F-measure** (Vilain et al., 1995) is based on the number of coreference *links* (pairs of mentions) common to H and R. Precision is the number of common links divided by the number of links in H. Recall is the number of common links divided by the number of links in R; This makes MUC biased toward systems that produce large chains (and fewer entities), and it ignores singletons, since they don't involve links.

B^3

B^3 is mention-based rather than link-based. For each mention in the reference chain, we compute a precision and recall, and then we take a weighted sum over all N mentions in the document to compute a precision and recall for the entire task. For a given mention i, let R be the reference chain that includes i, and H the hypothesis chain that has i. The set of correct mentions in H is $H \cap R$. Precision for mention i is thus $\frac{|H \cap R|}{|H|}$, and recall for mention i thus $\frac{|H \cap R|}{|R|}$. The total precision is the weighted sum of the precision for mention i, weighted by a weight w_i. The total recall is the weighted sum of the recall for mention i, weighted by a weight w_i. Equivalently:

$$\text{Precision} = \sum_{i=1}^{N} w_i \frac{\#\ of\ correct\ mentions\ in\ hypothesis\ chain\ containing\ entity_i}{\#\ of\ mentions\ in\ hypothesis\ chain\ containing\ entity_i}$$

$$\text{Recall} = \sum_{i=1}^{N} w_i \frac{\#\ of\ correct\ mentions\ in\ hypothesis\ chain\ containing\ entity_i}{\#\ of\ mentions\ in\ reference\ chain\ containing\ entity_i}$$

The weight w_i for each entity can be set to different values to produce different versions of the algorithm.

Following a proposal from Denis and Baldridge (2009), the CoNLL coreference competitions were scored based on the average of MUC, CEAF-e, and B^3 (Pradhan et al. 2011, Pradhan et al. 2012b), and so it is common in many evaluation campaigns to report an average of these 3 metrics. See Luo and Pradhan (2016) for a detailed description of the entire set of metrics; reference implementations of these should be used rather than attempting to reimplement from scratch (Pradhan et al., 2014).

Alternative metrics have been proposed that deal with particular coreference domains or tasks. For example, consider the task of resolving mentions to named entities (persons, organizations, geopolitical entities), which might be useful for information extraction or knowledge base completion. A hypothesis chain that correctly contains all the pronouns referring to an entity, but has no version of the name itself, or is linked with a wrong name, is not useful for this task. We might instead want a metric that weights each mention by how informative it is (with names being most informative) (Chen and Ng, 2013) or a metric that considers a hypothesis to match a gold chain only if it contains at least one variant of a name (the NEC F1 metric of Agarwal et al. (2019)).

23.9 Winograd Schema problems

From early on in the field, researchers have noted that some cases of coreference are quite difficult, seeming to require world knowledge or sophisticated reasoning to solve. The problem was most famously pointed out by Winograd (1972) with the following example:

(23.72) The city council denied the demonstrators a permit because

a. they feared violence.

b. they advocated violence.

Winograd noticed that the antecedent that most readers preferred for the pronoun *they* in continuation (a) was *the city council*, but in (b) was *the demonstrators*. He suggested that this requires understanding that the second clause is intended as an explanation of the first clause, and also that our cultural frames suggest that city councils are perhaps more likely than demonstrators to fear violence and that demonstrators might be more likely to advocate violence.

Winograd schema

In an attempt to get the field of NLP to focus more on methods involving world knowledge and common-sense reasoning, Levesque (2011) proposed a challenge task called the **Winograd Schema Challenge**.[8] The problems in the challenge task are coreference problems designed to be easily disambiguated by the human reader, but hopefully not solvable by simple techniques such as selectional restrictions, or other basic word association methods.

The problems are framed as a pair of statements that differ in a single word or phrase, and a coreference question:

(23.73) The trophy didn't fit into the suitcase because it was too **large**.
Question: What was too **large**? Answer: The trophy

[8] Levesque's call was quickly followed up by Levesque et al. (2012) and Rahman and Ng (2012), a competition at the IJCAI conference (Davis et al., 2017), and a natural language inference version of the problem called WNLI (Wang et al., 2018a).

(23.74) The trophy didn't fit into the suitcase because it was too **small**.
Question: What was too **small**? Answer: The suitcase

The problems have the following characteristics:

1. The problems each have two parties
2. A pronoun preferentially refers to one of the parties, but could grammatically also refer to the other
3. A question asks which party the pronoun refers to
4. If one word in the question is changed, the human-preferred answer changes to the other party

The kind of world knowledge that might be needed to solve the problems can vary. In the trophy/suitcase example, it is knowledge about the physical world; that a bigger object cannot fit into a smaller object. In the original Winograd sentence, it is stereotypes about social actors like politicians and protesters. In examples like the following, it is knowledge about human actions like turn-taking or thanking.

(23.75) Bill passed the gameboy to John because his turn was [over/next]. Whose turn was [over/next]? Answers: Bill/John

(23.76) Joan made sure to thank Susan for all the help she had [given/received]. Who had [given/received] help? Answers: Susan/Joan.

Although the Winograd Schema was designed to require common-sense reasoning, a large percentage of the original set of problems can be solved by pretrained language models, fine-tuned on Winograd Schema sentences (Kocijan et al., 2019). Large pretrained language models encode an enormous amount of world or common-sense knowledge! The current trend is therefore to propose new datasets with increasingly difficult Winograd-like coreference resolution problems like KNOWREF (Emami et al., 2019), with examples like:

(23.77) Marcus is undoubtedly faster than Jarrett right now but in [his] prime the gap wasn't all that big.

In the end, it seems likely that some combination of language modeling and knowledge will prove fruitful; indeed, it seems that knowledge-based models overfit less to lexical idiosyncracies in Winograd Schema training sets (Trichelair et al., 2018),

23.10 Gender Bias in Coreference

As with other aspects of language processing, coreference models exhibit gender and other biases (Zhao et al. 2018a, Rudinger et al. 2018, Webster et al. 2018). For example the WinoBias dataset (Zhao et al., 2018a) uses a variant of the Winograd Schema paradigm to test the extent to which coreference algorithms are biased toward linking gendered pronouns with antecedents consistent with cultural stereotypes. As we summarized in Chapter 5, embeddings replicate societal biases in their training test, such as associating men with historically sterotypical male occupations like doctors, and women with stereotypical female occupations like secretaries (Caliskan et al. 2017, Garg et al. 2018).

A WinoBias sentence contain two mentions corresponding to stereotypically-male and stereotypically-female occupations and a gendered pronoun that must be linked to one of them. The sentence cannot be disambiguated by the gender of the pronoun, but a biased model might be distracted by this cue. Here is an example sentence:

(23.78) The secretary called the physician$_i$ and told him$_i$ about a new patient [pro-stereotypical]

(23.79) The secretary called the physician$_i$ and told her$_i$ about a new patient [anti-stereotypical]

Zhao et al. (2018a) consider a coreference system to be biased if it is more accurate at linking pronouns consistent with gender stereotypical occupations (e.g., *him* with *physician* in (23.78)) than linking pronouns inconsistent with gender-stereotypical occupations (e.g., *her* with *physician* in (23.79)). They show that coreference systems of all architectures (rule-based, feature-based machine learned, and end-to-end-neural) all show significant bias, performing on average 21 F_1 points worse in the anti-stereotypical cases.

One possible source of this bias is that female entities are significantly underrepresented in the OntoNotes dataset, used to train most coreference systems. Zhao et al. (2018a) propose a way to overcome this bias: they generate a second gender-swapped dataset in which all male entities in OntoNotes are replaced with female ones and vice versa, and retrain coreference systems on the combined original and swapped OntoNotes data, also using debiased GloVE embeddings (Bolukbasi et al., 2016). The resulting coreference systems no longer exhibit bias on the WinoBias dataset, without significantly impacting OntoNotes coreference accuracy. In a follow-up paper, Zhao et al. (2019) show that the same biases exist in ELMo contextualized word vector representations and coref systems that use them. They showed that retraining ELMo with data augmentation again reduces or removes bias in coreference systems on WinoBias.

Webster et al. (2018) introduces another dataset, GAP, and the task of Gendered Pronoun Resolution as a tool for developing improved coreference algorithms for gendered pronouns. GAP is a gender-balanced labeled corpus of 4,454 sentences with gendered ambiguous pronouns (by contrast, only 20% of the gendered pronouns in the English OntoNotes training data are feminine). The examples were created by drawing on naturally occurring sentences from Wikipedia pages to create hard to resolve cases with two named entities of the same gender and an ambiguous pronoun that may refer to either person (or neither), like the following:

(23.80) In May, Fujisawa joined Mari Motohashi's rink as the team's skip, moving back from Karuizawa to Kitami where **she** had spent her junior days.

Webster et al. (2018) show that modern coreference algorithms perform significantly worse on resolving feminine pronouns than masculine pronouns in GAP. Kurita et al. (2019) shows that a system based on BERT contextualized word representations shows similar bias.

23.11 Summary

This chapter introduced the task of **coreference resolution**.

- This is the task of linking together **mentions** in text which **corefer**, i.e. refer to the same **discourse entity** in the **discourse model**, resulting in a set of coreference **chains** (also called **clusters** or **entities**).
- Mentions can be **definite NPs** or **indefinite NPs**, **pronouns** (including **zero pronouns**) or **names**.

- The surface form of an entity mention is linked to its **information status** (**new**, **old**, or **inferrable**), and how **accessible** or **salient** the entity is.
- Some NPs are not referring expressions, such as pleonastic *it* in *It is raining*.
- Many corpora have human-labeled coreference annotations that can be used for supervised learning, including **OntoNotes** for English, Chinese, and Arabic, ARRAU for English, and **AnCora** for Spanish and Catalan.
- Mention detection can start with all nouns and named entities and then use **anaphoricity classifiers** or **referentiality classifiers** to filter out non-mentions.
- Three common architectures for coreference are **mention-pair**, **mention-rank**, and **entity-based**, each of which can make use of feature-based or neural classifiers.
- Modern coreference systems tend to be end-to-end, performing mention detection and coreference in a single end-to-end architecture.
- Algorithms learn representations for text spans and heads, and learn to compare anaphor spans with candidate antecedent spans.
- Entity linking is the task of associating a mention in text with the representation of some real-world entity in an ontology .
- Coreference systems are evaluated by comparing with gold entity labels using precision/recall metrics like **MUC**, **B**^{3}, **CEAF**, **BLANC**, or **LEA**.
- The **Winograd Schema Challenge** problems are difficult coreference problems that seem to require world knowledge or sophisticated reasoning to solve.
- Coreference systems exhibit **gender bias** which can be evaluated using datasets like Winobias and GAP.

Historical Notes

Coreference has been part of natural language processing since the 1970s (Woods et al. 1972, Winograd 1972). The discourse model and the entity-centric foundation of coreference was formulated by Karttunen (1969) (at the 3rd COLING conference), playing a role also in linguistic semantics (Heim 1982, Kamp 1981). But it was Bonnie Webber's 1978 dissertation and following work (Webber 1983) that explored the model's computational aspects, providing fundamental insights into how entities are represented in the discourse model and the ways in which they can license subsequent reference. Many of the examples she provided continue to challenge theories of reference to this day.

Hobbs algorithm

The **Hobbs algorithm**[9] is a tree-search algorithm that was the first in a long series of syntax-based methods for identifying reference robustly in naturally occurring text. The input to the Hobbs algorithm is a pronoun to be resolved, together with a syntactic (constituency) parse of the sentences up to and including the current sentence. The details of the algorithm depend on the grammar used, but can be understood from a simplified version due to Kehler et al. (2004) that just searches through the list of NPs in the current and prior sentences. This simplified Hobbs algorithm searches NPs in the following order: "(i) in the current sentence from right-to-left, starting with the first NP to the left of the pronoun, (ii) in the previous sentence from left-to-right, (iii) in two sentences prior from left-to-right, and (iv) in

[9] The simpler of two algorithms presented originally in Hobbs (1978).

the current sentence from left-to-right, starting with the first noun group to the right of the pronoun (for cataphora). The first noun group that agrees with the pronoun with respect to number, gender, and person is chosen as the antecedent" (Kehler et al., 2004).

Lappin and Leass (1994) was an influential entity-based system that used weights to combine syntactic and other features, extended soon after by Kennedy and Boguraev (1996) whose system avoids the need for full syntactic parses.

Approximately contemporaneously centering (Grosz et al., 1995) was applied to pronominal anaphora resolution by Brennan et al. (1987), and a wide variety of work followed focused on centering's use in coreference (Kameyama 1986, Di Eugenio 1990, Walker et al. 1994, Di Eugenio 1996, Strube and Hahn 1996, Kehler 1997a, Tetreault 2001, Iida et al. 2003). Kehler and Rohde (2013) show how centering can be integrated with coherence-driven theories of pronoun interpretation. See Chapter 24 for the use of centering in measuring discourse coherence.

Coreference competitions as part of the US DARPA-sponsored MUC conferences provided early labeled coreference datasets (the 1995 MUC-6 and 1998 MUC-7 corpora), and set the tone for much later work, choosing to focus exclusively on the simplest cases of *identity coreference* (ignoring difficult cases like bridging, metonymy, and part-whole) and drawing the community toward supervised machine learning and metrics like the MUC metric (Vilain et al., 1995). The later ACE evaluations produced labeled coreference corpora in English, Chinese, and Arabic that were widely used for model training and evaluation.

This DARPA work influenced the community toward supervised learning beginning in the mid-90s (Connolly et al. 1994, Aone and Bennett 1995, McCarthy and Lehnert 1995). Soon et al. (2001) laid out a set of basic features, extended by Ng and Cardie (2002b), and a series of machine learning models followed over the next 15 years. These often focused separately on pronominal anaphora resolution (Kehler et al. 2004, Bergsma and Lin 2006), full NP coreference (Cardie and Wagstaff 1999, Ng and Cardie 2002b, Ng 2005a) and definite NP reference (Poesio and Vieira 1998, Vieira and Poesio 2000), as well as separate anaphoricity detection (Bean and Riloff 1999, Bean and Riloff 2004, Ng and Cardie 2002a, Ng 2004), or singleton detection (de Marneffe et al., 2015).

The move from mention-pair to mention-ranking approaches was pioneered by Yang et al. (2003) and Iida et al. (2003) who proposed pairwise ranking methods, then extended by Denis and Baldridge (2008) who proposed to do ranking via a softmax over all prior mentions. The idea of doing mention detection, anaphoricity, and coreference jointly in a single end-to-end model grew out of the early proposal of Ng (2005b) to use a dummy antecedent for mention-ranking, allowing 'non-referential' to be a choice for coreference classifiers, Denis and Baldridge's 2007 joint system combining anaphoricity classifier probabilities with coreference probabilities, the Denis and Baldridge (2008) ranking model, and the Rahman and Ng (2009) proposal to train the two models jointly with a single objective.

Simple rule-based systems for coreference returned to prominence in the 2010s, partly because of their ability to encode entity-based features in a high-precision way (Zhou et al. 2004b, Haghighi and Klein 2009, Raghunathan et al. 2010, Lee et al. 2011, Lee et al. 2013, Hajishirzi et al. 2013) but in the end they suffered from an inability to deal with the semantics necessary to correctly handle cases of common noun coreference.

A return to supervised learning led to a number of advances in mention-ranking models which were also extended into neural architectures, for example using re-

inforcement learning to directly optimize coreference evaluation models Clark and Manning (2016a), doing end-to-end coreference all the way from span extraction (Lee et al. 2017b, Zhang et al. 2018). Neural models also were designed to take advantage of global entity-level information (Clark and Manning 2016b, Wiseman et al. 2016, Lee et al. 2018).

Coreference is also related to the task of **entity linking** discussed in Chapter 11. Coreference can help entity linking by giving more possible surface forms to help link to the right Wikipedia page, and conversely entity linking can help improve coreference resolution. Consider this example from Hajishirzi et al. (2013):

(23.81) [Michael Eisner]$_1$ and [Donald Tsang]$_2$ announced the grand opening of [[Hong Kong]$_3$ Disneyland]$_4$ yesterday. [Eisner]$_1$ thanked [the President]$_2$ and welcomed [fans]$_5$ to [the park]$_4$.

Integrating entity linking into coreference can help draw encyclopedic knowledge (like the fact that *Donald Tsang* is a president) to help disambiguate the mention *the President*. Ponzetto and Strube (2006) 2007 and Ratinov and Roth (2012) showed that such attributes extracted from Wikipedia pages could be used to build richer models of entity mentions in coreference. More recent research shows how to do linking and coreference jointly (Hajishirzi et al. 2013, Zheng et al. 2013) or even jointly with named entity tagging as well (Durrett and Klein 2014).

The coreference task as we introduced it involves a simplifying assumption that the relationship between an anaphor and its antecedent is one of *identity*: the two coreferring mentions refer to the identical discourse referent. In real texts, the relationship can be more complex, where different aspects of a discourse referent can be neutralized or refocused. For example (23.82) (Recasens et al., 2011) shows an example of **metonymy**, in which the capital city *Washington* is used metonymically to refer to the US. (23.83-23.84) show other examples (Recasens et al., 2011):

metonymy

(23.82) a strict interpretation of a policy requires **The U.S.** to notify foreign dictators of certain coup plots ... **Washington** rejected the bid ...

(23.83) I once crossed that border into Ashgh-Abad on Nowruz, the Persian New Year. In the South, everyone was celebrating **New Year**; to the North, **it** was a regular day.

(23.84) In France, **the president** is elected for a term of seven years, while in the United States **he** is elected for a term of four years.

For further linguistic discussions of these complications of coreference see Pustejovsky (1991), van Deemter and Kibble (2000), Poesio et al. (2006), Fauconnier and Turner (2008), Versley (2008), and Barker (2010).

Ng (2017) offers a useful compact history of machine learning models in coreference resolution. There are three excellent book-length surveys of anaphora/coreference resolution, covering different time periods: Hirst (1981) (early work until about 1981), Mitkov (2002) (1986-2001), and Poesio et al. (2016) (2001-2015).

Andy Kehler wrote the Discourse chapter for the 2000 first edition of this textbook, which we used as the starting point for the second-edition chapter, and there are some remnants of Andy's lovely prose still in this third-edition coreference chapter.

Exercises

CHAPTER

24 Discourse Coherence

> And even in our wildest and most wandering reveries, nay in our very dreams, we shall find, if we reflect, that the imagination ran not altogether at adventures, but that there was still a connection upheld among the different ideas, which succeeded each other. Were the loosest and freest conversation to be transcribed, there would immediately be transcribed, there would immediately be observed something which connected it in all its transitions.
>
> David Hume, *An enquiry concerning human understanding*, 1748

Orson Welles' movie *Citizen Kane* was groundbreaking in many ways, perhaps most notably in its structure. The story of the life of fictional media magnate Charles Foster Kane, the movie does not proceed in chronological order through Kane's life. Instead, the film begins with Kane's death (famously murmuring *"Rosebud"*) and is structured around flashbacks to his life inserted among scenes of a reporter investigating his death. The novel idea that the structure of a movie does not have to linearly follow the structure of the real timeline made apparent for 20th century cinematography the infinite possibilities and impact of different kinds of coherent narrative structures.

But coherent structure is not just a fact about movies or works of art. Like movies, language does not normally consist of isolated, unrelated sentences, but instead of collocated, structured, **coherent** groups of sentences. We refer to such a coherent structured group of sentences as a **discourse**, and we use the word **coherence** to refer to the relationship between sentences that makes real discourses different than just random assemblages of sentences. The chapter you are now reading is an example of a discourse, as is a news article, a conversation, a thread on social media, a Wikipedia page, and your favorite novel.

discourse
coherence

What makes a discourse coherent? If you created a text by taking random sentences each from many different sources and pasted them together, would that be a coherent discourse? Almost certainly not. Real discourses exhibit both **local coherence** and **global coherence**. Let's consider three ways in which real discourses are locally coherent;

local
global

First, sentences or clauses in real discourses are related to nearby sentences in systematic ways. Consider this example from Hobbs (1979):

(24.1) John took a train from Paris to Istanbul. He likes spinach.

This sequence is incoherent because it is unclear to a reader why the second sentence follows the first; what does liking spinach have to do with train trips? In fact, a reader might go to some effort to try to figure out how the discourse could be coherent; perhaps there is a French spinach shortage? The very fact that hearers try to identify such connections suggests that human discourse comprehension involves the need to establish this kind of coherence.

By contrast, in the following coherent example:

(24.2) Jane took a train from Paris to Istanbul. She had to attend a conference.

the second sentence gives a REASON for Jane's action in the first sentence. Structured relationships like REASON that hold between text units are called **coherence relations**, and coherent discourses are structured by many such coherence relations. Coherence relations are introduced in Section 24.1.

coherence relations

A second way a discourse can be locally coherent is by virtue of being "about" someone or something. In a coherent discourse some entities are **salient**, and the discourse focuses on them and doesn't go back and forth between multiple entities. This is called **entity-based coherence**. Consider the following incoherent passage, in which the salient entity seems to wildly swing from John to Jenny to the piano store to the living room, back to Jenny, then the piano again:

(24.3) John wanted to buy a piano for his living room.
Jenny also wanted to buy a piano.
He went to the piano store.
It was nearby.
The living room was on the second floor.
She didn't find anything she liked.
The piano he bought was hard to get up to that floor.

Centering Theory

Entity-based coherence models measure this kind of coherence by tracking salient entities across a discourse. For example **Centering Theory** (Grosz et al., 1995), the most influential theory of entity-based coherence, keeps track of which entities in the discourse model are salient at any point (salient entities are more likely to be pronominalized or to appear in prominent syntactic positions like subject or object). In Centering Theory, transitions between sentences that maintain the same salient entity are considered more coherent than ones that repeatedly shift between entities.

entity grid

The **entity grid** model of coherence (Barzilay and Lapata, 2008) is a commonly used model that realizes some of the intuitions of the Centering Theory framework. Entity-based coherence is introduced in Section 24.3.

topically coherent

Finally, discourses can be locally coherent by being **topically coherent**: nearby sentences are generally about the same topic and use the same or similar vocabulary to discuss these topics. Because topically coherent discourses draw from a single semantic field or topic, they tend to exhibit the surface property known as

lexical cohesion

lexical cohesion (Halliday and Hasan, 1976): the sharing of identical or semantically related words in nearby sentences. For example, the fact that the words *house*, *chimney*, *garret*, *closet*, and *window*— all of which belong to the same semantic field— appear in the two sentences in (24.4), or that they share the identical word *shingled*, is a cue that the two are tied together as a discourse:

(24.4) Before winter I built a **chimney**, and shingled the sides of my **house**...
I have thus a tight shingled and plastered **house**... with a **garret** and a **closet**, a large **window** on each side....

In addition to the local coherence between adjacent or nearby sentences, discourses also exhibit **global coherence**. Many genres of text are associated with particular conventional discourse structures. Academic articles might have sections describing the Methodology or Results. Stories might follow conventional plotlines or motifs. Persuasive essays have a particular claim they are trying to argue for, and an essay might express this claim together with a structured set of premises that support the argument and demolish potential counterarguments. We'll introduce versions of each of these kinds of global coherence.

Why do we care about the local or global coherence of a discourse? Since coherence is a property of a well-written text, coherence detection plays a part in any

task that requires measuring the **quality** of a text. For example coherence can help in pedagogical tasks like essay grading or essay quality measurement that are trying to grade how well-written a human essay is (Somasundaran et al. 2014, Feng et al. 2014, Lai and Tetreault 2018). Coherence can also help for summarization; knowing the coherence relationship between sentences can help know how to select information from them. Finally, detecting incoherent text may even play a role in mental health tasks like measuring symptoms of schizophrenia or other kinds of disordered language (Ditman and Kuperberg 2010, Elvevåg et al. 2007, Bedi et al. 2015, Iter et al. 2018).

24.1 Coherence Relations

Recall from the introduction the difference between passages (24.5) and (24.6).

(24.5) Jane took a train from Paris to Istanbul. She likes spinach.

(24.6) Jane took a train from Paris to Istanbul. She had to attend a conference.

The reason (24.6) is more coherent is that the reader can form a connection between the two sentences, in which the second sentence provides a potential REASON for the first sentences. This link is harder to form for (24.5). These connections between text spans in a discourse can be specified as a set of **coherence relations**. The next two sections describe two commonly used models of coherence relations and associated corpora: Rhetorical Structure Theory (RST), and the Penn Discourse TreeBank (PDTB).

coherence relation

24.1.1 Rhetorical Structure Theory

The most commonly used model of discourse organization is **Rhetorical Structure Theory** (**RST**) (Mann and Thompson, 1987). In RST relations are defined between two spans of text, generally a **nucleus** and a **satellite**. The nucleus is the unit that is more central to the writer's purpose and that is interpretable independently; the satellite is less central and generally is only interpretable with respect to the nucleus. Some symmetric relations, however, hold between two nuclei.

RST
nucleus
satellite

Below are a few examples of RST coherence relations, with definitions adapted from the RST Treebank Manual (Carlson and Marcu, 2001).

Reason: The nucleus is an action carried out by an animate agent and the satellite is the reason for the nucleus.

(24.7) [$_{\text{NUC}}$ Jane took a train from Paris to Istanbul.] [$_{\text{SAT}}$ She had to attend a conference.]

Elaboration: The satellite gives additional information or detail about the situation presented in the nucleus.

(24.8) [$_{\text{NUC}}$ Dorothy was from Kansas.] [$_{\text{SAT}}$ She lived in the midst of the great Kansas prairies.]

Evidence: The satellite gives additional information or detail about the situation presented in the nucleus. The information is presented with the goal of convince the reader to accept the information presented in the nucleus.

(24.9) [$_{\text{NUC}}$ Kevin must be here.] [$_{\text{SAT}}$ His car is parked outside.]

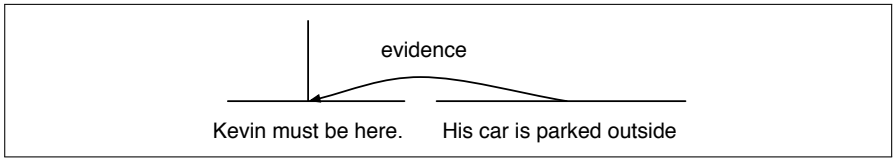
Attribution: The satellite gives the source of attribution for an instance of reported speech in the nucleus.

(24.10) [$_{SAT}$ Analysts estimated] [$_{NUC}$ that sales at U.S. stores declined in the quarter, too]

List: In this multinuclear relation, a series of nuclei is given, without contrast or explicit comparison:

(24.11) [$_{NUC}$ Billy Bones was the mate;] [$_{NUC}$ Long John, he was quartermaster]

RST relations are traditionally represented graphically; the asymmetric Nucleus-Satellite relation is represented with an arrow from the satellite to the nucleus:



We can also talk about the coherence of a larger text by considering the hierarchical structure between coherence relations. Figure 24.1 shows the rhetorical structure of a paragraph from Marcu (2000a) for the text in (24.12) from the *Scientific American* magazine.

(24.12) With its distant orbit–50 percent farther from the sun than Earth–and slim atmospheric blanket, Mars experiences frigid weather conditions. Surface temperatures typically average about -60 degrees Celsius (-76 degrees Fahrenheit) at the equator and can dip to -123 degrees C near the poles. Only the midday sun at tropical latitudes is warm enough to thaw ice on occasion, but any liquid water formed in this way would evaporate almost instantly because of the low atmospheric pressure.

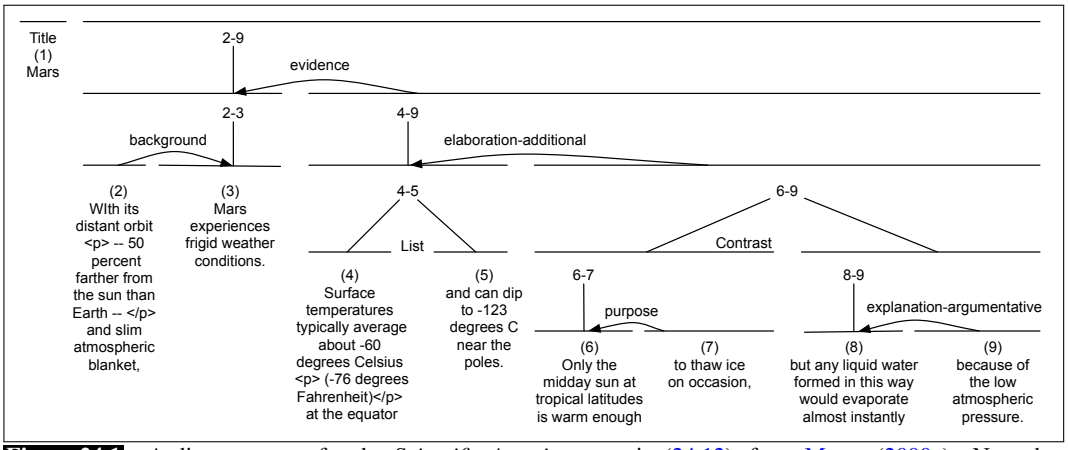


Figure 24.1 A discourse tree for the *Scientific American* text in (24.12), from Marcu (2000a). Note that asymmetric relations are represented with a curved arrow from the satellite to the nucleus.

EDU

The leaves in the Fig. 24.1 tree correspond to text spans of a sentence, clause or phrase that are called **elementary discourse units** or **EDU**s in RST; these units can also be referred to as **discourse segments**. Because these units may correspond to arbitrary spans of text, determining the boundaries of an EDU is an important task for extracting coherence relations. Roughly speaking, one can think of discourse

segments as being analogous to constituents in sentence syntax, and indeed as we'll see in Section 24.2 we generally draw on parsing algorithms to infer discourse structure.

There are corpora for many discourse coherence models; the RST Discourse TreeBank (Carlson et al., 2001) is the largest available discourse corpus. It consists of 385 English language documents selected from the Penn Treebank, with full RST parses for each one, using a large set of 78 distinct relations, grouped into 16 classes. RST treebanks exist also for Spanish, German, Basque, Dutch and Brazilian Portuguese (Braud et al., 2017).

Now that we've seen examples of coherence, we can see more clearly how a coherence relation can play a role in summarization or information extraction. For example, the nuclei of a text presumably express more important information than the satellites, which might be dropped in a summary.

24.1.2 Penn Discourse TreeBank (PDTB)

PDTB

The **Penn Discourse TreeBank** (**PDTB**) is a second commonly used dataset that embodies another model of coherence relations (Miltsakaki et al. 2004, Prasad et al. 2008, Prasad et al. 2014). PDTB labeling is *lexically grounded*. Instead of asking annotators to directly tag the coherence relation between text spans, they were given a list of **discourse connectives**, words that signal discourse relations, like *because*, *although*, *when*, *since*, or *as a result*. In a part of a text where these words marked a coherence relation between two text spans, the connective and the spans were then annotated, as in Fig. 24.13, where the phrase as a result signals a causal relationship between what PDTB calls *Arg1* (the first two sentences, here in italics) and **Arg2** (the third sentence, here in bold).

discourse connectives

(24.13) *Jewelry displays in department stores were often cluttered and uninspired. And the merchandise was, well, fake.* **As a result, marketers of faux gems steadily lost space in department stores to more fashionable rivals—cosmetics makers.**

(24.14) *In July, the Environmental Protection Agency imposed a gradual ban on virtually all uses of asbestos.* (implicit=as a result) **By 1997, almost all remaining uses of cancer-causing asbestos will be outlawed.**

Not all coherence relations are marked by an explicit discourse connective, and so the PDTB also annotates pairs of neighboring sentences with no explicit signal, like (24.14). The annotator first chooses the word or phrase that could have been its signal (in this case **as a result**), and then labels its sense. For example for the ambiguous discourse connective *since* annotators marked whether it is using a CAUSAL or a TEMPORAL sense.

The final dataset contains roughly 18,000 explicit relations and 16,000 implicit relations. Fig. 24.2 shows examples from each of the 4 major semantic classes, while Fig. 24.3 shows the full tagset.

Unlike the RST Discourse Treebank, which integrates these pairwise coherence relations into a global tree structure spanning an entire discourse, the PDTB does not annotate anything above the span-pair level, making no commitment with respect to higher-level discourse structure.

There are also treebanks using similar methods for other languages; (24.15) shows an example from the Chinese Discourse TreeBank (Zhou and Xue, 2015). Because Chinese has a smaller percentage of explicit discourse connectives than English (only 22% of all discourse relations are marked with explicit connectives,

Class	Type	Example
TEMPORAL	SYNCHRONOUS	The parishioners of St. Michael and All Angels stop to chat at the church door, as members here always have. (Implicit while) In the tower, five men and women pull rhythmically on ropes attached to the same five bells that first sounded here in 1614.
CONTINGENCY	REASON	Also unlike Mr. Ruder, Mr. Breeden appears to be in a position to get somewhere with his agenda. (implicit=because) **As a former White House aide who worked closely with Congress, he is savvy in the ways of Washington.**
COMPARISON	CONTRAST	The U.S. wants the removal of what it perceives as barriers to investment; Japan denies there are real barriers.
EXPANSION	CONJUNCTION	Not only do the actors stand outside their characters and make it clear they are at odds with them, but they often literally stand on their heads.

Figure 24.2 The four high-level semantic distinctions in the PDTB sense hierarchy

Temporal	**Comparison**
• Asynchronous	• Contrast (Juxtaposition, Opposition)
• Synchronous (Precedence, Succession)	•*Pragmatic Contrast (Juxtaposition, Opposition)*
	• Concession (Expectation, Contra-expectation)
	• *Pragmatic Concession*
Contingency	**Expansion**
• Cause (Reason, Result)	• *Exception*
• Pragmatic Cause (Justification)	• Instantiation
• *Condition (Hypothetical, General, Unreal Present/Past, Factual Present/Past)*	• Restatement (Specification, Equivalence, Generalization)
• *Pragmatic Condition (Relevance, Implicit Assertion)*	• Alternative (Conjunction, Disjunction, Chosen Alternative)
	• List

Figure 24.3 The PDTB sense hierarchy. There are four top-level classes, 16 types, and 23 subtypes (not all types have subtypes). 11 of the 16 types are commonly used for implicit argument classification; the 5 types in italics are too rare in implicit labeling to be used.

compared to 47% in English), annotators labeled this corpus by directly mapping pairs of sentences to 11 sense tags, without starting with a lexical discourse connector.

(24.15) [**Conn** 为] [**Arg2** 推动图们江地区开发]，[Arg1 韩国捐款一百万美元设立了图们江发展基金]
"[In order to] [**Arg2** promote the development of the Tumen River region], [**Arg1** South Korea donated one million dollars to establish the Tumen River Development Fund]."

These discourse treebanks have been used for shared tasks on multilingual discourse parsing (Xue et al., 2016).

24.2 Discourse Structure Parsing

discourse parsing

Given a sequence of sentences, how can we automatically determine the coherence relations between them? This task is often called **discourse parsing** (even though for PDTB we are only assigning labels to leaf spans and not building a full parse

tree as we do for RST).

24.2.1 EDU segmentation for RST parsing

RST parsing is generally done in two stages. The first stage, **EDU segmentation**, extracts the start and end of each EDU. The output of this stage would be a labeling like the following:

(24.16) [Mr. Rambo says]$_{e1}$ [that a 3.2-acre property]$_{e2}$ [overlooking the San Fernando Valley]$_{e3}$ [is priced at $4 million]$_{e4}$ [because the late actor Erroll Flynn once lived there.]$_{e5}$

Since EDUs roughly correspond to clauses, early models of EDU segmentation first ran a syntactic parser, and then post-processed the output. Modern systems generally use neural sequence models supervised by the gold EDU segmentation in datasets like the RST Discourse Treebank. Fig. 24.4 shows an example architecture simplified from the algorithm of Lukasik et al. (2020) that predicts for each token whether or not it is a break. Here the input sentence is passed through an encoder and then passed through a linear layer and a softmax to produce a sequence of 0s and 1, where 1 indicates the start of an EDU.

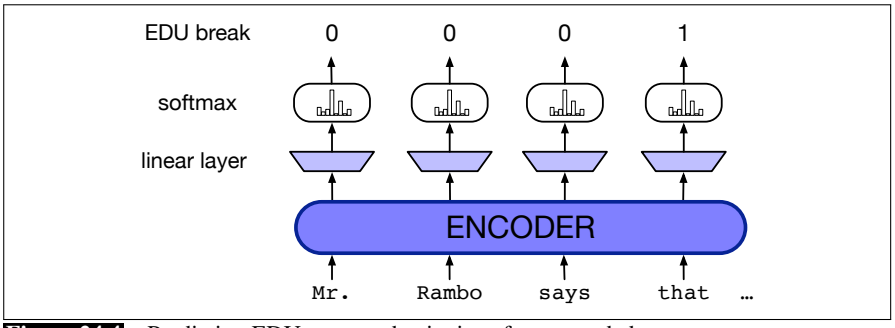


Figure 24.4 Predicting EDU segment beginnings from encoded text.

24.2.2 RST parsing

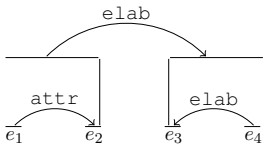
Tools for building RST coherence structure for a discourse have long been based on syntactic parsing algorithms like shift-reduce parsing (Marcu, 1999). Many modern RST parsers since Ji and Eisenstein (2014) draw on the neural syntactic parsers we saw in Chapter 19, using representation learning to build representations for each span, and training a parser to choose the correct shift and reduce actions based on the gold parses in the training set.

We'll describe the shift-reduce parser of Yu et al. (2018). The parser state consists of a stack and a queue, and produces this structure by taking a series of actions on the states. Actions include:

- **shift**: pushes the first EDU in the queue onto the stack creating a single-node subtree.
- **reduce**(l,d): merges the top two subtrees on the stack, where l is the coherence relation label, and d is the nuclearity direction, $d \in \{NN, NS, SN\}$.

