CHAPTER

6 Vector Semantics and 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, 157).

distributional hypothesis

In this chapter we introduce **vector semantics**, which instantiates this linguistic hypothesis by learning representations of the meaning of words, called **embeddings**, directly from their distributions in texts. These representations are used in every natural language processing application that makes use of meaning, and the **static embeddings** we introduce here underlie the more powerful dynamic or **contextualized embeddings** like **BERT** that we will see in Chapter 11.

vector semantics
embeddings

These word representations 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 the input, instead of creating representations by hand via **feature engineering**, is an important focus of NLP research (Bengio et al., 2013).

representation learning

6.1 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 Chapter 18 and Chapter 10.

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

wordform

sung or *carpets* or *sing* or *duermes* are called **wordforms**.

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 types "mouse info" into a search engine looking for a pet or a tool?). Chapter 18 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

synonym

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

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. We often say in this case that the two words have the same

propositional meaning

propositional meaning.

principle of contrast

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.

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.

similarity

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*):

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

relatedness association

similarity. One such class of connections is called word **relatedness** (Budanitsky and Hirst, 2006), also traditionally called word **association** in psychology.

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

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 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 Chapter 18 we'll introduce more relations between senses like **hypernymy** or **IS-A**, **antonymy** (opposites) and **meronymy** (part-whole relations).

Semantic Frames and Roles Closely related to semantic fields is the idea of a **semantic frame**. A semantic frame is a set of words that denote perspectives or participants in a particular type of event. A commercial transaction, for example, is a kind of event in which one entity trades money to another entity in return for some good or service, after which the good changes hands or perhaps the service is performed. This event can be encoded lexically by using verbs like *buy* (the event from the perspective of the buyer), *sell* (from the perspective of the seller), *pay* (focusing on the monetary aspect), or nouns like *buyer*. Frames have semantic roles (like *buyer, seller, goods, money*), and words in a sentence can take on these roles.

semantic frame

Knowing that *buy* and *sell* have this relation makes it possible for a system to know that a sentence like *Sam bought the book from Ling* could be paraphrased as *Ling sold the book to Sam*, and that Sam has the role of the *buyer* in the frame and Ling the *seller*. Being able to recognize such paraphrases is important for question answering, and can help in shifting perspective for machine translation.

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 (*happy*) while others have negative connotations (*sad*). 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 Chapter 4, 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.

connotations

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	8.05	5.5	7.38
music	7.67	5.57	6.5
heartbreak	2.45	5.65	3.58
cub	6.71	3.95	4.24

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 [2.45, 5.65, 3.58]) was the first expression of the vector semantics models that we introduce next.

6.2 Vector Semantics

vector semantics

Vectors 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.

using context to derive meaning

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:

(6.1) Ongchoi is delicious sauteed with garlic.
(6.2) Ongchoi is superb over rice
(6.3) ...ongchoi leaves with salty sauces...

all of this is stored in a vector

And suppose that you had seen many of these context words in other contexts:

(6.4) ...spinach sauteed with garlic over rice...
(6.5) ...chard stems and leaves are delicious...
(6.6) ...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 do the same thing computationally by just counting words in the context of *ongchoi*.

The idea of vector semantics is to represent a word as a point in a multidimensional semantic space that is derived (in ways we'll see) from the distributions of word neighbors. Vectors for representing words are called **embeddings** (although the term is sometimes more strictly applied only to dense vectors like word2vec (Section 6.8), rather than sparse tf-idf or PPMI vectors (Section 6.3-Section 6.6)). The word "embedding" derives 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.

embeddings

[1] It's in fact *Ipomoea aquatica*, a relative of morning glory sometimes called *water spinach* in English.

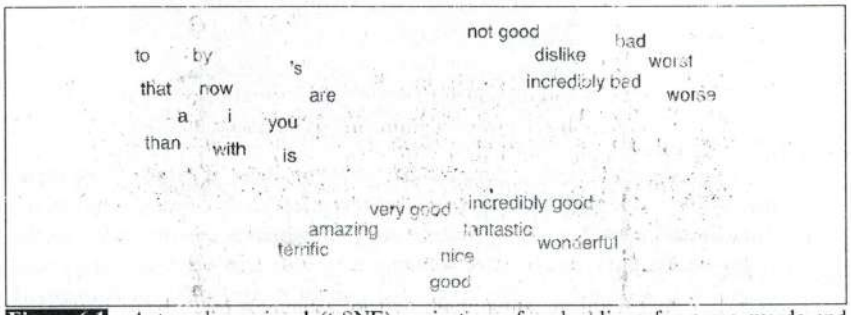


Figure 6.1 A two-dimensional (t-SNE) projection of embeddings for some words and phrases, showing that words with similar meanings are nearby in space. The original 60-dimensional embeddings were trained for sentiment analysis. Simplified from [illegible] et al. (2015) with colors added for explanation.

Fig. 6.1 shows a visualization of embeddings learned for sentiment analysis, showing the location of selected words projected down from 60-dimensional space into a two dimensional space. Notice the distinct regions containing positive words, negative words, and neutral function words.

The fine-grained model of word similarity of vector semantics offers enormous power to NLP applications. NLP applications like the sentiment classifiers of Chapter 4 or Chapter 5 depend on the same words appearing in the training and test sets. But by representing words as embeddings, classifiers can assign sentiment as long as it sees some words with *similar meanings*. And as we'll see, vector semantic models can be learned automatically from text without supervision.

In this chapter we'll introduce the two most commonly used models. In the **tf-idf** model, an important baseline, the meaning of a word is defined by a simple function of the counts of nearby words. 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 introduce the **word2vec** model family for constructing short, **dense** vectors that have 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 like question answering, summarization, or automatic essay grading.

6.3 Words and Vectors

"The most important attributes of a vector in 3-space are {Location, Location, Location}"
Randall Munroe, `https://xkcd.com/2358/`

Vector or distributional models of meaning are generally based on a **co-occurrence matrix**, a way of representing how often words co-occur. We'll look at two popular matrices: the term-document matrix and the term-term matrix.

6.3.1 Vectors and documents

term-document matrix

In a **term-document matrix**, each row represents a word in the vocabulary and each column represents a document from some collection of documents. Fig. 6.2 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*.

	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 6.2 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.

vector space model

The term-document matrix of Fig. 6.2 was first defined as part of the **vector space model** of information retrieval (Salton, 1971). In this model, a document is represented as a count vector, a column in Fig. 6.3.

vector

To review some basic linear algebra, a **vector** is, at heart, just a list or array of numbers. So *As You Like It* is represented as the list [1,114,36,20] (the first **column vector** in Fig. 6.3) and *Julius Caesar* is represented as the list [7,62,1,2] (the third column vector). A **vector space** is a collection of vectors, characterized by their **dimension**. In the example in Fig. 6.3, the document vectors are of dimension 4, just so they fit on the page; in real term-document matrices, the vectors representing each document would have dimensionality $|V|$, the vocabulary size.

vector space

dimension

The ordering of the numbers in a vector space indicates different meaningful dimensions on which documents vary. Thus the first dimension for both these vectors corresponds to the number of times the word *battle* occurs, and we can 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 6.3 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.

We can think of the vector for a document as a point in $|V|$-dimensional space; thus the documents in Fig. 6.3 are points in 4-dimensional space. Since 4-dimensional spaces are hard to visualize, Fig. 6.4 shows a visualization in two dimensions; we've arbitrarily chosen the dimensions corresponding to the words *battle* and *fool*.

Term-document matrices were originally defined as a means of finding similar documents for the task of document **information retrieval**. 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]. This is clear with the raw numbers; in the first dimension (battle) the comedies have low numbers and the others have high numbers, and we can see it visually in Fig. 6.4; we'll see very shortly how to quantify this intuition more formally.

A real term-document matrix, of course, wouldn't just have 4 rows and columns, let alone 2. More generally, the term-document matrix has $|V|$ rows (one for each word type in the vocabulary) and D columns (one for each document in the collec-

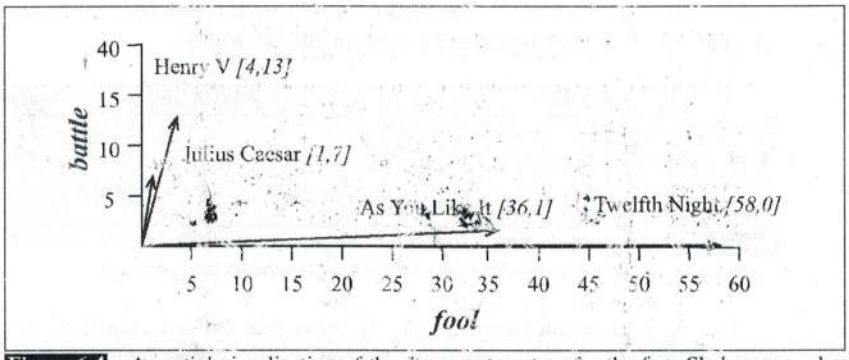


Figure 6.4 A spatial visualization of the document vectors for the four Shakespeare play documents, showing just two of the dimensions, corresponding to the words *battle* and *fool*. The comedies have high values for the *fool* dimension and low values for the *battle* dimension.

tion); as we'll see, vocabulary sizes are generally in the tens of thousands, and the number of documents can be enormous (think about all the pages on the web).

information retrieval

Information retrieval (IR) is the task of finding the document d from the D documents in some collection that best matches a query q. For IR we'll therefore also represent a query by a vector, also of length $|V|$, and we'll need a way to compare two vectors to find how similar they are. (Doing IR will also require efficient ways to store and manipulate these vectors by making use of the convenient fact that these vectors are sparse, i.e., mostly zeros).

Later in the chapter we'll introduce some of the components of this vector comparison process: the tf-idf term weighting, and the cosine similarity metric.

6.3.2 Words as vectors: document dimensions

We've seen that documents can be represented as vectors in a vector space. But vector semantics can also be used to represent the meaning of *words*. We do this by associating each word with a word vector— a **row vector** rather than a column vector, hence with different dimensions, as shown in Fig. 6.5. The four dimensions of the vector for *fool*, [36,58,1,4], correspond to the four Shakespeare plays. Word counts in the same four dimensions are used to form the vectors for the other 3 words: *wit*, [20,15,2,3]; *battle*, [1,0,7,13]; and *good* [114,80,62,89].

row vector

	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 6.5 The term-document matrix for four words in four Shakespeare plays. The red boxes show that each word is represented as a row vector of length four.

For documents, we saw that similar documents had similar vectors, because similar documents tend to have similar words. This same principle applies to words: similar words have similar vectors because they tend to occur in similar documents. The term-document matrix thus lets us represent the meaning of a word by the documents it tends to occur in.

6.3.3 Words as vectors: word dimensions

word-word matrix

An alternative to using the term-document matrix to represent words as vectors of document counts, is to use the **term-term matrix**, also called the **word-word matrix** or the **term-context matrix**, in which the columns are labeled by words rather than documents. 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 in some context in some training corpus. The context could be the document, in which case the cell represents the number of times the two words appear in the same document. It is most common, however, to use smaller contexts, generally a window around the word, for example of 4 words to the left and 4 words to the right, in which case the cell represents the number of times (in some training corpus) the column word occurs in such a ± 4 word window around the row word. For example here is one example each of some words in their windows:

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 (say **strawberry**) and count the context words around it, we get a word-word co-occurrence matrix. Fig. 6.6 shows a simplified subset of the word-word co-occurrence matrix for these four words computed from the Wikipedia corpus (Davies, 2015).