As well as the **pop root** operation, to remove the final tree from the stack.

Fig. 24.6 shows the actions the parser takes to build the structure in Fig. 24.5.



e_1: American Telephone & Telegraph Co. said it
e_2: will lay off 75 to 85 technicians here , effective Nov. 1.
e_3: The workers install , maintain and repair its private branch exchanges,
e_4: which are large intracompany telephone networks.

Figure 24.5 Example RST discourse tree, showing four EDUs. Figure from Yu et al. (2018).

Step	Stack	Queue	Action	Relation
1	$\varnothing$	e_1, e_2, e_3, e_4	SH	$\varnothing$
2	e_1	e_2, e_3, e_4	SH	$\varnothing$
3	e_1, e_2	e_3, e_4	RD (attr, SN)	$\varnothing$
4	$e_{1:2}$	e_3, e_4	SH	$\widehat{e_1 \mathbf{e_2}}$
5	$e_{1:2}, e_3$	e_4	SH	$\widehat{e_1 \mathbf{e_2}}$
6	$e_{1:2}, e_3, e_4$	$\varnothing$	RD (elab, NS)	$\widehat{e_1 \mathbf{e_2}}$
7	$e_{1:2}, e_{3:4}$	$\varnothing$	RD (elab, SN)	$\widehat{e_1 \mathbf{e_2}}, \widehat{\mathbf{e_3} e_4}$
8	$e_{1:4}$	$\varnothing$	PR	$\widehat{e_1 \mathbf{e_2}}, \widehat{\mathbf{e_3} e_4}, \widehat{e_{1:2} \mathbf{e_{3:4}}}$

Figure 24.6 Parsing the example of Fig. 24.5 using a shift-reduce parser. Figure from Yu et al. (2018).

The Yu et al. (2018) uses an encoder-decoder architecture, where the encoder represents the input span of words and EDUs using a hierarchical biLSTM. The first biLSTM layer represents the words inside an EDU, and the second represents the EDU sequence. Given an input sentence $w_1, w_2, ..., w_m$, the words can be represented as usual (by static embeddings, combinations with character embeddings or tags, or contextual embeddings) resulting in an input word representation sequence $\mathbf{x}^w_1, \mathbf{x}^w_2, ..., \mathbf{x}^w_m$. The result of the word-level biLSTM is then a sequence of $\mathbf{h^w}$ values:

$$\mathbf{h}^w_1, \mathbf{h}^w_2, ..., \mathbf{h}^w_m = \text{biLSTM}(\mathbf{x}^w_1, \mathbf{x}^w_2, ..., \mathbf{x}^w_m) \tag{24.17}$$

An EDU of span $w_s, w_{s+1}, ..., w_t$ then has biLSTM output representation $\mathbf{h}^w_s, \mathbf{h}^w_{s+1}, ..., \mathbf{h}^w_t$, and is represented by average pooling:

$$\mathbf{x}^e = \frac{1}{t-s+1} \sum_{k=s}^{t} \mathbf{h}^w_k \tag{24.18}$$

The second layer uses this input to compute a final representation of the sequence of EDU representations $\mathbf{h}^e$:

$$\mathbf{h}^e_1, \mathbf{h}^e_2, ..., \mathbf{h}^e_n = \text{biLSTM}(\mathbf{x}^e_1, \mathbf{x}^e_2, ..., \mathbf{x}^e_n) \tag{24.19}$$

The decoder is then a feedforward network $\mathbf{W}$ that outputs an action o based on a concatenation of the top three subtrees on the stack (s_o, s_1, s_2) plus the first EDU in the queue (q_0):

$$\mathbf{o} = \mathbf{W}(\mathbf{h}^{\text{t}}_{\text{s0}}, \mathbf{h}^{\text{t}}_{\text{s1}}, \mathbf{h}^{\text{t}}_{\text{s2}}, \mathbf{h}^{\text{e}}_{\text{q0}}) \tag{24.20}$$

where the representation of the EDU on the queue $\mathbf{h}^{\text{e}}_{\text{q0}}$ comes directly from the encoder, and the three hidden vectors representing partial trees are computed by average pooling over the encoder output for the EDUs in those trees:

$$\mathbf{h}^{\text{t}}_{\text{s}} = \frac{1}{j-i+1} \sum_{k=i}^{j} \mathbf{h}^e_k \tag{24.21}$$

Training first maps each RST gold parse tree into a sequence of oracle actions, and then uses the standard cross-entropy loss (with l_2 regularization) to train the system to take such actions. Give a state S and oracle action a, we first compute the decoder output using Eq. 24.20, apply a softmax to get probabilities:

$$p_a = \frac{\exp(\mathbf{o}_a)}{\sum_{a' \in A} \exp(\mathbf{o}_{a'})} \tag{24.22}$$

and then computing the cross-entropy loss:

$$L_{CE}() = -\log(p_a) + \frac{\lambda}{2}||\Theta||^2 \tag{24.23}$$

RST discourse parsers are evaluated on the test section of the RST Discourse Treebank, either with gold EDUs or end-to-end, using the RST-Pareval metrics (Marcu, 2000b). It is standard to first transform the gold RST trees into right-branching binary trees, and to report four metrics: trees with no labels (S for Span), labeled with nuclei (N), with relations (R), or both (F for Full), for each metric computing micro-averaged F_1 over all spans from all documents (Marcu 2000b, Morey et al. 2017).

24.2.3 PDTB discourse parsing

shallow discourse parsing

PDTB discourse parsing, the task of detecting PDTB coherence relations between spans, is sometimes called **shallow discourse parsing** because the task just involves flat relationships between text spans, rather than the full trees of RST parsing.

The set of four subtasks for PDTB discourse parsing was laid out by Lin et al. (2014) in the first complete system, with separate tasks for explicit (tasks 1-3) and implicit (task 4) connectives:

1. Find the discourse connectives (disambiguating them from non-discourse uses)
2. Find the two spans for each connective
3. Label the relationship between these spans
4. Assign a relation between every adjacent pair of sentences

Many systems have been proposed for Task 4: taking a pair of adjacent sentences as input and assign a coherence relation sense label as output. The setup often follows Lin et al. (2009) in assuming gold sentence span boundaries and assigning each adjacent span one of the 11 second-level PDTB tags or none (removing the 5 very rare tags of the 16 shown in italics in Fig. 24.3).

A simple but very strong algorithm for Task 4 is to represent each of the two spans by BERT embeddings and take the last layer hidden state corresponding to the position of the `[CLS]` token, pass this through a single layer tanh feedforward network and then a softmax for sense classification (Nie et al., 2019).

Each of the other tasks also have been addressed. Task 1 is to disambiguating discourse connectives from their non-discourse use. For example as Pitler and Nenkova (2009) point out, the word *and* is a discourse connective linking the two clauses by an elaboration/expansion relation in (24.24) while it's a non-discourse NP conjunction in (24.25):

(24.24) Selling picked up as previous buyers bailed out of their positions and aggressive short sellers—anticipating further declines—moved in.

(24.25) My favorite colors are blue and green.

Similarly, *once* is a discourse connective indicating a temporal relation in (24.26), but simply a non-discourse adverb meaning 'formerly' and modifying *used* in (24.27):

(24.26) The asbestos fiber, crocidolite, is unusually resilient once it enters the lungs, with even brief exposures to it causing symptoms that show up decades later, researchers said.

(24.27) A form of asbestos once used to make Kent cigarette filters has caused a high percentage of cancer deaths among a group of workers exposed to it more than 30 years ago, researchers reported.

Determining whether a word is a discourse connective is thus a special case of word sense disambiguation. Early work on disambiguation showed that the 4 PDTB high-level sense classes could be disambiguated with high (94%) accuracy used syntactic features from gold parse trees (Pitler and Nenkova, 2009). Recent work performs the task end-to-end from word inputs using a biLSTM-CRF with BIO outputs (B-CONN, I-CONN, O) (Yu et al., 2019).

For task 2, PDTB spans can be identified with the same sequence models used to find RST EDUs: a biLSTM sequence model with pretrained contextual embedding (BERT) inputs (Muller et al., 2019). Simple heuristics also do pretty well as a baseline at finding spans, since 93% of relations are either completely within a single sentence or span two adjacent sentences, with one argument in each sentence (Biran and McKeown, 2015).

24.3 Centering and Entity-Based Coherence

A second way a discourse can be coherent is by virtue of being "about" some entity. This idea that at each point in the discourse some entity is salient, and a discourse is coherent by continuing to discuss the same entity, appears early in functional linguistics and the psychology of discourse (Chafe 1976, Kintsch and Van Dijk 1978), and soon made its way to computational models. In this section we introduce two models of this kind of **entity-based coherence**: **Centering Theory** (Grosz et al., 1995), and the **entity grid** model of Barzilay and Lapata (2008).

entity-based

24.3.1 Centering

Centering Theory

Centering Theory (Grosz et al., 1995) is a theory of both discourse salience and discourse coherence. As a model of discourse salience, Centering proposes that at any given point in the discourse one of the entities in the discourse model is salient: it is being "centered" on. As a model of discourse coherence, Centering proposes that discourses in which adjacent sentences CONTINUE to maintain the same salient entity are more coherent than those which SHIFT back and forth between multiple entities (we will see that CONTINUE and SHIFT are technical terms in the theory).

The following two texts from Grosz et al. (1995) which have exactly the same propositional content but different saliences, can help in understanding the main Centering intuition.

(24.28) a. John went to his favorite music store to buy a piano.
b. He had frequented the store for many years.
c. He was excited that he could finally buy a piano.
d. He arrived just as the store was closing for the day.

(24.29) a. John went to his favorite music store to buy a piano.
b. It was a store John had frequented for many years.
c. He was excited that he could finally buy a piano.
d. It was closing just as John arrived.

While these two texts differ only in how the two entities (John and the store) are realized in the sentences, the discourse in (24.28) is intuitively more coherent than the one in (24.29). As Grosz et al. (1995) point out, this is because the discourse in (24.28) is clearly about one individual, John, describing his actions and feelings. The discourse in (24.29), by contrast, focuses first on John, then the store, then back to John, then to the store again. It lacks the "aboutness" of the first discourse.

backward-looking center

forward-looking center

Centering Theory realizes this intuition by maintaining two representations for each utterance U_n. The **backward-looking center** of U_n, denoted as $C_b(U_n)$, represents the current salient entity, the one being focused on in the discourse after U_n is interpreted. The **forward-looking centers** of U_n, denoted as $C_f(U_n)$, are a set of potential future salient entities, the discourse entities evoked by U_n any of which could serve as C_b (the salient entity) of the following utterance, i.e. $C_b(U_{n+1})$.

The set of forward-looking centers $C_f(U_n)$ are ranked according to factors like discourse salience and grammatical role (for example subjects are higher ranked than objects, which are higher ranked than all other grammatical roles). We call the highest-ranked forward-looking center C_p (for "preferred center"). C_p is a kind of prediction about what entity will be talked about next. Sometimes the next utterance indeed talks about this entity, but sometimes another entity becomes salient instead.

We'll use here the algorithm for centering presented in Brennan et al. (1987), which defines four intersentential relationships between a pair of utterances U_n and U_{n+1} that depend on the relationship between $C_b(U_{n+1})$, $C_b(U_n)$, and $C_p(U_{n+1})$; these are shown in Fig. 24.7.

	$C_b(U_{n+1}) = C_b(U_n)$ or undefined $C_b(U_n)$	$C_b(U_{n+1}) \neq C_b(U_n)$
$C_b(U_{n+1}) = C_p(U_{n+1})$	Continue	Smooth-Shift
$C_b(U_{n+1}) \neq C_p(U_{n+1})$	Retain	Rough-Shift

Figure 24.7 Centering Transitions for Rule 2 from Brennan et al. (1987).

The following rules are used by the algorithm:

Rule 1: If any element of $C_f(U_n)$ is realized by a pronoun in utterance U_{n+1}, then $C_b(U_{n+1})$ must be realized as a pronoun also.
Rule 2: Transition states are ordered. Continue is preferred to Retain is preferred to Smooth-Shift is preferred to Rough-Shift.

Rule 1 captures the intuition that pronominalization (including zero-anaphora) is a common way to mark discourse salience. If there are multiple pronouns in an utterance realizing entities from the previous utterance, one of these pronouns must realize the backward center C_b; if there is only one pronoun, it must be C_b.

Rule 2 captures the intuition that discourses that continue to center the same entity are more coherent than ones that repeatedly shift to other centers. The transition table is based on two factors: whether the backward-looking center C_b is the same from U_n to U_{n+1} and whether this discourse entity is the one that is preferred (C_p) in the new utterance U_{n+1}. If both of these hold, a CONTINUE relation, the speaker has been talking about the same entity and is going to continue talking about that

entity. In a RETAIN relation, the speaker intends to SHIFT to a new entity in a future utterance and meanwhile places the current entity in a lower rank C_f. In a SHIFT relation, the speaker is shifting to a new salient entity.

Let's walk though the start of (24.28) again, repeated as (24.30), showing the representations after each utterance is processed.

(24.30) John went to his favorite music store to buy a piano. (U_1)
He was excited that he could finally buy a piano. (U_2)
He arrived just as the store was closing for the day. (U_3)
It was closing just as John arrived (U_4)

Using the grammatical role hierarchy to order the C_f, for sentence U_1 we get:

$C_f(U_1)$: {John, music store, piano}
$C_p(U_1)$: John
$C_b(U_1)$: undefined

and then for sentence U_2:

$C_f(U_2)$: {John, piano}
$C_p(U_2)$: John
$C_b(U_2)$: John
Result: Continue ($C_p(U_2)$=$C_b(U_2)$; $C_b(U_1)$ undefined)

The transition from U_1 to U_2 is thus a CONTINUE. Completing this example is left as exercise (1) for the reader

24.3.2 Entity Grid model

Centering embodies a particular theory of how entity mentioning leads to coherence: that salient entities appear in subject position or are pronominalized, and that discourses are salient by means of continuing to mention the same entity in such ways.

entity grid

The **entity grid** model of Barzilay and Lapata (2008) is an alternative way to capture entity-based coherence: instead of having a top-down theory, the entity-grid model using machine learning to induce the patterns of entity mentioning that make a discourse more coherent.

The model is based around an **entity grid**, a two-dimensional array that represents the distribution of entity mentions across sentences. The rows represent sentences, and the columns represent discourse entities (most versions of the entity grid model focus just on nominal mentions). Each cell represents the possible appearance of an entity in a sentence, and the values represent whether the entity appears and its grammatical role. Grammatical roles are subject (S), object (O), neither (X), or absent (–); in the implementation of Barzilay and Lapata (2008), subjects of passives are represented with O, leading to a representation with some of the characteristics of thematic roles.

Fig. 24.8 from Barzilay and Lapata (2008) shows a grid for the text shown in Fig. 24.9. There is one row for each of the six sentences. The second column, for the entity 'trial', is O – – – X, showing that the trial appears in the first sentence as direct object, in the last sentence as an oblique, and does not appear in the middle sentences. The third column, for the entity Microsoft, shows that it appears as subject in sentence 1 (it also appears as the object of the preposition *against*, but entities that appear multiple times are recorded with their highest-ranked grammatical function). Computing the entity grids requires extracting entities and doing coreference

	Department	Trial	Microsoft	Evidence	Competitors	Markets	Products	Brands	Case	Netscape	Software	Tactics	Government	Suit	Earnings	
1	S	O	S	X	O	–	–	–	–	–	–	–	–	–	–	1
2	–	–	O	–	–	X	S	O	–	–	–	–	–	–	–	2
3	–	–	S	O	–	–	–	–	S	O	O	–	–	–	–	3
4	–	–	S	–	–	–	–	–	–	–	–	S	–	–	–	4
5	–	–	–	–	–	–	–	–	–	–	–	–	S	O	–	5
6	–	X	S	–	–	–	–	–	–	–	–	–	–	–	O	6

Figure 24.8 Part of the entity grid for the text in Fig. 24.9. Entities are listed by their head noun; each cell represents whether an entity appears as subject (S), object (O), neither (X), or is absent (–). Figure from Barzilay and Lapata (2008).

1 [The Justice Department]$_S$ is conducting an [anti-trust trial]$_O$ against [Microsoft Corp.]$_X$ with [evidence]$_X$ that [the company]$_S$ is increasingly attempting to crush [competitors]$_O$.
2 [Microsoft]$_O$ is accused of trying to forcefully buy into [markets]$_X$ where [its own products]$_S$ are not competitive enough to unseat [established brands]$_O$.
3 [The case]$_S$ revolves around [evidence]$_O$ of [Microsoft]$_S$ aggressively pressuring [Netscape]$_O$ into merging [browser software]$_O$.
4 [Microsoft]$_S$ claims [its tactics]$_S$ are commonplace and good economically.
5 [The government]$_S$ may file [a civil suit]$_O$ ruling that [conspiracy]$_S$ to curb [competition]$_O$ through [collusion]$_X$ is [a violation of the Sherman Act]$_O$.
6 [Microsoft]$_S$ continues to show [increased earnings]$_O$ despite [the trial]$_X$.

Figure 24.9 A discourse with the entities marked and annotated with grammatical functions. Figure from Barzilay and Lapata (2008).

resolution to cluster them into discourse entities (Chapter 23) as well as parsing the sentences to get grammatical roles.

In the resulting grid, columns that are dense (like the column for Microsoft) indicate entities that are mentioned often in the texts; sparse columns (like the column for earnings) indicate entities that are mentioned rarely.

In the entity grid model, coherence is measured by patterns of **local entity transition**. For example, Department is a subject in sentence 1, and then not mentioned in sentence 2; this is the transition [S –]. The transitions are thus sequences $\{$S,O X, –$\}^n$ which can be extracted as continuous cells from each column. Each transition has a probability; the probability of [S –] in the grid from Fig. 24.8 is 0.08 (it occurs 6 times out of the 75 total transitions of length two). Fig. 24.10 shows the distribution over transitions of length 2 for the text of Fig. 24.9 (shown as the first row d_1), and 2 other documents.

	SS	SO	SX	S–	OS	OO	OX	O–	XS	XO	XX	X–	–S	–O	–X	––
d_1	.01	.01	0	.08	.01	0	0	.09	0	0	0	.03	.05	.07	.03	.59
d_2	.02	.01	.01	.02	0	.07	0	.02	.14	.14	.06	.04	.03	.07	0.1	.36
d_3	.02	0	0	.03	.09	0	.09	.06	0	0	0	.05	.03	.07	.17	.39

Figure 24.10 A feature vector for representing documents using all transitions of length 2. Document d_1 is the text in Fig. 24.9. Figure from Barzilay and Lapata (2008).

The transitions and their probabilities can then be used as features for a machine learning model. This model can be a text classifier trained to produce human-labeled coherence scores (for example from humans labeling each text as coherent or incoherent). But such data is expensive to gather. Barzilay and Lapata (2005) introduced a simplifying innovation: coherence models can be trained by **self-supervision**: trained to distinguish the natural original order of sentences in a discourse from

a modified order (such as a randomized order). We turn to these evaluations in the next section.

24.3.3 Evaluating Neural and Entity-based coherence

Entity-based coherence models, as well as the neural models we introduce in the next section, are generally evaluated in one of two ways.

First, we can have humans rate the coherence of a document and train a classifier to predict these human ratings, which can be categorial (high/low, or high/mid/low) or continuous. This is the best evaluation to use if we have some end task in mind, like essay grading, where human raters are the correct definition of the final label.

Alternatively, since it's very expensive to get human labels, and we might not yet have an end-task in mind, we can use natural texts to do self-supervision. In self-supervision we pair up a natural discourse with a pseudo-document created by changing the ordering. Since naturally-ordered discourses are more coherent than random permutation (Lin et al., 2011), a successful coherence algorithm should prefer the original ordering.

Self-supervision has been implemented in 3 ways. In the **sentence order discrimination** task (Barzilay and Lapata, 2005), we compare a document to a random permutation of its sentences. A model is considered correct for an (original, permuted) test pair if it ranks the original document higher. Given k documents, we can compute n permutations, resulting in kn pairs each with one original document and one permutation, to use in training and testing.

In the **sentence insertion** task (Chen et al., 2007) we take a document, remove one of the n sentences s, and create $n-1$ copies of the document with s inserted into each position. The task is to decide which of the n documents is the one with the original ordering, distinguishing the original position for s from all other positions. Insertion is harder than discrimination since we are comparing documents that differ by only one sentence.

Finally, in the **sentence order reconstruction** task (Lapata, 2003), we take a document, randomize the sentences, and train the model to put them back in the correct order. Again given k documents, we can compute n permutations, resulting in kn pairs each with one original document and one permutation, to use in training and testing. Reordering is of course a much harder task than simple classification.

24.4 Representation learning models for local coherence

The third kind of local coherence is topical or semantic field coherence. Discourses cohere by talking about the same topics and subtopics, and drawing on the same semantic fields in doing so.

lexical cohesion

The field was pioneered by a series of unsupervised models in the 1990s of this kind of coherence that made use of **lexical cohesion** (Halliday and Hasan, 1976): the sharing of identical or semantically related words in nearby sentences. Morris and Hirst (1991) computed **lexical chains** of words (like *pine*, *bush trees*, *trunk*) that occurred through a discourse and that were related in Roget's Thesaurus (by being in the same category, or linked categories). They showed that the number and density of chain correlated with the topic structure. The **TextTiling** algorithm of Hearst (1997) computed the cosine between neighboring text spans (the normalized dot product of vectors of raw word counts), again showing that sentences or paragraph in

TextTiling

a subtopic have high cosine with each other, but not with sentences in a neighboring subtopic.

A third early model, the LSA Coherence method of Foltz et al. (1998) was the first to use embeddings, modeling the coherence between two sentences as the cosine between their LSA sentence embedding vectors[1], computing embeddings for a sentence s by summing the embeddings of its words w:

$$\begin{aligned} \text{sim}(s,t) &= \cos(\mathbf{s},\mathbf{t}) \\ &= \cos(\sum_{w\in s}\mathbf{w},\sum_{w\in t}\mathbf{w}) \end{aligned} \tag{24.31}$$

and defining the overall coherence of a text as the average similarity over all pairs of adjacent sentences s_i and s_{i+1}:

$$\text{coherence}(T) = \frac{1}{n-1}\sum_{i=1}^{n-1}\cos(s_i,s_{i+1}) \tag{24.32}$$

Modern neural representation-learning coherence models, beginning with Li et al. (2014), draw on the intuitions of these early unsupervised models for learning sentence representations and measuring how they change between neighboring sentences. But the new models also draw on the idea pioneered by Barzilay and Lapata (2005) of self-supervision. That is, unlike say coherence relation models, which train on hand-labeled representations for RST or PDTB, these models are trained to distinguish natural discourses from unnatural discourses formed by scrambling the order of sentences, thus using representation learning to discover the features that matter for at least the ordering aspect of coherence.

Here we present one such model, the local coherence discriminator (LCD) (Xu et al., 2019). Like early models, LCD computes the coherence of a text as the average of coherence scores between consecutive pairs of sentences. But unlike the early unsupervised models, LCD is a self-supervised model trained to discriminate consecutive sentence pairs (s_i,s_{i+1}) in the training documents (assumed to be coherent) from (constructed) incoherent pairs (s_i,s'). All consecutive pairs are positive examples, and the negative (incoherent) partner for a sentence s_i is another sentence uniformly sampled from the same document as s_i.

Fig. 24.11 describes the architecture of the model f_θ, which takes a sentence pair and returns a score, higher scores for more coherent pairs. Given an input sentence pair s and t, the model computes sentence embeddings $\mathbf{s}$ and $\mathbf{t}$ (using any sentence embeddings algorithm), and then concatenates four features of the pair: (1) the concatenation of the two vectors (2) their difference $\mathbf{s}-\mathbf{t}$; (3) the absolute value of their difference $|\mathbf{s}-\mathbf{t}|$; (4) their element-wise product $\mathbf{s}\odot\mathbf{t}$. These are passed through a one-layer feedforward network to output the coherence score.

The model is trained to make this coherence score higher for real pairs than for negative pairs. More formally, the training objective for a corpus C of documents d, each of which consists of a list of sentences s_i, is:

$$L_\theta = \sum_{d\in C}\sum_{s_i\in d}\mathop{\mathbb{E}}_{p(s'|s_i)}[L(f_\theta(s_i,s_{i+1}),f_\theta(s_i,s'))] \tag{24.33}$$

$\mathbb{E}_{p(s'|s_i)}$ is the expectation with respect to the negative sampling distribution conditioned on s_i: given a sentence s_i the algorithms samples a negative sentence s'

[1] See Chapter 5 for more on LSA embeddings; they are computed by applying SVD to the term-document matrix (each cell weighted by log frequency and normalized by entropy), and then the first 300 dimensions are used as the embedding.

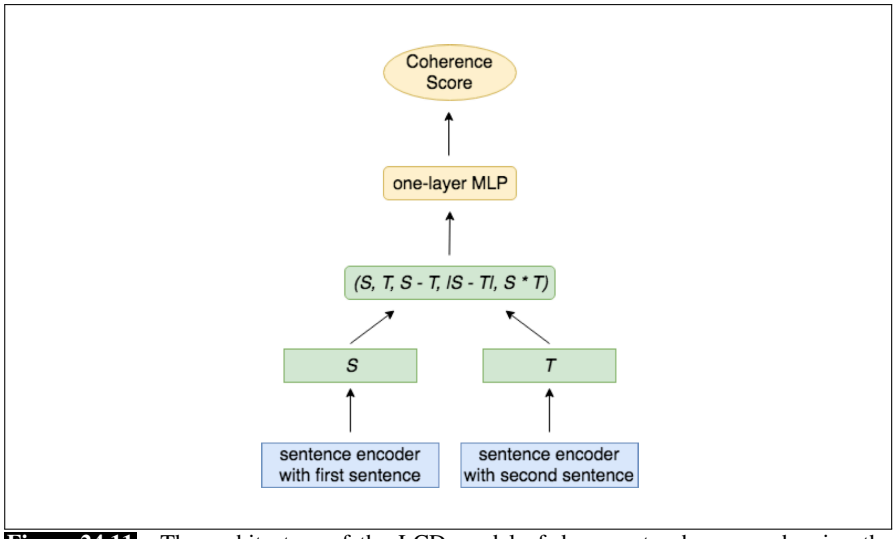


Figure 24.11 The architecture of the LCD model of document coherence, showing the computation of the score for a pair of sentences s and t. Figure from Xu et al. (2019).

uniformly over the other sentences in the same document. L is a loss function that takes two scores, one for a positive pair and one for a negative pair, with the goal of encouraging $f^+ = f_\theta(s_i, s_{i+1})$ to be high and $f^- = f_\theta(s_i, s'))$ to be low. Fig. 24.11 use the margin loss $l(f^+, f^-) = \max(0, \eta - f^+ + f^-)$ where η is the margin hyperparameter.

Xu et al. (2019) also give a useful baseline algorithm that itself has quite high performance in measuring perplexity: train an RNN language model on the data, and compute the log likelihood of sentence s_i in two ways, once given the preceding context (conditional log likelihood) and once with no context (marginal log likelihood). The difference between these values tells us how much the preceding context improved the predictability of s_i, a predictability measure of coherence.

Training models to predict longer contexts than just consecutive pairs of sentences can result in even stronger discourse representations. For example a Transformer language model trained with a contrastive sentence objective to predict text up to a distance of ± 2 sentences improves performance on various discourse coherence tasks (Iter et al., 2020).

Language-model style models are generally evaluated by the methods of Section 24.3.3, although they can also be evaluated on the RST and PDTB coherence relation tasks.

24.5 Global Coherence

A discourse must also cohere globally rather than just at the level of pairs of sentences. Consider stories, for example. The narrative structure of stories is one of the oldest kinds of global coherence to be studied. In his influential *Morphology of the Folktale*, Propp (1968) models the discourse structure of Russian folktales via a kind of plot grammar. His model includes a set of character categories he called **dramatis personae**, like Hero, Villain, Donor, or Helper, and a set of events he called **functions** (like "Villain commits kidnapping", "Donor tests Hero", or "Hero

is pursued") that have to occur in particular order, along with other components. Propp shows that the plots of each of the fairy tales he studies can be represented as a sequence of these functions, different tales choosing different subsets of functions, but always in the same order. Indeed Lakoff (1972) showed that Propp's model amounted to a discourse grammar of stories, and in recent computational work Finlayson (2016) demonstrates that some of these Proppian functions could be induced from corpora of folktale texts by detecting events that have similar actions across stories. Bamman et al. (2013) showed that generalizations over dramatis personae could be induced from movie plot summaries on Wikipedia. Their model induced latent personae from features like the actions the character takes (e.g., Villains strangle), the actions done to them (e.g., Villains are foiled and arrested) or the descriptive words used of them (Villains are evil).

In this section we introduce two kinds of such global discourse structure that have been widely studied computationally. The first is the structure of arguments: the way people attempt to convince each other in persuasive essays by offering claims and supporting premises. The second is somewhat related: the structure of scientific papers, and the way authors present their goals, results, and relationship to prior work in their papers.

24.5.1 Argumentation Structure

argumentation mining

The first type of global discourse structure is the structure of **arguments**. Analyzing people's argumentation computationally is often called **argumentation mining**.

pathos **ethos** **logos**

The study of arguments dates back to Aristotle, who in his Rhetorics described three components of a good argument: **pathos** (appealing to the emotions of the listener), **ethos** (appealing to the speaker's personal character), and **logos** (the logical structure of the argument).

Most of the discourse structure studies of argumentation have focused on **logos**, particularly via building and training on annotated datasets of persuasive essays or other arguments (Reed et al. 2008, Stab and Gurevych 2014a, Peldszus and Stede 2016, Habernal and Gurevych 2017, Musi et al. 2018). Such corpora, for example, often include annotations of argumentative components like **claims** (the central component of the argument that is controversial and needs support) and **premises** (the reasons given by the author to persuade the reader by supporting or attacking the claim or other premises), as well as the **argumentative relations** between them like SUPPORT and ATTACK.

claims **premises** **argumentative relations**

Consider the following example of a persuasive essay from Stab and Gurevych (2014b). The first sentence (1) presents a claim (in bold). (2) and (3) present two premises supporting the claim. (4) gives a premise supporting premise (3).

> "(1) **Museums and art galleries provide a better understanding about arts than Internet.** (2) In most museums and art galleries, detailed descriptions in terms of the background, history and author are provided. (3) Seeing an artwork online is not the same as watching it with our own eyes, as (4) the picture online does not show the texture or three-dimensional structure of the art, which is important to study."

Thus this example has three argumentative relations: SUPPORT(2,1), SUPPORT(3,1) and SUPPORT(4,3). Fig. 24.12 shows the structure of a much more complex argument.

While argumentation mining is clearly related to rhetorical structure and other kinds of coherence relations, arguments tend to be much less local; often a persua-

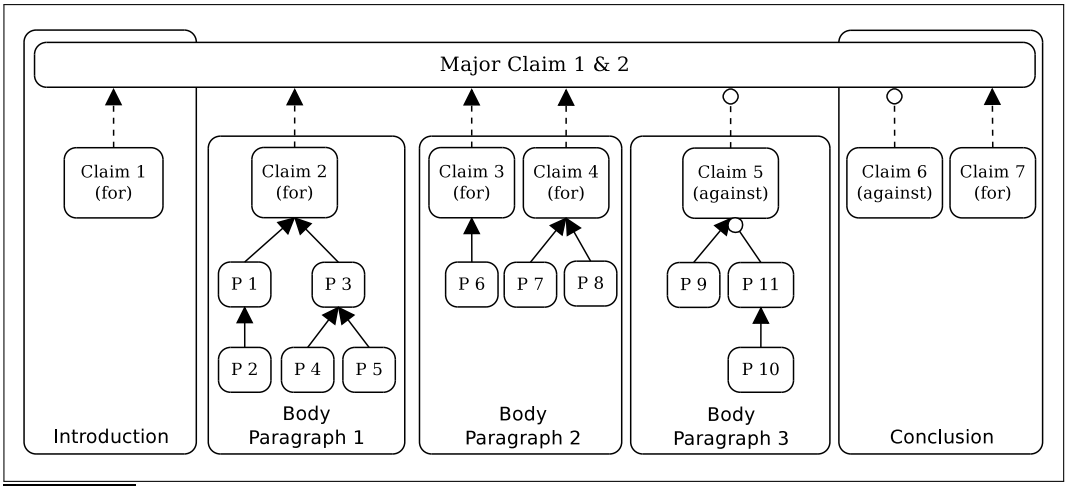


Figure 24.12 Argumentation structure of a persuasive essay. Arrows indicate argumentation relations, either of SUPPORT (with arrowheads) or ATTACK (with circleheads); P denotes premises. Figure from Stab and Gurevych (2017).

sive essay will have only a single main claim, with premises spread throughout the text, without the local coherence we see in coherence relations.

Algorithms for detecting argumentation structure often include classifiers for distinguishing claims, premises, or non-argumentation, together with relation classifiers for deciding if two spans have the SUPPORT, ATTACK, or neither relation (Peldszus and Stede, 2013). While these are the main focus of much computational work, there is also preliminary efforts on annotating and detecting richer semantic relationships (Park and Cardie 2014, Hidey et al. 2017) such as detecting **argumentation schemes**, larger-scale structures for argument like **argument from example**, or **argument from cause to effect**, or **argument from consequences** (Feng and Hirst, 2011).

argumentation schemes

Another important line of research is studying how these argument structure (or other features) are associated with the success or persuasiveness of an argument (Habernal and Gurevych 2016, Tan et al. 2016, Hidey et al. 2017. Indeed, while it is Aristotle's logos that is most related to discourse structure, Aristotle's ethos and pathos techniques are particularly relevant in the detection of mechanisms of this sort of **persuasion**. For example scholars have investigated the linguistic realization of features studied by social scientists like **reciprocity** (people return favors), **social proof** (people follow others' choices), **authority** (people are influenced by those with power), and **scarcity** (people value things that are scarce), all of which can be brought up in a persuasive argument (Cialdini, 1984). Rosenthal and McKeown (2017) showed that these features could be combined with argumentation structure to predict who influences whom on social media, Althoff et al. (2014) found that linguistic models of reciprocity and authority predicted success in online requests, while the semisupervised model of Yang et al. (2019) detected mentions of scarcity, commitment, and social identity to predict the success of peer-to-peer lending platforms.

persuasion

See Stede and Schneider (2018) for a comprehensive survey of argument mining.

24.5.2 The structure of scientific discourse

argumentative zoning

Scientific papers have a very specific global structure: somewhere in the course of the paper the authors must indicate a scientific goal, develop a method for a solution, provide evidence for the solution, and compare to prior work. One popular annotation scheme for modeling these rhetorical goals is the **argumentative zoning** model of Teufel et al. (1999) and Teufel et al. (2009), which is informed by the idea that each scientific paper tries to make a **knowledge claim** about a new piece of knowledge being added to the repository of the field (Myers, 1992). Sentences in a scientific paper can be assigned one of 15 tags; Fig. 24.13 shows 7 (shortened) examples of labeled sentences.

Category	Description	Example
AIM	Statement of specific research goal, or hypothesis of current paper	"The aim of this process is to examine the role that training plays in the tagging process"
OWN_METHOD	New Knowledge claim, own work: methods	"In order for it to be useful for our purposes, the following extensions must be made:"
OWN_RESULTS	Measurable/objective outcome of own work	"All the curves have a generally upward trend but always lie far below backoff (51% error rate)"
USE	Other work is used in own work	"We use the framework for the allocation and transfer of control of Whittaker...."
GAP_WEAK	Lack of solution in field, problem with other solutions	"Here, we will produce experimental evidence suggesting that this simple model leads to serious overestimates"
SUPPORT	Other work supports current work or is supported by current work	"Work similar to that described here has been carried out by Merialdo (1994), with broadly similar conclusions."
ANTISUPPORT	Clash with other's results or theory; superiority of own work	"This result challenges the claims of..."

Figure 24.13 Examples for 7 of the 15 labels from the Argumentative Zoning labelset (Teufel et al., 2009).

Teufel et al. (1999) and Teufel et al. (2009) develop labeled corpora of scientific articles from computational linguistics and chemistry, which can be used as supervision for training standard sentence-classification architecture to assign the 15 labels.

24.6 Summary

In this chapter we introduced local and global models for discourse **coherence**.

- Discourses are not arbitrary collections of sentences; they must be *coherent*. Among the factors that make a discourse coherent are coherence relations between the sentences, entity-based coherence, and topical coherence.
- Various sets of **coherence relations** and **rhetorical relations** have been proposed. The relations in Rhetorical Structure Theory (**RST**) hold between spans of text and are structured into a tree. Because of this, shift-reduce and other parsing algorithms are generally used to assign these structures. The Penn Discourse Treebank (**PDTB**) labels only relations between pairs of spans, and the labels are generally assigned by sequence models.
- **Entity-based coherence** captures the intuition that discourses are **about** an entity, and continue mentioning the entity from sentence to sentence. **Centering Theory** is a family of models describing how salience is modeled for

discourse entities, and hence how coherence is achieved by virtue of keeping the same discourse entities salient over the discourse. The **entity grid** model gives a more bottom-up way to compute which entity realization transitions lead to coherence.

- Many different genres have different types of **global coherence**. Persuasive essays have claims and premises that are extracted in the field of **argument mining**, scientific articles have structure related to aims, methods, results, and comparisons.

Historical Notes

Coherence relations arose from the independent development of a number of scholars, including Hobbs (1979) idea that coherence relations play an inferential role for the hearer, and the investigations by Mann and Thompson (1987) of the discourse structure of large texts. Other approaches to coherence relations and their extraction include Segmented Discourse Representation Theory (**SDRT**) (Asher and Lascarides 2003, Baldridge et al. 2007) and the Linguistic Discourse Model (Polanyi 1988, Scha and Polanyi 1988, Polanyi et al. 2004). Wolf and Gibson (2005) argue that coherence structure includes crossed bracketings, which make it impossible to represent as a tree, and propose a graph representation instead. A compendium of over 350 relations that have been proposed in the literature can be found in Hovy (1990).

SDRT

RST parsing was first proposed by Marcu (1997), and early work was rule-based, focused on discourse markers (Marcu, 2000a). The creation of the RST Discourse TreeBank (Carlson et al. 2001, Carlson and Marcu 2001) enabled a wide variety of machine learning algorithms, beginning with the shift-reduce parser of Marcu (1999) that used decision trees to choose actions, and continuing with a wide variety of machine learned parsing methods (Soricut and Marcu 2003, Sagae 2009, Hernault et al. 2010, Feng and Hirst 2014, Surdeanu et al. 2015, Joty et al. 2015) and chunkers (Sporleder and Lapata, 2005). Subba and Di Eugenio (2009) integrated sophisticated semantic information into RST parsing. Ji and Eisenstein (2014) first applied neural models to RST parsing neural models, leading to the modern set of neural RST models (Li et al. 2014, Li et al. 2016, Braud et al. 2017, Yu et al. 2018, inter alia) as well as neural segmenters (Wang et al. 2018b). and neural PDTB parsing models (Ji and Eisenstein 2015, Qin et al. 2016, Qin et al. 2017).

Barzilay and Lapata (2005) pioneered the idea of self-supervision for coherence: training a coherence model to distinguish true orderings of sentences from random permutations. Li et al. (2014) first applied this paradigm to neural sentence-representation, and many neural self-supervised models followed (Li and Jurafsky 2017, Logeswaran et al. 2018, Lai and Tetreault 2018, Xu et al. 2019, Iter et al. 2020)

Another aspect of global coherence is the global topic structure of a text, the way the topics shift over the course of the document. Barzilay and Lee (2004) introduced an HMM model for capturing topics for coherence, and later work expanded this intuition (Soricut and Marcu 2006, Elsner et al. 2007, Louis and Nenkova 2012, Li and Jurafsky 2017).

The relationship between explicit and implicit discourse connectives has been a fruitful one for research. Marcu and Echihabi (2002) first proposed to use sen-

tences with explicit relations to help provide training data for implicit relations, by removing the explicit relations and trying to re-predict them as a way of improving performance on implicit connectives; this idea was refined by Sporleder and Lascarides (2005), (Pitler et al., 2009), and Rutherford and Xue (2015). This relationship can also be used as a way to create discourse-aware representations. The DisSent algorithm (Nie et al., 2019) creates the task of predicting explicit discourse markers between two sentences. They show that representations learned to be good at this task also function as powerful sentence representations for other discourse tasks.

The idea of entity-based coherence seems to have arisen in multiple fields in the mid-1970s, in functional linguistics (Chafe, 1976), in the psychology of discourse processing (Kintsch and Van Dijk, 1978), and in the roughly contemporaneous work of Grosz, Sidner, Joshi, and their colleagues. Grosz (1977a) addressed the focus of attention that conversational participants maintain as the discourse unfolds. She defined two levels of focus; entities relevant to the entire discourse were said to be in *global* focus, whereas entities that are locally in focus (i.e., most central to a particular utterance) were said to be in *immediate* focus. Sidner 1979; 1983 described a method for tracking (immediate) discourse foci and their use in resolving pronouns and demonstrative noun phrases. She made a distinction between the current discourse focus and potential foci, which are the predecessors to the backward- and forward-looking centers of Centering theory, respectively. The name and further roots of the centering approach lie in papers by Joshi and Kuhn (1979) and Joshi and Weinstein (1981), who addressed the relationship between immediate focus and the inferences required to integrate the current utterance into the discourse model. Grosz et al. (1983) integrated this work with the prior work of Sidner and Grosz. This led to a manuscript on centering which, while widely circulated since 1986, remained unpublished until Grosz et al. (1995). A collection of centering papers appears in Walker et al. (1998). See Karamanis et al. (2004) and Poesio et al. (2004) for a deeper exploration of centering and its parameterizations, and the History section of Chapter 23 for more on the use of centering on coreference.

The grid model of entity-based coherence was first proposed by Barzilay and Lapata (2005) drawing on earlier work by Lapata (2003) and Barzilay, and then extended by them Barzilay and Lapata (2008) and others with additional features (Elsner and Charniak 2008, 2011, Feng et al. 2014, Lin et al. 2011) a model that projects entities into a global graph for the discourse (Guinaudeau and Strube 2013, Mesgar and Strube 2016), and a convolutional model to capture longer-range entity dependencies (Nguyen and Joty, 2017).

Theories of discourse coherence have also been used in algorithms for interpreting discourse-level linguistic phenomena, including verb phrase ellipsis and gapping (Asher 1993, Kehler 1993), and tense interpretation (Lascarides and Asher 1993, Kehler 1994, Kehler 2000). An extensive investigation into the relationship between coherence relations and discourse connectives can be found in Knott and Dale (1994).

Useful surveys of discourse processing and structure include Stede (2011) and Webber et al. (2012).

Andy Kehler wrote the Discourse chapter for the 2000 first edition of this textbook, which we used as the starting point for the second-edition chapter, and there are some remnants of Andy's lovely prose still in this third-edition coherence chapter.

Exercises

24.1 Finish the Centering Theory processing of the last two utterances of (24.30), and show how (24.29) would be processed. Does the algorithm indeed mark (24.29) as less coherent?

24.2 Select an editorial column from your favorite newspaper, and determine the discourse structure for a 10–20 sentence portion. What problems did you encounter? Were you helped by superficial cues the speaker included (e.g., discourse connectives) in any places?

CHAPTER

25 Conversation and its Structure

Conversation is an intricate and complex joint activity, and conversations have structure. This is true of all conversations, whether they are conversations between people or conversations between people and language models. Understanding the structure of human conversations is an important social science and linguistic task. The concepts we introduce in studying human conversation can also be a useful tool for analyzing human-LLM conversations.

[This draft is the initial stub of a chapter that will introduce different kinds of conversational structure and how to annotate them computationally.]

25.1 Properties of Human Conversation

What are the conversational phenomena that take place when humans converse with each other? Are conversations between humans and machines different? Consider what goes on in the conversation between a human travel agent and a human client excerpted in Fig. 25.1.

C_1:	. . . I need to travel in May.
A_2:	And, what day in May did you want to travel?
C_3:	OK uh I need to be there for a meeting that's from the 12th to the 15th.
A_4:	And you're flying into what city?
C_5:	Seattle.
A_6:	And what time would you like to leave Pittsburgh?
C_7:	Uh hmm I don't think there's many options for non-stop.
A_8:	Right. There's three non-stops today.
C_9:	What are they?
A_{10}:	The first one departs PGH at 10:00am arrives Seattle at 12:05 their time. The second flight departs PGH at 5:55pm, arrives Seattle at 8pm. And the last flight departs PGH at 8:15pm arrives Seattle at 10:28pm.
C_{11}:	OK I'll take the 5ish flight on the night before on the 11th.
A_{12}:	On the 11th? OK. Departing at 5:55pm arrives Seattle at 8pm, U.S. Air flight 115.
C_{13}:	OK.
A_{14}:	And you said returning on May 15th?
C_{15}:	Uh, yeah, at the end of the day.
A_{16}:	OK. There's #two non-stops . . . #
C_{17}:	#Act. . . actually #, what day of the week is the 15th?
A_{18}:	It's a Friday.
C_{19}:	Uh hmm. I would consider staying there an extra day til Sunday.
A_{20}:	OK. . . OK. On Sunday I have . . .

Figure 25.1 Part of a phone conversation between a human travel agent (A) and human client (C). The passages framed by # in A_{16} and C_{17} indicate overlaps in speech.

25.1.1 Turns

turn

A dialogue is a sequence of **turns** (C_1, A_2, C_3, and so on), each a single contribution from one speaker to the dialogue (as if in a game: I take a turn, then you take a turn, then me, and so on). There are 20 turns in Fig. 25.1. A turn can consist of a sentence (like C_1), although it might be as short as a single word (C_{13}) or as long as multiple sentences (A_{10}).

Turn structure has important implications for spoken dialogue. A human has to know when to stop talking; the client interrupts (in A_{16} and C_{17}), so a system that was performing this role must know to stop talking (and that the user might be making a correction).

The same issues come up for LLMs; a system also has to know when to start talking. For example, most of the time in conversation, speakers start their turns almost immediately after the other speaker finishes, without a long pause, because people are can usually predict when the other person is about to finish talking.

endpointing

Spoken language models must also detect whether a user is done speaking, so they can process the utterance and respond. This task—called **endpointing** or **endpoint detection**— can be quite challenging because of noise and because people often pause in the middle of turns.

25.1.2 Speech Acts

A key insight into conversation—due originally to the philosopher Wittgenstein (1953) but worked out more fully by Austin (1962)—is that each utterance in a dialogue is a kind of **action** being performed by the speaker. These actions are commonly called **speech acts** or **dialogue acts**: here's one taxonomy consisting of 4 major classes (Bach and Harnish, 1979):

speech acts

Constatives:	committing the speaker to something's being the case (*answering*, *claiming*, *confirming*, *denying*, *disagreeing*, *stating*)
Directives:	attempts by the speaker to get the addressee to do something (*advising*, *asking*, *forbidding*, *inviting*, *ordering*, *requesting*)
Commissives:	committing the speaker to some future course of action (*promising, planning, vowing, betting, opposing*)
Acknowledgments:	express the speaker's attitude regarding the hearer with respect to some social action (*apologizing*, *greeting*, *thanking*, *accepting an acknowledgment*)

A user asking a person or a dialogue system to do something ('Turn up the music') is issuing a DIRECTIVE. Asking a question that requires an answer is also a way of issuing a DIRECTIVE: in a sense when the system says (A_2) "what day in May did you want to travel?" it's as if the system is (very politely) commanding the user to answer. By contrast, a user stating a constraint (like C_1 'I need to travel in May') is issuing a CONSTATIVE. A user thanking the system is issuing an ACKNOWLEDGMENT. The speech act expresses an important component of the intention of the speaker (or writer) in saying what they said.

25.1.3 Grounding

common ground
grounding

A dialogue is not just a series of independent speech acts, but rather a collective act performed by the speaker and the hearer. Like all collective acts, it's important for the participants to establish what they both agree on, called the **common ground** (Stalnaker, 1978). Speakers do this by **grounding** each other's utterances. Ground-

ing means acknowledging that the hearer has understood the speaker (Clark, 1996). (People need grounding for non-linguistic actions as well; the reason an elevator button lights up when it's pressed is to acknowledge that the elevator has indeed been called, essentially grounding your action of pushing the button (Norman, 1988).)

Grounding is also important when the hearer needs to indicate that the speaker has *not* succeeded in performing an action. If the hearer has problems in understanding, she must indicate these problems to the speaker, again so that mutual understanding can eventually be achieved.

contribution

How is closure achieved? Clark and Schaefer (1989) introduce the idea that each joint linguistic act or **contribution** has two phases, called **presentation** and **acceptance**. In the first phase, a speaker presents the hearer with an utterance, performing a sort of speech act. In the acceptance phase, the hearer has to ground the utterance, indicating to the speaker whether understanding was achieved.

What methods can the hearer B use to ground the speaker A's utterance? Clark and Schaefer (1989) discuss a continuum of methods ordered from weakest to strongest:

Continued attention:	B shows she is continuing to attend and therefore remains satisfied with A's presentation.
Next contribution:	B starts in on the next relevant contribution.
Acknowledgment:	B nods or says a continuer like *uh-huh*, *yeah*, or the like, or an **assessment** like *that's great*.
Demonstration:	B demonstrates all or part of what she has understood A to mean, for example, by **reformulating** (paraphrasing) A's utterance or by **collaborative completion** of A's utterance.
Display:	B displays verbatim all or part of A's presentation.

Examples of these kind of grounding occur in the travel agent conversation. We can ground by explicitly saying "OK", as the agent does in A_8 or A_{10}. Or we can ground by repeating what the other person says; in utterance A_2 the agent repeats "in May", demonstrating her understanding to the client. Or notice that when the client answers a question, the agent begins the next question with "And". The "And" implies that the new question is 'in addition' to the old question, again indicating to the client that the agent has successfully understood the answer to the last question.

This particular fragment doesn't have an example of an *acknowledgment*, but there's an example in another fragment:

C: He wants to fly from Boston to Baltimore
A: **Uh huh**

continuer
backchannel

The word *uh-huh* here is a **continuer**, also often called an **acknowledgment token** or a **backchannel**. A continuer is a (short) optional utterance that acknowledges the content of the utterance of the other and that doesn't require an acknowledgment by the other (Yngve, 1970; Jefferson, 1984; Schegloff, 1982; Ward and Tsukahara, 2000).

25.1.4 Subdialogues and Dialogue Structure

conversation analysis

Conversations have structure. Consider, for example, the local structure between speech acts discussed in the field of **conversation analysis** (Sacks et al., 1974). QUESTIONS set up an expectation for an ANSWER. PROPOSALS are followed by ACCEPTANCE (or REJECTION). COMPLIMENTS ("Nice jacket!") often give rise to DOWNPLAYERS ("Oh, this old thing?"). These pairs, called **adjacency pairs**, are

adjacency pair

composed of a **first pair part** and a **second pair part** (Schegloff, 1968), and these expectations can help systems decide what actions to take.

side sequence
subdialogue

However, dialogue acts aren't always followed immediately by their second pair part. The two parts can be separated by a **side sequence** (Jefferson 1972) or **subdialogue**. For example utterances C_{17} to A_{20} constitute a **correction subdialogue** (Litman 1985, Litman and Allen 1987, Chu-Carroll and Carberry 1998):

C_{17}: #Act... actually#, what day of the week is the 15th?
A_{18}: It's a Friday.
C_{19}: Uh hmm. I would consider staying there an extra day til Sunday.
A_{20}: OK... OK. On Sunday I have ...

The question in C_{17} interrupts the prior discourse, in which the agent was looking for a May 15 return flight. The agent must answer the question and also realize that ''I would consider staying...til Sunday" means that the client would probably like to change their plan, and now go back to finding return flights, but for the 17th.

Another side sequence is the **clarification question**, which can form a subdialogue between a REQUEST and a RESPONSE. This is especially common in dialogue systems where speech recognition errors causes the system to have to ask for clarifications or repetitions like the following:

User: What do you have going to UNKNOWN_WORD on the 5th?
System: Let's see, going where on the 5th?
User: Going to Hong Kong.
System: OK, here are some flights...

presequence

In addition to side-sequences, questions often have **presequences**, like the following example where a user starts with a question about the system's capabilities ("Can you make train reservations") before making a request.

User: Can you make train reservations?
System: Yes I can.
User: Great, I'd like to reserve a seat on the 4pm train to New York.

25.1.5 Initiative

initiative

Sometimes a conversation is completely controlled by one participant. For example a reporter interviewing a chef might ask questions, and the chef responds. We say that the reporter in this case has the conversational **initiative** (Carbonell, 1970; Nickerson, 1976). In normal human-human dialogue, however, it's more common for initiative to shift back and forth between the participants, as they sometimes answer questions, sometimes ask them, sometimes take the conversations in new directions, sometimes not. You may ask me a question, and then I respond asking you to clarify something you said, which leads the conversation in all sorts of ways. We call such interactions **mixed initiative** (Carbonell, 1970).

Full mixed initiative, while the norm for human-human conversations, can be difficult for dialogue systems. The most primitive dialogue systems tend to use **system-initiative**, where the system asks a question and the user can't do anything until they answer it, or *user-initiative* like simple search engines, where the user specifies a query and the system passively responds. Even modern large language model-based dialogue systems, which come much closer to using full mixed initiative, often don't have completely natural initiative switching. Getting this right is an important goal for modern systems.

25.1.6 Inference and Implicature

Inference is also important in dialogue understanding. Consider the client's response C_2, repeated here:

A_2: And, what day in May did you want to travel?

C_3: OK uh I need to be there for a meeting that's from the 12th to the 15th.

Notice that the client does not in fact answer the agent's question. The client merely mentions a meeting at a certain time. What is it that licenses the agent to infer that the client is mentioning this meeting so as to inform the agent of the travel dates?

The speaker seems to expect the hearer to draw certain inferences; in other words, the speaker is communicating more information than seems to be present in the uttered words. This kind of example was pointed out by Grice (1975, 1978) as part of his theory of **conversational implicature**. **Implicature** means a particular class of licensed inferences. Grice proposed that what enables hearers to draw these inferences is that conversation is guided by a set of **maxims**, general heuristics that play a guiding role in the interpretation of conversational utterances. One such maxim is the maxim of **relevance** which says that speakers attempt to be relevant, they don't just utter random speech acts. When the client mentions a meeting on the 12th, the agent reasons 'There must be some relevance for mentioning this meeting. What could it be?'. The agent knows that one precondition for having a meeting (at least before Web conferencing) is being at the place where the meeting is held, and therefore that maybe the meeting is a reason for the travel, and if so, then since people like to arrive the day before a meeting, the agent should infer that the flight should be on the 11th.

implicature

relevance

These subtle characteristics of human conversations (**turns**, **speech acts**, **grounding**, **dialogue structure**, **initiative**, and **implicature**) are among the reasons it is difficult to build dialogue systems that can carry on natural conversations with humans. Many of these challenges are active areas of dialogue systems research.

25.2 Dialog Acts and Corpora

The ideas of speech acts and grounding are combined in a single kind of action called a **dialogue act**, a tag which represents the interactive function of the sentence being tagged.

dialogue act

Dialog acts can be used to analyze human-human conversation or human-LLM conversation. Both the nature of the participants and the type of dialogue (task-based or not task-based) influence the development of dialogue act tagsets.

Figure 25.2 shows a domain-specific tagset for the task of two people scheduling meetings. It has tags specific to the domain of scheduling, such as SUGGEST, used for the proposal of a particular date to meet, and ACCEPT and REJECT, used for acceptance or rejection of a proposal for a date, but also tags that have a more general function, like CLARIFY, used to request a user to clarify an ambiguous proposal.

Figure 25.3 shows a tagset for a restaurant recommendation system, and Fig. 25.4 shows these tags labeling a sample dialogue from the HIS system (Young et al., 2010). This example also shows the content of each dialogue acts, which are the slot fillers being communicated.

There are a number of more general and domain-independent dialogue act tagsets. In the DAMSL (Dialogue Act Markup in Several Layers) architecture inspired by

Tag	Example
THANK	*Thanks*
GREET	*Hello Dan*
INTRODUCE	*It's me again*
BYE	*Alright bye*
REQUEST-COMMENT	*How does that look?*
SUGGEST	*from thirteenth through seventeenth June*
REJECT	*No Friday I'm booked all day*
ACCEPT	*Saturday sounds fine*
REQUEST-SUGGEST	*What is a good day of the week for you?*
INIT	*I wanted to make an appointment with you*
GIVE_REASON	*Because I have meetings all afternoon*
FEEDBACK	*Okay*
DELIBERATE	*Let me check my calendar here*
CONFIRM	*Okay, that would be wonderful*
CLARIFY	*Okay, do you mean Tuesday the 23rd?*
DIGRESS	*[we could meet for lunch] and eat lots of ice cream*
MOTIVATE	*We should go to visit our subsidiary in Munich*
GARBAGE	*Oops, I-*

Figure 25.2 The 18 high-level dialogue acts for a meeting scheduling task, from the Verbmobil-1 system (Jekat et al., 1995).

Tag	Sys	User	Description
HELLO($a = x, b = y, \ldots$)	✓	✓	Open a dialogue and give info $a = x, b = y, \ldots$
INFORM($a = x, b = y, \ldots$)	✓	✓	Give info $a = x, b = y, \ldots$
REQUEST($a, b = x, \ldots$)	✓	✓	Request value for a given $b = x, \ldots$
REQALTS($a = x, \ldots$)	χ	✓	Request alternative with $a = x, \ldots$
CONFIRM($a = x, b = y, \ldots$)	✓	✓	Explicitly confirm $a = x, b = y, \ldots$
CONFREQ($a = x, \ldots, d$)	✓	χ	Implicitly confirm $a = x, \ldots$ and request value of d
SELECT($a = x, a = y$)	✓	χ	Implicitly confirm $a = x, \ldots$ and request value of d
AFFIRM($a = x, b = y, \ldots$)	✓	✓	Affirm and give further info $a = x, b = y, \ldots$
NEGATE($a = x$)	χ	✓	Negate and give corrected value $a = x$
DENY($a = x$)	χ	✓	Deny that $a = x$
BYE()	✓	✓	Close a dialogue

Figure 25.3 Dialogue acts used by the HIS restaurant recommendation system of Young et al. (2010). The Sys and User columns indicate which acts are valid as system outputs and user inputs, respectively.

the work of Clark and Schaefer (1989), Allwood et al. (1992), and (Allwood, 1995), each utterance is tagged for two types of functions, **forward-looking functions** like speech act functions, and **backward-looking** functions, like grounding and answering, which "look back" to the interlocutor's previous utterance (Allen and Core, 1997; Walker et al., 1996; Carletta et al., 1997; Core et al., 1999).

	Utterance	Dialogue act
U:	Hi, I am looking for somewhere to eat.	`hello(task = find,type=restaurant)`
S:	You are looking for a restaurant. What type of food do you like?	`confreq(type = restaurant, food)`
U:	I'd like an Italian somewhere near the museum.	`inform(food = Italian, near=museum)`
S:	Roma is a nice Italian restaurant near the museum.	`inform(name = "Roma", type = restaurant, food = Italian, near = museum)`
U:	Is it reasonably priced?	`confirm(pricerange = moderate)`
S:	Yes, Roma is in the moderate price range.	`affirm(name = "Roma", pricerange = moderate)`
U:	What is the phone number?	`request(phone)`
S:	The number of Roma is 385456.	`inform(name = "Roma", phone = "385456")`
U:	Ok, thank you goodbye.	bye()

Figure 25.4 A sample dialogue from the HIS System of Young et al. (2010) using the dialogue acts in Fig. 25.3.

Forward Looking Function	
STATEMENT	a claim made by the speaker
INFO-REQUEST	a question by the speaker
CHECK	a question for confirming information
INFLUENCE-ON-ADDRESSEE	(=Bach's directives)
OPEN-OPTION	a weak suggestion or listing of options
ACTION-DIRECTIVE	an actual command
INFLUENCE-ON-SPEAKER	(=Austin's commissives)
OFFER	speaker offers to do something, (subject to confirmation)
COMMIT	speaker is committed to doing something
CONVENTIONAL	other
OPENING	greetings
CLOSING	farewells
THANKING	thanking and responding to thanks

The backward looking function of DAMSL focuses on the relationship of an utterance to previous utterances by the other speaker. These include accepting and rejecting proposals (since DAMSL is focused on task-oriented dialogue), and grounding and repair acts:

Backward Looking Function	
AGREEMENT	speaker's response to previous proposal
ACCEPT	accepting the proposal
ACCEPT-PART	accepting some part of the proposal
MAYBE	neither accepting nor rejecting the proposal
REJECT-PART	rejecting some part of the proposal
REJECT	rejecting the proposal
HOLD	putting off response, usually via subdialogue
ANSWER	answering a question
UNDERSTANDING	whether speaker understood previous
SIGNAL-NON-UNDER.	speaker didn't understand
SIGNAL-UNDER.	speaker did understand
ACK	demonstrated via continuer or assessment
REPEAT-REPHRASE	demonstrated via repetition or reformulation
COMPLETION	demonstrated via collaborative completion

Fig. 25.5 shows a labeling of parts of our sample conversation using versions of

the DAMSL Forward and Backward tags.

[assert]	C_1:	. . . I need to travel in May.
[info-req,ack]	A_2:	And, what day in May did you want to travel?
[assert, answer]	C_3:	OK uh I need to be there for a meeting that's from the 12th to the 15th.
[info-req,ack]	A_4:	And you're flying into what city?
[assert,answer]	C_5:	Seattle.
[info-req,ack]	A_6:	And what time would you like to leave Pittsburgh?
[check,hold]	C_7:	Uh hmm I don't think there's many options for non-stop.
[accept,ack]	A_7:	Right.
[assert]		There's three non-stops today.
[info-req]	C_8:	What are they?
[assert, open-option]	A_9:	The first one departs PGH at 10:00am arrives Seattle at 12:05 their time. The second flight departs PGH at 5:55pm, arrives Seattle at 8pm. And the last flight departs PGH at 8:15pm arrives Seattle at 10:28pm.
[accept,ack]	C_{10}:	OK I'll take the 5ish flight on the night before on the 11th.
[check,ack]	A_{11}:	On the 11th?
[assert,ack]		OK. Departing at 5:55pm arrives Seattle at 8pm, U.S. Air flight 115.
[ack]	C_{12}:	OK.

Figure 25.5 A potential DAMSL labeling of the beginning of the conversational fragment in Fig. 25.1.

Bibliography

Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Abney, S. P., R. E. Schapire, and Y. Singer. 1999. Boosting applied to tagging and PP attachment. *EMNLP/VLC*.

Agarwal, O., S. Subramanian, A. Nenkova, and D. Roth. 2019. Evaluation of named entity coreference. *Workshop on Computational Models of Reference, Anaphora and Coreference*.

Aggarwal, C. C. and C. Zhai. 2012. A survey of text classification algorithms. In C. C. Aggarwal and C. Zhai, eds, *Mining text data*, 163–222. Springer.

Agichtein, E. and L. Gravano. 2000. Snowball: Extracting relations from large plain-text collections. *Proceedings of the 5th ACM International Conference on Digital Libraries*.

Agirre, E., C. Banea, C. Cardie, D. Cer, M. Diab, A. Gonzalez-Agirre, W. Guo, I. Lopez-Gazpio, M. Maritxalar, R. Mihalcea, G. Rigau, L. Uria, and J. Wiebe. 2015. SemEval-2015 task 2: Semantic textual similarity, English, Spanish and pilot on interpretability. *SemEval-15*.

Agirre, E., M. Diab, D. Cer, and A. Gonzalez-Agirre. 2012. SemEval-2012 task 6: A pilot on semantic textual similarity. *SemEval-12*.

Agirre, E. and D. Martinez. 2001. Learning class-to-class selectional preferences. *CoNLL*.

Ahia, O., S. Kumar, H. Gonen, J. Kasai, D. Mortensen, N. Smith, and Y. Tsvetkov. 2023. Do all languages cost the same? tokenization in the era of commercial language models. *EMNLP*.

Aho, A. V. and J. D. Ullman. 1972. *The Theory of Parsing, Translation, and Compiling*, volume 1. Prentice Hall.

Algoet, P. H. and T. M. Cover. 1988. A sandwich proof of the Shannon-McMillan-Breiman theorem. *The Annals of Probability*, 16(2):899–909.

Allen, J. 1984. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–154.

Allen, J. and M. Core. 1997. Draft of DAMSL: Dialog act markup in several layers. Unpublished manuscript.

Allen, J., M. S. Hunnicut, and D. H. Klatt. 1987. *From Text to Speech: The MITalk system*. Cambridge University Press.

Allwood, J. 1995. An activity-based approach to pragmatics. *Gothenburg Papers in Theoretical Linguistics*, 76:1–38.

Allwood, J., J. Nivre, and E. Ahlsén. 1992. On the semantics and pragmatics of linguistic feedback. *Journal of Semantics*, 9:1–26.

Althoff, T., C. Danescu-Niculescu-Mizil, and D. Jurafsky. 2014. How to ask for a favor: A case study on the success of altruistic requests. *ICWSM 2014*.

An, J., H. Kwak, and Y.-Y. Ahn. 2018. SemAxis: A lightweight framework to characterize domain-specific word semantics beyond sentiment. *ACL*.

Anastasopoulos, A. and G. Neubig. 2020. Should all cross-lingual embeddings speak English? *ACL*.

Anthropic. 2025. Release notes: System prompts. `https://docs.anthropic.com/en/release-notes/system-prompts`.

Antoniak, M. and D. Mimno. 2018. Evaluating the stability of embedding-based word similarities. *TACL*, 6:107–119.

Aone, C. and S. W. Bennett. 1995. Evaluating automated and manual acquisition of anaphora resolution strategies. *ACL*.

Ardila, R., M. Branson, K. Davis, M. Kohler, J. Meyer, M. Henretty, R. Morais, L. Saunders, F. Tyers, and G. Weber. 2020. Common voice: A massively-multilingual speech corpus. *LREC*.

Ariel, M. 2001. Accessibility theory: An overview. In T. Sanders, J. Schilperoord, and W. Spooren, eds, *Text Representation: Linguistic and Psycholinguistic Aspects*, 29–87. Benjamins.

Arkadiev, P. M. 2020. *Morphology in typology: Historical retrospect, state of the art, and prospects*. Oxford.

Artetxe, M. and H. Schwenk. 2019. Massively multilingual sentence embeddings for zero-shot cross-lingual transfer and beyond. *TACL*, 7:597–610.

Asher, N. 1993. *Reference to Abstract Objects in Discourse*. Studies in Linguistics and Philosophy (SLAP) 50, Kluwer.

Asher, N. and A. Lascarides. 2003. *Logics of Conversation*. Cambridge University Press.

Atal, B. S. and S. Hanauer. 1971. Speech analysis and synthesis by prediction of the speech wave. *JASA*, 50:637–655.

Austin, J. L. 1962. *How to Do Things with Words*. Harvard University Press.

Ba, J. L., J. R. Kiros, and G. E. Hinton. 2016. Layer normalization. *NeurIPS workshop*.

Baayen, R. H. 2001. *Word frequency distributions*. Springer.

Baayen, R. H., R. Piepenbrock, and L. Gulikers. 1995. *The CELEX Lexical Database (Release 2) [CD-ROM]*. Linguistic Data Consortium, University of Pennsylvania [Distributor].

Babbage, C. 1864. *Passages from the Life of a Philosopher*. Longman.

Baccianella, S., A. Esuli, and F. Sebastiani. 2010. Sentiwordnet 3.0: An enhanced lexical resource for sentiment analysis and opinion mining. *LREC*.

Bach, K. and R. Harnish. 1979. *Linguistic communication and speech acts*. MIT Press.

Backus, J. W. 1959. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. *Information Processing: Proceedings of the International Conference on Information Processing, Paris*. UNESCO.

Backus, J. W. 1996. Transcript of question and answer session. In R. L. Wexelblat, ed., *History of Programming Languages*, page 162. Academic Press.

Bada, M., M. Eckert, D. Evans, K. Garcia, K. Shipley, D. Sitnikov, W. A. Baumgartner, K. B. Cohen, K. Verspoor, J. A. Blake, and L. E. Hunter. 2012. Concept annotation in the

craft corpus. *BMC bioinformatics*, 13(1):161.

Baevski, A., Y. Zhou, A. Mohamed, and M. Auli. 2020. wav2vec 2.0: A framework for self-supervised learning of speech representations. *NeurIPS*, volume 33.

Bagga, A. and B. Baldwin. 1998. Algorithms for scoring coreference chains. *LREC Workshop on Linguistic Coreference*.

Bahdanau, D., K. H. Cho, and Y. Bengio. 2015. Neural machine translation by jointly learning to align and translate. *ICLR 2015*.

Bahdanau, D., J. Chorowski, D. Serdyuk, P. Brakel, and Y. Bengio. 2016. End-to-end attention-based large vocabulary speech recognition. *ICASSP*.

Bahl, L. R. and R. L. Mercer. 1976. Part of speech assignment by a statistical decision algorithm. *Proceedings IEEE International Symposium on Information Theory*.

Bahl, L. R., F. Jelinek, and R. L. Mercer. 1983. A maximum likelihood approach to continuous speech recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(2):179–190.

Bai, Y., A. Jones, K. Ndousse, A. Askell, A. Chen, N. DasSarma, D. Drain, S. Fort, D. Ganguli, T. Henighan, N. Joseph, S. Kadavath, J. Kernion, T. Conerly, S. El-Showk, N. Elhage, Z. Hatfield-Dodds, D. Hernandez, T. Hume, S. Johnston, S. Kravec, L. Lovitt, N. Nanda, C. Olsson, D. Amodei, T. Brown, J. Clark, S. McCandlish, C. Olah, B. Mann, and J. Kaplan. 2022. Training a helpful and harmless assistant with reinforcement learning from human feedback.

Bajaj, P., D. Campos, N. Craswell, L. Deng, J. G. ando Xiaodong Liu, R. Majumder, A. McNamara, B. Mitra, T. Nguye, M. Rosenberg, X. Song, A. Stoica, S. Tiwary, and T. Wang. 2016. MS MARCO: A human generated MAchine Reading COmprehension dataset. *NeurIPS*.

Baker, C. F., C. J. Fillmore, and J. B. Lowe. 1998. The Berkeley FrameNet project. *COLING/ACL*.

Baker, J. K. 1975a. The DRAGON system – An overview. *IEEE Transactions on ASSP*, ASSP-23(1):24–29.

Baker, J. K. 1975b. Stochastic modeling for automatic speech understanding. In D. R. Reddy, ed., *Speech Recognition*. Academic Press.

Baldridge, J., N. Asher, and J. Hunter. 2007. Annotation for and robust parsing of discourse structure on unrestricted texts. *Zeitschrift für Sprachwissenschaft*, 26:213–239.

Bamman, D., O. Lewke, and A. Mansoor. 2020. An annotated dataset of coreference in English literature. *LREC*.

Bamman, D., B. O'Connor, and N. A. Smith. 2013. Learning latent personas of film characters. *ACL*.

Bamman, D., S. Popat, and S. Shen. 2019. An annotated dataset of literary entities. *NAACL HLT*.

Banerjee, S. and A. Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. *Proceedings of ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for MT and/or Summarization*.

Banko, M., M. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. 2007. Open information extraction for the web. *IJCAI*.

Bañón, M., P. Chen, B. Haddow, K. Heafield, H. Hoang, M. Esplà-Gomis, M. L. Forcada, A. Kamran, F. Kirefu, P. Koehn, S. Ortiz Rojas, L. Pla Sempere, G. Ramírez-Sánchez, E. Sarrías, M. Strelec, B. Thompson, W. Waites, D. Wiggins, and J. Zaragoza. 2020. ParaCrawl: Web-scale acquisition of parallel corpora. *ACL*.

Bar-Hillel, Y. 1960. The present status of automatic translation of languages. In F. Alt, ed., *Advances in Computers 1*, 91–163. Academic Press.

Barker, C. 2010. Nominals don't provide criteria of identity. In M. Rathert and A. Alexiadou, eds, *The Semantics of Nominalizations across Languages and Frameworks*, 9–24. Mouton.

Barrett, L. F., B. Mesquita, K. N. Ochsner, and J. J. Gross. 2007. The experience of emotion. *Annual Review of Psychology*, 58:373–403.

Barzilay, R. and M. Lapata. 2005. Modeling local coherence: An entity-based approach. *ACL*.

Barzilay, R. and M. Lapata. 2008. Modeling local coherence: An entity-based approach. *Computational Linguistics*, 34(1):1–34.

Barzilay, R. and L. Lee. 2004. Catching the drift: Probabilistic content models, with applications to generation and summarization. *HLT-NAACL*.

Baum, L. E. and J. A. Eagon. 1967. An inequality with applications to statistical estimation for probabilistic functions of Markov processes and to a model for ecology. *Bulletin of the American Mathematical Society*, 73(3):360–363.

Baum, L. E. and T. Petrie. 1966. Statistical inference for probabilistic functions of finite-state Markov chains. *Annals of Mathematical Statistics*, 37(6):1554–1563.

Bazell, C. E. 1952/1966. The correspondence fallacy in structural linguistics. In E. P. Hamp, F. W. Householder, and R. Austerlitz, eds, *Studies by Members of the English Department, Istanbul University (3), reprinted in Readings in Linguistics II (1966)*, 271–298. University of Chicago Press.

Bean, D. and E. Riloff. 1999. Corpus-based identification of non-anaphoric noun phrases. *ACL*.

Bean, D. and E. Riloff. 2004. Unsupervised learning of contextual role knowledge for coreference resolution. *HLT-NAACL*.

Beckman, M. E. and G. M. Ayers. 1997. Guidelines for ToBI labelling. Unpublished manuscript, Ohio State University, `http://www.ling.ohio-state.edu/research/phonetics/E_ToBI/`.

Beckman, M. E. and J. Hirschberg. 1994. The ToBI annotation conventions. Manuscript, Ohio State University.

Bedi, G., F. Carrillo, G. A. Cecchi, D. F. Slezak, M. Sigman, N. B. Mota, S. Ribeiro, D. C. Javitt, M. Copelli, and C. M. Corcoran. 2015. Automated analysis of free speech predicts psychosis onset in high-risk youths. *npj Schizophrenia*, 1.

Bejček, E., E. Hajičová, J. Hajič, P. Jínová, V. Kettnerová, V. Kolářová, M. Mikulová, J. Mírovský, A. Nedoluzhko, J. Panevová, L. Poláková, M. Ševčíková, J. Štěpánek, and Š. Zikánová. 2013. Prague dependency treebank 3.0. Technical report, Institute of Formal and Applied Linguistics, Charles University in Prague. LINDAT/CLARIN digital library at Institute of Formal and Applied Linguistics, Charles University in Prague.

Bellegarda, J. R. 1997. A latent semantic analysis framework for large-span language modeling. *EUROSPEECH*.

Bellegarda, J. R. 2000. Exploiting latent semantic information in statistical language modeling. *Proceedings of the IEEE*, 89(8):1279–1296.

Bellman, R. 1957. *Dynamic Programming*. Princeton University Press.

Bellman, R. 1984. *Eye of the Hurricane: an autobiography*. World Scientific Singapore.

Bender, E. M. 2019. The #BenderRule: On naming the languages we study and why it matters. Blog post.