	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 6.6 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.

Note in Fig. 6.6 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*. Fig. 6.7 shows a spatial visualization.

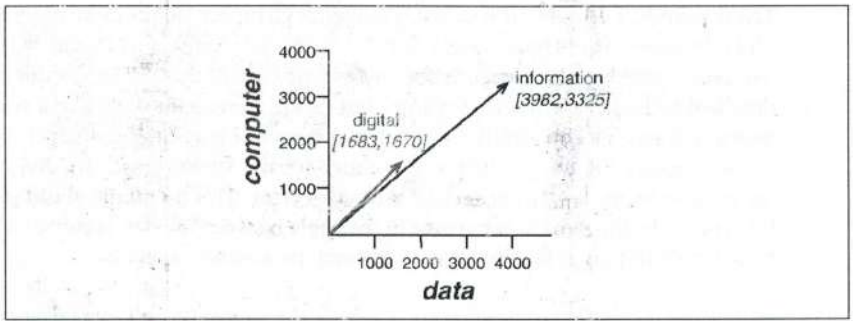


Figure 6.7 A spatial visualization of word vectors for *digital* and *information*, showing just two of the dimensions, corresponding to the words *data* and *computer*.

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.

Now that we have some intuitions, let's move on to examine the details of computing word similarity. Afterwards we'll discuss methods for weighting cells.

6.4 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 as documents, 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**:

$$\text{dot product}(\mathbf{v},\mathbf{w}) = \mathbf{v}\cdot\mathbf{w} = \sum_{i=1}^{N} v_i w_i = v_1 w_1 + v_2 w_2 + \ldots + v_N w_N \tag{6.7}$$

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.

vector length

This raw dot product, however, has a problem as a similarity metric: it favors **long** vectors. The **vector length** is defined as

$$|\mathbf{v}| = \sqrt{\sum_{i=1}^{N} v_i^2} \tag{6.8}$$

vector length

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 $\mathbf{b}$:

$$\mathbf{a}\cdot\mathbf{b} = |\mathbf{a}||\mathbf{b}|\cos\theta$$
$$\frac{\mathbf{a}\cdot\mathbf{b}}{|\mathbf{a}||\mathbf{b}|} = \cos\theta \tag{6.9}$$

normalized dot product = cos(θ between 2 vectors)

cosine

The **cosine** similarity metric between two vectors $\mathbf{v}$ and $\mathbf{w}$ thus can be computed as:

$a\cdot b = |a||b|\cos\theta$

$$\text{cosine}(\mathbf{v},\mathbf{w}) = \frac{\mathbf{v}\cdot\mathbf{w}}{|\mathbf{v}||\mathbf{w}|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2}\sqrt{\sum_{i=1}^{N} w_i^2}} \tag{6.10}$$

ok just subbed Formula 6.8 into 6.9

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 **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

$$\cos(\text{cherry},\text{information}) = \frac{442*5+8*3982+2*3325}{\sqrt{442^2+8^2+2^2}\sqrt{5^2+3982^2+3325^2}} = .018$$

$$\cos(\text{digital},\text{information}) = \frac{5*5+1683*3982+1670*3325}{\sqrt{5^2+1683^2+1670^2}\sqrt{5^2+3982^2+3325^2}} = .996$$

The model decides that *information* is way closer to *digital* than it is to *cherry*, a result that seems sensible. Fig. 6.8 shows a visualization.

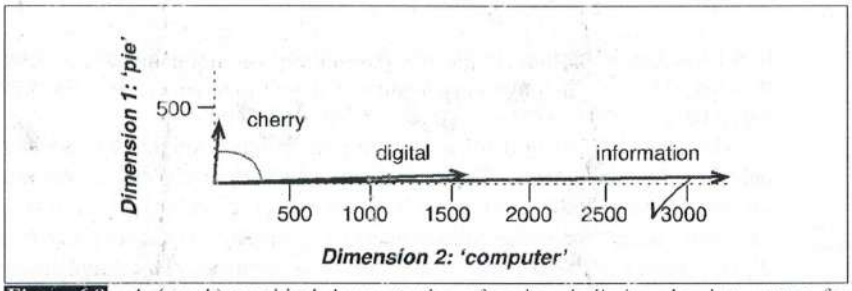


Figure 6.8 A (rough) graphical demonstration of cosine similarity, showing vectors for three words (*cherry*, *digital*, and *information*) in the two dimensional space defined by counts of the words *computer* and *pie* nearby. The figure doesn't show the cosine, but it highlights the angles; note that 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.

6.5 TF-IDF: Weighing terms in the vector

The co-occurrence matrices above represent each cell by frequencies, either of words with documents (Fig. 6.5), or words with other words (Fig. 6.6). But raw frequency

is not the best measure of association between words. Raw frequency is very skewed and not very discriminative. If we want to know what kinds of contexts are shared by *cherry* and *strawberry* but not by *digital* and *information*, we're not going to get good discrimination from words like *the*, *it*, or *they*, which occur frequently with all sorts of words and aren't informative about any particular word. We saw this also in Fig. 6.3 for the Shakespeare corpus; the dimension for the word *good* is not very discriminative between plays; *good* is simply a frequent word and has roughly equivalent high frequencies in each of the plays.

It's a bit of a paradox. Words that occur nearby frequently (maybe *pie* nearby *cherry*) are more important than words that only appear once or twice. Yet words that are too frequent—ubiquitous, like *the* or *good*— are unimportant. How can we balance these two conflicting constraints?

There are two common solutions to this problem: in this section we'll describe the **tf-idf** weighting, usually used when the dimensions are documents. In the next we introduce the **PPMI** algorithm (usually used when the dimensions are words).

The **tf-idf weighting** (the '-' here is a hyphen, not a minus sign) is the product of two terms, each term capturing one of these two intuitions:

term frequency

The first is the **term frequency** (Luhn, 1957): the frequency of the word t in the document d. We can just use the raw count as the term frequency:

$$\text{tf}_{t,d} = \text{count}(t,d) \tag{6.11}$$

More commonly we squash the raw frequency a bit, by using the $\log_{10}$ of the frequency instead. 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. Because we can't take the log of 0, we normally add 1 to the count:[2]

$$\text{tf}_{t,d} = \log_{10}(\text{count}(t,d)+1) \tag{6.12}$$

If we use log weighting, terms which occur 0 times in a document would have $\text{tf} = \log_{10}(1) = 0$, 10 times in a document $\text{tf} = \log_{10}(11) = 1.04$, 100 times $\text{tf} = \log_{10}(101) = 2.004$, 1000 times $\text{tf} = 3.00044$, and so on.

The second factor in tf-idf is used to give a higher weight to words that occur only in a few documents. Terms that are limited to a few documents are useful for discriminating those documents from the rest of the collection; terms that occur frequently across the entire collection aren't as helpful. The **document frequency** df_t of a term t is the number of documents it occurs in. Document frequency is not the same as the **collection frequency** of a term, which is the total number of times the word appears in the whole collection in any document. Consider in the collection of Shakespeare's 37 plays the two words *Romeo* and *action*. The words have identical collection frequencies (they both occur 113 times in all the plays) but very different document frequencies, since Romeo only occurs in a single play. If our goal is to find documents about the romantic tribulations of Romeo, the word *Romeo* should be highly weighted, but not *action*:

document frequency

	Collection Frequency	Document Frequency
Romeo	113	1
action	113	31

We emphasize discriminative words like *Romeo* via the **inverse document frequency** or **idf** term weight (Sparck Jones, 1972). The idf is defined using the frac-

idf

[2] Or we can use this alternative: $\text{tf}_{t,d} = \begin{cases} 1+\log_{10}\text{count}(t,d) & \text{if count}(t,d)>0 \\ 0 & \text{otherwise} \end{cases}$

tion N/df_t, 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 1 is assigned to terms that occur in all the documents. It's usually clear what counts as a document: in Shakespeare we would use a play; when processing a collection of encyclopedia articles like Wikipedia, the document is a Wikipedia page; in processing newspaper articles, the document is a single article. Occasionally your corpus might not have appropriate document divisions and you might need to break up the corpus into documents yourself for the purposes of computing idf.

Because of the large number of documents in many collections, this measure too is usually squashed with a log function. The resulting definition for inverse document frequency (idf) is thus

$$\mathrm{idf}_t = \log_{10}\left(\frac{N}{\mathrm{df}_t}\right) \qquad idf_t = \log_{10}\left(\frac{N}{df_t}\right)$$

Here are some idf values for some words in the Shakespeare corpus, ranging from extremely informative words which occur in only one play like *Romeo*, to those that occur in a few like *salad* or *Falstaff*, to those which 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*.[3]

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

tf-idf

The **tf-idf** weighted value $w_{t,d}$ for word t in document d thus combines term frequency $\mathrm{tf}_{t,d}$ (defined either by Eq. 6.11 or by Eq. 6.12) with idf from Eq. 6.13:

$$w_{t,d} = \mathrm{tf}_{t,d} \times \mathrm{idf}_t \tag{6.14}$$

Fig. 6.9 applies tf-idf weighting to the Shakespeare term-document matrix in Fig. 6.2, using the tf equation Eq. 6.12. Note that the tf-idf values for the dimension corresponding to the word *good* have now all become 0; since this word appears in every document, the tf-idf weighting leads it to be ignored. Similarly, the word *fool*, which appears in 36 out of the 37 plays, has a much lower weight.

The tf-idf weighting is the way for weighting co-occurrence matrices in information retrieval, but also plays a role in many other aspects of natural language processing. It's also a great baseline, the simple thing to try first. We'll look at other weightings like PPMI (Positive Pointwise Mutual Information) in Section 6.6.

[3] *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).

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	0.074	0	0.22	0.28
good	0	0	0	0
fool	0.019	0.021	0.0036	0.0083
wit	0.049	0.044	0.018	0.022

Figure 6.9 A tf-idf weighted term-document matrix for four words in four Shakespeare plays, using the counts in Fig. 6.2. For example the 0.049 value for *wit* in *As You Like It* is the product of $\text{tf} = \log_{10}(20+1) = 1.322$ and $\text{idf} \approx .037$. Note that the idf weighting has eliminated the importance of the ubiquitous word *good* and vastly reduced the impact of the almost-ubiquitous word *fool*.

6.6 Pointwise Mutual Information (PMI)

pointwise mutual information

An alternative weighting function to tf-idf, PPMI (positive pointwise mutual information), is used for term-term-matrices, when the vector dimensions correspond to words rather than documents. PPMI draws on the intuition that the best way to weigh the association between two words is to ask how much **more** the two words co-occur in our corpus than we would have a priori expected them to appear by chance.