Bender, E. M., B. Friedman, and A. McMillan-Major. 2021. A guide for writing data statements for natural language processing. http://techpolicylab.uw.edu/data-statements/.

Bender, E. M. and A. Koller. 2020. Climbing towards NLU: On meaning, form, and understanding in the age of data. *ACL*.

Bengio, Y., A. Courville, and P. Vincent. 2013. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828.

Bengio, Y., R. Ducharme, and P. Vincent. 2000. A neural probabilistic language model. *NeurIPS*.

Bengio, Y., R. Ducharme, P. Vincent, and C. Jauvin. 2003. A neural probabilistic language model. *JMLR*, 3:1137–1155.

Bengio, Y., P. Lamblin, D. Popovici, and H. Larochelle. 2007. Greedy layer-wise training of deep networks. *NeurIPS*.

Bengio, Y., H. Schwenk, J.-S. Senécal, F. Morin, and J.-L. Gauvain. 2006. Neural probabilistic language models. In *Innovations in Machine Learning*, 137–186. Springer.

Bengtson, E. and D. Roth. 2008. Understanding the value of features for coreference resolution. *EMNLP*.

Bennett, R. and E. Elfner. 2019. The syntax–prosody interface. *Annual Review of Linguistics*, 5:151–171.

Bentivogli, L., M. Cettolo, M. Federico, and C. Federmann. 2018. Machine translation human evaluation: an investigation of evaluation based on post-editing and its relation with direct assessment. *ICSLT*.

Berant, J., A. Chou, R. Frostig, and P. Liang. 2013. Semantic parsing on freebase from question-answer pairs. *EMNLP*.

Berg-Kirkpatrick, T., D. Burkett, and D. Klein. 2012. An empirical investigation of statistical significance in NLP. *EMNLP*.

Berger, A., S. A. Della Pietra, and V. J. Della Pietra. 1996. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–71.

Bergsma, S. and D. Lin. 2006. Bootstrapping path-based pronoun resolution. *COLING/ACL*.

Bergsma, S., D. Lin, and R. Goebel. 2008a. Discriminative learning of selectional preference from unlabeled text. *EMNLP*.

Bergsma, S., D. Lin, and R. Goebel. 2008b. Distributional identification of non-referential pronouns. *ACL*.

Bethard, S. 2013. ClearTK-TimeML: A minimalist approach to TempEval 2013. *SemEval-13*.

Bhat, I., R. A. Bhat, M. Shrivastava, and D. Sharma. 2017. Joining hands: Exploiting monolingual treebanks for parsing of code-mixing data. *EACL*.

Bianchi, F., M. Suzgun, G. Attanasio, P. Rottger, D. Jurafsky, T. Hashimoto, and J. Zou. 2024. Safety-tuned LLaMAs: Lessons from improving the safety of large language models that follow instructions. *ICLR*.

Bickel, B. 2003. Referential density in discourse and syntactic typology. *Language*, 79(2):708–736.

Bickmore, T. W., H. Trinh, S. Olafsson, T. K. O'Leary, R. Asadi, N. M. Rickles, and R. Cruz. 2018. Patient and consumer safety risks when using conversational assistants for medical information: An observational study of Siri, Alexa, and Google Assistant. *Journal of Medical Internet Research*, 20(9):e11510.

Bikel, D. M., S. Miller, R. Schwartz, and R. Weischedel. 1997. Nymble: A high-performance learning name-finder. *ANLP*.

Biran, O. and K. McKeown. 2015. PDTB discourse parsing as a tagging task: The two taggers approach. *SIGDIAL*.

Bird, S., E. Klein, and E. Loper. 2009. *Natural Language Processing with Python*. O'Reilly.

Bisani, M. and H. Ney. 2004. Bootstrap estimates for confidence intervals in ASR performance evaluation. *ICASSP*.

Bishop, C. M. 2006. *Pattern recognition and machine learning*. Springer.

Bisk, Y., A. Holtzman, J. Thomason, J. Andreas, Y. Bengio, J. Chai, M. Lapata, A. Lazaridou, J. May, A. Nisnevich, N. Pinto, and J. Turian. 2020. Experience grounds language. *EMNLP*.

Bizer, C., J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. 2009. DBpedia—A crystallization point for the Web of Data. *Web Semantics: science, services and agents on the world wide web*, 7(3):154–165.

Björkelund, A. and J. Kuhn. 2014. Learning structured perceptrons for coreference resolution with latent antecedents and non-local features. *ACL*.

Black, A. W. and P. Taylor. 1994. CHATR: A generic speech synthesis system. *COLING*.

Black, E., S. P. Abney, D. Flickinger, C. Gdaniec, R. Grishman, P. Harrison, D. Hindle, R. Ingria, F. Jelinek, J. L. Klavans, M. Y. Liberman, M. P. Marcus, S. Roukos, B. Santorini, and T. Strzalkowski. 1991. A procedure for quantitatively comparing the syntactic coverage of English grammars. *Speech and Natural Language Workshop*.

Blei, D. M., A. Y. Ng, and M. I. Jordan. 2003. Latent Dirichlet allocation. *JMLR*, 3(5):993–1022.

Blodgett, S. L., S. Barocas, H. Daumé III, and H. Wallach. 2020. Language (technology) is power: A critical survey of "bias" in NLP. *ACL*.

Blodgett, S. L., L. Green, and B. O'Connor. 2016. Demographic dialectal variation in social media: A case study of African-American English. *EMNLP*.

Blodgett, S. L. and B. O'Connor. 2017. Racial disparity in natural language processing: A case study of social media African-American English. *FAT/ML Workshop, KDD*.

Bloomfield, L. 1914. *An Introduction to the Study of Language*. Henry Holt and Company.

Bloomfield, L. 1933. *Language*. University of Chicago Press.

Bobrow, D. G., R. M. Kaplan, M. Kay, D. A. Norman, H. Thompson, and T. Winograd. 1977. GUS, A frame driven dialog system. *Artificial Intelligence*, 8:155–173.

Bobrow, D. G. and D. A. Norman. 1975. Some principles of memory schemata. In D. G. Bobrow and A. Collins, eds, *Representation and Understanding*. Academic Press.

Boersma, P. and D. Weenink. 2005. Praat: doing phonetics by computer (version 4.3.14). [Computer program]. Retrieved May 26, 2005, from http://www.praat.org/.

Bojanowski, P., E. Grave, A. Joulin, and T. Mikolov. 2017. Enriching word vectors with subword information. *TACL*, 5:135–146.

Bollacker, K., C. Evans, P. Paritosh, T. Sturge, and J. Taylor. 2008. Freebase: a collaboratively created graph database for structuring human knowledge. *SIGMOD 2008*.

Bolukbasi, T., K.-W. Chang, J. Zou, V. Saligrama, and A. T. Kalai. 2016. Man is to computer programmer as woman is to homemaker? Debiasing word embeddings. *NeurIPS*.

Bommasani, R., D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, E. Brynjolfsson, S. Buch,

D. Card, R. Castellon, N. S. Chatterji, A. S. Chen, K. A. Creel, J. Davis, D. Demszky, C. Donahue, M. Doumbouya, E. Durmus, S. Ermon, J. Etchemendy, K. Ethayarajh, L. Fei-Fei, C. Finn, T. Gale, L. E. Gillespie, K. Goel, N. D. Goodman, S. Grossman, N. Guha, T. Hashimoto, P. Henderson, J. Hewitt, D. E. Ho, J. Hong, K. Hsu, J. Huang, T. F. Icard, S. Jain, D. Jurafsky, P. Kalluri, S. Karamcheti, G. Keeling, F. Khani, O. Khattab, P. W. Koh, M. S. Krass, R. Krishna, R. Kuditipudi, A. Kumar, F. Ladhak, M. Lee, T. Lee, J. Leskovec, I. Levent, X. L. Li, X. Li, T. Ma, A. Malik, C. D. Manning, S. P. Mirchandani, E. Mitchell, Z. Munyikwa, S. Nair, A. Narayan, D. Narayanan, B. Newman, A. Nie, J. C. Niebles, H. Nilforoshan, J. F. Nyarko, G. Ogut, L. Orr, I. Papadimitriou, J. S. Park, C. Piech, E. Portelance, C. Potts, A. Raghunathan, R. Reich, H. Ren, F. Rong, Y. H. Roohani, C. Ruiz, J. Ryan, C. R'e, D. Sadigh, S. Sagawa, K. Santhanam, A. Shih, K. P. Srinivasan, A. Tamkin, R. Taori, A. W. Thomas, F. Tramèr, R. E. Wang, W. Wang, B. Wu, J. Wu, Y. Wu, S. M. Xie, M. Yasunaga, J. You, M. A. Zaharia, M. Zhang, T. Zhang, X. Zhang, Y. Zhang, L. Zheng, K. Zhou, and P. Liang. 2021. On the opportunities and risks of foundation models. *ArXiv*.

Booth, T. L. 1969. Probabilistic representation of formal languages. *IEEE Conference Record of the 1969 Tenth Annual Symposium on Switching and Automata Theory*.

Borges, J. L. 1964. The analytical language of john wilkins. In *Other inquisitions 1937–1952*. University of Texas Press. Trans. Ruth L. C. Simms.

Bostrom, K. and G. Durrett. 2020. Byte pair encoding is suboptimal for language model pretraining. *EMNLP*.

Bourlard, H. and N. Morgan. 1994. *Connectionist Speech Recognition: A Hybrid Approach*. Kluwer.

Bradley, R. A. and M. E. Terry. 1952. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39:324–345.

Brants, T. 2000. TnT: A statistical part-of-speech tagger. *ANLP*.

Brants, T., A. C. Popat, P. Xu, F. J. Och, and J. Dean. 2007. Large language models in machine translation. *EMNLP/CoNLL*.

Braud, C., M. Coavoux, and A. Søgaard. 2017. Cross-lingual RST discourse parsing. *EACL*.

Bréal, M. 1897. *Essai de Sémantique: Science des significations*. Hachette.

Brennan, S. E., M. W. Friedman, and C. Pollard. 1987. A centering approach to pronouns. *ACL*.

Brin, S. 1998. Extracting patterns and relations from the World Wide Web. *Proceedings World Wide Web and Databases International Workshop, Number 1590 in LNCS*. Springer.

Brockmann, C. and M. Lapata. 2003. Evaluating and combining approaches to selectional preference acquisition. *EACL*.

Broschart, J. 1997. Why Tongan does it differently. *Linguistic Typology*, 1:123–165.

Brown, P. F., J. Cocke, S. A. Della Pietra, V. J. Della Pietra, F. Jelinek, J. D. Lafferty, R. L. Mercer, and P. S. Roossin. 1990. A statistical approach to machine translation. *Computational Linguistics*, 16(2):79–85.

Brown, P. F., S. A. Della Pietra, V. J. Della Pietra, and R. L. Mercer. 1993. The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics*, 19(2):263–311.

Brown, T., B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. 2020. Language models are few-shot learners. *NeurIPS*, volume 33.

Brysbaert, M., A. B. Warriner, and V. Kuperman. 2014. Concreteness ratings for 40 thousand generally known English word lemmas. *Behavior Research Methods*, 46(3):904–911.

Bu, H., J. Du, X. Na, B. Wu, and H. Zheng. 2017. AISHELL-1: An open-source Mandarin speech corpus and a speech recognition baseline. *O-COCOSDA Proceedings*.

Buchholz, S. and E. Marsi. 2006. Conll-x shared task on multilingual dependency parsing. *CoNLL*.

Budanitsky, A. and G. Hirst. 2006. Evaluating WordNet-based measures of lexical semantic relatedness. *Computational Linguistics*, 32(1):13–47.

Bullinaria, J. A. and J. P. Levy. 2007. Extracting semantic representations from word co-occurrence statistics: A computational study. *Behavior research methods*, 39(3):510–526.

Bullinaria, J. A. and J. P. Levy. 2012. Extracting semantic representations from word co-occurrence statistics: stop-lists, stemming, and SVD. *Behavior research methods*, 44(3):890–907.

Caliskan, A., J. J. Bryson, and A. Narayanan. 2017. Semantics derived automatically from language corpora contain human-like biases. *Science*, 356(6334):183–186.

Callison-Burch, C., M. Osborne, and P. Koehn. 2006. Re-evaluating the role of BLEU in machine translation research. *EACL*.

Canavan, A., D. Graff, and G. Zipperlen. 1997. CALLHOME American English speech LDC97S42. Linguistic Data Consortium.

Cao, Y., Y. Kang, C. Wang, and L. Sun. 2024. Instruction mining: Instruction data selection for tuning large language models. *First Conference on Language Modeling*.

Carbonell, J. R. 1970. AI in CAI: An artificial-intelligence approach to computer-assisted instruction. *IEEE transactions on man-machine systems*, 11(4):190–202.

Cardie, C. 1993. A case-based approach to knowledge acquisition for domain specific sentence analysis. *AAAI*.

Cardie, C. 1994. *Domain-Specific Knowledge Acquisition for Conceptual Sentence Analysis*. Ph.D. thesis, University of Massachusetts, Amherst, MA. Available as CMP-SCI Technical Report 94-74.

Cardie, C. and K. Wagstaff. 1999. Noun phrase coreference as clustering. *EMNLP/VLC*.

Carletta, J., N. Dahlbäck, N. Reithinger, and M. A. Walker. 1997. Standards for dialogue coding in natural language processing. Technical Report 167, Dagstuhl Seminars. Report from Dagstuhl seminar number 9706.

Carlini, N., F. Tramer, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. Brown, D. Song, U. Erlingsson, et al. 2021. Extracting training data from large language models. *30th USENIX Security Symposium (USENIX Security 21)*.

Carlson, G. N. 1977. *Reference to kinds in English*. Ph.D. thesis, University of Massachusetts, Amherst. Forward.

Carlson, L. and D. Marcu. 2001. Discourse tagging manual. Technical Report ISI-TR-545, ISI.

Carlson, L., D. Marcu, and M. E. Okurowski. 2001. Building a discourse-tagged corpus in the framework of rhetorical structure theory. *SIGDIAL*.

Carreras, X. and L. Màrquez. 2005. Introduction to the CoNLL-2005 shared task: Semantic role labeling. *CoNLL*.

Chafe, W. L. 1976. Givenness, contrastiveness, definiteness, subjects, topics, and point of view. In C. N. Li, ed., *Subject and Topic*, 25–55. Academic Press.

Chambers, N. 2013. NavyTime: Event and time ordering from raw text. *SemEval-13*.

Chambers, N., T. Cassidy, B. McDowell, and S. Bethard. 2014. Dense event ordering with a multi-pass architecture. *TACL*, 2:273–284.

Chambers, N. and D. Jurafsky. 2010. Improving the use of pseudo-words for evaluating selectional preferences. *ACL*.

Chambers, N. and D. Jurafsky. 2011. Template-based information extraction without the templates. *ACL*.

Chan, W., N. Jaitly, Q. Le, and O. Vinyals. 2016. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. *ICASSP*.

Chandioux, J. 1976. MÉTÉO: un système opérationnel pour la traduction automatique des bulletins météorologiques destinés au grand public. *Meta*, 21:127–133.

Chang, A. X. and C. D. Manning. 2012. SUTime: A library for recognizing and normalizing time expressions. *LREC*.

Chang, K.-W., R. Samdani, and D. Roth. 2013. A constrained latent variable model for coreference resolution. *EMNLP*.

Chang, K.-W., R. Samdani, A. Rozovskaya, M. Sammons, and D. Roth. 2012. Illinois-Coref: The UI system in the CoNLL-2012 shared task. *CoNLL*.

Chaplot, D. S. and R. Salakhutdinov. 2018. Knowledge-based word sense disambiguation using topic models. *AAAI*.

Charniak, E. 1997. Statistical parsing with a context-free grammar and word statistics. *AAAI*.

Charniak, E., C. Hendrickson, N. Jacobson, and M. Perkowitz. 1993. Equations for part-of-speech tagging. *AAAI*.

Che, W., Z. Li, Y. Li, Y. Guo, B. Qin, and T. Liu. 2009. Multilingual dependency-based syntactic and semantic parsing. *CoNLL*.

Chen, C. and V. Ng. 2013. Linguistically aware coreference evaluation metrics. *IJCNLP*.

Chen, D., A. Fisch, J. Weston, and A. Bordes. 2017a. Reading Wikipedia to answer open-domain questions. *ACL*.

Chen, D. and C. Manning. 2014. A fast and accurate dependency parser using neural networks. *EMNLP*.

Chen, E., B. Snyder, and R. Barzilay. 2007. Incremental text structuring with online hierarchical ranking. *EMNLP/CoNLL*.

Chen, S., C. Wang, Y. Wu, Z. Zhang, L. Zhou, S. Liu, Z. Chen, Y. Liu, H. Wang, J. Li, L. He, S. Zhao, and F. Wei. 2025. Neural codec language models are zero-shot text to speech synthesizers. *IEEE Trans. on TASLP*, 33:705–718.

Chen, S. F. and J. Goodman. 1999. An empirical study of smoothing techniques for language modeling. *Computer Speech and Language*, 13:359–394.

Chen, X., Z. Shi, X. Qiu, and X. Huang. 2017b. Adversarial multi-criteria learning for Chinese word segmentation. *ACL*.

Cheng, J., L. Dong, and M. Lapata. 2016. Long short-term memory-networks for machine reading. *EMNLP*.

Cheng, M., E. Durmus, and D. Jurafsky. 2023. Marked personas: Using natural language prompts to measure stereotypes in language models. *ACL*.

Cheng, M., C. Lee, P. Khadpe, S. Yu, D. Han, and D. Jurafsky. 2025. Sycophantic ai decreases prosocial intentions and promotes dependence. ArXiv preprint.

Chiang, D. 2005. A hierarchical phrase-based model for statistical machine translation. *ACL*.

Chinchor, N., L. Hirschman, and D. L. Lewis. 1993. Evaluating Message Understanding systems: An analysis of the third Message Understanding Conference. *Computational Linguistics*, 19(3):409–449.

Chiticariu, L., M. Danilevsky, Y. Li, F. Reiss, and H. Zhu. 2018. SystemT: Declarative text understanding for enterprise. *NAACL HLT*, volume 3.

Chiticariu, L., Y. Li, and F. R. Reiss. 2013. Rule-Based Information Extraction is Dead! Long Live Rule-Based Information Extraction Systems! *EMNLP*.

Chiu, J. P. C. and E. Nichols. 2016. Named entity recognition with bidirectional LSTM-CNNs. *TACL*, 4:357–370.

Cho, K., B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. 2014. Learning phrase representations using RNN encoder–decoder for statistical machine translation. *EMNLP*.

Choe, D. K. and E. Charniak. 2016. Parsing as language modeling. *EMNLP*.

Choi, J. D. and M. Palmer. 2011a. Getting the most out of transition-based dependency parsing. *ACL*.

Choi, J. D. and M. Palmer. 2011b. Transition-based semantic role labeling using predicate argument clustering. *Proceedings of the ACL 2011 Workshop on Relational Models of Semantics*.

Choi, J. D., J. Tetreault, and A. Stent. 2015. It depends: Dependency parser comparison using a web-based evaluation tool. *ACL*.

Chomsky, N. 1956. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124.

Chomsky, N. 1956/1975. *The Logical Structure of Linguistic Theory*. Plenum.

Chomsky, N. 1957. *Syntactic Structures*. Mouton.

Chomsky, N. 1963. Formal properties of grammars. In R. D. Luce, R. Bush, and E. Galanter, eds, *Handbook of Mathematical Psychology*, volume 2, 323–418. Wiley.

Chomsky, N. 1981. *Lectures on Government and Binding*. Foris.

Chorowski, J., D. Bahdanau, K. Cho, and Y. Bengio. 2014. End-to-end continuous speech recognition using attention-based recurrent NN: First results. *NeurIPS Deep Learning and Representation Learning Workshop*.

Chou, W., C.-H. Lee, and B. H. Juang. 1993. Minimum error rate training based on *n*-best string models. *ICASSP*.

Christodoulopoulos, C., S. Goldwater, and M. Steedman. 2010. Two decades of unsupervised POS induction: How far have we come? *EMNLP*.

Chu, Y.-J. and T.-H. Liu. 1965. On the shortest arborescence of a directed graph. *Science Sinica*, 14:1396–1400.

Chu-Carroll, J. and S. Carberry. 1998. Collaborative response generation in planning dialogues. *Computational Linguistics*, 24(3):355–400.

Church, K. W. 1988. A stochastic parts program and noun phrase parser for unrestricted text. *ANLP*.

Church, K. W. 1989. A stochastic parts program and noun phrase parser for unrestricted text. *ICASSP*.

Church, K. W. 1994. Unix for Poets. Slides from 2nd ELSNET Summer School and unpublished paper ms.

Church, K. W. and W. A. Gale. 1991. A comparison of the enhanced Good-Turing and deleted estimation methods for estimating probabilities of English bigrams. *Computer Speech and Language*, 5:19–54.

Cialdini, R. B. 1984. *Influence: The psychology of persuasion*. Morrow.

Cieri, C., D. Miller, and K. Walker. 2004. The Fisher corpus: A resource for the next generations of speech-to-text. *LREC*.

Clark, E. 1987. The principle of contrast: A constraint on language acquisition. In B. MacWhinney, ed., *Mechanisms of language acquisition*, 1–33. LEA.

Clark, H. H. 1996. *Using Language*. Cambridge University Press.

Clark, H. H. and J. E. Fox Tree. 2002. Using uh and um in spontaneous speaking. *Cognition*, 84:73–111.

Clark, H. H. and E. F. Schaefer. 1989. Contributing to discourse. *Cognitive Science*, 13:259–294.

Clark, J. and C. Yallop. 1995. *An Introduction to Phonetics and Phonology*, 2nd edition. Blackwell.

Clark, J. H., E. Choi, M. Collins, D. Garrette, T. Kwiatkowski, V. Nikolaev, and J. Palomaki. 2020a. TyDi QA: A benchmark for information-seeking question answering in typologically diverse languages. *TACL*, 8:454–470.

Clark, K., M.-T. Luong, Q. V. Le, and C. D. Manning. 2020b. Electra: Pre-training text encoders as discriminators rather than generators. *ICLR*.

Clark, K. and C. D. Manning. 2015. Entity-centric coreference resolution with model stacking. *ACL*.

Clark, K. and C. D. Manning. 2016a. Deep reinforcement learning for mention-ranking coreference models. *EMNLP*.

Clark, K. and C. D. Manning. 2016b. Improving coreference resolution by learning entity-level distributed representations. *ACL*.

Clark, S., J. R. Curran, and M. Osborne. 2003. Bootstrapping POS-taggers using unlabelled data. *CoNLL*.

CMU. 1993. The Carnegie Mellon Pronouncing Dictionary v0.1. Carnegie Mellon University.

Cobbe, K., V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, C. Hesse, and J. Schulman. 2021. Training verifiers to solve math word problems. ArXiv preprint.

Coccaro, N. and D. Jurafsky. 1998. Towards better integration of semantic predictors in statistical language modeling. *ICSLP*.

Coenen, A., E. Reif, A. Yuan, B. Kim, A. Pearce, F. Viégas, and M. Wattenberg. 2019. Visualizing and measuring the geometry of bert. *NeurIPS*.

Coleman, J. 2005. *Introducing Speech and Language Processing*. Cambridge University Press.

Collins, M. 1999. *Head-Driven Statistical Models for Natural Language Parsing*. Ph.D. thesis, University of Pennsylvania, Philadelphia.

Collobert, R. and J. Weston. 2007. Fast semantic extraction using a novel neural network architecture. *ACL*.

Collobert, R. and J. Weston. 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. *ICML*.

Collobert, R., J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. 2011. Natural language processing (almost) from scratch. *JMLR*, 12:2493–2537.

Comrie, B. 1989. *Language Universals and Linguistic Typology*, 2nd edition. Blackwell.

Conneau, A., K. Khandelwal, N. Goyal, V. Chaudhary, G. Wenzek, F. Guzmán, E. Grave, M. Ott, L. Zettlemoyer, and V. Stoyanov. 2020. Unsupervised cross-lingual representation learning at scale. *ACL*.

Conneau, A., M. Ma, S. Khanuja, Y. Zhang, V. Axelrod, S. Dalmia, J. Riesa, C. Rivera, and A. Bapna. 2023. Fleurs: Few-shot learning evaluation of universal representations of speech. *IEEE SLT*.

Connolly, D., J. D. Burger, and D. S. Day. 1994. A machine learning approach to anaphoric reference. *Proceedings of the International Conference on New Methods in Language Processing (NeMLaP)*.

Cooley, J. W. and J. W. Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301.

Cooper, F. S., A. M. Liberman, and J. M. Borst. 1951. The interconversion of audible and visible patterns as a basis for research in the perception of speech. *Proceedings of the National Academy of Sciences*, 37(5):318–325.

Cordier, B. 1965. Factor-analysis of correspondences. *COLING 1965*.

Core, M., M. Ishizaki, J. D. Moore, C. Nakatani, N. Reithinger, D. R. Traum, and S. Tutiya. 1999. The Report of the 3rd workshop of the Discourse Resource Initiative. Technical Report No.3 CC-TR-99-1, Chiba Corpus Project, Chiba, Japan.

Costa-jussà, M. R., J. Cross, O. Çelebi, M. Elbayad, K. Heafield, K. Heffernan, E. Kalbassi, J. Lam, D. Licht, J. Maillard, A. Sun, S. Wang, G. Wenzek, A. Youngblood, B. Akula, L. Barrault, G. M. Gonzalez, P. Hansanti, J. Hoffman, S. Jarrett, K. R. Sadagopan, D. Rowe, S. Spruit, C. Tran, P. Andrews, N. F. Ayan, S. Bhosale, S. Edunov, A. Fan, C. Gao, V. Goswami, F. Guzmán, P. Koehn, A. Mourachko, C. Ropers, S. Saleem, H. Schwenk, J. Wang, and NLLB Team. 2022. No language left behind: Scaling human-centered machine translation. ArXiv.

Cover, T. M. and J. A. Thomas. 1991. *Elements of Information Theory*. Wiley.

Covington, M. 2001. A fundamental algorithm for dependency parsing. *Proceedings of the 39th Annual ACM Southeast Conference*.

Cox, D. 1969. *Analysis of Binary Data*. Chapman and Hall, London.

Craven, M. and J. Kumlien. 1999. Constructing biological knowledge bases by extracting information from text sources. *ISMB-99*.

Crawford, K. 2017. The trouble with bias. Keynote at NeurIPS.

Croft, W. 1990. *Typology and Universals*. Cambridge University Press.

Crosbie, J. and E. Shutova. 2022. Induction heads as an essential mechanism for pattern matching in in-context learning. ArXiv preprint.

Cross, J. and L. Huang. 2016. Span-based constituency parsing with a structure-label system and provably optimal dynamic oracles. *EMNLP*.

Cruse, D. A. 2004. *Meaning in Language: an Introduction to Semantics and Pragmatics*. Oxford University Press. Second edition.

Cucerzan, S. 2007. Large-scale named entity disambiguation based on Wikipedia data. *EMNLP/CoNLL*.

Cui, G., L. Yuan, N. Ding, G. Yao, B. He, W. Zhu, Y. Ni, G. Xie, R. Xie, Y. Lin, Z. Liu, and M. Sun. 2024. Ultrafeedback: boosting language models with scaled ai feedback. *ICML 2024*.

Dahl, G. E., T. N. Sainath, and G. E. Hinton. 2013. Improving deep neural networks for LVCSR using rectified linear units and dropout. *ICASSP*.

Dahl, G. E., D. Yu, L. Deng, and A. Acero. 2012. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on audio, speech, and language processing*, 20(1):30–42.

Dahl, M., V. Magesh, M. Suzgun, and D. E. Ho. 2024. Large legal fictions: Profiling legal hallucinations in large language models. *Journal of Legal Analysis*, 16:64–93.

Dai, A. M. and Q. V. Le. 2015. Semi-supervised sequence learning. *NeurIPS*.

Das, S. R. and M. Y. Chen. 2001. Yahoo! for Amazon: Sentiment parsing from small talk on the web. EFA 2001 Barcelona Meetings. http://ssrn.com/abstract=276189.

David, Jr., E. E. and O. G. Selfridge. 1962. Eyes and ears for computers. *Proceedings of the IRE (Institute of Radio Engineers)*, 50:1093–1101.

Davidson, T., D. Bhattacharya, and I. Weber. 2019. Racial bias in hate speech and abusive language detection datasets. *Third Workshop on Abusive Language Online*.

Davies, M. 2012. Expanding horizons in historical linguistics with the 400-million word Corpus of Historical American English. *Corpora*, 7(2):121–157.

Davies, M. 2015. The Wikipedia Corpus: 4.6 million articles, 1.9 billion words. Adapted from Wikipedia. https://www.english-corpora.org/wiki/.

Davies, M. 2020. The Corpus of Contemporary American English (COCA): One billion words, 1990-2019. https://www.english-corpora.org/coca/.

Davis, E., L. Morgenstern, and C. L. Ortiz. 2017. The first Winograd schema challenge at IJCAI-16. *AI Magazine*, 38(3):97–98.

Davis, K. H., R. Biddulph, and S. Balashek. 1952. Automatic recognition of spoken digits. *JASA*, 24(6):637–642.

Davis, S. and P. Mermelstein. 1980. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE Transactions on ASSP*, 28(4):357–366.

Deerwester, S. C., S. T. Dumais, G. W. Furnas, R. A. Harshman, T. K. Landauer, K. E. Lochbaum, and L. Streeter. 1988. Computer information retrieval using latent semantic structure: US Patent 4,839,853.

Deerwester, S. C., S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. 1990. Indexing by latent semantics analysis. *JASIS*, 41(6):391–407.

Défossez, A., J. Copet, G. Synnaeve, and Y. Adi. 2023. High fidelity neural audio compression. *TMLR*.

DeJong, G. F. 1982. An overview of the FRUMP system. In W. G. Lehnert and M. H. Ringle, eds, *Strategies for Natural Language Processing*, 149–176. LEA.

Denes, P. 1959. The design and operation of the mechanical speech recognizer at University College London. *Journal of the British Institution of Radio Engineers*, 19(4):219–234. Appears together with companion paper (Fry 1959).

Deng, L., G. Hinton, and B. Kingsbury. 2013. New types of deep neural network learning for speech recognition and related applications: An overview. *ICASSP*.

Deng, Y. and W. Byrne. 2005. HMM word and phrase alignment for statistical machine translation. *HLT-EMNLP*.

Denis, P. and J. Baldridge. 2007. Joint determination of anaphoricity and coreference resolution using integer programming. *NAACL-HLT*.

Denis, P. and J. Baldridge. 2008. Specialized models and ranking for coreference resolution. *EMNLP*.

Denis, P. and J. Baldridge. 2009. Global joint models for coreference resolution and named entity classification. *Procesamiento del Lenguaje Natural*, 42.

DeRose, S. J. 1988. Grammatical category disambiguation by statistical optimization. *Computational Linguistics*, 14:31–39.

Devlin, J., M.-W. Chang, K. Lee, and K. Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. *NAACL HLT*.

Di Eugenio, B. 1990. Centering theory and the Italian pronominal system. *COLING*.

Di Eugenio, B. 1996. The discourse functions of Italian subjects: A centering approach. *COLING*.

Dias Oliva, T., D. Antonialli, and A. Gomes. 2021. Fighting hate speech, silencing drag queens? artificial intelligence in content moderation and risks to lgbtq voices online. *Sexuality & Culture*, 25:700–732.

Dinan, E., G. Abercrombie, A. S. Bergman, S. Spruit, D. Hovy, Y.-L. Boureau, and V. Rieser. 2021. Anticipating safety issues in e2e conversational ai: Framework and tooling. ArXiv.

Dinan, E., A. Fan, A. Williams, J. Urbanek, D. Kiela, and J. Weston. 2020. Queens are powerful too: Mitigating gender bias in dialogue generation. *EMNLP*.

Ditman, T. and G. R. Kuperberg. 2010. Building coherence: A framework for exploring the breakdown of links across clause boundaries in schizophrenia. *Journal of neurolinguistics*, 23(3):254–269.

Dixon, L., J. Li, J. Sorensen, N. Thain, and L. Vasserman. 2018. Measuring and mitigating unintended bias in text classification. *2018 AAAI/ACM Conference on AI, Ethics, and Society*.

Dixon, N. and H. Maxey. 1968. Terminal analog synthesis of continuous speech using the diphone method of segment assembly. *IEEE Transactions on Audio and Electroacoustics*, 16(1):40–50.

Do, Q. N. T., S. Bethard, and M.-F. Moens. 2017. Improving implicit semantic role labeling by predicting semantic frame arguments. *IJCNLP*.

Doddington, G. 2002. Automatic evaluation of machine translation quality using n-gram co-occurrence statistics. *HLT*.

Dodge, J., S. Gururangan, D. Card, R. Schwartz, and N. A. Smith. 2019. Show your work: Improved reporting of experimental results. *EMNLP*.

Dodge, J., M. Sap, A. Marasović, W. Agnew, G. Ilharco, D. Groeneveld, M. Mitchell, and M. Gardner. 2021. Documenting large webtext corpora: A case study on the colossal clean crawled corpus. *EMNLP*.

Dong, L. and M. Lapata. 2016. Language to logical form with neural attention. *ACL*.

Dorr, B. 1994. Machine translation divergences: A formal description and proposed solution. *Computational Linguistics*, 20(4):597–633.

Dostert, L. 1955. The Georgetown-I.B.M. experiment. In *Machine Translation of Languages: Fourteen Essays*, 124–135. MIT Press.

Doumbouya, M. K. B., D. Jurafsky, and C. D. Manning. 2025. Tversky neural networks: Psychologically plausible deep learning with differentiable tversky similarity. ArXiv preprint.

Dowty, D. R. 1979. *Word Meaning and Montague Grammar*. D. Reidel.

Dozat, T. and C. D. Manning. 2017. Deep biaffine attention for neural dependency parsing. *ICLR*.

Dozat, T. and C. D. Manning. 2018. Simpler but more accurate semantic dependency parsing. *ACL*.

Dozat, T., P. Qi, and C. D. Manning. 2017. Stanford's graph-based neural dependency parser at the CoNLL 2017 shared task. *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*.

Dror, R., G. Baumer, M. Bogomolov, and R. Reichart. 2017. Replicability analysis for natural language processing: Testing significance with multiple datasets. *TACL*, 5:471––486.

Dror, R., L. Peled-Cohen, S. Shlomov, and R. Reichart. 2020. *Statistical Significance Testing for Natural Language Processing*, volume 45 of *Synthesis Lectures on Human Language Technologies*. Morgan & Claypool.

Dryer, M. S. and M. Haspelmath, eds. 2013. *The World Atlas of Language Structures Online*. Max Planck Institute for Evolutionary Anthropology, Leipzig. Available online at http://wals.info.

Durrett, G. and D. Klein. 2013. Easy victories and uphill battles in coreference resolution. *EMNLP*.

Durrett, G. and D. Klein. 2014. A joint model for entity analysis: Coreference, typing, and linking. *TACL*, 2:477–490.

Earley, J. 1968. *An Efficient Context-Free Parsing Algorithm*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA.

Earley, J. 1970. An efficient context-free parsing algorithm. *CACM*, 6(8):451–455.

Edmonds, J. 1967. Optimum branchings. *Journal of Research of the National Bureau of Standards B*, 71(4):233–240.

Edunov, S., M. Ott, M. Auli, and D. Grangier. 2018. Understanding back-translation at scale. *EMNLP*.

Efron, B. and R. J. Tibshirani. 1993. *An introduction to the bootstrap*. CRC press.

Egghe, L. 2007. Untangling Herdan's law and Heaps' law: Mathematical and informetric arguments. *JASIST*, 58(5):702–709.

Eisner, J. 1996. Three new probabilistic models for dependency parsing: An exploration. *COLING*.

Ekman, P. 1999. Basic emotions. In T. Dalgleish and M. J. Power, eds, *Handbook of Cognition and Emotion*, 45–60. Wiley.

Elhage, N., N. Nanda, C. Olsson, T. Henighan, N. Joseph, B. Mann, A. Askell, Y. Bai, A. Chen, T. Conerly, N. DasSarma, D. Drain, D. Ganguli, Z. Hatfield-Dodds, D. Hernandez, A. Jones, J. Kernion, L. Lovitt, K. Ndousse, D. Amodei, T. Brown, J. Clark, J. Kaplan, S. McCandlish, and C. Olah. 2021. A mathematical framework for transformer circuits. White paper.

Elman, J. L. 1990. Finding structure in time. *Cognitive science*, 14(2):179–211.

Elsner, M., J. Austerweil, and E. Charniak. 2007. A unified local and global model for discourse coherence. *NAACL-HLT*.

Elsner, M. and E. Charniak. 2008. Coreference-inspired coherence modeling. *ACL*.

Elsner, M. and E. Charniak. 2011. Extending the entity grid with entity-specific features. *ACL*.

Elvevåg, B., P. W. Foltz, D. R. Weinberger, and T. E. Goldberg. 2007. Quantifying incoherence in speech: an automated methodology and novel application to schizophrenia. *Schizophrenia research*, 93(1-3):304–316.

Emami, A. and F. Jelinek. 2005. A neural syntactic language model. *Machine learning*, 60(1):195–227.

Emami, A., P. Trichelair, A. Trischler, K. Suleman, H. Schulz, and J. C. K. Cheung. 2019. The KNOWREF coreference corpus: Removing gender and number cues for difficult pronominal anaphora resolution. *ACL*.

Erk, K. 2007. A simple, similarity-based model for selectional preferences. *ACL*.

Ethayarajh, K. 2019. How contextual are contextualized word representations? Comparing the geometry of BERT, ELMo, and GPT-2 embeddings. *EMNLP*.

Ethayarajh, K., D. Duvenaud, and G. Hirst. 2019a. Towards understanding linear word analogies. *ACL*.