Pointwise mutual information (Fano, 1961)[4] is one of the most important concepts in NLP. It is a measure of how often two events x and y occur, compared with what we would expect if they were independent:

$$I(x,y) = \log_2 \frac{P(x,y)}{P(x)P(y)} \tag{6.16}$$

The pointwise mutual information between a target word w and a context word c (Church and Hanks 1989, Church and Hanks 1990) is then defined as:

$$\text{PMI}(w,c) = \log_2 \frac{P(w,c)}{P(w)P(c)} \tag{6.17}$$

The numerator tells us how often we observed the two words together (assuming we compute probability by using the MLE). The denominator tells us how often we would **expect** the two words to co-occur assuming they each occurred independently; recall that the probability of two independent events both occurring is just the product of the probabilities of the two events. Thus, the ratio gives us an estimate of how much more the two words co-occur than we expect by chance. PMI is a useful tool whenever we need to find words that are strongly associated.

PMI values range from negative to positive infinity. But negative PMI values (which imply things are co-occurring *less often* than we would expect by chance) tend to be unreliable unless our corpora are enormous. To distinguish whether two words whose individual probability is each 10^{-6} occur together less often than chance, we would need to be certain that the probability of the two occurring together is significantly different than 10^{-12}, and this kind of granularity would require an enormous corpus. Furthermore it's not clear whether it's even possible to evaluate such scores of 'unrelatedness' with human judgments. For this reason it

[4] PMI is based on the **mutual information** between two random variables X and Y, defined as:

$$I(X,Y) = \sum_x \sum_y P(x,y) \log_2 \frac{P(x,y)}{P(x)P(y)} \tag{6.15}$$

In a confusion of terminology, Fano used the phrase *mutual information* to refer to what we now call *pointwise mutual information* and the phrase *expectation of the mutual information* for what we now call *mutual information*

PPMI

is more common to use Positive PMI (called **PPMI**) which replaces all negative PMI values with zero (Church and Hanks 1989, Dagan et al. 1993, Niwa and Nitta 1994)[5]:

$$\text{PPMI}(w,c) = \max(\log_2 \frac{P(w,c)}{P(w)P(c)}, 0) \tag{6.18}$$

More formally, let's assume we have a co-occurrence matrix F with W rows (words) and C columns (contexts), where f_{ij} gives the number of times word w_i occurs in context c_j. This can be turned into a PPMI matrix where PPMI_{ij} gives the PPMI value of word w_i with context c_j (which we can also express as $\text{PPMI}(w_i, c_j)$ or $\text{PPMI}(w = i, c = j)$) as follows:

$$p_{ij} = \frac{f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}}, \quad p_{i*} = \frac{\sum_{j=1}^{C} f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}}, \quad p_{*j} = \frac{\sum_{i=1}^{W} f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}} \tag{6.19}$$

$$\text{PPMI}_{ij} = \max(\log_2 \frac{p_{ij}}{p_{i*}p_{*j}}, 0) \tag{6.20}$$

Let's see some **PPMI** calculations. We'll use Fig. 6.10, which repeats Fig. 6.6 plus all the count marginals, and let's pretend for ease of calculation that these are the only words/contexts that matter.

	computer	data	result	pie	sugar	count(w)
cherry	2	8	9	442	25	486
strawberry	0	0	1	60	19	80
digital	1670	1683	85	5	4	3447
information	3325	3982	378	5	13	7703
count(context)	4997	5673	473	512	61	11716

Figure 6.10 Co-occurrence counts for four words in 5 contexts in the Wikipedia corpus, together with the marginals, pretending for the purpose of this calculation that no other words/contexts matter.

Thus for example we could compute **PPMI**(information,data), assuming we pretended that Fig. 6.6 encompassed all the relevant word contexts/dimensions, as follows:

$$\begin{aligned} P(\text{w=information, c=data}) &= \frac{3982}{11716} = .3399 \\ P(\text{w=information}) &= \frac{7703}{11716} = .6575 \\ P(\text{c=data}) &= \frac{5673}{11716} = .4842 \\ \text{PPMI(information,data)} &= \log 2(.3399/(.6575 * .4842)) = .0944 \end{aligned}$$

Fig. 6.11 shows the joint probabilities computed from the counts in Fig. 6.10, and Fig. 6.12 shows the PPMI values. Not surprisingly, *cherry* and *strawberry* are highly associated with both *pie* and *sugar*, and *data* is mildly associated with *information*.

PMI has the problem of being biased toward infrequent events; very rare words tend to have very high PMI values. One way to reduce this bias toward low frequency

[5] Positive PMI also cleanly solves the problem of what to do with zero counts, using 0 to replace the $-\infty$ from $\log(0)$.

	p(w,context)					p(w)
	computer	data	result	pie	sugar	p(w)
cherry	0.0002	0.0007	0.0008	0.0377	0.0021	0.0415
strawberry	0.0000	0.0000	0.0001	0.0051	0.0016	0.0068
digital	0.1425	0.1436	0.0073	0.0004	0.0003	0.2942
information	0.2838	0.3399	0.0323	0.0004	0.0011	0.6575
p(context)	0.4265	0.4842	0.0404	0.0437	0.0052	

Figure 6.11 Replacing the counts in Fig. 6.6 with joint probabilities, showing the marginals around the outside.

	computer	data	result	pie	sugar
cherry	0	0	0	4.38	3.30
strawberry	0	0	0	4.10	5.51
digital	0.18	0.01	0	0	0
information	0.02	0.09	0.28	0	0

Figure 6.12 The PPMI matrix showing the association between words and context words, computed from the counts in Fig. 6.11. Note that most of the 0 PPMI values are ones that had a negative PMI; for example PMI(*cherry,computer*) = -6.7, meaning that *cherry* and *computer* co-occur on Wikipedia less often than we would expect by chance, and with PPMI we replace negative values by zero.

events is to slightly change the computation for $P(c)$, using a different function $P_\alpha(c)$ that raises the probability of the context word to the power of α:

$$\text{PPMI}_\alpha(w,c) = \max(\log_2 \frac{P(w,c)}{P(w)P_\alpha(c)}, 0) \tag{6.21}$$

$$P_\alpha(c) = \frac{count(c)^\alpha}{\sum_c count(c)^\alpha} \tag{6.22}$$

Levy et al. (2015) found that a setting of $\alpha = 0.75$ improved performance of embeddings on a wide range of tasks (drawing on a similar weighting used for skip-grams described below in Eq. 6.32). This works because raising the count to $\alpha = 0.75$ increases the probability assigned to rare contexts, and hence lowers their PMI ($P_\alpha(c) > P(c)$ when c is rare).

Another possible solution is Laplace smoothing: Before computing PMI, a small constant k (values of 0.1-3 are common) is added to each of the counts, shrinking (discounting) all the non-zero values. The larger the k, the more the non-zero counts are discounted.

6.7 Applications of the tf-idf or PPMI vector models

In summary, the vector semantics model we've described so far represents a target word as a vector with dimensions corresponding either to the documents in a large collection (the term-document matrix) or to the counts of words in some neighboring window (the term-term matrix). The values in each dimension are counts, weighted by tf-idf (for term-document matrices) or PPMI (for term-term matrices), and the vectors are sparse (since most values are zero).

The model computes the similarity between two words x and y by taking the cosine of their tf-idf or PPMI vectors; high cosine, high similarity. This entire model

is sometimes referred to as the **tf-idf** model or the **PPMI** model, after the weighting function.

The tf-idf model of meaning is often used for document functions like deciding if two documents are similar. We represent a document by taking the vectors of all the words in the document, and computing the **centroid** of all those vectors. The centroid is the multidimensional version of the mean; the centroid of a set of vectors is a single vector that has the minimum sum of squared distances to each of the vectors in the set. Given k word vectors $w_1, w_2, ..., w_k$, the centroid **document vector** d is:

centroid

document vector

$$d = \frac{w_1 + w_2 + ... + w_k}{k} \tag{6.23}$$

Given two documents, we can then compute their document vectors d_1 and d_2, and estimate the similarity between the two documents by $\cos(d_1, d_2)$. Document similarity is also useful for all sorts of applications: information retrieval, plagiarism detection, news recommender systems, and even for digital humanities tasks like comparing different versions of a text to see which are similar to each other.

Either the PPMI model or the tf-idf model can be used to compute 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 find 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.

6.8 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 or documents in a collection. 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 $|V|$ or number of documents D we've seen. 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 on-

line with code and pretrained embeddings. Word2vec embeddings are **static embeddings**, meaning that the method learns one fixed embedding for each word in the vocabulary. In Chapter 11 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.

static embeddings

The intuition of word2vec is that instead of counting how often each word w occurs near, say, *apricot*, we'll instead train a classifier on a binary prediction task: "Is word w 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.

6.8.1 The classifier

as we learned in WK 6 of the sml textbook

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*):

$$P(+|w,c) \tag{6.24}$$

The probability that word c is not a real context word for w is just 1 minus Eq. 6.24:

$$P(-|w,c) = 1 - P(+|w,c) \tag{6.25}$$

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:

$$Similarity(w,c) \approx \mathbf{c} \cdot \mathbf{w} \tag{6.26}$$

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:

$$\sigma(x) = \frac{1}{1+\exp(-x)} \tag{6.27}$$

We model the probability that word c is a real context word for target word w as:

$$P(+|w,c) = \sigma(\mathbf{c} \cdot \mathbf{w}) = \frac{1}{1+\exp(-\mathbf{c} \cdot \mathbf{w})} \tag{6.28}$$

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:

$$\begin{aligned} P(-|w,c) &= 1 - P(+|w,c) \\ &= \sigma(-\mathbf{c} \cdot \mathbf{w}) = \frac{1}{1+\exp(\mathbf{c} \cdot \mathbf{w})} \end{aligned} \tag{6.29}$$

Equation 6.28 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:

$$P(+|w,c_{1:L}) = \prod_{i=1}^{L} \sigma(\mathbf{c_i} \cdot \mathbf{w}) \tag{6.30}$$

$$\log P(+|w,c_{1:L}) = \sum_{i=1}^{L} \log \sigma(\mathbf{c_i} \cdot \mathbf{w}) \tag{6.31}$$

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. 6.13 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.[6] Let's now turn to learning these embeddings (which is the real goal of training this classifier in the first place).

[6] In principle the target matrix and the context matrix could use different vocabularies, but we'll simplify by assuming one shared vocabulary V.

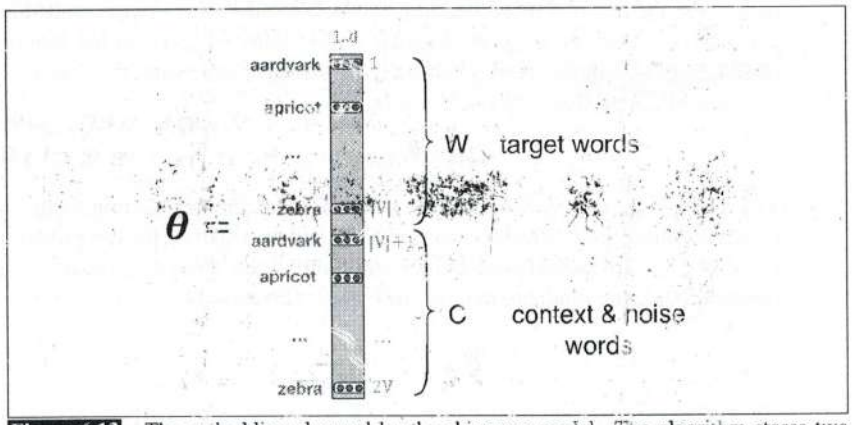


Figure 6.13 The embeddings learned by the skipgram model. The algorithm stores two embeddings for each word, the target embedding (sometimes called the input embedding) and the context embedding (sometimes called the output embedding). The parameter θ that the algorithm learns is thus a matrix of $2|V|$ vectors, each of dimension d, formed by concatenating two matrices, the target embeddings $\mathbf{W}$ and the context+noise embeddings $\mathbf{C}$.

6.8.2 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 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 frequency $p_\alpha(w)$, where α is a weight. If we were sampling according to unweighted frequency $p(w)$, 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 $\alpha = .75$, i.e. use the

weighting $p^{\frac{3}{4}}(w)$:

$$P_\alpha(w) = \frac{count(w)^\alpha}{\sum_{w'} count(w')^\alpha} \tag{6.32}$$

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 two events, $P(a) = .99$ and $P(b) = .01$:

$$\begin{aligned} P_\alpha(a) &= \frac{.99^{.75}}{.99^{.75} + .01^{.75}} = .97 \\ P_\alpha(b) &= \frac{.01^{.75}}{.99^{.75} + .01^{.75}} = .03 \end{aligned} \tag{6.33}$$

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 $c_{neg_1} \ldots c_{neg_k}$, 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:

$$\begin{aligned} L_{CE} &= -\log \left[P(+|w, c_{pos}) \prod_{i=1}^{k} P(-|w, c_{neg_i}) \right] \\ &= -\left[\log P(+|w, c_{pos}) + \sum_{i=1}^{k} \log P(-|w, c_{neg_i}) \right] \\ &= -\left[\log P(+|w, c_{pos}) + \sum_{i=1}^{k} \log \left(1 - P(+|w, c_{neg_i})\right) \right] \\ &= -\left[\log \sigma(c_{pos} \cdot w) + \sum_{i=1}^{k} \log \sigma(-c_{neg_i} \cdot w) \right] \end{aligned} \tag{6.34}$$

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. 6.14 shows the intuition of one step of learning.

To get the gradient, we need to take the derivative of Eq. 6.34 with respect to the different embeddings. It turns out the derivatives are the following (we leave the

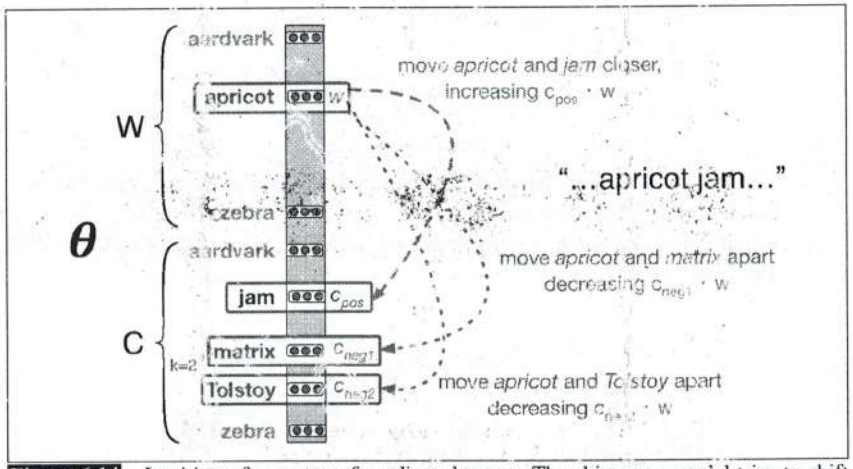


Figure 6.14 Intuition of one step of gradient descent. The skip-gram model tries to shift embeddings so the target embeddings (here for *apricot*) are closer to (have a higher dot product with) context embeddings for nearby words (here *jam*) and further from (lower dot product with) context embeddings for noise words that don't occur nearby (here *Tolstoy* and *matrix*).

proof as an exercise at the end of the chapter):

$$\frac{\partial L_{CE}}{\partial c_{pos}} = [\sigma(\mathbf{c}_{pos} \cdot \mathbf{w}) - 1]\mathbf{w} \tag{6.35}$$

$$\frac{\partial L_{CE}}{\partial c_{neg}} = [\sigma(\mathbf{c}_{neg} \cdot \mathbf{w})]\mathbf{w} \tag{6.36}$$

$$\frac{\partial L_{CE}}{\partial w} = [\sigma(\mathbf{c}_{pos} \cdot \mathbf{w}) - 1]\mathbf{c}_{pos} + \sum_{i=1}^{k}[\sigma(\mathbf{c}_{neg_i} \cdot \mathbf{w})]\mathbf{c}_{neg_i} \tag{6.37}$$

The update equations going from time step t to $t+1$ in stochastic gradient descent are thus:

$$\mathbf{c}_{pos}^{t+1} = \mathbf{c}_{pos}^{t} - \eta[\sigma(\mathbf{c}_{pos}^{t} \cdot \mathbf{w}^{t}) - 1]\mathbf{w}^{t} \tag{6.38}$$

$$\mathbf{c}_{neg}^{t+1} = \mathbf{c}_{neg}^{t} - \eta[\sigma(\mathbf{c}_{neg}^{t} \cdot \mathbf{w}^{t})]\mathbf{w}^{t} \tag{6.39}$$

$$\mathbf{w}^{t+1} = \mathbf{w}^{t} - \eta\left[[\sigma(\mathbf{c}_{pos} \cdot \mathbf{w}^{t}) - 1]\mathbf{c}_{pos} + \sum_{i=1}^{k}[\sigma(\mathbf{c}_{neg_i} \cdot \mathbf{w}^{t})]\mathbf{c}_{neg_i}\right] \tag{6.40}$$

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 maximize the objective in Eq. 6.34 by making the updates in (Eq. 6.39)-(Eq. 6.40).

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 L affects the performance of skip-gram embeddings, and experiments often tune the parameter L on a devset.

6.8.3 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`.

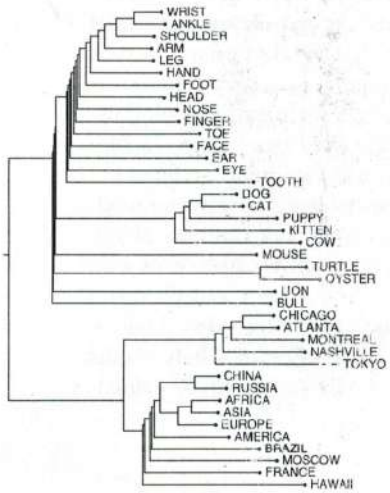
Another very widely used static embedding model is GloVe (Pennington et al., 2014), short for Global Vectors, because the model is based on capturing global corpus statistics. GloVe is based on ratios of probabilities from the word-word co-occurrence matrix, combining the intuitions of count-based models like PPMI while also capturing the linear structures used by methods like word2vec.

It turns out that dense embeddings like word2vec actually have an elegant mathematical relationship with sparse embeddings like PPMI, in which word2vec can be seen as implicitly optimizing a shifted version of a PPMI matrix (Levy and Goldberg, 2014c).

6.9 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 the GloVe embeddings 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. 6.1 showed one such visualization, as does Fig. 6.16, using a projection method called t-SNE (van der Maaten and Hinton, 2008).

6.10 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 tf-idf 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

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*.

second-order co-occurrence

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 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. 6.15, 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-

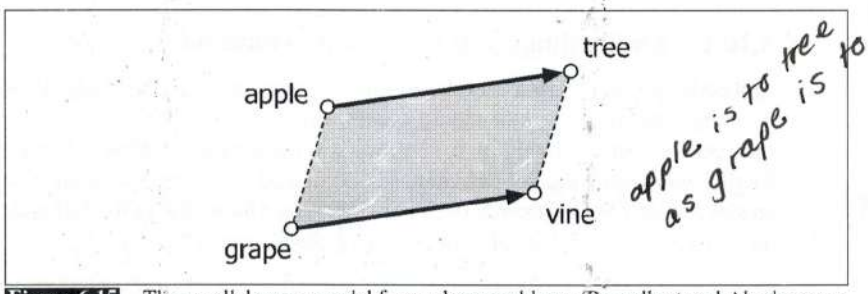


Figure 6.15 The parallelogram model for analogy problems (Rumelhart and Abrahamson, 1973): the location of $\overrightarrow{\text{vine}}$ can be found by subtracting $\overrightarrow{\text{apple}}$ from $\overrightarrow{\text{tree}}$ and adding $\overrightarrow{\text{grape}}$.

ing representations of relations like MALE-FEMALE, or CAPITAL-CITY-OF, or even COMPARATIVE/SUPERLATIVE, as shown in Fig. 6.16 from GloVe.

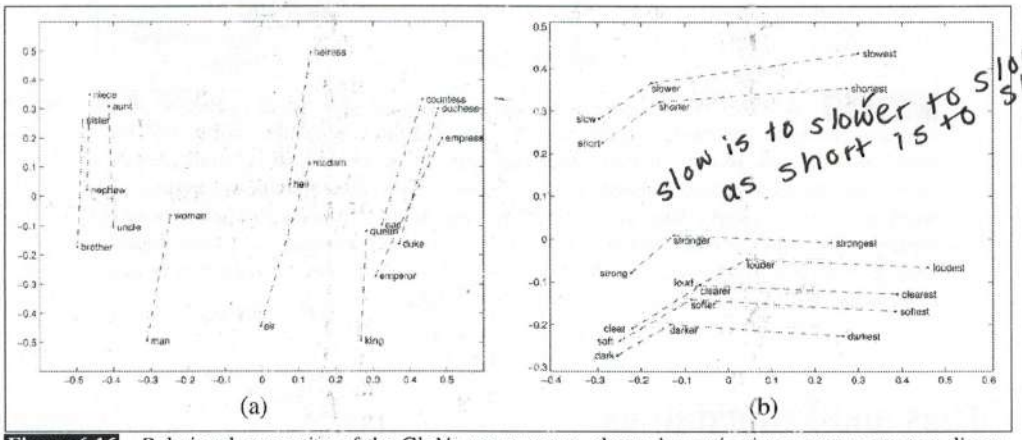


Figure 6.16 Relational properties of the GloVe vector space, shown by projecting vectors onto two dimensions. (a) $\overrightarrow{\text{king}} - \overrightarrow{\text{man}} + \overrightarrow{\text{woman}}$ is close to $\overrightarrow{\text{queen}}$. (b) offsets seem to capture comparative and superlative morphology (Pennington et al., 2014).

For a **a** : **b** :: **a*** : **b*** problem, meaning the algorithm is given vectors **a**, **b**, and **a*** and must find **b***, the parallelogram method is thus:

$$\hat{\mathbf{b}}^* = \underset{\mathbf{x}}{\text{argmin}}\ \text{distance}(\mathbf{x}, \mathbf{b} - \mathbf{a} + \mathbf{a}^*) \tag{6.41}$$

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.