Ethayarajh, K., D. Duvenaud, and G. Hirst. 2019b. Understanding undesirable word embedding associations. *ACL*.

Ethayarajh, K. and D. Jurafsky. 2020. Utility is in the eye of the user: A critique of NLP leaderboards. *EMNLP*.

Ethayarajh, K., H. C. Zhang, and S. Behzad. 2022. Stanford human preferences dataset v2 (shp-2).

Etzioni, O., M. Cafarella, D. Downey, A.-M. Popescu, T. Shaked, S. Soderland, D. S. Weld, and A. Yates. 2005. Unsupervised named-entity extraction from the web: An experimental study. *Artificial Intelligence*, 165(1):91–134.

Evans, N. 2000. Word classes in the world's languages. In G. Booij, C. Lehmann, and J. Mugdan, eds, *Morphology: A Handbook on Inflection and Word Formation*, 708–732. Mouton.

Fader, A., S. Soderland, and O. Etzioni. 2011. Identifying relations for open information extraction. *EMNLP*.

Fan, A., S. Bhosale, H. Schwenk, Z. Ma, A. El-Kishky, S. Goyal, M. Baines, O. Celebi, G. Wenzek, V. Chaudhary, N. Goyal, T. Birch, V. Liptchinsky, S. Edunov, M. Auli, and A. Joulin. 2021. Beyond english-centric multilingual machine translation. *JMLR*, 22(107):1–48.

Fant, G. M. 1951. Speech communication research. *Ing. Vetenskaps Akad. Stockholm, Sweden*, 24:331–337.

Fant, G. M. 1960. *Acoustic Theory of Speech Production*. Mouton.

Fant, G. M. 1986. Glottal flow: Models and interaction. *Journal of Phonetics*, 14:393–399.

Fant, G. M. 2004. *Speech Acoustics and Phonetics*. Kluwer.

Fast, E., B. Chen, and M. S. Bernstein. 2016. Empath: Understanding Topic Signals in Large-Scale Text. *CHI*.

Fauconnier, G. and M. Turner. 2008. *The way we think: Conceptual blending and the mind's hidden complexities*. Basic Books.

Feldman, J. A. and D. H. Ballard. 1982. Connectionist models and their properties. *Cognitive Science*, 6:205–254.

Fellbaum, C., ed. 1998. *WordNet: An Electronic Lexical Database*. MIT Press.

Feng, V. W. and G. Hirst. 2011. Classifying arguments by scheme. *ACL*.

Feng, V. W. and G. Hirst. 2014. A linear-time bottom-up discourse parser with constraints and post-editing. *ACL*.

Feng, V. W., Z. Lin, and G. Hirst. 2014. The impact of deep hierarchical discourse structures in the evaluation of text coherence. *COLING*.

Fernandes, E. R., C. N. dos Santos, and R. L. Milidiú. 2012. Latent structure perceptron with feature induction for unrestricted coreference resolution. *CoNLL*.

Ferragina, P. and U. Scaiella. 2011. Fast and accurate annotation of short texts with wikipedia pages. *IEEE Software*, 29(1):70–75.

Ferro, L., L. Gerber, I. Mani, B. Sundheim, and G. Wilson. 2005. Tides 2005 standard for the annotation of temporal expressions. Technical report, MITRE.

Ferrucci, D. A. 2012. Introduction to "This is Watson". *IBM Journal of Research and Development*, 56(3/4):1:1–1:15.

Field, A. and Y. Tsvetkov. 2019. Entity-centric contextual affective analysis. *ACL*.

Fillmore, C. J. 1966. A proposal concerning English prepositions. In F. P. Dinneen, ed., *17th annual Round Table*, volume 17 of *Monograph Series on Language and Linguistics*, 19–34. Georgetown University Press.

Fillmore, C. J. 1968. The case for case. In E. W. Bach and R. T. Harms, eds, *Universals in Linguistic Theory*, 1–88. Holt, Rinehart & Winston.

Fillmore, C. J. 1985. Frames and the semantics of understanding. *Quaderni di Semantica*, VI(2):222–254.

Fillmore, C. J. 2003. Valency and semantic roles: the concept of deep structure case. In V. Agel, L. M. Eichinger, H. W. Eroms, P. Hellwig, H. J. Heringer, and H. Lobin, eds, *Dependenz und Valenz: Ein internationales Handbuch der zeitgenössischen Forschung*, chapter 36, 457–475. Walter de Gruyter.

Fillmore, C. J. 2012. ACL lifetime achievement award: Encounters with language. *Computational Linguistics*, 38(4):701–718.

Fillmore, C. J. and C. F. Baker. 2009. A frames approach to semantic analysis. In B. Heine and H. Narrog, eds, *The Oxford Handbook of Linguistic Analysis*, 313–340. Oxford University Press.

Fillmore, C. J., C. R. Johnson, and M. R. L. Petruck. 2003. Background to FrameNet. *International journal of lexicography*, 16(3):235–250.

Finkelstein, L., E. Gabrilovich, Y. Matias, E. Rivlin, Z. Solan, G. Wolfman, and E. Ruppin. 2002. Placing search in context: The concept revisited. *ACM Transactions on Information Systems*, 20(1):116—-131.

Finlayson, M. A. 2016. Inferring Propp's functions from semantically annotated text. *The Journal of American Folklore*, 129(511):55–77.

Firth, J. R. 1957. A synopsis of linguistic theory 1930–1955. In *Studies in Linguistic Analysis*. Philological Society. Reprinted in Palmer, F. (ed.) 1968. Selected Papers of J. R. Firth. Longman, Harlow.

Fitt, S. 2002. Unisyn lexicon. http://www.cstr.ed.ac.uk/projects/unisyn/.

Flanagan, J. L. 1972. *Speech Analysis, Synthesis, and Perception*. Springer.

Flanagan, J. L., K. Ishizaka, and K. L. Shipley. 1975. Synthesis of speech from a dynamic model of the vocal cords and vocal tract. *The Bell System Technical Journal*, 54(3):485–506.

Foland, W. and J. H. Martin. 2016. CU-NLP at SemEval-2016 task 8: AMR parsing using LSTM-based recurrent neural networks. *SemEval-2016*.

Foland, Jr., W. R. and J. H. Martin. 2015. Dependency-based semantic role labeling using convolutional neural networks. **SEM 2015*.

Foltz, P. W., W. Kintsch, and T. K. Landauer. 1998. The measurement of textual coherence with latent semantic analysis. *Discourse processes*, 25(2-3):285–307.

Fox, B. A. 1993. *Discourse Structure and Anaphora: Written and Conversational English*. Cambridge.

Francis, W. N. and H. Kučera. 1982. *Frequency Analysis of English Usage*. Houghton Mifflin, Boston.

Franz, A. and T. Brants. 2006. All our n-gram are belong to you. https://research.google/blog/all-our-n-gram-are-belong-to-you/.

Friedman, B. and D. G. Hendry. 2019. *Value Sensitive Design: Shaping Technology with Moral Imagination*. MIT Press.

Friedman, B., D. G. Hendry, and A. Borning. 2017. A survey of value sensitive design methods. *Foundations and Trends in Human-Computer Interaction*, 11(2):63–125.

Fry, D. B. 1955. Duration and intensity as physical correlates of linguistic stress. *JASA*, 27:765–768.

Fry, D. B. 1959. Theoretical aspects of mechanical speech recognition. *Journal of the British Institution of Radio Engineers*, 19(4):211–218. Appears together with companion paper (Denes 1959).

Furnas, G. W., T. K. Landauer, L. M. Gomez, and S. T. Dumais. 1987. The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971.

Gabow, H. N., Z. Galil, T. Spencer, and R. E. Tarjan. 1986. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122.

Gaddy, D., M. Stern, and D. Klein. 2018. What's going on in neural constituency parsers? an analysis. *NAACL HLT*.

Gage, P. 1994. A new algorithm for data compression. *The C Users Journal*, 12(2):23–38.

Gale, W. A. and K. W. Church. 1994. What is wrong with adding one? In N. Oostdijk and P. de Haan, eds, *Corpus-Based Research into Language*, 189–198. Rodopi.

Gale, W. A. and K. W. Church. 1991. A program for aligning sentences in bilingual corpora. *ACL*.

Gale, W. A. and K. W. Church. 1993. A program for aligning sentences in bilingual corpora. *Computational Linguistics*, 19:75–102.

Gale, W. A., K. W. Church, and D. Yarowsky. 1992a. One sense per discourse. *HLT*.

Gale, W. A., K. W. Church, and D. Yarowsky. 1992b. Work on statistical methods for word sense disambiguation. *AAAI Fall Symposium on Probabilistic Approaches to Natural Language*.

Gao, L., T. Hoppe, A. Thite, S. Biderman, C. Foster, N. Nabeshima, S. Black, J. Phang, S. Presser, L. Golding, H. He, and C. Leahy. 2020. The Pile: An 800GB dataset of diverse text for language modeling. ArXiv preprint.

Gao, T., H. Yen, J. Yu, and D. Chen. 2023. Enabling large language models to generate text with citations. *EMNLP*.

Garg, N., L. Schiebinger, D. Jurafsky, and J. Zou. 2018. Word embeddings quantify 100 years of gender and ethnic stereotypes. *Proceedings of the National Academy of Sciences*, 115(16):E3635–E3644.

Garside, R. 1987. The CLAWS word-tagging system. In R. Garside, G. Leech, and G. Sampson, eds, *The Computational Analysis of English*, 30–41. Longman.

Garside, R., G. Leech, and A. McEnery. 1997. *Corpus Annotation*. Longman.

Gebru, T., J. Morgenstern, B. Vecchione, J. W. Vaughan, H. Wallach, H. Daumé III, and K. Crawford. 2020. Datasheets for datasets. ArXiv.

Gehman, S., S. Gururangan, M. Sap, Y. Choi, and N. A. Smith. 2020. RealToxicityPrompts: Evaluating neural toxic degeneration in language models. *Findings of EMNLP*.

Gemmeke, J. F., D. P. Ellis, D. Freedman, A. Jansen, W. Lawrence, R. C. Moore, M. Plakal, and M. Ritter. 2017. Audio Set: An ontology and human-labeled dataset for audio events. *ICASSP*.

Gerber, M. and J. Y. Chai. 2010. Beyond nombank: A study of implicit arguments for nominal predicates. *ACL*.

Gers, F. A., J. Schmidhuber, and F. Cummins. 2000. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471.

Geva, M., R. Schuster, J. Berant, and O. Levy. 2021. Transformer feed-forward layers are key-value memories. *EMNLP*.

Gil, D. 2000. Syntactic categories, cross-linguistic variation and universal grammar. In P. M. Vogel and B. Comrie, eds, *Approaches to the Typology of Word Classes*, 173–216. Mouton.

Gildea, D. and D. Jurafsky. 2000. Automatic labeling of semantic roles. *ACL*.

Gildea, D. and D. Jurafsky. 2002. Automatic labeling of semantic roles. *Computational Linguistics*, 28(3):245–288.

Gildea, D. and M. Palmer. 2002. The necessity of syntactic parsing for predicate argument recognition. *ACL*.

Giles, C. L., G. M. Kuhn, and R. J. Williams. 1994. Dynamic recurrent neural networks: Theory and applications. *IEEE Trans. Neural Netw. Learning Syst.*, 5(2):153–156.

Gillick, L. and S. J. Cox. 1989. Some statistical issues in the comparison of speech recognition algorithms. *ICASSP*.

Girard, G. 1718. *La justesse de la langue françoise: ou les différentes significations des mots qui passent pour synonimes*. Laurent d'Houry, Paris.

Giuliano, V. E. 1965. The interpretation of word associations. *Statistical Association Methods For Mechanized Documentation. Symposium Proceedings. Washington, D.C., USA, March 17, 1964*. https://nvlpubs.nist.gov/nistpubs/Legacy/MP/nbsmiscellaneouspub269.pdf.

Gladkova, A., A. Drozd, and S. Matsuoka. 2016. Analogy-based detection of morphological and semantic relations with word embeddings: what works and what doesn't. *NAACL Student Research Workshop*.

Glenberg, A. M. and D. A. Robertson. 2000. Symbol grounding and meaning: A comparison of high-dimensional and embodied theories of meaning. *Journal of memory and language*, 43(3):379–401.

Godfrey, J., E. Holliman, and J. McDaniel. 1992. SWITCHBOARD: Telephone speech corpus for research and development. *ICASSP*.

Goel, V. and W. Byrne. 2000. Minimum bayes-risk automatic speech recognition. *Computer Speech & Language*, 14(2):115–135.

Goffman, E. 1974. *Frame analysis: An essay on the organization of experience*. Harvard University Press.

Gonen, H. and Y. Goldberg. 2019. Lipstick on a pig: Debiasing methods cover up systematic gender biases in word embeddings but do not remove them. *NAACL HLT*.

Goodfellow, I., Y. Bengio, and A. Courville. 2016. *Deep Learning*. MIT Press.

Goodman, J. 2006. A bit of progress in language modeling: Extended version. Technical Report MSR-TR-2001-72, Machine Learning and Applied Statistics Group, Microsoft Research, Redmond, WA.

Gould, S. J. 1980. *The Panda's Thumb*. Penguin Group.

Goyal, N., C. Gao, V. Chaudhary, P.-J. Chen, G. Wenzek, D. Ju, S. Krishnan, M. Ranzato, F. Guzmán, and A. Fan. 2022. The flores-101 evaluation benchmark for low-resource and multilingual machine translation. *TACL*, 10:522–538.

Graff, D. 1997. The 1996 Broadcast News speech and language-model corpus. *Proceedings DARPA Speech Recognition Workshop*.

Graves, A. 2012. Sequence transduction with recurrent neural networks. *ICASSP*.

Graves, A. 2013. Generating sequences with recurrent neural networks. ArXiv.

Graves, A., S. Fernández, F. Gomez, and J. Schmidhuber. 2006. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. *ICML*.

Graves, A., S. Fernández, M. Liwicki, H. Bunke, and J. Schmidhuber. 2007. Unconstrained on-line handwriting recognition with recurrent neural networks. *NeurIPS*.

Graves, A. and N. Jaitly. 2014. Towards end-to-end speech recognition with recurrent neural networks. *ICML*.

Graves, A., A.-r. Mohamed, and G. Hinton. 2013. Speech recognition with deep recurrent neural networks. *ICASSP*.

Graves, A. and J. Schmidhuber. 2005. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, 18(5-6):602–610.

Graves, A., G. Wayne, and I. Danihelka. 2014. Neural Turing machines. ArXiv.

Gray, R. M. 1984. Vector quantization. *IEEE Transactions on ASSP*, ASSP-1(2):4–29.

Green, B. F., A. K. Wolf, C. Chomsky, and K. Laughery. 1961. Baseball: An automatic question answerer. *Proceedings of the Western Joint Computer Conference 19*.

Greenberg, J. H. 1960. A quantitative approach to the morphological typology of language. *International journal of American linguistics*, 26(3):178–194.

Greenberg, S., D. Ellis, and J. Hollenback. 1996. Insights into spoken language gleaned from phonetic transcription of the Switchboard corpus. *ICSLP*.

Greene, B. B. and G. M. Rubin. 1971. Automatic grammatical tagging of English. Department of Linguistics, Brown University, Providence, Rhode Island.

Greenwald, A. G., D. E. McGhee, and J. L. K. Schwartz. 1998. Measuring individual differences in implicit cognition: the implicit association test. *Journal of personality and social psychology*, 74(6):1464–1480.

Grenager, T. and C. D. Manning. 2006. Unsupervised discovery of a statistical verb lexicon. *EMNLP*.

Grice, H. P. 1975. Logic and conversation. In P. Cole and J. L. Morgan, eds, *Speech Acts: Syntax and Semantics Volume 3*, 41–58. Academic Press.

Grice, H. P. 1978. Further notes on logic and conversation. In P. Cole, ed., *Pragmatics: Syntax and Semantics Volume 9*, 113–127. Academic Press.

Grishman, R. and B. Sundheim. 1995. Design of the MUC-6 evaluation. *MUC-6*.

Grosz, B. J. 1977a. The representation and use of focus in a system for understanding dialogs. *IJCAI-77*. Morgan Kaufmann.

Grosz, B. J. 1977b. *The Representation and Use of Focus in Dialogue Understanding*. Ph.D. thesis, University of California, Berkeley.

Grosz, B. J., A. K. Joshi, and S. Weinstein. 1983. Providing a unified account of definite noun phrases in English. *ACL*.

Grosz, B. J., A. K. Joshi, and S. Weinstein. 1995. Centering: A framework for modeling the local coherence of discourse. *Computational Linguistics*, 21(2):203–225.

Gruber, J. S. 1965. *Studies in Lexical Relations*. Ph.D. thesis, MIT.

Grünewald, S., A. Friedrich, and J. Kuhn. 2021. Applying Occam's

razor to transformer-based dependency parsing: What works, what doesn't, and what is really necessary. *IWPT*.

Guinaudeau, C. and M. Strube. 2013. Graph-based local coherence modeling. *ACL*.

Gundel, J. K., N. Hedberg, and R. Zacharski. 1993. Cognitive status and the form of referring expressions in discourse. *Language*, 69(2):274–307.

Gururangan, S., A. Marasović, S. Swayamdipta, K. Lo, I. Beltagy, D. Downey, and N. A. Smith. 2020. Don't stop pretraining: Adapt language models to domains and tasks. *ACL*.

Gusfield, D. 1997. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press.

Guyon, I. and A. Elisseeff. 2003. An introduction to variable and feature selection. *JMLR*, 3:1157–1182.

Haber, J. and M. Poesio. 2020. Assessing polyseme sense similarity through co-predication acceptability and contextualised embedding distance. **SEM*.

Habernal, I. and I. Gurevych. 2016. Which argument is more convincing? Analyzing and predicting convincingness of Web arguments using bidirectional LSTM. *ACL*.

Habernal, I. and I. Gurevych. 2017. Argumentation mining in user-generated web discourse. *Computational Linguistics*, 43(1):125–179.

Haghighi, A. and D. Klein. 2009. Simple coreference resolution with rich syntactic and semantic features. *EMNLP*.

Hajishirzi, H., L. Zilles, D. S. Weld, and L. Zettlemoyer. 2013. Joint coreference resolution and named-entity linking with multi-pass sieves. *EMNLP*.

Hajič, J. 1998. *Building a Syntactically Annotated Corpus: The Prague Dependency Treebank*, 106–132. Karolinum.

Hajič, J. 2000. Morphological tagging: Data vs. dictionaries. *NAACL*.

Hajič, J., M. Ciaramita, R. Johansson, D. Kawahara, M. A. Martí, L. Màrquez, A. Meyers, J. Nivre, S. Padó, J. Štěpánek, P. Stranǎḱ, M. Surdeanu, N. Xue, and Y. Zhang. 2009. The conll-2009 shared task: Syntactic and semantic dependencies in multiple languages. *CoNLL*.

Hakkani-Tür, D., K. Oflazer, and G. Tür. 2002. Statistical morphological disambiguation for agglutinative languages. *Journal of Computers and Humanities*, 36(4):381–410.

Halliday, M. A. K. and R. Hasan. 1976. *Cohesion in English*. Longman. English Language Series, Title No. 9.

Hamilton, W. L., K. Clark, J. Leskovec, and D. Jurafsky. 2016a. Inducing domain-specific sentiment lexicons from unlabeled corpora. *EMNLP*.

Hamilton, W. L., J. Leskovec, and D. Jurafsky. 2016b. Diachronic word embeddings reveal statistical laws of semantic change. *ACL*.

Hannun, A. 2017. Sequence modeling with CTC. *Distill*, 2(11).

Hannun, A. Y., A. L. Maas, D. Jurafsky, and A. Y. Ng. 2014. First-pass large vocabulary continuous speech recognition using bi-directional recurrent DNNs. ArXiv preprint arXiv:1408.2873.

Harris, C. M. 1953. A study of the building blocks in speech. *JASA*, 25(5):962–969.

Harris, Z. S. 1946. From morpheme to utterance. *Language*, 22(3):161–183.

Harris, Z. S. 1954. Distributional structure. *Word*, 10:146–162.

Harris, Z. S. 1962. *String Analysis of Sentence Structure*. Mouton, The Hague.

Hashimoto, T., M. Srivastava, H. Namkoong, and P. Liang. 2018. Fairness without demographics in repeated loss minimization. *ICML*.

Hastie, T., R. J. Tibshirani, and J. H. Friedman. 2001. *The Elements of Statistical Learning*. Springer.

Hatzivassiloglou, V. and K. McKeown. 1997. Predicting the semantic orientation of adjectives. *ACL*.

Hatzivassiloglou, V. and J. Wiebe. 2000. Effects of adjective orientation and gradability on sentence subjectivity. *COLING*.

Haviland, S. E. and H. H. Clark. 1974. What's new? Acquiring new information as a process in comprehension. *Journal of Verbal Learning and Verbal Behaviour*, 13:512–521.

Hawkins, J. A. 1978. *Definiteness and indefiniteness: a study in reference and grammaticality prediction*. Croom Helm Ltd.

Hayashi, T., R. Yamamoto, K. Inoue, T. Yoshimura, S. Watanabe, T. Toda, K. Takeda, Y. Zhang, and X. Tan. 2020. ESPnet-TTS: Unified, reproducible, and integratable open source end-to-end text-to-speech toolkit. *ICASSP*.

He, L., K. Lee, M. Lewis, and L. Zettlemoyer. 2017. Deep semantic role labeling: What works and what's next. *ACL*.

He, W., K. Liu, J. Liu, Y. Lyu, S. Zhao, X. Xiao, Y. Liu, Y. Wang, H. Wu, Q. She, X. Liu, T. Wu, and H. Wang. 2018. DuReader: a Chinese machine reading comprehension dataset from real-world applications. *Workshop on Machine Reading for Question Answering*.

Heafield, K. 2011. KenLM: Faster and smaller language model queries. *Workshop on Statistical Machine Translation*.

Heafield, K., I. Pouzyrevsky, J. H. Clark, and P. Koehn. 2013. Scalable modified Kneser-Ney language model estimation. *ACL*.

Heaps, H. S. 1978. *Information retrieval. Computational and theoretical aspects*. Academic Press.

Hearst, M. A. 1992a. Automatic acquisition of hyponyms from large text corpora. *COLING*.

Hearst, M. A. 1992b. Automatic acquisition of hyponyms from large text corpora. *COLING*.

Hearst, M. A. 1997. Texttiling: Segmenting text into multi-paragraph subtopic passages. *Computational Linguistics*, 23:33–64.

Hearst, M. A. 1998. Automatic discovery of WordNet relations. In C. Fellbaum, ed., *WordNet: An Electronic Lexical Database*. MIT Press.

Heim, I. 1982. *The semantics of definite and indefinite noun phrases*. Ph.D. thesis, University of Massachusetts at Amherst.

Heinz, J. M. and K. N. Stevens. 1961. On the properties of voiceless fricative consonants. *JASA*, 33:589–596.

Hellrich, J., S. Buechel, and U. Hahn. 2019. Modeling word emotion in historical language: Quantity beats supposed stability in seed word selection. *3rd Joint SIGHUM Workshop on Computational Linguistics for Cultural Heritage, Social Sciences, Humanities and Literature*.

Hellrich, J. and U. Hahn. 2016. Bad company—Neighborhoods in neural embedding spaces considered harmful. *COLING*.

Henderson, J. 1994. *Description Based Parsing in a Connectionist Network*. Ph.D. thesis, University of Pennsylvania, Philadelphia, PA.

Henderson, J. 2003. Inducing history representations for broad coverage statistical parsing. *HLT-NAACL-03*.

Henderson, J. 2004. Discriminative training of a neural network statistical parser. *ACL*.

Henderson, P., J. Hu, J. Romoff, E. Brunskill, D. Jurafsky, and J. Pineau. 2020. Towards the systematic reporting of the energy and

carbon footprints of machine learning. *Journal of Machine Learning Research*, 21(248):1–43.

Henderson, P., X. Li, D. Jurafsky, T. Hashimoto, M. A. Lemley, and P. Liang. 2023. Foundation models and fair use. *JMLR*, 24(400):1–79.

Henderson, P., K. Sinha, N. Angelard-Gontier, N. R. Ke, G. Fried, R. Lowe, and J. Pineau. 2017. Ethical challenges in data-driven dialogue systems. *AAAI/ACM AI Ethics and Society Conference*.

Hendrickx, I., S. N. Kim, Z. Kozareva, P. Nakov, D. Ó Séaghdha, S. Padó, M. Pennacchiotti, L. Romano, and S. Szpakowicz. 2009. Semeval-2010 task 8: Multi-way classification of semantic relations between pairs of nominals. *5th International Workshop on Semantic Evaluation*.

Hendrix, G. G., C. W. Thompson, and J. Slocum. 1973. Language processing via canonical verbs and semantic models. *Proceedings of IJCAI-73*.

Herdan, G. 1960. *Type-token mathematics*. Mouton.

Hermann, K. M., T. Kocisky, E. Grefenstette, L. Espeholt, W. Kay, M. Suleyman, and P. Blunsom. 2015. Teaching machines to read and comprehend. *NeurIPS*.

Hernault, H., H. Prendinger, D. A. duVerle, and M. Ishizuka. 2010. Hilda: A discourse parser using support vector machine classification. *Dialogue & Discourse*, 1(3).

Hidey, C., E. Musi, A. Hwang, S. Muresan, and K. McKeown. 2017. Analyzing the semantic types of claims and premises in an online persuasive forum. *4th Workshop on Argument Mining*.

Hill, F., R. Reichart, and A. Korhonen. 2015. Simlex-999: Evaluating semantic models with (genuine) similarity estimation. *Computational Linguistics*, 41(4):665–695.

Hinton, G. E. 1986. Learning distributed representations of concepts. *COGSCI*.

Hinton, G. E., S. Osindero, and Y.-W. Teh. 2006. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554.

Hinton, G. E., N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. ArXiv preprint arXiv:1207.0580.

Hirschman, L., M. Light, E. Breck, and J. D. Burger. 1999. Deep Read: A reading comprehension system. *ACL*.

Hirst, G. 1981. *Anaphora in Natural Language Understanding: A survey*. Number 119 in Lecture notes in computer science. Springer-Verlag.

Hirst, G. 1987. *Semantic Interpretation and the Resolution of Ambiguity*. Cambridge University Press.

Hjelmslev, L. 1969. *Prologomena to a Theory of Language*. University of Wisconsin Press. Translated by Francis J. Whitfield; original Danish edition 1943.

Hobbs, J. R. 1978. Resolving pronoun references. *Lingua*, 44:311–338.

Hobbs, J. R. 1979. Coherence and coreference. *Cognitive Science*, 3:67–90.

Hobbs, J. R., D. E. Appelt, J. Bear, D. Israel, M. Kameyama, M. E. Stickel, and M. Tyson. 1997. FASTUS: A cascaded finite-state transducer for extracting information from natural-language text. In E. Roche and Y. Schabes, eds, *Finite-State Language Processing*, 383–406. MIT Press.

Hochreiter, S. and J. Schmidhuber. 1997. Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Hofmann, T. 1999. Probabilistic latent semantic indexing. *SIGIR-99*.

Hofmann, V., P. R. Kalluri, D. Jurafsky, and S. King. 2024. Ai generates covertly racist decisions about people based on their dialect. *Nature*, 633(8028):147–154.

Holtzman, A., J. Buys, L. Du, M. Forbes, and Y. Choi. 2020. The curious case of neural text degeneration. *ICLR*.

Hopcroft, J. E. and J. D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Hou, Y., K. Markert, and M. Strube. 2018. Unrestricted bridging resolution. *Computational Linguistics*, 44(2):237–284.

Householder, F. W. 1995. Dionysius Thrax, the *technai*, and Sextus Empiricus. In E. F. K. Koerner and R. E. Asher, eds, *Concise History of the Language Sciences*, 99–103. Elsevier Science.

Hovy, E. H. 1990. Parsimonious and profligate approaches to the question of discourse structure relations. *Proceedings of the 5th International Workshop on Natural Language Generation*.

Hovy, E. H., M. P. Marcus, M. Palmer, L. A. Ramshaw, and R. Weischedel. 2006. OntoNotes: The 90% solution. *HLT-NAACL*.

Hsu, W.-N., B. Bolte, Y.-H. H. Tsai, K. Lakhotia, R. Salakhutdinov, and A. Mohamed. 2021. Hubert: Self-supervised speech representation learning by masked prediction of hidden units. *IEEE/ACM TASLP*, 29:3451–3460.

Hu, M. and B. Liu. 2004. Mining and summarizing customer reviews. *SIGKDD-04*.

Huang, E. H., R. Socher, C. D. Manning, and A. Y. Ng. 2012. Improving word representations via global context and multiple word prototypes. *ACL*.

Huang, Z., W. Xu, and K. Yu. 2015. Bidirectional LSTM-CRF models for sequence tagging. *arXiv preprint arXiv:1508.01991*.

Huffman, S. 1996. Learning information extraction patterns from examples. In S. Wertmer, E. Riloff, and G. Scheller, eds, *Connectionist, Statistical, and Symbolic Approaches to Learning Natural Language Processing*, 246–260. Springer.

Hunt, A. J. and A. W. Black. 1996. Unit selection in a concatenative speech synthesis system using a large speech database. *ICASSP*.

Hutchins, W. J. 1986. *Machine Translation: Past, Present, Future*. Ellis Horwood, Chichester, England.

Hutchins, W. J. 1997. From first conception to first demonstration: The nascent years of machine translation, 1947–1954. A chronology. *Machine Translation*, 12:192–252.

Hutchins, W. J. and H. L. Somers. 1992. *An Introduction to Machine Translation*. Academic Press.

Hutchinson, B., V. Prabhakaran, E. Denton, K. Webster, Y. Zhong, and S. Denuyl. 2020. Social biases in NLP models as barriers for persons with disabilities. *ACL*.

Hymes, D. 1974. Ways of speaking. In R. Bauman and J. Sherzer, eds, *Explorations in the ethnography of speaking*, 433–451. Cambridge University Press.

Iida, R., K. Inui, H. Takamura, and Y. Matsumoto. 2003. Incorporating contextual cues in trainable models for coreference resolution. *EACL Workshop on The Computational Treatment of Anaphora*.

Irsoy, O. and C. Cardie. 2014. Opinion mining with deep recurrent neural networks. *EMNLP*.

Ischen, C., T. Araujo, H. Voorveld, G. van Noort, and E. Smit. 2019. Privacy concerns in chatbot interactions. *International Workshop on Chatbot Research and Design*.

ISO8601. 2004. Data elements and interchange formats—information interchange—representation of dates and times. Technical report, International Organization for Standards (ISO).

Itakura, F. 1975. Minimum prediction residual principle applied to speech recognition. *IEEE Transactions on ASSP*, ASSP-32:67–72.

Iter, D., K. Guu, L. Lansing, and D. Jurafsky. 2020. Pretraining with contrastive sentence objectives improves discourse performance of language models. *ACL*.

Iter, D., J. Yoon, and D. Jurafsky. 2018. Automatic detection of incoherent speech for diagnosing schizophrenia. *Fifth Workshop on Computational Linguistics and Clinical Psychology*.

Iyer, S., I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. *ACL*.

Iyer, S., X. V. Lin, R. Pasunuru, T. Mihaylov, D. Simig, P. Yu, K. Shuster, T. Wang, Q. Liu, P. S. Koura, X. Li, B. O'Horo, G. Pereyra, J. Wang, C. Dewan, A. Celikyilmaz, L. Zettlemoyer, and V. Stoyanov. 2022. Opt-iml: Scaling language model instruction meta learning through the lens of generalization. ArXiv preprint.

Izacard, G., P. Lewis, M. Lomeli, L. Hosseini, F. Petroni, T. Schick, J. Dwivedi-Yu, A. Joulin, S. Riedel, and E. Grave. 2022. Few-shot learning with retrieval augmented language models. ArXiv preprint.

Jackendoff, R. 1983. *Semantics and Cognition*. MIT Press.

Jacobs, P. S. and L. F. Rau. 1990. SCISOR: A system for extracting information from on-line news. *CACM*, 33(11):88–97.

Jaech, A., G. Mulcaire, S. Hathi, M. Ostendorf, and N. A. Smith. 2016. Hierarchical character-word models for language identification. *ACL Workshop on NLP for Social Media*.

Jaitly, N., P. Nguyen, A. Senior, and V. Vanhoucke. 2012. Application of pretrained deep neural networks to large vocabulary speech recognition. *INTERSPEECH*.

Jauhiainen, T., M. Lui, M. Zampieri, T. Baldwin, and K. Lindén. 2019. Automatic language identification in texts: A survey. *JAIR*, 65(1):675–682.

Jefferson, G. 1972. Side sequences. In D. Sudnow, ed., *Studies in social interaction*, 294–333. Free Press, New York.

Jefferson, G. 1984. Notes on a systematic deployment of the acknowledgement tokens 'yeah' and 'mm hm'. *Papers in Linguistics*, 17(2):197–216.

Jeffreys, H. 1948. *Theory of Probability*, 2nd edition. Clarendon Press. Section 3.23.

Jekat, S., A. Klein, E. Maier, I. Maleck, M. Mast, and J. Quantz. 1995. Dialogue acts in verbmobil. Verbmobil–Report–65–95.

Jelinek, F. 1969. A fast sequential decoding algorithm using a stack. *IBM Journal of Research and Development*, 13:675–685.

Jelinek, F. 1990. Self-organized language modeling for speech recognition. In A. Waibel and K.-F. Lee, eds, *Readings in Speech Recognition*, 450–506. Morgan Kaufmann. Originally distributed as IBM technical report in 1985.

Jelinek, F. and R. L. Mercer. 1980. Interpolated estimation of Markov source parameters from sparse data. In E. S. Gelsema and L. N. Kanal, eds, *Proceedings, Workshop on Pattern Recognition in Practice*, 381–397. North Holland.

Jelinek, F., R. L. Mercer, and L. R. Bahl. 1975. Design of a linguistic statistical decoder for the recognition of continuous speech. *IEEE Transactions on Information Theory*, IT-21(3):250–256.

Ji, H. and R. Grishman. 2011. Knowledge base population: Successful approaches and challenges. *ACL*.

Ji, H., R. Grishman, and H. T. Dang. 2010. Overview of the tac 2011 knowledge base population track. *TAC-11*.

Ji, Y. and J. Eisenstein. 2014. Representation learning for text-level discourse parsing. *ACL*.

Ji, Y. and J. Eisenstein. 2015. One vector is not enough: Entity-augmented distributed semantics for discourse relations. *TACL*, 3:329–344.

Jia, R. and P. Liang. 2016. Data recombination for neural semantic parsing. *ACL*.

Jia, S., T. Meng, J. Zhao, and K.-W. Chang. 2020. Mitigating gender bias amplification in distribution by posterior regularization. *ACL*.

Jiang, C., B. Qi, X. Hong, D. Fu, Y. Cheng, F. Meng, M. Yu, B. Zhou, and J. Zhou. 2024. On large language models' hallucination with regard to known facts. *NAACL HLT*.

Johnson, J., M. Douze, and H. Jégou. 2017. Billion-scale similarity search with GPUs. ArXiv preprint arXiv:1702.08734.

Johnson, K. 2003. *Acoustic and Auditory Phonetics*, 2nd edition. Blackwell.

Johnson, W. E. 1932. Probability: deductive and inductive problems (appendix to). *Mind*, 41(164):421–423.

Johnson-Laird, P. N. 1983. *Mental Models*. Harvard University Press, Cambridge, MA.

Jones, M. P. and J. H. Martin. 1997. Contextual spelling correction using latent semantic analysis. *ANLP*.

Jones, R., A. McCallum, K. Nigam, and E. Riloff. 1999. Bootstrapping for text learning tasks. *IJCAI-99 Workshop on Text Mining: Foundations, Techniques and Applications*.

Jones, T. 2015. Toward a description of African American Vernacular English dialect regions using "Black Twitter". *American Speech*, 90(4):403–440.

Joos, M. 1950. Description of language design. *JASA*, 22:701–708.

Jordan, M. 1986. Serial order: A parallel distributed processing approach. Technical Report ICS Report 8604, University of California, San Diego.

Joshi, A. K. and P. Hopely. 1999. A parser from antiquity. In A. Kornai, ed., *Extended Finite State Models of Language*, 6–15. Cambridge University Press.

Joshi, A. K. and S. Kuhn. 1979. Centered logic: The role of entity centered sentence representation in natural language inferencing. *IJCAI-79*.

Joshi, A. K. and S. Weinstein. 1981. Control of inference: Role of some aspects of discourse structure – centering. *IJCAI-81*.

Joshi, M., D. Chen, Y. Liu, D. S. Weld, L. Zettlemoyer, and O. Levy. 2020. SpanBERT: Improving pre-training by representing and predicting spans. *TACL*, 8:64–77.

Joshi, M., O. Levy, D. S. Weld, and L. Zettlemoyer. 2019. BERT for coreference resolution: Baselines and analysis. *EMNLP*.

Joty, S., G. Carenini, and R. T. Ng. 2015. CODRA: A novel discriminative framework for rhetorical analysis. *Computational Linguistics*, 41(3):385–435.

Jurafsky, D. 2014. *The Language of Food*. W. W. Norton, New York.

Jurafsky, D., V. Chahuneau, B. R. Routledge, and N. A. Smith. 2014. Narrative framing of consumer sentiment in online restaurant reviews. *First Monday*, 19(4).

Jurafsky, D., C. Wooters, G. Tajchman, J. Segal, A. Stolcke, E. Fosler, and N. Morgan. 1994. The Berkeley restaurant project. *ICSLP*.

Jurgens, D., S. M. Mohammad, P. Turney, and K. Holyoak. 2012. SemEval-2012 task 2: Measuring degrees of relational similarity. **SEM 2012*.