6.10.1 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. 6.17 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., 2012) and the Corpus of Historical American English (Davies, 2012).

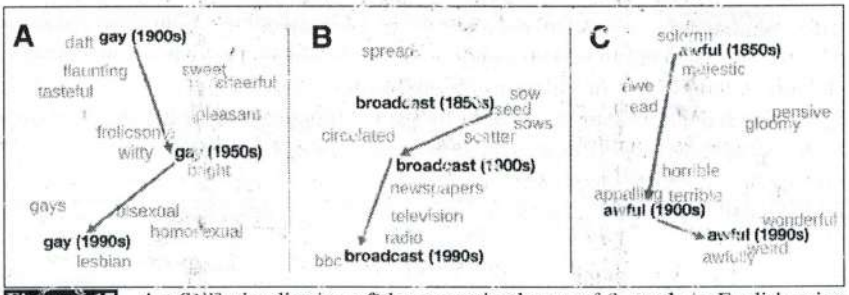


Figure 6.17 A t-SNE visualization of the semantic change of 3 words in English using word2vec vectors. The modern sense of each word, and the grey context words, are computed from the most recent (modern) time-point embedding space. Earlier points are computed from earlier historical embedding spaces. 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., 2016).

6.11 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 'man' - 'computer programmer' + '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.[7] 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., 2018). 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.

6.12 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.

[7] 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 difficult dataset that quantifies similarity (*cup, mug*) rather than relatedness (*cup, coffee*), and including both 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 Section **??**. 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 24, 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).

6.13 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**, in each 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 **term-document** matrix has a row for each word (**term**) in the vocabulary and a column for each document. 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. Two sparse weightings are common: the **tf-idf** weighting which weights each cell by its **term frequency** and **inverse document frequency**, and **PPMI** (pointwise positive mutual information), which is most common for for word-context matrices.

- Dense vector models 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.

Bibliographical and 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.

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, suggesting 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 data set, 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 (1992). 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 (1992). 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) 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

7 Neural Networks and Neural Language Models

> "[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 human 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 words 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 6. Nets that are very deep are particularly good at representation learning. For that reason deep neural nets are the right tool for large scale problems that offer sufficient data to learn features automatically.

In this chapter we'll introduce feedforward networks as classifiers, and also apply them to the simple task of language modeling: assigning probabilities to word sequences and predicting upcoming words. In subsequent chapters we'll introduce many other aspects of neural models, such as **recurrent neural networks** and the **Transformer** (Chapter 9), contextual embeddings like **BERT** (Chapter 11), and **encoder-decoder** models and **attention** (Chapter 10).

7.1 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.

bias term

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 $x_1 \ldots x_n$, a unit has a set of corresponding weights $w_1 \ldots w_n$ and a bias b, so the weighted sum z can be represented as:

$$z = b + \sum_i w_i x_i \tag{7.1}$$

vector

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**:

$$z = \mathbf{w} \cdot \mathbf{x} + b \tag{7.2}$$

As defined in Eq. 7.2, z is just a real valued number.

activation

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:

$$y = a = f(z)$$

sigmoid

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 5:

$$y = \sigma(z) = \frac{1}{1+e^{-z}} \tag{7.3}$$

The sigmoid (shown in Fig. 7.1) 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 **??** will be handy for learning.

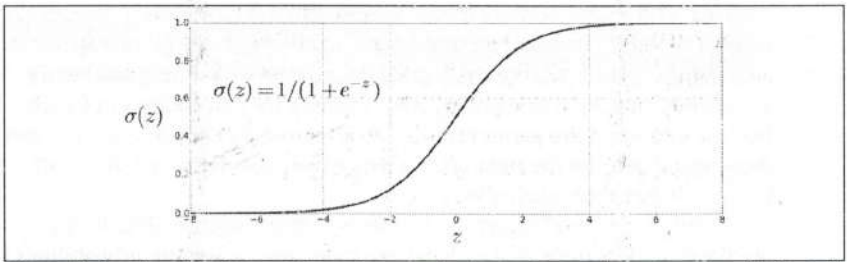


Figure 7.1 The sigmoid function takes a real value and maps it to the range $[0,1]$. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

Substituting Eq. 7.2 into Eq. 7.3 gives us the output of a neural unit:

$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1+\exp(-(\mathbf{w} \cdot \mathbf{x} + b))} \tag{7.4}$$

Fig. 7.2 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.

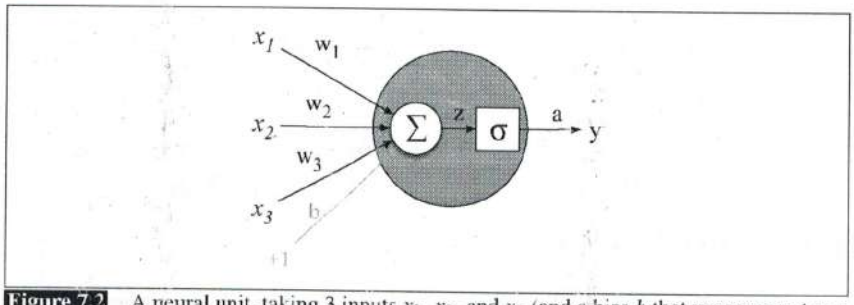


Figure 7.2 A neural unit, taking 3 inputs x_1, x_2, and x_3 (and a bias b that we represent as a weight for an input clamped at +1) and producing an output y. We include some convenient intermediate variables: the output of the summation, z, and the output of the sigmoid, a. In this case the output of 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:

$$\mathbf{w} = [0.2, 0.3, 0.9]$$
$$b = 0.5$$

What would this unit do with the following input vector:

$$\mathbf{x} = [0.5, 0.6, 0.1]$$

The resulting output y would be:

$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1+e^{-(\mathbf{w} \cdot \mathbf{x}+b)}} = \frac{1}{1+e^{-(.5*.2+.6*.3+.1*.9+.5)}} = \frac{1}{1+e^{-0.87}} = .70$$

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. 7.3a; tanh is a variant of the sigmoid that ranges from -1 to +1:

tanh

$$y = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{7.5}$$

The simplest activation function, and perhaps the most commonly used, is the rectified linear unit, also called the **ReLU**, shown in Fig. 7.3b. It's just the same as z when z is positive, and 0 otherwise:

ReLU

$$y = \text{ReLU}(z) = max(z, 0) \tag{7.6}$$

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

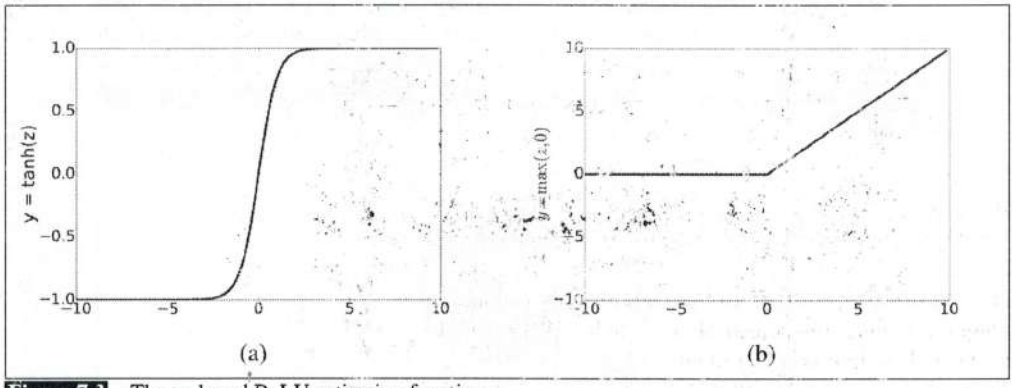


Figure 7.3 The tanh and ReLU activation functions.

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 7.6, 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

7.2 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 does **not** have a 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. 7.2):

$$y = \begin{cases} 0, & \text{if } \mathbf{w}\cdot\mathbf{x}+b \leq 0 \\ 1, & \text{if } \mathbf{w}\cdot\mathbf{x}+b > 0 \end{cases} \tag{7.7}$$

It's very easy to build a perceptron that can compute the logical AND and OR functions of its binary inputs; Fig. 7.4 shows the necessary weights.

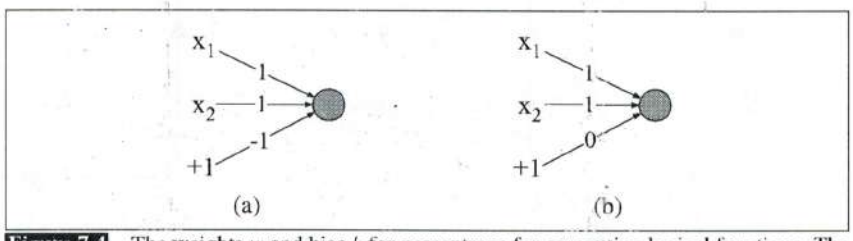


Figure 7.4 The weights w and bias b for perceptrons for computing logical functions. The inputs are shown as x_1 and x_2 and the bias as a special node with value $+1$ which is multiplied with the bias weight b. (a) logical AND, showing weights $w_1 = 1$ and $w_2 = 1$ and bias weight $b = -1$. (b) logical OR, showing 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. 7.5 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

7.2.1 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. 7.6 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 for each ReLU that correctly computes 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].

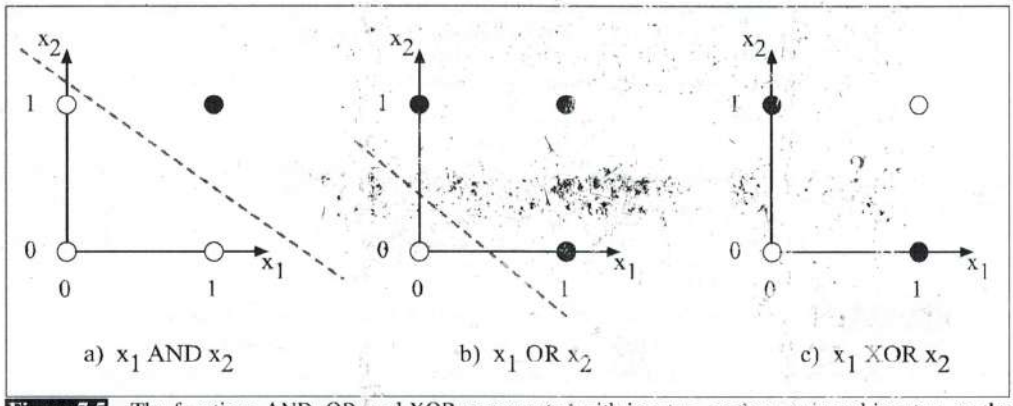


Figure 7.5 The functions AND, OR, and XOR, represented with input x_1 on the x-axis and input x_2 on the y axis. Filled circles represent perceptron outputs of 1, and white circles perceptron outputs of 0. There is no way to draw a line that correctly separates the two categories for XOR. Figure styled after Russell and Norvig (2002).

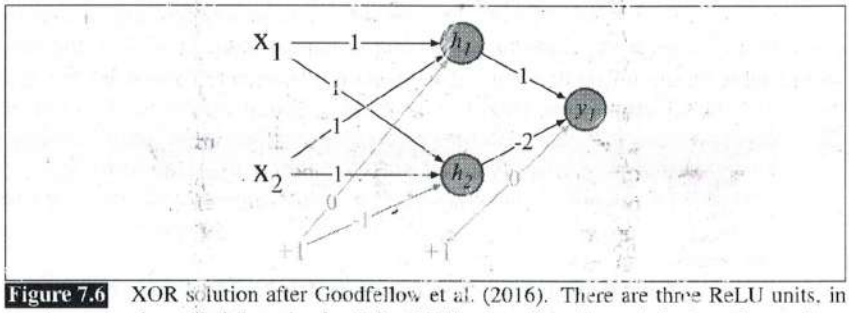


Figure 7.6 XOR solution after Goodfellow et al. (2016). There are three ReLU units, in two layers; we've called them h_1, h_2 (h for "hidden layer") and y_1. As before, the numbers on the arrows represent the weights w for each unit, and we represent the bias b as a weight on a unit clamped to +1, with 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 **h** vector for the inputs $\mathbf{x} = [0, 0]$ was $[0, 0]$. Fig. 7.7b shows the values of the **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. 7.6. But for real examples the weights for neural networks are learned automatically using the error backpropagation algorithm to be introduced in Section 7.6. 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.

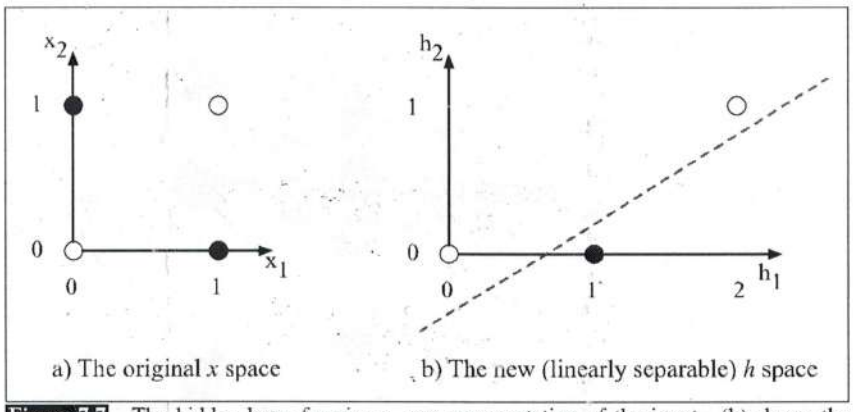


Figure 7.7 The hidden layer forming a new representation of the input. (b) shows the representation of the hidden layer, **h**, compared to the original input representation **x** in (a). Notice that the input point [0, 1] has been collapsed with the input point [1, 0], making it possible to linearly separate the positive and negative cases of XOR. After Goodfellow et al. (2016).

7.3 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 9 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 **MLPs**); this is a technical misnomer, since the units in modern multilayer networks aren't perceptrons (perceptrons are purely linear, but modern networks are made up of units with non-linearities like sigmoids), but at some point the name stuck.

Simple feedforward networks have three kinds of nodes: input units, hidden units, and output units. Fig. 7.8 shows a picture.

The input layer **x** is a vector of simple scalar values just as we saw in Fig. 7.2.

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 7.1, 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 **W** and a single bias vector b for the whole layer (see Fig. 7.8). Each element $\mathbf{W}_{ji}$ of the weight matrix **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 **W** for the weights of the entire layer is that now the hidden layer computation for a feedforward network can be done very efficiently with simple matrix operations. In fact, the computation only has three steps: multiplying the weight matrix by the input vector **x**, adding the bias vector **b**,

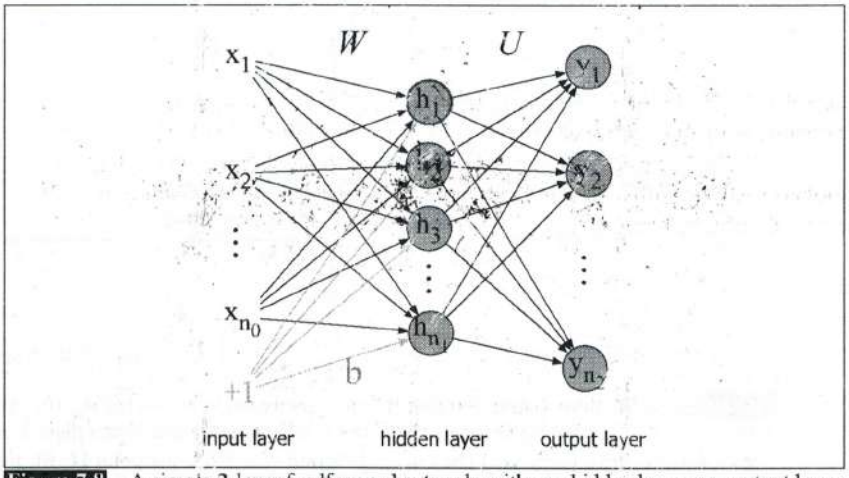


Figure 7.8 A simple 2-layer feedforward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

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):

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{7.8}$$

Notice that we're applying the σ function here to a vector, while in Eq. 7.3 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 $[n_0, 1]$. 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. $[n_1, n_0]$.

Take a moment to convince yourself that the matrix multiplication in Eq. 7.8 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 7.2, 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 ?? of Chapter 5. More generally for any vector $\mathbf{z}$ of dimensionality d, the softmax is defined as:

normalizing

softmax

$$\text{softmax}(\mathbf{z}_i) = \frac{\exp(\mathbf{z}_i)}{\sum_{j=1}^{d} \exp(\mathbf{z}_j)} \quad 1 \le i \le d \tag{7.9}$$

Thus for example given a vector

$$\mathbf{z} = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1], \tag{7.10}$$

The softmax function will normalize it to a probability distribution (shown rounded):

$$\text{softmax}(\mathbf{z}) = [0.055, 0.090, 0.0067, 0.10, 0.74, 0.010]. \tag{7.11}$$

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 5.

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 5 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 $\mathbf{b}$:

$$\begin{aligned} \mathbf{h} &= \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \\ \mathbf{z} &= \mathbf{U}\mathbf{h} \\ \mathbf{y} &= \text{softmax}(\mathbf{z}) \end{aligned} \tag{7.12}$$

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.

7.3.1 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 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. 7.12 as follows.

$$\begin{aligned}
\mathbf{z}^{[1]} &= \mathbf{W}^{[1]}\mathbf{a}^{[0]} + \mathbf{b}^{[1]} \\
\mathbf{a}^{[1]} &= g^{[1]}(\mathbf{z}^{[1]}) \\
\mathbf{z}^{[2]} &= \mathbf{W}^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]} \\
\mathbf{a}^{[2]} &= g^{[2]}(\mathbf{z}^{[2]}) \\
\hat{\mathbf{y}} &= \mathbf{a}^{[2]}
\end{aligned} \tag{7.13}$$

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
  $\mathbf{z}^{[i]} = \mathbf{W}^{[i]}\,\mathbf{a}^{[i-1]} + \mathbf{b}^{[i]}$
  $\mathbf{a}^{[i]} = g^{[i]}(\mathbf{z}^{[i]})$
$\hat{\mathbf{y}} = \mathbf{a}^{[n]}$

The activation functions $g(\cdot)$ are generally different at the final layer. Thus $g^{[2]}$ might be softmax for multinomial classification or sigmoid for binary classification, while ReLU or tanh might be the activation function $g(\cdot)$ at the internal layers.

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:

$$\begin{aligned}
\mathbf{z}^{[2]} &= \mathbf{W}^{[2]}\mathbf{z}^{[1]} + \mathbf{b}^{[2]} \\
&= \mathbf{W}^{[2]}(\mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}) + \mathbf{b}^{[2]} \\
&= \mathbf{W}^{[2]}\mathbf{W}^{[1]}\mathbf{x} + \mathbf{W}^{[2]}\mathbf{b}^{[1]} + \mathbf{b}^{[2]} \\
&= \mathbf{W}'\mathbf{x} + \mathbf{b}'
\end{aligned} \tag{7.14}$$

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 often 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

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{7.15}$$

we'll use:

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x}) \tag{7.16}$$

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:

$$\mathbf{h}_j = \sigma\left(\sum_{i=1}^{n_0} \mathbf{W}_{ji}\mathbf{x}_i + \mathbf{b}_j\right), \tag{7.17}$$

we'll instead use:

$$\mathbf{h}_j = \sigma\left(\sum_{i=0}^{n_0} \mathbf{W}_{ji}\mathbf{x}_i\right), \tag{7.18}$$

where the value $\mathbf{W}_{j0}$ replaces what had been $\mathbf{b}_j$. Fig. 7.9 shows a visualization.

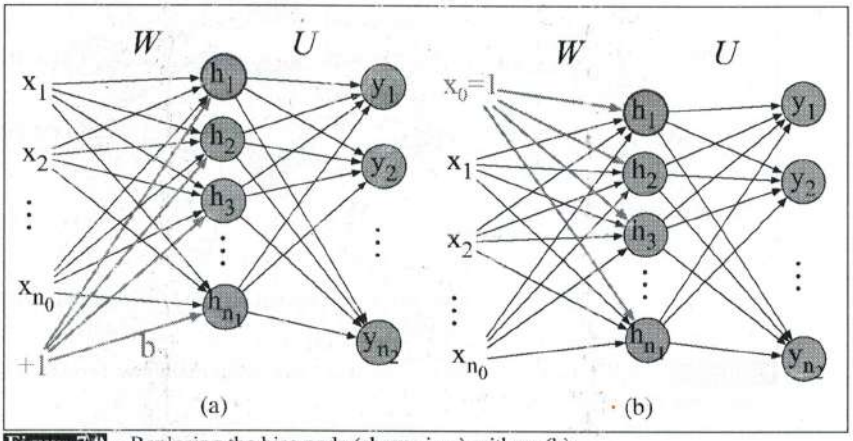


Figure 7.9 Replacing the bias node (shown in a) with x_0 (b).

We'll continue showing the bias as b when we go over the learning algorithm in Section 7.6, but then we'll switch to this simplified notation without explicit bias terms for the rest of the book.

7.4 Feedforward networks for NLP: Classification

Let's see how to apply feedforward networks to NLP tasks! In this section we'll look at classification tasks like sentiment analysis; in the next section we'll introduce neural language modeling.