Jurgens, D., Y. Tsvetkov, and D. Jurafsky. 2017. Incorporating dialectal variability for socially equitable language identification. *ACL*.

Justeson, J. S. and S. M. Katz. 1991. Co-occurrences of antonymous adjectives and their contexts. *Computational linguistics*, 17(1):1–19.

Kalchbrenner, N. and P. Blunsom. 2013. Recurrent continuous translation models. *EMNLP*.

Kameyama, M. 1986. A property-sharing constraint in centering. *ACL*.

Kamp, H. 1981. A theory of truth and semantic representation. In J. Groenendijk, T. Janssen, and M. Stokhof, eds, *Formal Methods in the Study of Language*, 189–222. Mathematical Centre, Amsterdam.

Kamphuis, C., A. P. de Vries, L. Boytsov, and J. Lin. 2020. Which BM25 do you mean? a large-scale reproducibility study of scoring variants. *European Conference on Information Retrieval*.

Kane, S. K., M. R. Morris, A. Paradiso, and J. Campbell. 2017. "at times avuncular and cantankerous, with the reflexes of a mongoose": Understanding self-expression through augmentative and alternative communication devices. *CSCW*.

Kaplan, J., S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. 2020. Scaling laws for neural language models. ArXiv preprint.

Kaplan, R. M. 1973. A general syntactic processor. In R. Rustin, ed., *Natural Language Processing*, 193–241. Algorithmics Press.

Karamanis, N., M. Poesio, C. Mellish, and J. Oberlander. 2004. Evaluating centering-based metrics of coherence for text structuring using a reliably annotated corpus. *ACL*.

Karita, S., N. Chen, T. Hayashi, T. Hori, H. Inaguma, Z. Jiang, M. Someki, N. E. Y. Soplin, R. Yamamoto, X. Wang, S. Watanabe, T. Yoshimura, and W. Zhang. 2019. A comparative study on transformer vs RNN in speech applications. *IEEE ASRU-19*.

Karlsson, F., A. Voutilainen, J. Heikkilä, and A. Anttila, eds. 1995. *Constraint Grammar: A Language-Independent System for Parsing Unrestricted Text*. Mouton de Gruyter.

Karpukhin, V., B. Oğuz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W.-t. Yih. 2020. Dense passage retrieval for open-domain question answering. *EMNLP*.

Karttunen, L. 1969. Discourse referents. *COLING*. Preprint No. 70.

Karttunen, L. 1999. Comments on Joshi. In A. Kornai, ed., *Extended Finite State Models of Language*, 16–18. Cambridge University Press.

Kasami, T. 1965. An efficient recognition and syntax analysis algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA.

Katz, J. J. and J. A. Fodor. 1963. The structure of a semantic theory. *Language*, 39:170–210.

Kay, M. 1967. Experiments with a powerful parser. *COLING*.

Kay, M. 1973. The MIND system. In R. Rustin, ed., *Natural Language Processing*, 155–188. Algorithmics Press.

Kay, M. 1982. Algorithm schemata and data structures in syntactic processing. In S. Allén, ed., *Text Processing: Text Analysis and Generation, Text Typology and Attribution*, 327–358. Almqvist and Wiksell, Stockholm.

Kay, M. and M. Röscheisen. 1988. Text-translation alignment. Technical Report P90-00143, Xerox Palo Alto Research Center, Palo Alto, CA.

Kay, M. and M. Röscheisen. 1993. Text-translation alignment. *Computational Linguistics*, 19:121–142.

Kehler, A. 1993. The effect of establishing coherence in ellipsis and anaphora resolution. *ACL*.

Kehler, A. 1994. Temporal relations: Reference or discourse coherence? *ACL*.

Kehler, A. 1997a. Current theories of centering for pronoun interpretation: A critical evaluation. *Computational Linguistics*, 23(3):467–475.

Kehler, A. 1997b. Probabilistic coreference in information extraction. *EMNLP*.

Kehler, A. 2000. *Coherence, Reference, and the Theory of Grammar*. CSLI Publications.

Kehler, A., D. E. Appelt, L. Taylor, and A. Simma. 2004. The (non)utility of predicate-argument frequencies for pronoun interpretation. *HLT-NAACL*.

Kehler, A. and H. Rohde. 2013. A probabilistic reconciliation of coherence-driven and centering-driven theories of pronoun interpretation. *Theoretical Linguistics*, 39(1-2):1–37.

Keller, F. and M. Lapata. 2003. Using the web to obtain frequencies for unseen bigrams. *Computational Linguistics*, 29:459–484.

Kendall, T. and C. Farrington. 2020. The Corpus of Regional African American Language. Version 2020.05. Eugene, OR: The Online Resources for African American Language Project. http://oraal.uoregon.edu/coraal.

Kennedy, C. and B. K. Boguraev. 1996. Anaphora for everyone: Pronominal anaphora resolution without a parser. *COLING*.

Khandelwal, U., O. Levy, D. Jurafsky, L. Zettlemoyer, and M. Lewis. 2019. Generalization through memorization: Nearest neighbor language models. *ICLR*.

Khattab, O., C. Potts, and M. Zaharia. 2021. Relevance-guided supervision for OpenQA with ColBERT. *TACL*, 9:929–944.

Khattab, O., A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, S. Haq, A. Sharma, T. T. Joshi, H. Moazam, H. Miller, M. Zaharia, and C. Potts. 2024. DSPy: Compiling declarative language model calls into self-improving pipelines. *ICLR*.

Khattab, O. and M. Zaharia. 2020. ColBERT: Efficient and effective passage search via contextualized late interaction over BERT. *SIGIR*.

Kiela, D., M. Bartolo, Y. Nie, D. Kaushik, A. Geiger, Z. Wu, B. Vidgen, G. Prasad, A. Singh, P. Ringshia, Z. Ma, T. Thrush, S. Riedel, Z. Waseem, P. Stenetorp, R. Jia, M. Bansal, C. Potts, and A. Williams. 2021. Dynabench: Rethinking benchmarking in NLP. *NAACL HLT*.

Kiela, D. and S. Clark. 2014. A systematic study of semantic vector space model parameters. *EACL 2nd Workshop on Continuous Vector Space Models and their Compositionality (CVSC)*.

Kim, E. 2019. Optimize computational efficiency of skip-gram with negative sampling. https://aegis4048.github.io/optimize_computational_efficiency_of_skip-gram_with_negative_sampling.

Kim, S. M. and E. H. Hovy. 2004. Determining the sentiment of opinions. *COLING*.

King, S. 2020. From African American Vernacular English to African American Language: Rethinking the study of race and language in African Americans' speech. *Annual Review of Linguistics*, 6:285–300.

Kingma, D. and J. Ba. 2015. Adam: A method for stochastic optimization. *ICLR 2015*.

Kintsch, W. and T. A. Van Dijk. 1978. Toward a model of text comprehension and production. *Psychological review*, 85(5):363–394.

Kiperwasser, E. and Y. Goldberg. 2016. Simple and accurate dependency parsing using bidirectional LSTM feature representations. *TACL*, 4:313–327.

Kipper, K., H. T. Dang, and M. Palmer. 2000. Class-based construction of a verb lexicon. *AAAI*.

Kiritchenko, S. and S. M. Mohammad. 2017. Best-worst scaling more reliable than rating scales: A case study on sentiment intensity annotation. *ACL*.

Kiritchenko, S. and S. M. Mohammad. 2018. Examining gender and race bias in two hundred sentiment analysis systems. **SEM*.

Kiss, T. and J. Strunk. 2006. Unsupervised multilingual sentence boundary detection. *Computational Linguistics*, 32(4):485–525.

Kitaev, N., S. Cao, and D. Klein. 2019. Multilingual constituency parsing with self-attention and pre-training. *ACL*.

Kitaev, N. and D. Klein. 2018. Constituency parsing with a self-attentive encoder. *ACL*.

Klatt, D. H. 1975. Voice onset time, friction, and aspiration in word-initial consonant clusters. *Journal of Speech and Hearing Research*, 18:686–706.

Klatt, D. H. 1977. Review of the ARPA speech understanding project. *JASA*, 62(6):1345–1366.

Klatt, D. H. 1982. The Klattalk text-to-speech conversion system. *ICASSP*.

Kleene, S. C. 1951. Representation of events in nerve nets and finite automata. Technical Report RM-704, RAND Corporation. RAND Research Memorandum.

Kleene, S. C. 1956. Representation of events in nerve nets and finite automata. In C. Shannon and J. McCarthy, eds, *Automata Studies*, 3–41. Princeton University Press.

Klein, S. and R. F. Simmons. 1963. A computational approach to grammatical coding of English words. *Journal of the ACM*, 10(3):334–347.

Knott, A. and R. Dale. 1994. Using linguistic phenomena to motivate a set of coherence relations. *Discourse Processes*, 18(1):35–62.

Kocijan, V., A.-M. Cretu, O.-M. Camburu, Y. Yordanov, and T. Lukasiewicz. 2019. A surprisingly robust trick for the Winograd Schema Challenge. *ACL*.

Kocmi, T., C. Federmann, R. Grundkiewicz, M. Junczys-Dowmunt, H. Matsushita, and A. Menezes. 2021. To ship or not to ship: An extensive evaluation of automatic metrics for machine translation. ArXiv.

Koehn, P. 2005. Europarl: A parallel corpus for statistical machine translation. *MT summit, vol. 5*.

Koehn, P., H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, C. Dyer, O. Bojar, A. Constantin, and E. Herbst. 2006. Moses: Open source toolkit for statistical machine translation. *ACL*.

Koehn, P., F. J. Och, and D. Marcu. 2003. Statistical phrase-based translation. *HLT-NAACL*.

Koenig, W., H. K. Dunn, and L. Y. Lacy. 1946. The sound spectrograph. *JASA*, 18:19–49.

Kolhatkar, V., A. Roussel, S. Dipper, and H. Zinsmeister. 2018. Anaphora with non-nominal antecedents in computational linguistics: A survey. *Computational Linguistics*, 44(3):547–612.

Kreutzer, J., I. Caswell, L. Wang, A. Wahab, D. van Esch, N. Ulzii-Orshikh, A. Tapo, N. Subramani, A. Sokolov, C. Sikasote, M. Setyawan, S. Sarin, S. Samb, B. Sagot, C. Rivera, A. Rios, I. Papadimitriou, S. Osei, P. O. Suarez, I. Orife, K. Ogueji, A. N. Rubungo, T. Q. Nguyen, M. Müller, A. Müller, S. H. Muhammad, N. Muhammad, A. Mnyakeni, J. Mirzakhalov, T. Matangira, C. Leong, N. Lawson, S. Kudugunta, Y. Jernite, M. Jenny, O. Firat, B. F. P. Dossou, S. Dlamini, N. de Silva, S. Çabuk Ballı, S. Biderman, A. Battisti, A. Baruwa, A. Bapna, P. Baljekar, I. A. Azime, A. Awokoya, D. Ataman, O. Ahia, O. Ahia, S. Agrawal, and M. Adeyemi. 2022. Quality at a glance: An audit of web-crawled multilingual datasets. *TACL*, 10:50–72.

Kruskal, J. B. 1983. An overview of sequence comparison. In D. Sankoff and J. B. Kruskal, eds, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, 1–44. Addison-Wesley.

Kudo, T. 2018. Subword regularization: Improving neural network translation models with multiple subword candidates. *ACL*.

Kudo, T. and Y. Matsumoto. 2002. Japanese dependency analysis using cascaded chunking. *CoNLL*.

Kudo, T. and J. Richardson. 2018a. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *EMNLP*.

Kudo, T. and J. Richardson. 2018b. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *EMNLP*.

Kullback, S. and R. A. Leibler. 1951. On information and sufficiency. *Annals of Mathematical Statistics*, 22:79–86.

Kulmizev, A., M. de Lhoneux, J. Gontrum, E. Fano, and J. Nivre. 2019. Deep contextualized word embeddings in transition-based and graph-based dependency parsing - a tale of two parsers revisited. *EMNLP*.

Kumar, S. and W. Byrne. 2004. Minimum Bayes-risk decoding for statistical machine translation. *HLT-NAACL*.

Kummerfeld, J. K. and D. Klein. 2013. Error-driven analysis of challenges in coreference resolution. *EMNLP*.

Kuno, S. 1965. The predictive analyzer and a path elimination technique. *CACM*, 8(7):453–462.

Kupiec, J. 1992. Robust part-of-speech tagging using a hidden Markov model. *Computer Speech and Language*, 6:225–242.

Kurebito, M. 2017. Koryak. In M. Fortescue, M. Mithun, and N. Evans, eds, *Oxford Handbook of Polysynthesis*. Oxford.

Kurita, K., N. Vyas, A. Pareek, A. W. Black, and Y. Tsvetkov. 2019. Quantifying social biases in contextual word representations. *1st ACL Workshop on Gender Bias for Natural Language Processing*.

Kwiatkowski, T., J. Palomaki, O. Redfield, M. Collins, A. Parikh, C. Alberti, D. Epstein, I. Polosukhin, J. Devlin, K. Lee, K. Toutanova, L. Jones, M. Kelcey, M.-W. Chang, A. M. Dai, J. Uszkoreit, Q. Le, and S. Petrov. 2019. Natural questions: A benchmark for question answering research. *TACL*, 7:452–466.

Ladefoged, P. 1993. *A Course in Phonetics*. Harcourt Brace Jovanovich. (3rd ed.).

Ladefoged, P. 1996. *Elements of Acoustic Phonetics*, 2nd edition. University of Chicago.

Lafferty, J. D., A. McCallum, and F. C. N. Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. *ICML*.

Lai, A. and J. Tetreault. 2018. Discourse coherence in the wild: A dataset, evaluation and methods. *SIGDIAL*.

Lake, B. M. and G. L. Murphy. 2021. Word meaning in minds and machines. *Psychological Review*. In press.

Lakoff, G. 1965. *On the Nature of Syntactic Irregularity*. Ph.D. thesis, Indiana University. Published as *Irregularity in Syntax*. Holt, Rinehart, and Winston, New York, 1970.

Lakoff, G. 1972. Structural complexity in fairy tales. In *The Study of Man*, 128–50. School of Social Sciences, University of California, Irvine, CA.

Lakoff, G. and M. Johnson. 1980. *Metaphors We Live By*. University of Chicago Press, Chicago, IL.

Lambert, N., L. Tunstall, N. Rajani, and T. Thrush. 2023. Huggingface h4 stack exchange preference dataset.

Lample, G., M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer. 2016. Neural architectures for named entity recognition. *NAACL HLT*.

Lample, G. and A. Conneau. 2019. Cross-lingual language model pretraining. *NeurIPS*, volume 32.

Lan, Z., M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut. 2020. ALBERT: A lite BERT for self-supervised learning of language representations. *ICLR*.

Landauer, T. K. and S. T. Dumais. 1997. A solution to Plato's problem: The Latent Semantic Analysis theory of acquisition, induction, and representation of knowledge. *Psychological Review*, 104:211–240.

Landauer, T. K., D. Laham, B. Rehder, and M. E. Schreiner. 1997. How well can passage meaning be derived without using word order? A comparison of Latent Semantic Analysis and humans. *COGSCI*.

Lang, J. and M. Lapata. 2014. Similarity-driven semantic role induction via graph partitioning. *Computational Linguistics*, 40(3):633–669.

Lang, K. J., A. H. Waibel, and G. E. Hinton. 1990. A time-delay neural network architecture for isolated word recognition. *Neural networks*, 3(1):23–43.

Lapata, M. 2003. Probabilistic text structuring: Experiments with sentence ordering. *ACL*.

Lapesa, G. and S. Evert. 2014. A large scale evaluation of distributional semantic models: Parameters, interactions and model selection. *TACL*, 2:531–545.

Lappin, S. and H. Leass. 1994. An algorithm for pronominal anaphora resolution. *Computational Linguistics*, 20(4):535–561.

Lascarides, A. and N. Asher. 1993. Temporal interpretation, discourse relations, and common sense entailment. *Linguistics and Philosophy*, 16(5):437–493.

Lawrence, W. 1953. The synthesis of speech from signals which have a low information rate. In W. Jackson, ed., *Communication Theory*, 460–469. Butterworth.

LDC. 1998. *LDC Catalog: Hub4 project*. University of Pennsylvania. www.ldc.upenn.edu/Catalog/LDC98S71.html.

LeCun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. 1989. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551.

Lee, D. D. and H. S. Seung. 1999. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788–791.

Lee, H., A. Chang, Y. Peirsman, N. Chambers, M. Surdeanu, and D. Jurafsky. 2013. Deterministic coreference resolution based on entity-centric, precision-ranked rules. *Computational Linguistics*, 39(4):885–916.

Lee, H., Y. Peirsman, A. Chang, N. Chambers, M. Surdeanu, and D. Jurafsky. 2011. Stanford's multi-pass sieve coreference resolution system at the CoNLL-2011 shared task. *CoNLL*.

Lee, H., M. Surdeanu, and D. Jurafsky. 2017a. A scaffolding approach to coreference resolution integrating statistical and rule-based models. *Natural Language Engineering*, 23(5):733–762.

Lee, K., M.-W. Chang, and K. Toutanova. 2019. Latent retrieval for weakly supervised open domain question answering. *ACL*.

Lee, K., L. He, M. Lewis, and L. Zettlemoyer. 2017b. End-to-end neural coreference resolution. *EMNLP*.

Lee, K., L. He, and L. Zettlemoyer. 2018. Higher-order coreference resolution with coarse-to-fine inference. *NAACL HLT*.

Lehiste, I., ed. 1967. *Readings in Acoustic Phonetics*. MIT Press.

Lehnert, W. G., C. Cardie, D. Fisher, E. Riloff, and R. Williams. 1991. Description of the CIRCUS system as used for MUC-3. *MUC-3*.

Levenshtein, V. I. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, 10(8):707–710. Original in *Doklady Akademii Nauk SSSR* 163(4): 845–848 (1965).

Levesque, H. 2011. The Winograd Schema Challenge. *Logical Formalizations of Commonsense Reasoning — Papers from the AAAI 2011 Spring Symposium (SS-11-06)*.

Levesque, H., E. Davis, and L. Morgenstern. 2012. The Winograd Schema Challenge. *KR-12*.

Levin, B. 1977. Mapping sentences to case frames. Technical Report 167, MIT AI Laboratory. AI Working Paper 143.

Levin, B. 1993. *English Verb Classes and Alternations: A Preliminary Investigation*. University of Chicago Press.

Levin, B. and M. Rappaport Hovav. 2005. *Argument Realization*. Cambridge University Press.

Levy, O. and Y. Goldberg. 2014a. Dependency-based word embeddings. *ACL*.

Levy, O. and Y. Goldberg. 2014b. Linguistic regularities in sparse and explicit word representations. *CoNLL*.

Levy, O. and Y. Goldberg. 2014c. Neural word embedding as implicit matrix factorization. *NeurIPS*.

Levy, O., Y. Goldberg, and I. Dagan. 2015. Improving distributional similarity with lessons learned from word embeddings. *TACL*, 3:211–225.

Li, A., F. Zheng, W. Byrne, P. Fung, T. Kamm, L. Yi, Z. Song, U. Ruhi, V. Venkataramani, and X. Chen. 2000. CASS: A phonetically transcribed corpus of Mandarin spontaneous speech. *ICSLP*.

Li, B. Z., S. Min, S. Iyer, Y. Mehdad, and W.-t. Yih. 2020. Efficient one-pass end-to-end entity linking for questions. *EMNLP*.

Li, J. and D. Jurafsky. 2017. Neural net models of open-domain discourse coherence. *EMNLP*.

Li, J., R. Li, and E. H. Hovy. 2014. Recursive deep models for discourse parsing. *EMNLP*.

Li, Q., T. Li, and B. Chang. 2016. Discourse parsing with attention-based hierarchical neural networks. *EMNLP*.

Li, X., Y. Meng, X. Sun, Q. Han, A. Yuan, and J. Li. 2019. Is word segmentation necessary for deep learning of Chinese representations? *ACL*.

Liang, P., R. Bommasani, T. Lee, D. Tsipras, D. Soylu, M. Yasunaga, Y. Zhang, D. Narayanan, Y. Wu, A. Kumar, B. Newman, B. Yuan, B. Yan, C. Zhang, C. Cosgrove,

C. D. Manning, C. Ré, D. Acosta-Navas, D. A. Hudson, E. Zelikman, E. Durmus, F. Ladhak, F. Rong, H. Ren, H. Yao, J. Wang, K. Santhanam, L. Orr, L. Zheng, M. Yuksekgonul, M. Suzgun, N. Kim, N. Guha, N. Chatterji, O. Khattab, P. Henderson, Q. Huang, R. Chi, S. M. Xie, S. Santurkar, S. Ganguli, T. Hashimoto, T. Icard, T. Zhang, V. Chaudhary, W. Wang, X. Li, Y. Mai, Y. Zhang, and Y. Koreeda. 2023. Holistic evaluation of language models. *Transactions on Machine Learning Research*.

Liberman, A. M., P. C. Delattre, and F. S. Cooper. 1952. The role of selected stimulus variables in the perception of the unvoiced stop consonants. *American Journal of Psychology*, 65:497–516.

Lin, D. 2003. Dependency-based evaluation of minipar. *Workshop on the Evaluation of Parsing Systems*.

Lin, Y., J.-B. Michel, E. Aiden Lieberman, J. Orwant, W. Brockman, and S. Petrov. 2012a. Syntactic annotations for the Google books NGram corpus. *ACL*.

Lin, Y., J.-B. Michel, E. Lieberman Aiden, J. Orwant, W. Brockman, and S. Petrov. 2012b. Syntactic annotations for the Google Books NGram corpus. *ACL*.

Lin, Z., M.-Y. Kan, and H. T. Ng. 2009. Recognizing implicit discourse relations in the Penn Discourse Treebank. *EMNLP*.

Lin, Z., H. T. Ng, and M.-Y. Kan. 2011. Automatically evaluating text coherence using discourse relations. *ACL*.

Lin, Z., H. T. Ng, and M.-Y. Kan. 2014. A pdtb-styled end-to-end discourse parser. *Natural Language Engineering*, 20(2):151–184.

Ling, W., C. Dyer, A. W. Black, I. Trancoso, R. Fermandez, S. Amir, L. Marujo, and T. Luís. 2015. Finding function in form: Compositional character models for open vocabulary word representation. *EMNLP*.

Linzen, T. 2016. Issues in evaluating semantic spaces using word analogies. *1st Workshop on Evaluating Vector-Space Representations for NLP*.

Lison, P. and J. Tiedemann. 2016. Opensubtitles2016: Extracting large parallel corpora from movie and tv subtitles. *LREC*.

Litman, D. J. 1985. *Plan Recognition and Discourse Analysis: An Integrated Approach for Understanding Dialogues*. Ph.D. thesis, University of Rochester, Rochester, NY.

Litman, D. J. and J. Allen. 1987. A plan recognition model for subdialogues in conversation. *Cognitive Science*, 11:163–200.

Liu, A., J. Hayase, V. Hofmann, S. Oh, N. A. Smith, and Y. Choi. 2025. SuperBPE: Space travel for language models. ArXiv preprint.

Liu, B. and L. Zhang. 2012. A survey of opinion mining and sentiment analysis. In C. C. Aggarwal and C. Zhai, eds, *Mining text data*, 415–464. Springer.

Liu, H., J. Dacon, W. Fan, H. Liu, Z. Liu, and J. Tang. 2020. Does gender matter? Towards fairness in dialogue systems. *COLING*.

Liu, J., S. Min, L. Zettlemoyer, Y. Choi, and H. Hajishirzi. 2024. Infini-gram: Scaling unbounded n-gram language models to a trillion tokens. ArXiv preprint.

Liu, Y., C. Sun, L. Lin, and X. Wang. 2016. Learning natural language inference using bidirectional LSTM model and inner-attention. ArXiv.

Liu, Y., P. Fung, Y. Yang, C. Cieri, S. Huang, and D. Graff. 2006. HKUST/MTS: A very large scale Mandarin telephone speech corpus. *International Conference on Chinese Spoken Language Processing*.

Liu, Y., M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. 2019. RoBERTa: A robustly optimized BERT pretraining approach. ArXiv preprint arXiv:1907.11692.

Llama Team. 2024. The llama 3 herd of models.

Logeswaran, L., H. Lee, and D. Radev. 2018. Sentence ordering and coherence modeling using recurrent neural networks. *AAAI*.

Longpre, S., L. Hou, T. Vu, A. Webson, H. W. Chung, Y. Tay, D. Zhou, Q. V. Le, B. Zoph, J. Wei, and A. Roberts. 2023. The Flan collection: Designing data and methods for effective instruction tuning. *ICML*.

Longpre, S., R. Mahari, A. Lee, C. Lund, H. Oderinwale, W. Brannon, N. Saxena, N. Obeng-Marnu, T. South, C. Hunter, et al. 2024a. Consent in crisis: The rapid decline of the ai data commons. ArXiv preprint.

Longpre, S., G. Yauney, E. Reif, K. Lee, A. Roberts, B. Zoph, D. Zhou, J. Wei, K. Robinson, D. Mimno, and D. Ippolito. 2024b. A pretrainer's guide to training data: Measuring the effects of data age, domain coverage, quality, & toxicity. *NAACL HLT*.

Louis, A. and A. Nenkova. 2012. A coherence model based on syntactic patterns. *EMNLP*.

Loureiro, D. and A. Jorge. 2019. Language modelling makes sense: Propagating representations through WordNet for full-coverage word sense disambiguation. *ACL*.

Louviere, J. J., T. N. Flynn, and A. A. J. Marley. 2015. *Best-worst scaling: Theory, methods and applications*. Cambridge University Press.

Lovins, J. B. 1968. Development of a stemming algorithm. *Mechanical Translation and Computational Linguistics*, 11(1–2):9–13.

Lowerre, B. T. 1976. *The Harpy Speech Recognition System*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA.

Lukasik, M., B. Dadachev, K. Papineni, and G. Simões. 2020. Text segmentation by cross segment attention. *EMNLP*.

Luo, X. 2005. On coreference resolution performance metrics. *EMNLP*.

Luo, X. and S. Pradhan. 2016. Evaluation metrics. In M. Poesio, R. Stuckardt, and Y. Versley, eds, *Anaphora resolution: Algorithms, resources, and applications*, 141–163. Springer.

Luo, X., S. Pradhan, M. Recasens, and E. H. Hovy. 2014. An extension of BLANC to system mentions. *ACL*.

Ma, X. and E. H. Hovy. 2016. End-to-end sequence labeling via bi-directional LSTM-CNNs-CRF. *ACL*.

Maas, A., Z. Xie, D. Jurafsky, and A. Y. Ng. 2015. Lexicon-free conversational speech recognition with neural networks. *NAACL HLT*.

Maas, A. L., A. Y. Hannun, and A. Y. Ng. 2013. Rectifier nonlinearities improve neural network acoustic models. *ICML*.

Maas, A. L., P. Qi, Z. Xie, A. Y. Hannun, C. T. Lengerich, D. Jurafsky, and A. Y. Ng. 2017. Building dnn acoustic models for large vocabulary speech recognition. *Computer Speech & Language*, 41:195–213.

Magerman, D. M. 1995. Statistical decision-tree models for parsing. *ACL*.

Mairesse, F. and M. A. Walker. 2008. Trainable generation of big-five personality styles through data-driven parameter estimation. *ACL*.

Mann, W. C. and S. A. Thompson. 1987. Rhetorical structure theory: A theory of text organization. Technical Report RS-87-190, Information Sciences Institute.

Manning, C. D. 2011. Part-of-speech tagging from 97% to 100%: Is it time for some linguistics? *CICLing 2011*.

Manning, C. D., P. Raghavan, and H. Schütze. 2008. *Introduction to Information Retrieval*. Cambridge.

Manning, C. D., M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, and D. McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. *ACL*.

Marcu, D. 1997. The rhetorical parsing of natural language texts. *ACL*.

Marcu, D. 1999. A decision-based approach to rhetorical parsing. *ACL*.

Marcu, D. 2000a. The rhetorical parsing of unrestricted texts: A surface-based approach. *Computational Linguistics*, 26(3):395–448.

Marcu, D., ed. 2000b. *The Theory and Practice of Discourse Parsing and Summarization*. MIT Press.

Marcu, D. and A. Echihabi. 2002. An unsupervised approach to recognizing discourse relations. *ACL*.

Marcu, D. and W. Wong. 2002. A phrase-based, joint probability model for statistical machine translation. *EMNLP*.

Marcus, M. P. 1980. *A Theory of Syntactic Recognition for Natural Language*. MIT Press.

Marcus, M. P., B. Santorini, and M. A. Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn treebank. *Computational Linguistics*, 19(2):313–330.

Marie, B., A. Fujita, and R. Rubino. 2021. Scientific credibility of machine translation research: A meta-evaluation of 769 papers. *ACL*.

Markov, A. A. 1913. Essai d'une recherche statistique sur le texte du roman "Eugene Onegin" illustrant la liaison des epreuve en chain ('Example of a statistical investigation of the text of "Eugene Onegin" illustrating the dependence between samples in chain'). *Izvistia Imperatorskoi Akademii Nauk (Bulletin de l'Académie Impériale des Sciences de St.-Pétersbourg)*, 7:153–162.

de Marneffe, M.-C., T. Dozat, N. Silveira, K. Haverinen, F. Ginter, J. Nivre, and C. D. Manning. 2014. Universal Stanford dependencies: A cross-linguistic typology. *LREC*.

de Marneffe, M.-C., B. MacCartney, and C. D. Manning. 2006. Generating typed dependency parses from phrase structure parses. *LREC*.

de Marneffe, M.-C. and C. D. Manning. 2008. The Stanford typed dependencies representation. *COLING Workshop on Cross-Framework and Cross-Domain Parser Evaluation*.

de Marneffe, M.-C., C. D. Manning, J. Nivre, and D. Zeman. 2021. Universal Dependencies. *Computational Linguistics*, 47(2):255–308.

de Marneffe, M.-C., M. Recasens, and C. Potts. 2015. Modeling the lifespan of discourse entities with application to coreference resolution. *JAIR*, 52:445–475.

Màrquez, L., X. Carreras, K. C. Litkowski, and S. Stevenson. 2008. Semantic role labeling: An introduction to the special issue. *Computational linguistics*, 34(2):145–159.

Marshall, I. 1983. Choice of grammatical word-class without global syntactic analysis: Tagging words in the LOB corpus. *Computers and the Humanities*, 17:139–150.

Marshall, I. 1987. Tag selection using probabilistic methods. In R. Garside, G. Leech, and G. Sampson, eds, *The Computational Analysis of English*, 42–56. Longman.

Martschat, S. and M. Strube. 2014. Recall error analysis for coreference resolution. *EMNLP*.

Martschat, S. and M. Strube. 2015. Latent structures for coreference resolution. *TACL*, 3:405–418.

Mathis, D. A. and M. C. Mozer. 1995. On the computational utility of consciousness. *NeurIPS*. MIT Press.

McCallum, A., D. Freitag, and F. C. N. Pereira. 2000. Maximum entropy Markov models for information extraction and segmentation. *ICML*.

McCallum, A. and W. Li. 2003. Early results for named entity recognition with conditional random fields, feature induction and web-enhanced lexicons. *CoNLL*.

McCarthy, J. F. and W. G. Lehnert. 1995. Using decision trees for coreference resolution. *IJCAI-95*.

McClelland, J. L. and J. L. Elman. 1986. The TRACE model of speech perception. *Cognitive Psychology*, 18:1–86.

McClelland, J. L. and D. E. Rumelhart, eds. 1986. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 2: *Psychological and Biological Models*. MIT Press.

McCulloch, W. S. and W. Pitts. 1943. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133.

McDonald, R., K. Crammer, and F. C. N. Pereira. 2005a. Online large-margin training of dependency parsers. *ACL*.

McDonald, R. and J. Nivre. 2011. Analyzing and integrating dependency parsers. *Computational Linguistics*, 37(1):197–230.

McDonald, R., F. C. N. Pereira, K. Ribarov, and J. Hajič. 2005b. Non-projective dependency parsing using spanning tree algorithms. *HLT-EMNLP*.

McGuffie, K. and A. Newhouse. 2020. The radicalization risks of GPT-3 and advanced neural language models. ArXiv preprint arXiv:2009.06807.

McLuhan, M. 1964. *Understanding Media: The Extensions of Man*. New American Library.

Melamud, O., J. Goldberger, and I. Dagan. 2016. context2vec: Learning generic context embedding with bidirectional LSTM. *CoNLL*.

Meng, K., D. Bau, A. Andonian, and Y. Belinkov. 2022. Locating and editing factual associations in GPT. *NeurIPS*, volume 36.

Merialdo, B. 1994. Tagging English text with a probabilistic model. *Computational Linguistics*, 20(2):155–172.

Mesgar, M. and M. Strube. 2016. Lexical coherence graph modeling using word embeddings. *ACL*.

Meyers, A., R. Reeves, C. Macleod, R. Szekely, V. Zielinska, B. Young, and R. Grishman. 2004. The nombank project: An interim report. *NAACL/HLT Workshop: Frontiers in Corpus Annotation*.

Mihalcea, R. and A. Csomai. 2007. Wikify!: Linking documents to encyclopedic knowledge. *CIKM 2007*.

Mikheev, A., M. Moens, and C. Grover. 1999. Named entity recognition without gazetteers. *EACL*.

Mikolov, T. 2012. *Statistical language models based on neural networks*. Ph.D. thesis, Brno University of Technology.

Mikolov, T., K. Chen, G. S. Corrado, and J. Dean. 2013a. Efficient estimation of word representations in vector space. *ICLR 2013*.

Mikolov, T., M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur. 2010. Recurrent neural network based language model. *INTERSPEECH*.

Mikolov, T., S. Kombrink, L. Burget, J. H. Černockỳ, and S. Khudanpur. 2011. Extensions of recurrent neural network language model. *ICASSP*.

Mikolov, T., I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. 2013b. Distributed representations of words and phrases and their compositionality. *NeurIPS*.

Mikolov, T., W.-t. Yih, and G. Zweig. 2013c. Linguistic regularities in continuous space word representations. *NAACL HLT*.

Miller, G. A. and P. E. Nicely. 1955. An analysis of perceptual confusions among some English consonants. *JASA*, 27:338–352.

Miller, G. A. and J. G. Beebe-Center. 1956. Some psychological methods for evaluating the quality of translations. *Mechanical Translation*, 3:73–80.

Miller, G. A. and W. G. Charles. 1991. Contextual correlates of semantics similarity. *Language and Cognitive Processes*, 6(1):1–28.

Miller, G. A. and N. Chomsky. 1963. Finitary models of language users. In R. D. Luce, R. R. Bush, and E. Galanter, eds, *Handbook of Mathematical Psychology*, volume II, 419–491. John Wiley.

Miller, G. A. and J. A. Selfridge. 1950. Verbal context and the recall of meaningful material. *American Journal of Psychology*, 63:176–185.

Milne, D. and I. H. Witten. 2008. Learning to link with wikipedia. *CIKM 2008*.

Miltsakaki, E., R. Prasad, A. K. Joshi, and B. L. Webber. 2004. The Penn Discourse Treebank. *LREC*.

Min, S., X. Lyu, A. Holtzman, M. Artetxe, M. Lewis, H. Hajishirzi, and L. Zettlemoyer. 2022. Rethinking the role of demonstrations: What makes in-context learning work? *EMNLP*.

Minsky, M. 1974. A framework for representing knowledge. Technical Report 306, MIT AI Laboratory. Memo 306.

Minsky, M. and S. Papert. 1969. *Perceptrons*. MIT Press.

Mintz, M., S. Bills, R. Snow, and D. Jurafsky. 2009. Distant supervision for relation extraction without labeled data. *ACL IJCNLP*.

Mirza, P. and S. Tonelli. 2016. CATENA: CAusal and TEmporal relation extraction from NAtural language texts. *COLING*.

Mishra, S., D. Khashabi, C. Baral, and H. Hajishirzi. 2022. Cross-task generalization via natural language crowdsourcing instructions. *ACL*.

Mitchell, M., S. Wu, A. Zaldivar, P. Barnes, L. Vasserman, B. Hutchinson, E. Spitzer, I. D. Raji, and T. Gebru. 2019. Model cards for model reporting. *ACM FAccT*.

Mitkov, R. 2002. *Anaphora Resolution*. Longman.

Mohamed, A., G. E. Dahl, and G. E. Hinton. 2009. Deep Belief Networks for phone recognition. *NIPS Workshop on Deep Learning for Speech Recognition and Related Applications*.

Mohammad, S. M. 2018a. Obtaining reliable human ratings of valence, arousal, and dominance for 20,000 English words. *ACL*.

Mohammad, S. M. 2018b. Word affect intensities. *LREC*.

Mohammad, S. M. and P. D. Turney. 2013. Crowdsourcing a word-emotion association lexicon. *Computational Intelligence*, 29(3):436–465.

Monroe, B. L., M. P. Colaresi, and K. M. Quinn. 2008. Fightin'words: Lexical feature selection and evaluation for identifying the content of political conflict. *Political Analysis*, 16(4):372–403.

Moors, A., P. C. Ellsworth, K. R. Scherer, and N. H. Frijda. 2013. Appraisal theories of emotion: State of the art and future development. *Emotion Review*, 5(2):119–124.

Moosavi, N. S. and M. Strube. 2016. Which coreference evaluation metric do you trust? A proposal for a link-based entity aware metric. *ACL*.

Morey, M., P. Muller, and N. Asher. 2017. How much progress have we made on RST discourse parsing? a replication study of recent results on the rst-dt. *EMNLP*.

Morgan, A. A., L. Hirschman, M. Colosimo, A. S. Yeh, and J. B. Colombe. 2004. Gene name identification and normalization using a model organism database. *Journal of Biomedical Informatics*, 37(6):396–410.