Let's begin with a simple 2-layer sentiment classifier. You might imagine taking our logistic regression classifier from Chapter 5, which corresponds to a 1-layer network, and just adding a hidden layer. The input element x_i could be scalar features like those in Fig. ??, e.g., x_1 = count(words ∈ doc), x_2 = count(positive lexicon words ∈ doc), x_3 = 1 if "no" ∈ doc, and so on. And the output layer $\hat{y}$ could have two nodes (one each for positive and negative), or 3 nodes (positive, negative, neutral), in which case $\hat{y}_1$ would be the estimated probability of positive sentiment, $\hat{y}_2$ the probability of negative and $\hat{y}_3$ 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).

$$\begin{aligned}\mathbf{x} &= [x_1, x_2, \ldots x_N] \quad \text{(each } x_i \text{ is a hand-designed feature)}\\ \mathbf{h} &= \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})\\ \mathbf{z} &= \mathbf{U}\mathbf{h}\\ \hat{\mathbf{y}} &= \text{softmax}(\mathbf{z})\end{aligned} \tag{7.19}$$

Fig. 7.10 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.

more flexibility

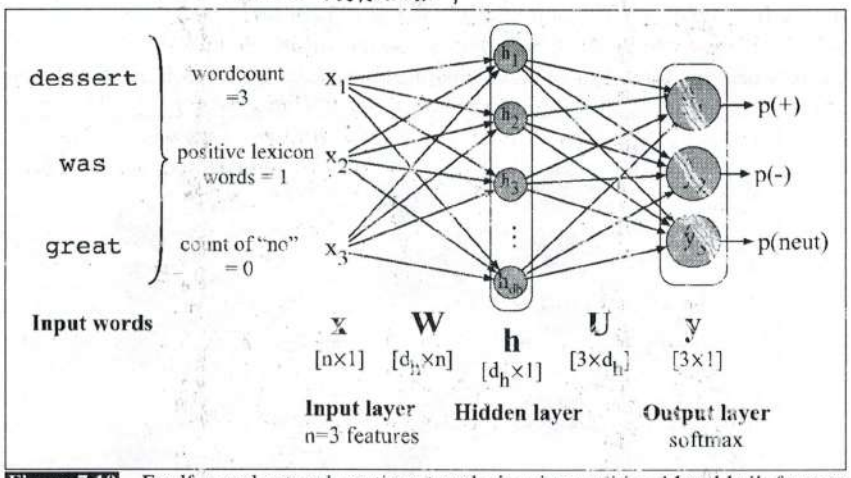


Figure 7.10 Feedforward network sentiment analysis using traditional hand-built features of the input text.

Most neural NLP applications do something different, however. Instead of using hand-built human-engineered features as the input to our classifier, we draw on deep learning's ability to learn features from the data by representing words as embeddings, like the word2vec or GloVe embeddings we saw in Chapter 6. There are various ways to represent an input for classification. One simple baseline is to apply some sort of **pooling** function to the embeddings of all the words in the input. For example, for a text with n input words/tokens $w_1, \ldots, w_n$, we can turn the n embeddings $\mathbf{e}(w_1), \ldots, \mathbf{e}(w_n)$ (each of dimensionality d) into a single embedding also of dimensionality d by just summing the embeddings, or by taking their mean (summing and then dividing by n):

pooling

$$\mathbf{x}_{mean} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{e}(w_i) \tag{7.20}$$

There are many other options, like taking 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. Here are the equations for this classifier assuming mean pooling; the architecture is sketched in Fig. 7.11:

$$\begin{aligned}\mathbf{x} &= \text{mean}(\mathbf{e}(w_1), \mathbf{e}(w_2), \ldots, \mathbf{e}(w_n)) \\ \mathbf{h} &= \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \\ \mathbf{z} &= \mathbf{U}\mathbf{h} \\ \hat{\mathbf{y}} &= \text{softmax}(\mathbf{z})\end{aligned} \tag{7.21}$$

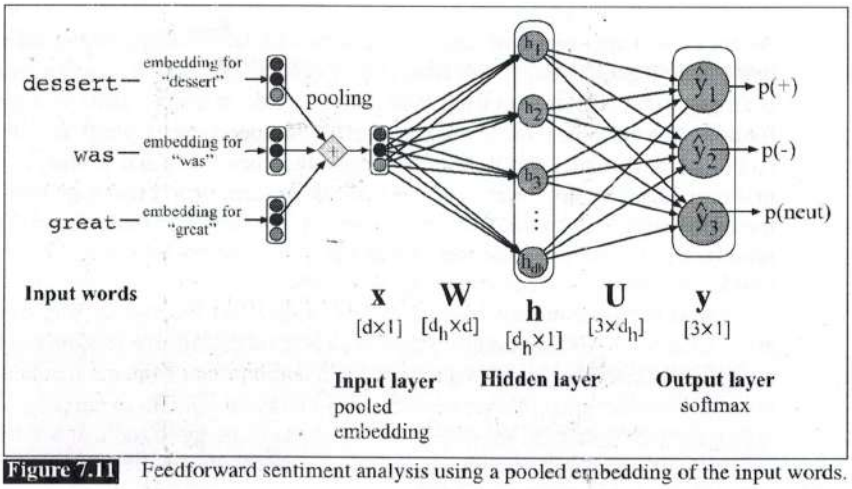


Figure 7.11 Feedforward sentiment analysis using a pooled embedding of the input words.

While Eq. 7.21 shows how to a classify a single example x, in practice we want to efficiently classify an entire test set of m examples. We do this by vectoring 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 pooled embedding for input example $x^{(i)}$ (i.e., the vector $\mathbf{x}^{(i)}$). If the dimensionality of our pooled input embedding is d, $\mathbf{X}$ will be a matrix of shape $[m \times d]$.

We will then need to slightly modify Eq. 7.21. $\mathbf{X}$ is of shape $[m \times d]$ and $\mathbf{W}$ is of shape $[d_h \times d]$, so we'll have to 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 $[m \times d_h]$. The bias vector $\mathbf{b}$ from Eq. 7.21 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:

$$\begin{aligned}\mathbf{H} &= \sigma(\mathbf{X}\mathbf{W}^\intercal + \mathbf{b}) \\ \mathbf{Z} &= \mathbf{H}\mathbf{U}^\intercal \\ \hat{\mathbf{Y}} &= \text{softmax}(\mathbf{Z})\end{aligned} \tag{7.22}$$

The idea of using word2vec or GloVe embeddings as our input representation—and more generally the idea of relying on another algorithm to have already learned

pretraining

an embedding representation for our input words—is called **pretraining**. Using pretrained embedding representations, whether simple static word embeddings like word2vec or the much more powerful contextual embeddings we'll introduce in Chapter 11, is one of the central ideas of deep learning. (It's also, possible, however, to train the word embeddings as part of an NLP task; we'll talk about how to do this in Section 7.7 in the context of the neural language modeling task.)

7.5 Feedforward Neural Language Modeling

As our second application of feedforward networks, let's consider **language modeling**: predicting upcoming words from prior word context. Neural language modeling is an important NLP task in itself, and it plays a role in many important algorithms for tasks like machine translation, summarization, speech recognition, grammar correction, and dialogue. We'll describe simple feedforward neural language models, first introduced by Bengio et al. (2003). While modern neural language models use more powerful architectures like the recurrent nets or transformer networks to be introduced in Chapter 9, the feedforward language model introduces many of the important concepts of neural language modeling.

Neural language models have many advantages over the n-gram language models of Chapter 3. Compared to n-gram models, neural language models can handle much longer histories, can generalize better over contexts of similar words, and are more accurate at word-prediction. On the other hand, neural net language models are much more complex, are slower and need more energy to train, and are less interpretable than n-gram models, so for many (especially smaller) tasks an n-gram language model is still the right tool.

A feedforward neural LM 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:

$$P(w_t|w_1,\ldots,w_{t-1}) \approx P(w_t|w_{t-N+1},\ldots,w_{t-1}) \tag{7.23}$$

In the following examples we'll use a 4-gram example, so we'll show a 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".

7.5.1 Forward inference in the neural language model

forward inference

Let's walk through **forward inference** or **decoding** for neural language models.

Forward inference is the task, given an input, of running a forward pass on the network to produce a probability distribution over possible outputs, in this case next words.