Morgan, N. and H. Bourlard. 1990. Continuous speech recognition using multilayer perceptrons with hidden markov models. *ICASSP*.

Morgan, N. and H. A. Bourlard. 1995. Neural networks for statistical recognition of continuous speech. *Proceedings of the IEEE*, 83(5):742–772.

Morris, J. and G. Hirst. 1991. Lexical cohesion computed by thesaural relations as an indicator of the structure of text. *Computational Linguistics*, 17(1):21–48.

Mousavi, P., G. Maimon, A. Moumen, D. Petermann, J. Shi, H. Wu, H. Yang, A. Kuznetsova, A. Ploujnikov, R. Marxer, B. Ramabhadran, B. Elizalde, L. Lugosch, J. Li, C. Subakan, P. Woodland, M. Kim, H. yi Lee, S. Watanabe, Y. Adi, and M. Ravanelli. 2025. Discrete audio tokens: More than a survey! ArXiv preprint.

Muller, P., C. Braud, and M. Morey. 2019. ToNy: Contextual embeddings for accurate multilingual discourse segmentation of full documents. *Workshop on Discourse Relation Parsing and Treebanking*.

Murphy, K. P. 2012. *Machine learning: A probabilistic perspective*. MIT Press.

Musi, E., M. Stede, L. Kriese, S. Muresan, and A. Rocci. 2018. A multi-layer annotated corpus of argumentative text: From argument schemes to discourse relations. *LREC*.

Myers, G. 1992. "In this paper we report...": Speech acts and scientific facts. *Journal of Pragmatics*, 17(4):295–313.

Nádas, A. 1984. Estimation of probabilities in the language model of the IBM speech recognition system. *IEEE Transactions on ASSP*, 32(4):859–861.

Nadeem, M., A. Bethke, and S. Reddy. 2021. StereoSet: Measuring stereotypical bias in pretrained language models. *ACL*.

Nash-Webber, B. L. 1975. The role of semantics in automatic speech understanding. In D. G. Bobrow and A. Collins, eds, *Representation and Understanding*, 351–382. Academic Press.

Naur, P., J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijnagaarden, and M. Woodger. 1960. Report on the algorithmic language ALGOL 60. *CACM*, 3(5):299–314. Revised in CACM 6:1, 1-17, 1963.

Neff, G. and P. Nagy. 2016. Talking to bots: Symbiotic agency and the case of Tay. *International Journal of Communication*, 10:4915–4931.

Nekoto, W., V. Marivate, T. Matsila, T. Fasubaa, T. Kolawole, T. Fagbohungbe, S. O. Akinola, S. H. Muhammad, S. Kabongo, S. Osei, S. Freshia, R. A. Niyongabo, R. M. P. Ogayo, O. Ahia, M. Meressa, M. Adeyemi, M. Mokgesi-Selinga, L. Okegbemi, L. J. Martinus, K. Tajudeen, K. Degila, K. Ogueji, K. Siminyu, J. Kreutzer, J. Webster, J. T. Ali, J. A. I. Orife, I. Ezeani, I. A. Dangana, H. Kamper, H. Elsahar, G. Duru, G. Kioko, E. Murhabazi, E. van Biljon, D. Whitenack, C. Onyefuluchi, C. Emezue, B. Dossou, B. Sibanda, B. I. Bassey, A. Olabiyi, A. Ramkilowan, A. Öktem, A. Akinfaderin, and A. Bashir. 2020. Participatory research for low-resourced machine translation: A case study in African languages. *Findings of EMNLP*. The authors use the forall symbol to represent the whole Masakhane community.

Ng, H. T., L. H. Teo, and J. L. P. Kwan. 2000. A machine learning approach to answering questions for reading comprehension tests. *EMNLP*.

Ng, V. 2004. Learning noun phrase anaphoricity to improve coreference

resolution: Issues in representation and optimization. *ACL*.

Ng, V. 2005a. Machine learning for coreference resolution: From local classification to global ranking. *ACL*.

Ng, V. 2005b. Supervised ranking for pronoun resolution: Some recent improvements. *AAAI*.

Ng, V. 2010. Supervised noun phrase coreference research: The first fifteen years. *ACL*.

Ng, V. 2017. Machine learning for entity coreference resolution: A retrospective look at two decades of research. *AAAI*.

Ng, V. and C. Cardie. 2002a. Identifying anaphoric and non-anaphoric noun phrases to improve coreference resolution. *COLING*.

Ng, V. and C. Cardie. 2002b. Improving machine learning approaches to coreference resolution. *ACL*.

Nguyen, D. T. and S. Joty. 2017. A neural local coherence model. *ACL*.

Nickerson, R. S. 1976. On conversational interaction with computers. *Proceedings of the ACM/SIGGRAPH workshop on User-oriented design of interactive graphics systems*.

Nie, A., E. Bennett, and N. Goodman. 2019. DisSent: Learning sentence representations from explicit discourse relations. *ACL*.

Nielsen, M. A. 2015. *Neural networks and Deep learning*. Determination Press USA.

Nigam, K., J. D. Lafferty, and A. McCallum. 1999. Using maximum entropy for text classification. *IJCAI-99 workshop on machine learning for information filtering*.

Nirenburg, S., H. L. Somers, and Y. Wilks, eds. 2002. *Readings in Machine Translation*. MIT Press.

Nissim, M., S. Dingare, J. Carletta, and M. Steedman. 2004. An annotation scheme for information status in dialogue. *LREC*.

NIST. 1990. TIMIT Acoustic-Phonetic Continuous Speech Corpus. National Institute of Standards and Technology Speech Disc 1-1.1. NIST Order No. PB91-505065.

NIST. 2005. Speech recognition scoring toolkit (sctk) version 2.1. `http://www.nist.gov/speech/tools/`.

NIST. 2007. Matched Pairs Sentence-Segment Word Error (MAPSSWE) Test.

Nivre, J. 2007. Incremental non-projective dependency parsing. *NAACL-HLT*.

Nivre, J. 2003. An efficient algorithm for projective dependency parsing. *IWPT-03*.

Nivre, J. 2006. *Inductive Dependency Parsing*. Springer.

Nivre, J. 2009. Non-projective dependency parsing in expected linear time. *ACL IJCNLP*.

Nivre, J., J. Hall, S. Kübler, R. McDonald, J. Nilsson, S. Riedel, and D. Yuret. 2007a. The conll 2007 shared task on dependency parsing. *EMNLP/CoNLL*.

Nivre, J., J. Hall, J. Nilsson, A. Chanev, G. Eryigit, S. Kübler, S. Marinov, and E. Marsi. 2007b. Maltparser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(02):95–135.

Nivre, J. and J. Nilsson. 2005. Pseudo-projective dependency parsing. *ACL*.

Nivre, J. and M. Scholz. 2004. Deterministic dependency parsing of english text. *COLING*.

Noreen, E. W. 1989. *Computer Intensive Methods for Testing Hypothesis*. Wiley.

Norman, D. A. 1988. *The Design of Everyday Things*. Basic Books.

Norvig, P. 1991. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91–98.

Nosek, B. A., M. R. Banaji, and A. G. Greenwald. 2002a. Harvesting implicit group attitudes and beliefs from a demonstration web site. *Group Dynamics: Theory, Research, and Practice*, 6(1):101.

Nosek, B. A., M. R. Banaji, and A. G. Greenwald. 2002b. Math=male, me=female, therefore math≠ me. *Journal of personality and social psychology*, 83(1):44.

Nostalgebraist. 2020. Interpreting gpt: the logit lens. White paper.

Ocal, M., A. Perez, A. Radas, and M. Finlayson. 2022. Holistic evaluation of automatic TimeML annotators. *LREC*.

Och, F. J. 1998. *Ein beispielsbasierter und statistischer Ansatz zum maschinellen Lernen von natürlichsprachlicher Übersetzung*. Ph.D. thesis, Universität Erlangen-Nürnberg, Germany. Diplomarbeit (diploma thesis).

Och, F. J. 2003. Minimum error rate training in statistical machine translation. *ACL*.

Och, F. J. and H. Ney. 2002. Discriminative training and maximum entropy models for statistical machine translation. *ACL*.

Och, F. J. and H. Ney. 2003. A systematic comparison of various statistical alignment models. *Computational Linguistics*, 29(1):19–51.

Och, F. J. and H. Ney. 2004. The alignment template approach to statistical machine translation. *Computational Linguistics*, 30(4):417–449.

Olive, J. P. 1977. Rule synthesis of speech from dyadic units. *ICASSP77*.

Olsson, C., N. Elhage, N. Nanda, N. Joseph, N. DasSarma, T. Henighan, B. Mann, A. Askell, Y. Bai, A. Chen, et al. 2022. In-context learning and induction heads. ArXiv preprint.

Oppenheim, A. V., R. W. Schafer, and T. G. J. Stockham. 1968. Nonlinear filtering of multiplied and convolved signals. *Proceedings of the IEEE*, 56(8):1264–1291.

Oravecz, C. and P. Dienes. 2002. Efficient stochastic part-of-speech tagging for Hungarian. *LREC*.

Osgood, C. E., G. J. Suci, and P. H. Tannenbaum. 1957. *The Measurement of Meaning*. University of Illinois Press.

Ostendorf, M., P. Price, and S. Shattuck-Hufnagel. 1995. The Boston University Radio News Corpus. Technical Report ECS-95-001, Boston University.

Ouyang, L., J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe. 2022. Training language models to follow instructions with human feedback. *NeurIPS*, volume 35.

Packard, D. W. 1973. Computer-assisted morphological analysis of ancient Greek. *COLING*.

Palmer, M., D. Gildea, and N. Xue. 2010. Semantic role labeling. *Synthesis Lectures on Human Language Technologies*, 3(1):1–103.

Palmer, M., P. Kingsbury, and D. Gildea. 2005. The proposition bank: An annotated corpus of semantic roles. *Computational Linguistics*, 31(1):71–106.

Panayotov, V., G. Chen, D. Povey, and S. Khudanpur. 2015. Librispeech: an ASR corpus based on public domain audio books. *ICASSP*.

Pang, B. and L. Lee. 2008. Opinion mining and sentiment analysis. *Foundations and trends in information retrieval*, 2(1-2):1–135.

Pang, B., L. Lee, and S. Vaithyanathan. 2002. Thumbs up? Sentiment classification using machine learning techniques. *EMNLP*.

Papadimitriou, I., K. Lopez, and D. Jurafsky. 2023. Multilingual BERT has an accent: Evaluating English influences on fluency in multilingual models. *EACL Findings*.

Papineni, K., S. Roukos, T. Ward, and W.-J. Zhu. 2002. Bleu: A method for automatic evaluation of machine translation. *ACL*.

Park, J. H., J. Shin, and P. Fung. 2018. Reducing gender bias in abusive language detection. *EMNLP*.

Park, J. and C. Cardie. 2014. Identifying appropriate support for propositions in online user comments. *First workshop on argumentation mining*.

Parrish, A., A. Chen, N. Nangia, V. Padmakumar, J. Phang, J. Thompson, P. M. Htut, and S. Bowman. 2022. BBQ: A hand-built bias benchmark for question answering. *Findings of ACL 2022*.

Paszke, A., S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. 2017. Automatic differentiation in pytorch. *NIPS-W*.

Peldszus, A. and M. Stede. 2013. From argument diagrams to argumentation mining in texts: A survey. *International Journal of Cognitive Informatics and Natural Intelligence (IJCINI)*, 7(1):1–31.

Peldszus, A. and M. Stede. 2016. An annotated corpus of argumentative microtexts. *1st European Conference on Argumentation*.

Peng, Y., J. Tian, B. Yan, D. Berrebbi, X. Chang, X. Li, J. Shi, S. Arora, W. Chen, R. Sharma, W. Zhang, Y. Sudo, M. Shakee, J. weon Jung, S. Maiti, and S. Watanabe. 2023. Reproducing whisper-style training using an open-source toolkit and publicly available data. *ASRU*.

Penn, G. and P. Kiparsky. 2012. On Pāṇini and the generative capacity of contextualized replacement systems. *COLING*.

Pennebaker, J. W., R. J. Booth, and M. E. Francis. 2007. *Linguistic Inquiry and Word Count: LIWC 2007*. Austin, TX.

Pennington, J., R. Socher, and C. D. Manning. 2014. GloVe: Global vectors for word representation. *EMNLP*.

Percival, W. K. 1976. On the historical source of immediate constituent analysis. In J. D. McCawley, ed., *Syntax and Semantics Volume 7, Notes from the Linguistic Underground*, 229–242. Academic Press.

Peters, M., M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. 2018. Deep contextualized word representations. *NAACL HLT*.

Peterson, G. E. and H. L. Barney. 1952. Control methods used in a study of the vowels. *JASA*, 24:175–184.

Peterson, G. E., W. S.-Y. Wang, and E. Sivertsen. 1958. Segmentation techniques in speech synthesis. *JASA*, 30(8):739–742.

Peterson, J. C., D. Chen, and T. L. Griffiths. 2020. Parallelograms revisited: Exploring the limitations of vector space models for simple analogies. *Cognition*, 205.

Petroni, F., T. Rocktäschel, S. Riedel, P. Lewis, A. Bakhtin, Y. Wu, and A. Miller. 2019. Language models as knowledge bases? *EMNLP*.

Petrov, S., D. Das, and R. McDonald. 2012. A universal part-of-speech tagset. *LREC*.

Petrov, S. and R. McDonald. 2012. Overview of the 2012 shared task on parsing the web. *Notes of the First Workshop on Syntactic Analysis of Non-Canonical Language (SANCL)*, volume 59.

Picard, R. W. 1995. Affective computing. Technical Report 321, MIT Media Lab Perceputal Computing Technical Report. Revised November 26, 1995.

Pierce, J. R., J. B. Carroll, E. P. Hamp, D. G. Hays, C. F. Hockett, A. G. Oettinger, and A. J. Perlis. 1966. *Language and Machines: Computers in Translation and Linguistics*. ALPAC report. National Academy of Sciences, National Research Council, Washington, DC.

Pilehvar, M. T. and J. Camacho-Collados. 2019. WiC: the word-in-context dataset for evaluating context-sensitive meaning representations. *NAACL HLT*.

Pitler, E., A. Louis, and A. Nenkova. 2009. Automatic sense prediction for implicit discourse relations in text. *ACL IJCNLP*.

Pitler, E. and A. Nenkova. 2009. Using syntax to disambiguate explicit discourse connectives in text. *ACL IJCNLP*.

Pitt, M. A., L. Dilley, K. Johnson, S. Kiesling, W. D. Raymond, E. Hume, and E. Fosler-Lussier. 2007. Buckeye corpus of conversational speech (2nd release). Department of Psychology, Ohio State University (Distributor).

Pitt, M. A., K. Johnson, E. Hume, S. Kiesling, and W. D. Raymond. 2005. The buckeye corpus of conversational speech: Labeling conventions and a test of transcriber reliability. *Speech Communication*, 45:90–95.

Plutchik, R. 1962. *The emotions: Facts, theories, and a new model*. Random House.

Plutchik, R. 1980. A general psychoevolutionary theory of emotion. In R. Plutchik and H. Kellerman, eds, *Emotion: Theory, Research, and Experience, Volume 1*, 3–33. Academic Press.

Poesio, M., R. Stevenson, B. Di Eugenio, and J. Hitzeman. 2004. Centering: A parametric theory and its instantiations. *Computational Linguistics*, 30(3):309–363.

Poesio, M., R. Stuckardt, and Y. Versley. 2016. *Anaphora resolution: Algorithms, resources, and applications*. Springer.

Poesio, M., P. Sturt, R. Artstein, and R. Filik. 2006. Underspecification and anaphora: Theoretical issues and preliminary evidence. *Discourse processes*, 42(2):157–175.

Poesio, M. and R. Vieira. 1998. A corpus-based investigation of definite description use. *Computational Linguistics*, 24(2):183–216.

Polanyi, L. 1988. A formal model of the structure of discourse. *Journal of Pragmatics*, 12.

Polanyi, L., C. Culy, M. van den Berg, G. L. Thione, and D. Ahn. 2004. A rule based approach to discourse parsing. *Proceedings of SIGDIAL*.

Pollard, C. and I. A. Sag. 1994. *Head-Driven Phrase Structure Grammar*. University of Chicago Press.

Ponzetto, S. P. and M. Strube. 2006. Exploiting semantic role labeling, WordNet and Wikipedia for coreference resolution. *HLT-NAACL*.

Ponzetto, S. P. and M. Strube. 2007. Knowledge derived from Wikipedia for computing semantic relatedness. *JAIR*, 30:181–212.

Popović, M. 2015. chrF: character n-gram F-score for automatic MT evaluation. *Proceedings of the Tenth Workshop on Statistical Machine Translation*.

Popp, D., R. A. Donovan, M. Crawford, K. L. Marsh, and M. Peele. 2003. Gender, race, and speech style stereotypes. *Sex Roles*, 48(7-8):317–325.

Post, M. 2018. A call for clarity in reporting BLEU scores. *WMT 2018*.

Potts, C. 2011. On the negativity of negation. In N. Li and D. Lutz, eds, *Proceedings of Semantics and Linguistic Theory 20*, 636–659. CLC Publications, Ithaca, NY.

Povey, D., A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, J. Silovský, G. Stemmer, and K. Veselý. 2011. The Kaldi speech recognition toolkit. *ASRU*.

Pradhan, S., E. H. Hovy, M. P. Marcus, M. Palmer, L. Ramshaw, and R. Weischedel. 2007a. OntoNotes: A unified relational semantic representation. *Proceedings of ICSC*.

Pradhan, S., E. H. Hovy, M. P. Marcus, M. Palmer, L. A. Ramshaw, and R. M. Weischedel. 2007b. Ontonotes: a unified relational semantic representation. *Int. J. Semantic Computing*, 1(4):405–419.

Pradhan, S., X. Luo, M. Recasens, E. H. Hovy, V. Ng, and M. Strube. 2014. Scoring coreference partitions of predicted mentions: A reference implementation. *ACL*.

Pradhan, S., A. Moschitti, N. Xue, H. T. Ng, A. Björkelund, O. Uryupina, Y. Zhang, and Z. Zhong. 2013. Towards robust linguistic analysis using OntoNotes. *CoNLL*.

Pradhan, S., A. Moschitti, N. Xue, O. Uryupina, and Y. Zhang. 2012a. CoNLL-2012 shared task: Modeling multilingual unrestricted coreference in OntoNotes. *CoNLL*.

Pradhan, S., A. Moschitti, N. Xue, O. Uryupina, and Y. Zhang. 2012b. Conll-2012 shared task: Modeling multilingual unrestricted coreference in OntoNotes. *CoNLL*.

Pradhan, S., L. Ramshaw, M. P. Marcus, M. Palmer, R. Weischedel, and N. Xue. 2011. CoNLL-2011 shared task: Modeling unrestricted coreference in OntoNotes. *CoNLL*.

Pradhan, S., L. Ramshaw, R. Weischedel, J. MacBride, and L. Micciulla. 2007c. Unrestricted coreference: Identifying entities and events in OntoNotes. *Proceedings of ICSC 2007*.

Pradhan, S., W. Ward, K. Hacioglu, J. H. Martin, and D. Jurafsky. 2005. Semantic role labeling using different syntactic views. *ACL*.

Prasad, R., N. Dinesh, A. Lee, E. Miltsakaki, L. Robaldo, A. K. Joshi, and B. L. Webber. 2008. The Penn Discourse TreeBank 2.0. *LREC*.

Prasad, R., B. L. Webber, and A. Joshi. 2014. Reflections on the Penn Discourse Treebank, comparable corpora, and complementary annotation. *Computational Linguistics*, 40(4):921–950.

Prates, M. O. R., P. H. Avelar, and L. C. Lamb. 2019. Assessing gender bias in machine translation: a case study with Google Translate. *Neural Computing and Applications*, 32:6363–6381.

Price, P. J., W. Fisher, J. Bernstein, and D. Pallet. 1988. The DARPA 1000-word resource management database for continuous speech recognition. *ICASSP*.

Price, P. J., M. Ostendorf, S. Shattuck-Hufnagel, and C. Fong. 1991. The use of prosody in syntactic disambiguation. *JASA*, 90(6).

Prince, E. 1981. Toward a taxonomy of given-new information. In P. Cole, ed., *Radical Pragmatics*, 223–255. Academic Press.

Propp, V. 1968. *Morphology of the Folktale*, 2nd edition. University of Texas Press. Original Russian 1928. Translated by Laurence Scott.

Pundak, G. and T. N. Sainath. 2016. Lower frame rate neural network acoustic models. *INTERSPEECH*.

Pustejovsky, J. 1991. The generative lexicon. *Computational Linguistics*, 17(4).

Pustejovsky, J., P. Hanks, R. Saurí, A. See, R. Gaizauskas, A. Setzer, D. Radev, B. Sundheim, D. S. Day, L. Ferro, and M. Lazo. 2003. The TIMEBANK corpus. *Proceedings of Corpus Linguistics 2003 Conference*. UCREL Technical Paper number 16.

Pustejovsky, J., R. Ingria, R. Saurí, J. Castaño, J. Littman, R. Gaizauskas, A. Setzer, G. Katz, and I. Mani. 2005. *The Specification Language TimeML*, chapter 27. Oxford.

Qin, L., Z. Zhang, and H. Zhao. 2016. A stacking gated neural architecture for implicit discourse relation classification. *EMNLP*.

Qin, L., Z. Zhang, H. Zhao, Z. Hu, and E. Xing. 2017. Adversarial connective-exploiting networks for implicit discourse relation classification. *ACL*.

Radford, A., J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever. 2023. Robust speech recognition via large-scale weak supervision. *ICML*.

Radford, A., J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. 2019. Language models are unsupervised multitask learners. OpenAI tech report.

Rafailov, R., A. Sharma, E. Mitchell, S. Ermon, C. D. Manning, and C. Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *NeurIPS*.

Raffel, C., N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *JMLR*, 21(140):1–67.

Raghunathan, K., H. Lee, S. Rangarajan, N. Chambers, M. Surdeanu, D. Jurafsky, and C. D. Manning. 2010. A multi-pass sieve for coreference resolution. *EMNLP*.

Rahman, A. and V. Ng. 2009. Supervised models for coreference resolution. *EMNLP*.

Rahman, A. and V. Ng. 2012. Resolving complex cases of definite pronouns: the Winograd Schema challenge. *EMNLP*.

Rajpurkar, P., R. Jia, and P. Liang. 2018. Know what you don't know: Unanswerable questions for SQuAD. *ACL*.

Rajpurkar, P., J. Zhang, K. Lopyrev, and P. Liang. 2016. SQuAD: 100,000+ questions for machine comprehension of text. *EMNLP*.

Ram, O., Y. Levine, I. Dalmedigos, D. Muhlgay, A. Shashua, K. Leyton-Brown, and Y. Shoham. 2023. In-context retrieval-augmented language models. ArXiv preprint.

Ramshaw, L. A. and M. P. Marcus. 1995. Text chunking using transformation-based learning. *Proceedings of the 3rd Annual Workshop on Very Large Corpora*.

Rashkin, H., E. Bell, Y. Choi, and S. Volkova. 2017. Multilingual connotation frames: A case study on social media for targeted sentiment analysis and forecast. *ACL*.

Rashkin, H., S. Singh, and Y. Choi. 2016. Connotation frames: A data-driven investigation. *ACL*.

Ratinov, L. and D. Roth. 2012. Learning-based multi-sieve coreference resolution with knowledge. *EMNLP*.

Ratnaparkhi, A. 1996. A maximum entropy part-of-speech tagger. *EMNLP*.

Ratnaparkhi, A. 1997. A linear observed time statistical parser based on maximum entropy models. *EMNLP*.

Rawls, J. 2001. *Justice as fairness: A restatement*. Harvard University Press.

Recasens, M. and E. H. Hovy. 2011. BLANC: Implementing the Rand index for coreference evaluation. *Natural Language Engineering*, 17(4):485–510.

Recasens, M., E. H. Hovy, and M. A. Martí. 2011. Identity, non-identity, and near-identity: Addressing the complexity of coreference. *Lingua*, 121(6):1138–1152.

Recasens, M. and M. A. Martí. 2010. AnCora-CO: Coreferentially annotated corpora for Spanish and Catalan. *Language Resources and Evaluation*, 44(4):315–345.

Reed, C., R. Mochales Palau, G. Rowe, and M.-F. Moens. 2008. Language resources for studying argument. *LREC*.

Reeves, B. and C. Nass. 1996. *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*. Cambridge University Press.

Rehder, B., M. E. Schreiner, M. B. W. Wolfe, D. Laham, T. K. Landauer, and W. Kintsch. 1998. Using Latent Semantic Analysis to assess knowledge: Some technical considerations. *Discourse Processes*, 25(2-3):337–354.

Rei, R., C. Stewart, A. C. Farinha, and A. Lavie. 2020. COMET: A neural framework for MT evaluation. *EMNLP*.

Reichenbach, H. 1947. *Elements of Symbolic Logic*. Macmillan, New York.

Reichman, R. 1985. *Getting Computers to Talk Like You and Me*. MIT Press.

Renals, S., T. Hain, and H. Bourlard. 2007. Recognition and understanding of meetings: The AMI and AMIDA projects. *ASRU*.

Resnik, P. 1993. Semantic classes and syntactic ambiguity. *HLT*.

Resnik, P. 1996. Selectional constraints: An information-theoretic model and its computational realization. *Cognition*, 61:127–159.

Riedel, S., L. Yao, and A. McCallum. 2010. Modeling relations and their mentions without labeled text. In *Machine Learning and Knowledge Discovery in Databases*, 148–163. Springer.

Riedel, S., L. Yao, A. McCallum, and B. M. Marlin. 2013. Relation extraction with matrix factorization and universal schemas. *NAACL HLT*.

Riloff, E. 1993. Automatically constructing a dictionary for information extraction tasks. *AAAI*.

Riloff, E. 1996. Automatically generating extraction patterns from untagged text. *AAAI*.

Riloff, E. and R. Jones. 1999. Learning dictionaries for information extraction by multi-level bootstrapping. *AAAI*.

Riloff, E. and M. Schmelzenbach. 1998. An empirical approach to conceptual case frame acquisition. *Proceedings of the Sixth Workshop on Very Large Corpora*.

Riloff, E. and J. Shepherd. 1997. A corpus-based approach for building semantic lexicons. *EMNLP*.

Riloff, E. and M. Thelen. 2000. A rule-based question answering system for reading comprehension tests. *ANLP/NAACL workshop on reading comprehension tests*.

Riloff, E. and J. Wiebe. 2003. Learning extraction patterns for subjective expressions. *EMNLP*.

Ritter, A., O. Etzioni, and Mausam. 2010. A latent dirichlet allocation method for selectional preferences. *ACL*.

Ritter, A., L. Zettlemoyer, Mausam, and O. Etzioni. 2013. Modeling missing data in distant supervision for information extraction. *TACL*, 1:367–378.

Roberts, A., C. Raffel, and N. Shazeer. 2020. How much knowledge can you pack into the parameters of a language model? *EMNLP*.

Robertson, S., S. Walker, S. Jones, M. M. Hancock-Beaulieu, and M. Gatford. 1995. Okapi at TREC-3. *Overview of the Third Text REtrieval Conference (TREC-3)*.

Robins, R. H. 1967. *A Short History of Linguistics*. Indiana University Press, Bloomington.

Robinson, T. and F. Fallside. 1991. A recurrent error propagation network speech recognition system. *Computer Speech & Language*, 5(3):259–274.

Robinson, T., M. Hochberg, and S. Renals. 1996. The use of recurrent neural networks in continuous speech recognition. In C.-H. Lee, F. K. Soong, and K. K. Paliwal, eds, *Automatic speech and speaker recognition*, 233–258. Springer.

Rogers, A., M. Gardner, and I. Augenstein. 2023. QA dataset explosion: A taxonomy of NLP resources for question answering and reading comprehension. *ACM Computing Surveys*, 55(10):1–45.

Rohde, D. L. T., L. M. Gonnerman, and D. C. Plaut. 2006. An improved model of semantic similarity based on lexical co-occurrence. *CACM*, 8:627–633.

Rooth, M., S. Riezler, D. Prescher, G. Carroll, and F. Beil. 1999. Inducing a semantically annotated lexicon via EM-based clustering. *ACL*.

Rosenblatt, F. 1958. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386–408.

Rosenfeld, R. 1992. *Adaptive Statistical Language Modeling: A Maximum Entropy Approach*. Ph.D. thesis, Carnegie Mellon University.

Rosenfeld, R. 1996. A maximum entropy approach to adaptive statistical language modeling. *Computer Speech and Language*, 10:187–228.

Rosenthal, S. and K. McKeown. 2017. Detecting influencers in multiple online genres. *ACM Transactions on Internet Technology (TOIT)*, 17(2).

Rothe, S., S. Ebert, and H. Schütze. 2016. Ultradense Word Embeddings by Orthogonal Transformation. *NAACL HLT*.

Rudinger, R., J. Naradowsky, B. Leonard, and B. Van Durme. 2018. Gender bias in coreference resolution. *NAACL HLT*.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams. 1986. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, eds, *Parallel Distributed Processing*, volume 2, 318–362. MIT Press.

Rumelhart, D. E. and J. L. McClelland. 1986a. On learning the past tense of English verbs. In D. E. Rumelhart and J. L. McClelland, eds, *Parallel Distributed Processing*, volume 2, 216–271. MIT Press.

Rumelhart, D. E. and J. L. McClelland, eds. 1986b. *Parallel Distributed Processing*. MIT Press.

Rumelhart, D. E. and A. A. Abrahamson. 1973. A model for analogical reasoning. *Cognitive Psychology*, 5(1):1–28.

Rumelhart, D. E. and J. L. McClelland, eds. 1986c. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1: *Foundations*. MIT Press.

Ruppenhofer, J., M. Ellsworth, M. R. L. Petruck, C. R. Johnson, C. F. Baker, and J. Scheffczyk. 2016. FrameNet II: Extended theory and practice.

Ruppenhofer, J., C. Sporleder, R. Morante, C. F. Baker, and M. Palmer. 2010. Semeval-2010 task 10: Linking events and their participants in discourse. *5th International Workshop on Semantic Evaluation*.

Russell, J. A. 1980. A circumplex model of affect. *Journal of personality and social psychology*, 39(6):1161–1178.

Russell, S. and P. Norvig. 2002. *Artificial Intelligence: A Modern Approach*, 2nd edition. Prentice Hall.

Rust, P., J. Pfeiffer, I. Vulić, S. Ruder, and I. Gurevych. 2021. How good is your tokenizer? on the monolingual performance of multilingual language models. *ACL*.

Rutherford, A. and N. Xue. 2015. Improving the inference of implicit discourse relations via classifying explicit discourse connectives. *NAACL HLT*.

Sachan, D. S., M. Lewis, D. Yogatama, L. Zettlemoyer, J. Pineau, and M. Zaheer. 2023. Questions are all you need to train a dense passage retriever. *TACL*, 11:600–616.

Sacks, H., E. A. Schegloff, and G. Jefferson. 1974. A simplest systematics for the organization of turn-taking for conversation. *Language*, 50(4):696–735.

Sagae, K. 2009. Analysis of discourse structure with syntactic dependencies and data-driven shift-reduce parsing. *IWPT-09*.

Sagawa, S., P. W. Koh, T. B. Hashimoto, and P. Liang. 2020. Distributionally robust neural networks for group shifts: On the importance of regularization for worst-case generalization. *ICLR*.

Sagisaka, Y. 1988. Speech synthesis by rule using an optimal selection of non-uniform synthesis units. *ICASSP*.

Sagisaka, Y., N. Kaiki, N. Iwahashi, and K. Mimura. 1992. Atr – ν-talk speech synthesis system. *ICSLP*.

Sakoe, H. and S. Chiba. 1971. A dynamic programming approach to continuous speech recognition. *Proceedings of the Seventh International Congress on Acoustics*, volume 3. Akadémiai Kiadó.

Sakoe, H. and S. Chiba. 1984. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on ASSP*, ASSP-26(1):43–49.

Salomaa, A. 1969. Probabilistic and weighted grammars. *Information and Control*, 15:529–544.

Salton, G. 1971. *The SMART Retrieval System: Experiments in Automatic Document Processing*. Prentice Hall.

Sampson, G. 1987. Alternative grammatical coding systems. In R. Garside, G. Leech, and G. Sampson, eds, *The Computational Analysis of English*, 165–183. Longman.

Sankoff, D. and W. Labov. 1979. On the uses of variable rules. *Language in society*, 8(2-3):189–222.

Sap, M., D. Card, S. Gabriel, Y. Choi, and N. A. Smith. 2019. The risk of racial bias in hate speech detection. *ACL*.

Sap, M., M. C. Prasettio, A. Holtzman, H. Rashkin, and Y. Choi. 2017. Connotation frames of power and agency in modern films. *EMNLP*.

Saurí, R., J. Littman, B. Knippen, R. Gaizauskas, A. Setzer, and J. Pustejovsky. 2006. TimeML annotation guidelines version 1.2.1. Manuscript.

Scha, R. and L. Polanyi. 1988. An augmented context free grammar for discourse. *COLING*.

Schank, R. C. and R. P. Abelson. 1975. Scripts, plans, and knowledge. *Proceedings of IJCAI-75*.

Schank, R. C. and R. P. Abelson. 1977. *Scripts, Plans, Goals and Understanding*. Lawrence Erlbaum.

Schegloff, E. A. 1968. Sequencing in conversational openings. *American Anthropologist*, 70:1075–1095.

Schegloff, E. A. 1982. Discourse as an interactional achievement: Some uses of 'uh huh' and other things that come between sentences. In D. Tannen, ed., *Analyzing Discourse: Text and Talk*, 71–93. Georgetown University Press, Washington, D.C.

Scherer, K. R. 2000. Psychological models of emotion. In J. C. Borod, ed., *The neuropsychology of emotion*, 137–162. Oxford.

Schiebinger, L. 2013. Machine translation: Analyzing gender. `http://genderedinnovations.stanford.edu/case-studies/nlp.html#tabs-2`.

Schiebinger, L. 2014. Scientific research must take gender into account. *Nature*, 507(7490):9.

Schluter, N. 2018. The word analogy testing caveat. *NAACL HLT*.

Schmidt, C. W., V. Reddy, C. Tanner, and Y. Pinter. 2025. Boundless byte pair encoding: Breaking the pre-tokenization barrier. *COLM*.

Schone, P. and D. Jurafsky. 2000. Knowlege-free induction of morphology using latent semantic analysis. *CoNLL*.

Schone, P. and D. Jurafsky. 2001a. Is knowledge-free induction of multiword unit dictionary headwords a solved problem? *EMNLP*.

Schone, P. and D. Jurafsky. 2001b. Knowledge-free induction of inflectional morphologies. *NAACL*.

Schuster, M. and K. Nakajima. 2012. Japanese and Korean voice search. *ICASSP*.

Schuster, M. and K. K. Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45:2673–2681.

Schütze, H. 1992a. Context space. *AAAI Fall Symposium on Probabilistic Approaches to Natural Language*.

Schütze, H. 1992b. Dimensions of meaning. *Proceedings of Supercomputing '92*. IEEE Press.

Schütze, H. 1997. *Ambiguity Resolution in Language Learning – Computational and Cognitive Models*. CSLI, Stanford, CA.

Schütze, H., D. A. Hull, and J. Pedersen. 1995. A comparison of classifiers and document representations for the routing problem. *SIGIR-95*.

Schütze, H. and J. Pedersen. 1993. A vector model for syntagmatic and paradigmatic relatedness. *9th Annual Conference of the UW Centre for the New OED and Text Research*.

Schütze, H. and Y. Singer. 1994. Part-of-speech tagging using a variable memory Markov model. *ACL*.

Schwartz, H. A., J. C. Eichstaedt, M. L. Kern, L. Dziurzynski, S. M. Ramones, M. Agrawal, A. Shah, M. Kosinski, D. Stillwell, M. E. P. Seligman, and L. H. Ungar. 2013. Personality, gender, and age in the language of social media: The open-vocabulary approach. *PloS one*, 8(9):e73791.

Schwenk, H. 2007. Continuous space language models. *Computer Speech & Language*, 21(3):492–518.

Schwenk, H. 2018. Filtering and mining parallel data in a joint multilingual space. *ACL*.

Schwenk, H., D. Dechelotte, and J.-L. Gauvain. 2006. Continuous space language models for statistical machine translation. *COLING/ACL*.

Schwenk, H., G. Wenzek, S. Edunov, E. Grave, A. Joulin, and A. Fan. 2021. CCMatrix: Mining billions of high-quality parallel sentences on the web. *ACL*.

Séaghdha, D. O. 2010. Latent variable models of selectional preference. *ACL*.

Seddah, D., R. Tsarfaty, S. Kübler, M. Candito, J. D. Choi, R. Farkas, J. Foster, I. Goenaga, K. Gojenola, Y. Goldberg, S. Green, N. Habash, M. Kuhlmann, W. Maier, J. Nivre, A. Przepiórkowski, R. Roth, W. Seeker, Y. Versley, V. Vincze, M. Woliński, A. Wróblewska, and E. Villemonte de la Clérgerie. 2013. Overview of the SPMRL 2013 shared task: cross-framework evaluation of parsing morphologically rich languages. *4th Workshop on Statistical Parsing of Morphologically-Rich Languages*.

Sekine, S. and M. Collins. 1997. The evalb software. `http://cs.nyu.edu/cs/projects/proteus/evalb`.

Sellam, T., D. Das, and A. Parikh. 2020. BLEURT: Learning robust metrics for text generation. *ACL*.

Seneff, S. and V. W. Zue. 1988. Transcription and alignment of the TIMIT database. *Proceedings of the Second Symposium on Advanced Man-Machine Interface through Spoken Language*.

Sennrich, R., B. Haddow, and A. Birch. 2016. Neural machine translation of rare words with subword units. *ACL*.

Seo, M., A. Kembhavi, A. Farhadi, and H. Hajishirzi. 2017. Bidirectional attention flow for machine comprehension. *ICLR*.

Shannon, C. E. 1948. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423. Continued in the following volume.

Shannon, C. E. 1951. Prediction and entropy of printed English. *Bell System Technical Journal*, 30:50–64.

Sheil, B. A. 1976. Observations on context free parsing. *SMIL: Statistical Methods in Linguistics*, 1:71–109.

Sheng, E., K.-W. Chang, P. Natarajan, and N. Peng. 2019. The woman worked as a babysitter: On biases in language generation. *EMNLP*.

Shi, P. and J. Lin. 2019. Simple BERT models for relation extraction and semantic role labeling. ArXiv.

Shi, W., S. Min, M. Yasunaga, M. Seo, R. James, M. Lewis, L. Zettlemoyer, and W.-t. Yih. 2023. REPLUG: Retrieval-augmented black-box language models. ArXiv preprint.

Shoup, J. E. 1980. Phonological aspects of speech recognition. In W. A. Lea, ed., *Trends in Speech Recognition*, 125–138. Prentice Hall.

Sidner, C. L. 1979. Towards a computational theory of definite anaphora comprehension in English discourse. Technical Report 537, MIT Artificial Intelligence Laboratory, Cambridge, MA.

Sidner, C. L. 1983. Focusing in the comprehension of definite anaphora. In M. Brady and R. C. Berwick, eds, *Computational Models of Discourse*, 267–330. MIT Press.

Silverman, K., M. E. Beckman, J. F. Pitrelli, M. Ostendorf, C. W. Wightman, P. J. Price, J. B. Pierrehumbert, and J. Hirschberg. 1992. ToBI: A standard for labelling English prosody. *ICSLP*.

Simmons, R. F. 1965. Answering English questions by computer: A survey. *CACM*, 8(1):53–70.

Simmons, R. F. 1973. Semantic networks: Their computation and use for understanding English sentences. In R. C. Schank and K. M. Colby, eds, *Computer Models of Thought and Language*, 61–113. W.H. Freeman & Co.

Simmons, R. F., S. Klein, and K. McConlogue. 1964. Indexing and dependency logic for answering English questions. *American Documentation*, 15(3):196–204.

Simons, G. F. and C. D. Fennig. 2018. Ethnologue: Languages of the world, 21st edition. SIL International.

Singh, S., F. Vargus, D. D'souza, B. F. Karlsson, A. Mahendiran, W.-Y. Ko, H. Shandilya, J. Patel, D. Mataciunas, L. O'Mahony, M. Zhang, R. Hettiarachchi, J. Wilson, M. Machado, L. S. Moura, D. Krzemiński, H. Fadaei, I. Ergün, I. Okoh, A. Alaagib, O. Mudannayake, Z. Alyafeai, V. M. Chien, S. Ruder, S. Guthikonda, E. A. Alghamdi, S. Gehrmann, N. Muennighoff, M. Bartolo, J. Kreutzer, A. ÜÜstün, M. Fadaee, and S. Hooker. 2024. Aya dataset: An open-access collection for multilingual instruction tuning. ArXiv preprint.

Sleator, D. and D. Temperley. 1993. Parsing English with a link grammar. *IWPT-93*.

Sloan, M. C. 2010. Aristotle's Nicomachean Ethics as the original locus for the Septem Circumstantiae. *Classical Philology*, 105(3):236–251.

Slobin, D. I. 1996. Two ways to travel. In M. Shibatani and S. A. Thompson, eds, *Grammatical Constructions: Their Form and Meaning*, 195–220. Clarendon Press.

Smolensky, P. 1988. On the proper treatment of connectionism. *Behavioral and brain sciences*, 11(1):1–23.

Smolensky, P. 1990. Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial intelligence*, 46(1-2):159–216.

Snover, M., B. Dorr, R. Schwartz, L. Micciulla, and J. Makhoul. 2006. A study of translation edit rate with targeted human annotation. *AMTA-2006*.

Snow, R., D. Jurafsky, and A. Y. Ng. 2005. Learning syntactic patterns for automatic hypernym discovery. *NeurIPS*.

Socher, R., J. Bauer, C. D. Manning, and A. Y. Ng. 2013. Parsing with compositional vector grammars. *ACL*.

Socher, R., C. C.-Y. Lin, A. Y. Ng, and C. D. Manning. 2011. Parsing natural scenes and natural language with recursive neural networks. *ICML*.

Soderland, S., D. Fisher, J. Aseltine, and W. G. Lehnert. 1995. CRYSTAL: Inducing a conceptual dictionary. *IJCAI-95*.

Søgaard, A. 2010. Simple semi-supervised training of part-of-speech taggers. *ACL*.

Søgaard, A. and Y. Goldberg. 2016. Deep multi-task learning with low level tasks supervised at lower layers. *ACL*.

Søgaard, A., A. Johannsen, B. Plank, D. Hovy, and H. M. Alonso. 2014. What's in a p-value in NLP? *CoNLL*.

Soldaini, L., R. Kinney, A. Bhagia, D. Schwenk, D. Atkinson, R. Authur, B. Bogin, K. Chandu, J. Dumas, Y. Elazar, V. Hofmann, A. H. Jha, S. Kumar, L. Lucy, X. Lyu, N. Lambert, I. Magnusson, J. Morrison, N. Muennighoff, A. Naik, C. Nam, M. E. Peters, A. Ravichander, K. Richardson, Z. Shen, E. Strubell, N. Subramani, O. Tafjord, P. Walsh, L. Zettlemoyer, N. A. Smith, H. Hajishirzi, I. Beltagy, D. Groeneveld, J. Dodge, and K. Lo. 2024. Dolma: An open corpus of three trillion tokens for language model pretraining research. ArXiv preprint.

Solorio, T., E. Blair, S. Maharjan, S. Bethard, M. Diab, M. Ghoneim, A. Hawwari, F. AlGhamdi, J. Hirschberg, A. Chang, and P. Fung. 2014. Overview for the first shared task on language identification in code-switched data. *Workshop on Computational Approaches to Code Switching*.

Somasundaran, S., J. Burstein, and M. Chodorow. 2014. Lexical chaining for measuring discourse coherence quality in test-taker essays. *COLING*.

Soon, W. M., H. T. Ng, and D. C. Y. Lim. 2001. A machine learning approach to coreference resolution of noun phrases. *Computational Linguistics*, 27(4):521–544.

Soricut, R. and D. Marcu. 2003. Sentence level discourse parsing using syntactic and lexical information. *HLT-NAACL*.

Soricut, R. and D. Marcu. 2006. Discourse generation using utility-trained coherence models. *COLING/ACL*.

Sorokin, D. and I. Gurevych. 2018. Mixing context granularities for improved entity linking on question answering data across entity categories. **SEM*.

Sparck Jones, K. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21.

Sparck Jones, K. 1986. *Synonymy and Semantic Classification*. Edinburgh University Press, Edinburgh. Republication of 1964 PhD Thesis.

Sporleder, C. and A. Lascarides. 2005. Exploiting linguistic cues to classify rhetorical relations. *RANLP-05*.

Sporleder, C. and M. Lapata. 2005. Discourse chunking and its application to sentence compression. *EMNLP*.

Srivastava, N., G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *JMLR*, 15(1):1929–1958.

Stab, C. and I. Gurevych. 2014a. Annotating argument components and relations in persuasive essays. *COLING*.

Stab, C. and I. Gurevych. 2014b. Identifying argumentative discourse structures in persuasive essays. *EMNLP*.

Stab, C. and I. Gurevych. 2017. Parsing argumentation structures in persuasive essays. *Computational Linguistics*, 43(3):619–659.

Stalnaker, R. C. 1978. Assertion. In P. Cole, ed., *Pragmatics: Syntax and Semantics Volume 9*, 315–332. Academic Press.

Stamatatos, E. 2009. A survey of modern authorship attribution methods. *JASIST*, 60(3):538–556.

Stanovsky, G., N. A. Smith, and L. Zettlemoyer. 2019. Evaluating gender bias in machine translation. *ACL*.

Stede, M. 2011. *Discourse processing*. Morgan & Claypool.

Stede, M. and J. Schneider. 2018. *Argumentation Mining*. Morgan & Claypool.

Stern, M., J. Andreas, and D. Klein. 2017. A minimal span-based neural constituency parser. *ACL*.

Stevens, K. N. 1998. *Acoustic Phonetics*. MIT Press.

Stevens, K. N. and A. S. House. 1955. Development of a quantitative description of vowel articulation. *JASA*, 27:484–493.

Stevens, K. N. and A. S. House. 1961. An acoustical theory of vowel production and some of its implications. *Journal of Speech and Hearing Research*, 4:303–320.

Stevens, K. N., S. Kasowski, and G. M. Fant. 1953. An electrical analog of the vocal tract. *JASA*, 25(4):734–742.

Stevens, S. S. and J. Volkmann. 1940. The relation of pitch to frequency: A revised scale. *The American Journal of Psychology*, 53(3):329–353.

Stevens, S. S., J. Volkmann, and E. B. Newman. 1937. A scale for the measurement of the psychological magnitude pitch. *JASA*, 8:185–190.

Stiennon, N., L. Ouyang, J. Wu, D. M. Ziegler, R. Lowe, C. Voss, A. Radford, D. Amodei, and P. Christiano. 2020. Learning to summarize from human feedback. *Proceedings of the 34th International Conference on Neural Information Processing Systems*.

Stolcke, A. 1998. Entropy-based pruning of backoff language models. *Proc. DARPA Broadcast News Transcription and Understanding Workshop*.

Stolcke, A. 2002. SRILM – an extensible language modeling toolkit. *ICSLP*.

Stolcke, A., Y. Konig, and M. Weintraub. 1997. Explicit word error minimization in N-best list rescoring. *EUROSPEECH*, volume 1.

Stolz, W. S., P. H. Tannenbaum, and F. V. Carstensen. 1965. A stochastic approach to the grammatical coding of English. *CACM*, 8(6):399–405.

Stone, P., D. Dunphry, M. Smith, and D. Ogilvie. 1966. *The General Inquirer: A Computer Approach to Content Analysis*. MIT Press.

Strötgen, J. and M. Gertz. 2013. Multilingual and cross-domain temporal tagging. *Language Resources and Evaluation*, 47(2):269–298.

Strube, M. and U. Hahn. 1996. Functional centering. *ACL*.

Strubell, E., A. Ganesh, and A. McCallum. 2019. Energy and policy considerations for deep learning in NLP. *ACL*.

Su, Y., H. Sun, B. Sadler, M. Srivatsa, I. Gür, Z. Yan, and X. Yan. 2016. On generating characteristic-rich question sets for QA evaluation. *EMNLP*.

Subba, R. and B. Di Eugenio. 2009. An effective discourse parser that uses rich linguistic information. *NAACL HLT*.

Sukhbaatar, S., A. Szlam, J. Weston, and R. Fergus. 2015. End-to-end memory networks. *NeurIPS*.

Sundheim, B., ed. 1991. *Proceedings of MUC-3*.

Sundheim, B., ed. 1992. *Proceedings of MUC-4*.

Sundheim, B., ed. 1993. *Proceedings of MUC-5*. Baltimore, MD.

Sundheim, B., ed. 1995. *Proceedings of MUC-6*.

Surdeanu, M. 2013. Overview of the TAC2013 Knowledge Base Population evaluation: English slot filling and temporal slot filling. *TAC-13*.

Surdeanu, M., S. Harabagiu, J. Williams, and P. Aarseth. 2003. Using predicate-argument structures for information extraction. *ACL*.

Surdeanu, M., T. Hicks, and M. A. Valenzuela-Escarcega. 2015. Two practical rhetorical structure theory parsers. *NAACL HLT*.

Surdeanu, M., R. Johansson, A. Meyers, L. Màrquez, and J. Nivre. 2008. The CoNLL 2008 shared task on joint parsing of syntactic and semantic dependencies. *CoNLL*.

Sutskever, I., O. Vinyals, and Q. V. Le. 2014. Sequence to sequence learning with neural networks. *NeurIPS*.

Sutton, R. S. and A. G. Barto. 1998. *Reinforcement Learning: An Introduction*. MIT Press.

Suzgun, M., L. Melas-Kyriazi, and D. Jurafsky. 2023a. Follow the wisdom of the crowd: Effective text generation via minimum Bayes risk decoding. *Findings of ACL 2023*.

Suzgun, M., N. Scales, N. Schärli, S. Gehrmann, Y. Tay, H. W. Chung, A. Chowdhery, Q. Le, E. Chi, D. Zhou, and J. Wei. 2023b. Challenging BIG-bench tasks and whether chain-of-thought can solve them. *ACL Findings*.

Sweet, H. 1877. *A Handbook of Phonetics*. Clarendon Press.

Swier, R. and S. Stevenson. 2004. Unsupervised semantic role labelling. *EMNLP*.

Switzer, P. 1965. Vector images in document retrieval. *Statistical Association Methods For Mechanized Documentation. Symposium Proceedings. Washington, D.C., USA, March 17, 1964*. https://nvlpubs.nist.gov/nistpubs/Legacy/MP/nbsmiscellaneouspub269.pdf.

Syrdal, A. K., C. W. Wightman, A. Conkie, Y. Stylianou, M. Beutnagel, J. Schroeter, V. Strom, and K.-S. Lee. 2000. Corpus-based techniques in the AT&T NEXTGEN synthesis system. *ICSLP*.

Talmy, L. 1985. Lexicalization patterns: Semantic structure in lexical forms. In T. Shopen, ed., *Language Typology and Syntactic Description, Volume 3*. Cambridge University Press. Originally appeared as UC Berkeley Cognitive Science Program Report No. 30, 1980.

Talmy, L. 1991. Path to realization: A typology of event conflation. *BLS-91*.

Tan, C., V. Niculae, C. Danescu-Niculescu-Mizil, and L. Lee. 2016. Winning arguments: Interaction dynamics and persuasion strategies in good-faith online discussions. *WWW-16*.

Tannen, D. 1979. What's in a frame? Surface evidence for underlying expectations. In R. Freedle, ed., *New Directions in Discourse Processing*, 137–181. Ablex.

Taylor, P. 2009. *Text-to-Speech Synthesis*. Cambridge University Press.

Taylor, W. L. 1953. Cloze procedure: A new tool for measuring readability. *Journalism Quarterly*, 30:415–433.

Teranishi, R. and N. Umeda. 1968. Use of pronouncing dictionary in speech synthesis experiments. *6th International Congress on Acoustics*.

Tesnière, L. 1959. *Éléments de Syntaxe Structurale*. Librairie C. Klincksieck, Paris.

Tetreault, J. R. 2001. A corpus-based evaluation of centering and pronoun resolution. *Computational Linguistics*, 27(4):507–520.

Teufel, S., J. Carletta, and M. Moens. 1999. An annotation scheme for discourse-level argumentation in research articles. *EACL*.

Teufel, S., A. Siddharthan, and C. Batchelor. 2009. Towards domain-independent argumentative zoning: Evidence from chemistry and computational linguistics. *EMNLP*.

Thede, S. M. and M. P. Harper. 1999. A second-order hidden Markov model for part-of-speech tagging. *ACL*.

Thompson, B. and P. Koehn. 2019. Vecalign: Improved sentence alignment in linear time and space. *EMNLP*.

Thompson, K. 1968. Regular expression search algorithm. *CACM*, 11(6):419–422.

Tian, Y., V. Kulkarni, B. Perozzi, and S. Skiena. 2016. On the convergent properties of word embedding methods. ArXiv preprint arXiv:1605.03956.

Tibshirani, R. J. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288.

Timkey, W. and M. van Schijndel. 2021. All bark and no bite: Rogue dimensions in transformer language models obscure representational quality. *EMNLP*.

Titov, I. and E. Khoddam. 2014. Unsupervised induction of semantic roles within a reconstruction-error minimization framework. *NAACL HLT*.

Titov, I. and A. Klementiev. 2012. A Bayesian approach to unsupervised semantic role induction. *EACL*.

Tomkins, S. S. 1962. *Affect, imagery, consciousness: Vol. I. The positive affects*. Springer.

Toutanova, K., D. Klein, C. D. Manning, and Y. Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. *HLT-NAACL*.

Tria, F., V. Loreto, and V. D. Servedio. 2018. Zipf's, heaps' and taylor's laws are determined by the expansion into the adjacent possible. *Entropy*, 20(10):752.

Trichelair, P., A. Emami, J. C. K. Cheung, A. Trischler, K. Suleman, and F. Diaz. 2018. On the evaluation of common-sense reasoning in natural language understanding. *NeurIPS 2018 Workshop on Critiquing and Correcting Trends in Machine Learning*.

Trnka, K., D. Yarrington, J. McCaw, K. F. McCoy, and C. Pennington. 2007. The effects of word prediction on communication rate for AAC. *NAACL-HLT*.

Turian, J. P., L. Shen, and I. D. Melamed. 2003. Evaluation of machine translation and its evaluation. *Proceedings of MT Summit IX*.

Turian, J., L. Ratinov, and Y. Bengio. 2010. Word representations: a simple and general method for semi-supervised learning. *ACL*.

Turney, P. D. 2002. Thumbs up or thumbs down? Semantic orientation applied to unsupervised classification of reviews. *ACL*.

Turney, P. D. and M. Littman. 2003. Measuring praise and criticism: Inference of semantic orientation from association. *ACM Transactions on Information Systems (TOIS)*, 21:315–346.

Turney, P. D. and M. L. Littman. 2005. Corpus-based learning of analogies and semantic relations. *Machine Learning*, 60(1-3):251–278.

Umeda, N. 1976. Linguistic rules for text-to-speech synthesis. *Proceedings of the IEEE*, 64(4):443–451.

Umeda, N., E. Matui, T. Suzuki, and H. Omura. 1968. Synthesis of fairy tale using an analog vocal tract. *6th International Congress on Acoustics*.

Uryupina, O., R. Artstein, A. Bristot, F. Cavicchio, F. Delogu, K. J. Rodriguez, and M. Poesio. 2020. Annotating a broad range of anaphoric phenomena, in a variety of genres: The ARRAU corpus. *Natural Language Engineering*, 26(1):1–34.

Uszkoreit, J. 2017. Transformer: A novel neural network architecture for language understanding. Google Research blog post, Thursday August 31, 2017.

van Deemter, K. and R. Kibble. 2000. On coreferring: coreference in MUC and related annotation schemes. *Computational Linguistics*, 26(4):629–637.

Van Den Oord, A., O. Vinyals, and K. Kavukcuoglu. 2017. Neural discrete representation learning. *NeurIPS*.

van der Maaten, L. and G. E. Hinton. 2008. Visualizing high-dimensional data using t-SNE. *JMLR*, 9:2579–2605.

van Rijsbergen, C. J. 1975. *Information Retrieval*. Butterworths.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. 2017. Attention is all you need. *NeurIPS*.

Vauquois, B. 1968. A survey of formal grammars and algorithms for recognition and transformation in machine translation. *IFIP Congress 1968*.

Velichko, V. M. and N. G. Zagoruyko. 1970. Automatic recognition of 200 words. *International Journal of Man-Machine Studies*, 2:223–234.

Velikovich, L., S. Blair-Goldensohn, K. Hannan, and R. McDonald. 2010. The viability of web-derived polarity lexicons. *NAACL HLT*.

Vendler, Z. 1967. *Linguistics in Philosophy*. Cornell University Press.

Verhagen, M., R. Gaizauskas, F. Schilder, M. Hepple, J. Moszkowicz, and J. Pustejovsky. 2009. The TempEval challenge: Identifying temporal relations in text. *Language Resources and Evaluation*, 43(2):161–179.

Verhagen, M., I. Mani, R. Sauri, R. Knippen, S. B. Jang, J. Littman, A. Rumshisky, J. Phillips, and J. Pustejovsky. 2005. Automating temporal annotation with TARSQI. *ACL*.

Versley, Y. 2008. Vagueness and referential ambiguity in a large-scale annotated corpus. *Research on Language and Computation*, 6(3-4):333–353.

Vieira, R. and M. Poesio. 2000. An empirically based system for processing definite descriptions. *Computational Linguistics*, 26(4):539–593.

Vilain, M., J. D. Burger, J. Aberdeen, D. Connolly, and L. Hirschman. 1995. A model-theoretic coreference scoring scheme. *MUC-6*.

Vintsyuk, T. K. 1968. Speech discrimination by dynamic programming. *Cybernetics*, 4(1):52–57. Original Russian: Kibernetika 4(1):81-88. 1968.

Vinyals, O., Ł. Kaiser, T. Koo, S. Petrov, I. Sutskever, and G. Hinton. 2015. Grammar as a foreign language. *NeurIPS*.

Voorhees, E. M. and D. K. Harman. 2005. *TREC: Experiment and Evaluation in Information Retrieval*. MIT Press.

Voutilainen, A. 1999. Handcrafted rules. In H. van Halteren, ed., *Syntactic Wordclass Tagging*, 217–246. Kluwer.

Vrandečić, D. and M. Krötzsch. 2014. Wikidata: a free collaborative knowledge base. *CACM*, 57(10):78–85.

Wagner, R. A. and M. J. Fischer. 1974. The string-to-string correction problem. *Journal of the ACM*, 21:168–173.

Waibel, A., T. Hanazawa, G. Hinton, K. Shikano, and K. J. Lang. 1989. Phoneme recognition using time-delay neural networks. *IEEE Transactions on ASSP*, 37(3):328–339.

Walker, M. A., M. Iida, and S. Cote. 1994. Japanese discourse and the process of centering. *Computational Linguistics*, 20(2):193–232.

Walker, M. A., A. K. Joshi, and E. Prince, eds. 1998. *Centering in Discourse*. Oxford University Press.

Walker, M. A., E. Maier, J. Allen, J. Carletta, S. Condon, G. Flammia, J. Hirschberg, S. Isard, M. Ishizaki, L. Levin, S. Luperfoy, D. R. Traum, and S. Whittaker. 1996. Penn multiparty standard coding scheme: Draft annotation manual. `www.cis.upenn.edu/~ircs/discourse-tagging/newcoding.html`.

Wang, A., A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman. 2018a. Glue: A multi-task benchmark and analysis platform for natural language understanding. *ICLR*.

Wang, S. and C. D. Manning. 2012. Baselines and bigrams: Simple, good sentiment and topic classification. *ACL*.

Wang, W. and B. Chang. 2016. Graph-based dependency parsing with bidirectional LSTM. *ACL*.

Wang, Y., S. Li, and J. Yang. 2018b. Toward fast and accurate neural discourse segmentation. *EMNLP*.

Wang, Y., S. Mishra, P. Alipoormolabashi, Y. Kordi, A. Mirzaei, A. Naik, A. Ashok, A. S. Dhanasekaran, A. Arunkumar, D. Stap, E. Pathak, G. Karamanolakis, H. Lai, I. Purohit, I. Mondal, J. Anderson, K. Kuznia, K. Doshi, K. K. Pal, M. Patel, M. Moradshahi, M. Parmar, M. Purohit, N. Varshney, P. R. Kaza, P. Verma, R. S. Puri, R. Karia, S. Doshi, S. K. Sampat, S. Mishra, S. Reddy A, S. Patro, T. Dixit, and X. Shen. 2022. Super-NaturalInstructions: Generalization via declarative instructions on 1600+ NLP tasks. *EMNLP*.

Wang, Z., Y. Dong, J. Zeng, V. Adams, M. N. Sreedhar, D. Egert, O. Delalleau, J. Scowcroft, N. Kant, A. Swope, and O. Kuchaiev. 2024. HelpSteer: Multi-attribute helpfulness dataset for SteerLM. *NAACL HLT*.

Ward, N. and W. Tsukahara. 2000. Prosodic features which cue back-channel feedback in English and Japanese. *Journal of Pragmatics*, 32:1177–1207.

Watanabe, S., T. Hori, S. Karita, T. Hayashi, J. Nishitoba, Y. Unno, N. E. Y. Soplin, J. Heymann, M. Wiesner, N. Chen, A. Renduchintala, and T. Ochiai. 2018. ESPnet: End-to-end speech processing toolkit. *INTERSPEECH*.

Weaver, W. 1949/1955. Translation. In W. N. Locke and A. D. Boothe, eds, *Machine Translation of Languages*, 15–23. MIT Press. Reprinted from a memorandum written by Weaver in 1949.

Webber, B. L. 1978. *A Formal Approach to Discourse Anaphora*. Ph.D. thesis, Harvard University.

Webber, B. L. 1983. So what can we talk about now? In M. Brady and R. C. Berwick, eds, *Computational Models of Discourse*, 331–371. The MIT Press.

Webber, B. L. 1991. Structure and ostension in the interpretation of discourse deixis. *Language and Cognitive Processes*, 6(2):107–135.

Webber, B. L. and B. Baldwin. 1992. Accommodating context change. *ACL*.

Webber, B. L., M. Egg, and V. Kordoni. 2012. Discourse structure and language technology. *Natural Language Engineering*, 18(4):437–490.

Webber, B. L. 1988. Discourse deixis: Reference to discourse segments. *ACL*.

Webson, A. and E. Pavlick. 2022. Do prompt-based models really understand the meaning of their prompts? *NAACL HLT*.

Webster, K., M. Recasens, V. Axelrod, and J. Baldridge. 2018. Mind the GAP: A balanced corpus of gendered ambiguous pronouns. *TACL*, 6:605–617.

Wei, J., X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *NeurIPS*, volume 35.

Weischedel, R., M. Meteer, R. Schwartz, L. A. Ramshaw, and J. Palmucci. 1993. Coping with ambiguity and unknown words through probabilistic models. *Computational Linguistics*, 19(2):359–382.

Weizenbaum, J. 1966. ELIZA – A computer program for the study of natural language communication between man and machine. *CACM*, 9(1):36–45.

Weizenbaum, J. 1976. *Computer Power and Human Reason: From Judgement to Calculation*. W.H. Freeman & Co.

Wells, J. C. 1982. *Accents of English*. Cambridge University Press.

Werbos, P. 1974. *Beyond regression: new tools for prediction and analysis in the behavioral sciences*. Ph.D. thesis, Harvard University.

Werbos, P. J. 1990. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.

Weston, J., S. Chopra, and A. Bordes. 2015. Memory networks. *ICLR 2015*.

Widrow, B. and M. E. Hoff. 1960. Adaptive switching circuits. *IRE WESCON Convention Record*, volume 4.

Wiebe, J. 1994. Tracking point of view in narrative. *Computational Linguistics*, 20(2):233–287.

Wiebe, J. 2000. Learning subjective adjectives from corpora. *AAAI*.

Wiebe, J., R. F. Bruce, and T. P. O'Hara. 1999. Development and use of a gold-standard data set for subjectivity classifications. *ACL*.

Wierzbicka, A. 1992. *Semantics, Culture, and Cognition: University Human Concepts in Culture-Specific Configurations*. Oxford University Press.

Wierzbicka, A. 1996. *Semantics: Primes and Universals*. Oxford University Press.

Wilks, Y. 1973. An artificial intelligence approach to machine translation. In R. C. Schank and K. M. Colby, eds, *Computer Models of Thought and Language*, 114–151. W.H. Freeman.

Wilks, Y. 1975a. Preference semantics. In E. L. Keenan, ed., *The Formal Semantics of Natural Language*, 329–350. Cambridge Univ. Press.

Wilks, Y. 1975b. A preferential, pattern-seeking, semantics for natural language inference. *Artificial Intelligence*, 6(1):53–74.

Williams, A., N. Nangia, and S. Bowman. 2018. A broad-coverage challenge corpus for sentence understanding through inference. *NAACL HLT*.

Wilson, T., J. Wiebe, and P. Hoffmann. 2005. Recognizing contextual polarity in phrase-level sentiment analysis. *EMNLP*.

Winograd, T. 1972. *Understanding Natural Language*. Academic Press.

Winston, P. H. 1977. *Artificial Intelligence*. Addison Wesley.

Wiseman, S., A. M. Rush, and S. M. Shieber. 2016. Learning global features for coreference resolution. *NAACL HLT*.

Wiseman, S., A. M. Rush, S. M. Shieber, and J. Weston. 2015. Learning anaphoricity and antecedent ranking features for coreference resolution. *ACL*.

Witten, I. H. and T. C. Bell. 1991. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 37(4):1085–1094.

Witten, I. H. and E. Frank. 2005. *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd edition. Morgan Kaufmann.

Wittgenstein, L. 1953. *Philosophical Investigations. (Translated by Anscombe, G.E.M.)*. Blackwell.

Wolf, F. and E. Gibson. 2005. Representing discourse coherence: A corpus-based analysis. *Computational Linguistics*, 31(2):249–287.

Wolf, M. J., K. W. Miller, and F. S. Grodzinsky. 2017. Why we should have seen that coming: Comments on Microsoft's Tay "experiment," and wider implications. *The ORBIT Journal*, 1(2):1–12.

Woods, W. A. 1978. Semantics and quantification in natural language question answering. In M. Yovits, ed., *Advances in Computers*, 2–64. Academic.

Woods, W. A., R. M. Kaplan, and B. L. Nash-Webber. 1972. The lunar sciences natural language information system: Final report. Technical Report 2378, BBN.

Woodsend, K. and M. Lapata. 2015. Distributed representations for unsupervised semantic role labeling. *EMNLP*.

Wu, D. 1996. A polynomial-time algorithm for statistical machine translation. *ACL*.

Wu, F. and D. S. Weld. 2007. Autonomously semantifying Wikipedia. *CIKM-07*.

Wu, F. and D. S. Weld. 2010. Open information extraction using Wikipedia. *ACL*.

Wu, L., F. Petroni, M. Josifoski, S. Riedel, and L. Zettlemoyer. 2020. Scalable zero-shot entity linking with dense entity retrieval. *EMNLP*.

Wu, S. and M. Dredze. 2019. Beto, Bentz, Becas: The surprising cross-lingual effectiveness of BERT. *EMNLP*.

Wu, Y., M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Ł. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. S. Corrado, M. Hughes, and J. Dean. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. ArXiv preprint arXiv:1609.08144.

Wundt, W. 1900. *Völkerpsychologie: eine Untersuchung der Entwicklungsgesetze von Sprache, Mythus, und Sitte*. W. Engelmann, Leipzig. Band II: Die Sprache, Zweiter Teil.

Xu, A., E. Pathak, E. Wallace, S. Gururangan, M. Sap, and D. Klein. 2021. Detoxifying language models risks marginalizing minority voices. *NAACL HLT*.

Xu, J., D. Ju, M. Li, Y.-L. Boureau, J. Weston, and E. Dinan. 2020. Recipes for safety in open-domain chatbots. ArXiv preprint arXiv:2010.07079.

Xu, P., H. Saghir, J. S. Kang, T. Long, A. J. Bose, Y. Cao, and J. C. K. Cheung. 2019. A cross-domain transferable neural coherence model. *ACL*.

Xu, Y. 2005. Speech melody as articulatorily implemented communicative functions. *Speech communication*, 46(3-4):220–251.

Xue, N., H. T. Ng, S. Pradhan, A. Rutherford, B. L. Webber, C. Wang, and H. Wang. 2016. CoNLL 2016 shared task on multilingual shallow discourse parsing. *CoNLL-16 shared task*.

Xue, N. and M. Palmer. 2004. Calibrating features for semantic role labeling. *EMNLP*.

Yamada, H. and Y. Matsumoto. 2003. Statistical dependency analysis with support vector machines. *IWPT-03*.

Yang, D., J. Chen, Z. Yang, D. Jurafsky, and E. H. Hovy. 2019. Let's make your request more persuasive: Modeling persuasive strategies via semi-supervised neural nets on crowdfunding platforms. *NAACL HLT*.

Yang, X., G. Zhou, J. Su, and C. L. Tan. 2003. Coreference resolution using competition learning approach. *ACL*.

Yang, Y. and J. Pedersen. 1997. A comparative study on feature selection in text categorization. *ICML*.

Yih, W.-t., M. Richardson, C. Meek, M.-W. Chang, and J. Suh. 2016. The value of semantic parse labeling for knowledge base question answering. *ACL*.

Yngve, V. H. 1970. On getting a word in edgewise. *CLS-70*. University of Chicago.

Young, S. J., M. Gašić, S. Keizer, F. Mairesse, J. Schatzmann, B. Thomson, and K. Yu. 2010. The Hidden Information State model: A practical framework for POMDP-based spoken dialogue management. *Computer Speech & Language*, 24(2):150–174.

Younger, D. H. 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10:189–208.

Yu, N., M. Zhang, and G. Fu. 2018. Transition-based neural RST parsing with implicit syntax features. *COLING*.

Yu, Y., Y. Zhu, Y. Liu, Y. Liu, S. Peng, M. Gong, and A. Zeldes. 2019. GumDrop at the DISRPT2019 shared task: A model stacking approach to discourse unit segmentation and connective detection. *Workshop on Discourse Relation Parsing and Treebanking 2019*.

Yuan, J., M. Liberman, and C. Cieri. 2006. Towards an integrated understanding of speaking rate in conversation. *Interspeech*.

Zao-Sanders, M. 2025. How People Are Really Using Gen AI in 2025 — hbr.org. https://hbr.org/2025/04/how-people-are-really-using-gen-ai-in-2025. [Accessed 02-05-2025].

Zapirain, B., E. Agirre, L. Màrquez, and M. Surdeanu. 2013. Selectional preferences for semantic role classification. *Computational Linguistics*, 39(3):631–663.

Zelle, J. M. and R. J. Mooney. 1996. Learning to parse database queries using inductive logic programming. *AAAI*.

Zeman, D. 2008. Reusable tagset conversion using tagset drivers. *LREC*.

Zens, R. and H. Ney. 2007. Efficient phrase-table representation for machine translation with applications to online MT and speech translation. *NAACL-HLT*.

Zettlemoyer, L. and M. Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. *Uncertainty in Artificial Intelligence, UAI'05*.

Zettlemoyer, L. and M. Collins. 2007. Online learning of relaxed CCG grammars for parsing to logical form. *EMNLP/CoNLL*.

Zhang, A. K., K. Klyman, Y. Mai, Y. Levine, Y. Zhang, R. Bommasani, and P. Liang. 2025. Language model developers should report train-test overlap. *ICML*.

Zhang, R., C. N. dos Santos, M. Yasunaga, B. Xiang, and D. Radev. 2018. Neural coreference resolution

with deep biaffine attention by joint mention detection and mention clustering. *ACL*.

Zhang, T., V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi. 2020. BERTscore: Evaluating text generation with BERT. *ICLR 2020*.

Zhang, Y., V. Zhong, D. Chen, G. Angeli, and C. D. Manning. 2017. Position-aware attention and supervised data improve slot filling. *EMNLP*.

Zhao, H., W. Chen, C. Kit, and G. Zhou. 2009. Multilingual dependency learning: A huge feature engineering method to semantic dependency parsing. *CoNLL*.

Zhao, J., T. Wang, M. Yatskar, R. Cotterell, V. Ordonez, and K.-W. Chang. 2019. Gender bias in contextualized word embeddings. *NAACL HLT*.

Zhao, J., T. Wang, M. Yatskar, V. Ordonez, and K.-W. Chang. 2017. Men also like shopping: Reducing gender bias amplification using corpus-level constraints. *EMNLP*.

Zhao, J., T. Wang, M. Yatskar, V. Ordonez, and K.-W. Chang. 2018a. Gender bias in coreference resolution: Evaluation and debiasing methods. *NAACL HLT*.

Zhao, J., Y. Zhou, Z. Li, W. Wang, and K.-W. Chang. 2018b. Learning gender-neutral word embeddings. *EMNLP*.

Zheng, J., L. Vilnis, S. Singh, J. D. Choi, and A. McCallum. 2013. Dynamic knowledge-base alignment for coreference resolution. *CoNLL*.

Zhou, D., O. Bousquet, T. N. Lal, J. Weston, and B. Schölkopf. 2004a. Learning with local and global consistency. *NeurIPS*.

Zhou, G., J. Su, J. Zhang, and M. Zhang. 2005. Exploring various knowledge in relation extraction. *ACL*.

Zhou, J. and W. Xu. 2015a. End-to-end learning of semantic role labeling using recurrent neural networks. *ACL*.

Zhou, J. and W. Xu. 2015b. End-to-end learning of semantic role labeling using recurrent neural networks. *ACL*.

Zhou, K., K. Ethayarajh, D. Card, and D. Jurafsky. 2022. Problems with cosine as a measure of embedding similarity for high frequency words. *ACL*.

Zhou, K., J. Hwang, X. Ren, and M. Sap. 2024. Relying on the unreliable: The impact of language models' reluctance to express uncertainty. *ACL*.

Zhou, L., M. Ticrea, and E. H. Hovy. 2004b. Multi-document biography summarization. *EMNLP*.

Zhou, Y. and N. Xue. 2015. The Chinese Discourse TreeBank: a Chinese corpus annotated with discourse relations. *Language Resources and Evaluation*, 49(2):397–431.

Zhu, X. and Z. Ghahramani. 2002. Learning from labeled and unlabeled data with label propagation. Technical Report CMU-CALD-02, CMU.

Zhu, X., Z. Ghahramani, and J. Lafferty. 2003. Semi-supervised learning using gaussian fields and harmonic functions. *ICML*.

Zhu, Y., R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler. 2015. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. *IEEE International Conference on Computer Vision*.

Ziegler, D. M., N. Stiennon, J. Wu, T. B. Brown, A. Radford, D. Amodei, P. Christiano, and G. Irving. 2019. Fine-tuning language models from human preferences. *ArXiv*, abs/1909.08593.

Ziemski, M., M. Junczys-Dowmunt, and B. Pouliquen. 2016. The United Nations parallel corpus v1.0. *LREC*.

Subject Index