one-hot vector

We first represent each of the N previous words as a one-hot vector of length $|V|$, i.e., with one dimension for each word in the vocabulary. A **one-hot vector** is a vector that has one element equal to 1—in the dimension corresponding to that word's index in the vocabulary— while all the other elements are set to zero. Thus in a one-hot representation for the word "toothpaste", supposing it is V_5, i.e., 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|
```

The feedforward neural language model (sketched in Fig. 7.13) has a moving window that can see N words into the past. We'll let N equal 3, so the 3 words w_{t-1}, w_{t-2}, and w_{t-3} are each represented as a one-hot vector. We then multiply these one-hot vectors by the embedding matrix **E**. The embedding weight matrix **E** has a column for each word, each a column vector of d dimensions, and hence has dimensionality $d \times |V|$. Multiplying by a one-hot vector that has only one non-zero element $x_i = 1$ simply selects out the relevant column vector for word i, resulting in the embedding for word i, as shown in Fig. 7.12.

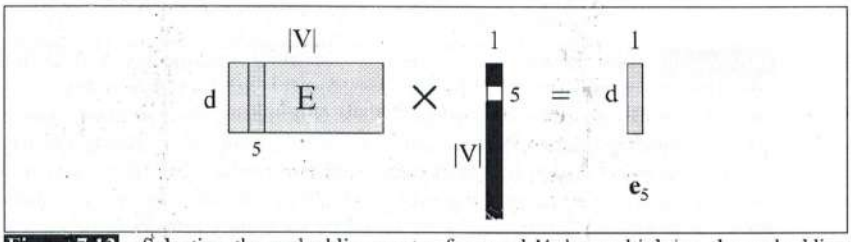


Figure 7.12 Selecting the embedding vector for word V_5 by multiplying the embedding matrix **E** with a one-hot vector with a 1 in index 5.

The 3 resulting embedding vectors are concatenated to produce **e**, the embedding layer. This is followed by a hidden layer and an output layer whose softmax produces 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).

Here's the algorithm in detail for our mini example:

embedding layer

1. **Select three embeddings from E**: Given the three previous words, we look up their indices, create 3 one-hot vectors, and then multiply each by the embedding matrix **E**. Consider w_{t-3}. The one-hot vector for 'for' (index 35) is multiplied by the embedding matrix **E**, to give the first part of the first hidden layer, the **embedding layer**. Since each column of the input matrix **E** is an embedding for a word, and the input is a one-hot column vector $\mathbf{x_i}$ for word V_i, the embedding layer for input w will be $\mathbf{Ex_i} = \mathbf{e_i}$, the embedding for word i. We now concatenate the three embeddings for the three context words to produce the embedding layer **e**.
2. **Multiply by W**: We multiply by W (and add b) and pass through the ReLU (or other) activation function to get the hidden layer h.
3. **Multiply by U**: h is now multiplied by U
4. **Apply softmax**: After the softmax, each node i in the output layer estimates the probability $P(w_t = i|w_{t-1}, w_{t-2}, w_{t-3})$

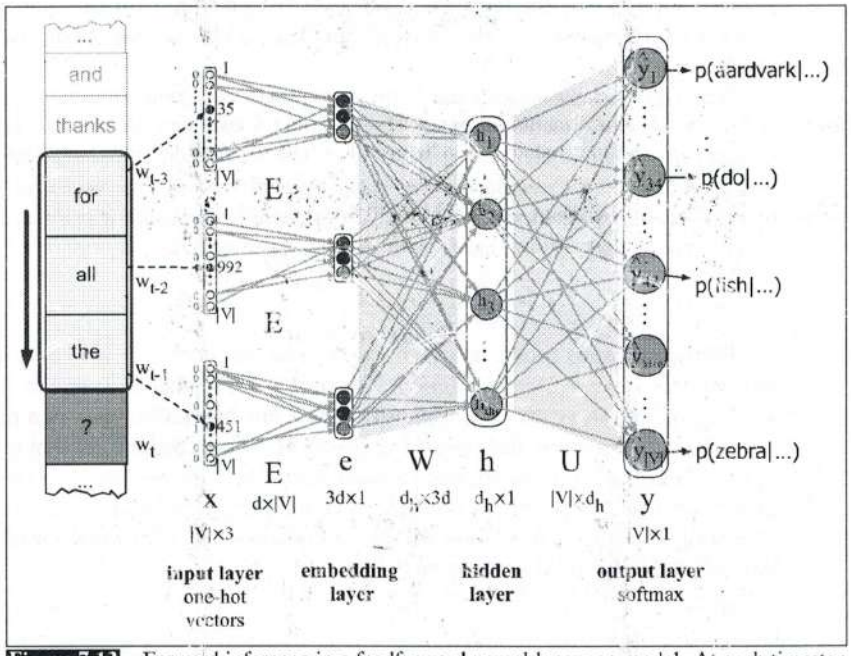


Figure 7.13 Forward inference in a feedforward neural language model. At each timestep t the network computes a d-dimensional embedding for each context word (by multiplying a one-hot vector by the embedding matrix E), and concatenates the 3 resulting embeddings to get the embedding layer $\mathbf{e}$. The embedding vector $\mathbf{e}$ is multiplied by a weight matrix $\mathbf{W}$ and then an activation function is applied element-wise to produce the hidden layer $\mathbf{h}$, which is then multiplied by another weight matrix $\mathbf{U}$. Finally, a softmax output layer predicts at each node i the probability that the next word w_t will be vocabulary word V_i.

In summary, the equations for a neural language model with a window size of 3, given one-hot input vectors for each input context word, are:

$$\begin{aligned}\mathbf{e} &= [\mathbf{E}\mathbf{x}_{t-3}; \mathbf{E}\mathbf{x}_{t-2}; \mathbf{E}\mathbf{x}_{t-1}] \\ \mathbf{h} &= \sigma(\mathbf{W}\mathbf{e} + \mathbf{b}) \\ \mathbf{z} &= \mathbf{U}\mathbf{h} \\ \hat{\mathbf{y}} &= \text{softmax}(\mathbf{z}) \end{aligned} \tag{7.24}$$

Note that we formed the embedding layer $\mathbf{e}$ by concatenating the 3 embeddings for the three context vectors; we'll often use semicolons to mean concatenation of vectors.

In the next section we'll introduce a general algorithm for training neural networks, and then return to how to specifically train the neural language model in Section 7.7.

7.6 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. 7.13, 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 5 for logistic regression, so the reader should be comfortable with that chapter before proceeding.

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 5.

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**.

7.6.1 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. ??:

$$L_{CE}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})] \tag{7.25}$$

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 5 on page ??. 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:

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{K} \mathbf{y}_k \log \hat{\mathbf{y}}_k \tag{7.26}$$

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. 7.26 will be 0 except for the term corresponding to the true class for which $\mathbf{y}_k = 1$:

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{K} \mathbb{1}\{\mathbf{y}_k = 1\} \log \hat{\mathbf{y}}_k$$

negative log likelihood loss

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**:

$$L_{CE}(\hat{\mathbf{y}},\mathbf{y}) = -\log \hat{\mathbf{y}}_c \quad \text{(where } c \text{ is the correct class)} \tag{7.27}$$

Plugging in the softmax formula from Eq. 7.9, and with K the number of classes:

$$L_{CE}(\hat{\mathbf{y}},\mathbf{y}) = -\log \frac{\exp(\mathbf{z}_c)}{\sum_{j=1}^{K}\exp(\mathbf{z}_j)} \quad \text{(where } c \text{ is the correct class)} \tag{7.28}$$

7.6.2 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. 7.29 (and derived in Section **??**):

$$\begin{aligned}\frac{\partial L_{CE}(\hat{\mathbf{y}},\mathbf{y})}{\partial w_j} &= (\hat{y}-y)\,\mathbf{x}_j \\ &= (\sigma(\mathbf{w}\cdot\mathbf{x}+b)-y)\,\mathbf{x}_j\end{aligned} \tag{7.29}$$

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. **??**, shown for a particular weight $\mathbf{w}_k$ and input $\mathbf{x}_i$

$$\begin{aligned}\frac{\partial L_{CE}(\hat{\mathbf{y}},\mathbf{y})}{\partial \mathbf{w}_{k,i}} &= -(\mathbf{y}_k-\hat{\mathbf{y}}_k)\mathbf{x}_i \\ &= -(\mathbf{y}_k - p(\mathbf{y}_k=1|\mathbf{x}))\mathbf{x}_i \\ &= -\left(\mathbf{y}_k - \frac{\exp(\mathbf{w_k}\cdot\mathbf{x}+b_k)}{\sum_{j=1}^{K}\exp(\mathbf{w_j}\cdot\mathbf{x}+b_j)}\right)\mathbf{x}_i\end{aligned} \tag{7.30}$$

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 back-propagation

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.

7.6.3 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. 7.14. 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.

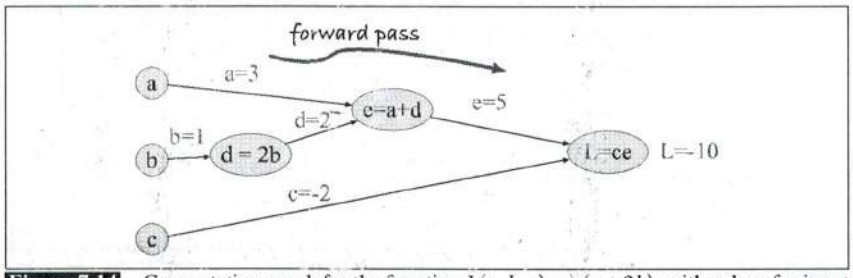


Figure 7.14 Computation graph for the function $L(a, b, c) = c(a+2b)$, with values for input nodes $a = 3$, $b = 1$, $c = -2$, showing the forward pass computation of L.

7.6.4 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:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx} \tag{7.31}$$

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:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dw} \cdot \frac{dw}{dx} \tag{7.32}$$

The intuition of backward differentiation is to pass gradients back from the final node to all the nodes in the graph. Fig. 7.15 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

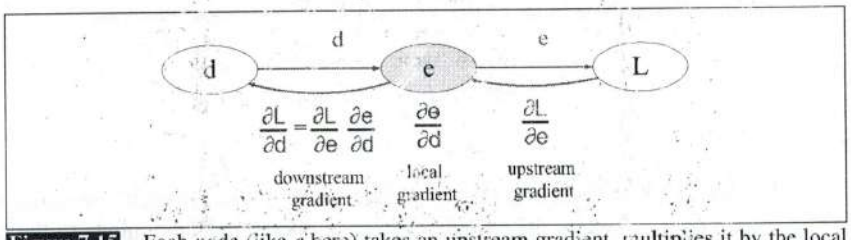


Figure 7.15 Each node (like e here) takes an upstream gradient, multiplies it by the local gradient (the gradient of its output with respect to its input), and uses the chain rule to compute a downstream gradient to be passed on to a prior node. A node may have multiple local gradients if it has multiple inputs.

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 $\frac{\partial L}{\partial c}$:

$$\frac{\partial L}{\partial c} = e \tag{7.33}$$

For the other two, we'll need to use the chain rule:

$$\begin{aligned}\frac{\partial L}{\partial a} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial a} \\ \frac{\partial L}{\partial b} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}\end{aligned} \tag{7.34}$$

Eq. 7.34 and Eq. 7.33 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. 7.16 shows the backward pass.

Backward differentiation for a neural network

Of course computation graphs for real neural networks are much more complex. Fig. 7.17 shows a sample computation graph for a 2-layer neural network with $n_0 =$

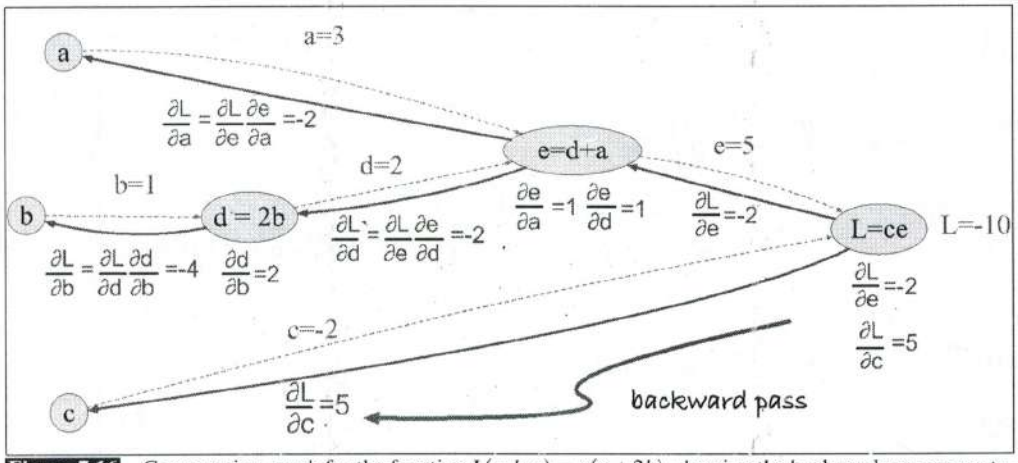


Figure 7.16 Computation graph for the function $L(a,b,c) = c(a+2b)$, showing the backward pass computation of $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$.

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:

$$\begin{aligned}
\mathbf{z}^{[1]} &= \mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]} \\
\mathbf{a}^{[1]} &= \text{ReLU}(\mathbf{z}^{[1]}) \\
z^{[2]} &= \mathbf{W}^{[2]}\mathbf{a}^{[1]} + b^{[2]} \\
a^{[2]} &= \sigma(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned} \tag{7.35}$$

For the backward pass we'll also need to compute the loss L. The loss function for binary sigmoid output from Eq. 7.25 is

$$L_{CE}(\hat{y},y) = -[y\log\hat{y} + (1-y)\log(1-\hat{y})] \tag{7.36}$$

Our output $\hat{y} = a^{[2]}$, so we can rephrase this as

$$L_{CE}(a^{[2]},y) = -\left[y\log a^{[2]} + (1-y)\log(1-a^{[2]})\right] \tag{7.37}$$

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 **??** the derivative of the sigmoid σ:

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z)) \tag{7.38}$$

We'll also need the derivatives of each of the other activation functions. The derivative of tanh is:

$$\frac{d\tanh(z)}{dz} = 1 - \tanh^2(z) \tag{7.39}$$

The derivative of the ReLU is

$$\frac{d\,\text{ReLU}(z)}{dz} = \begin{cases} 0 & for\ \ z < 0 \\ 1 & for\ \ z \geq 0 \end{cases} \tag{7.40}$$

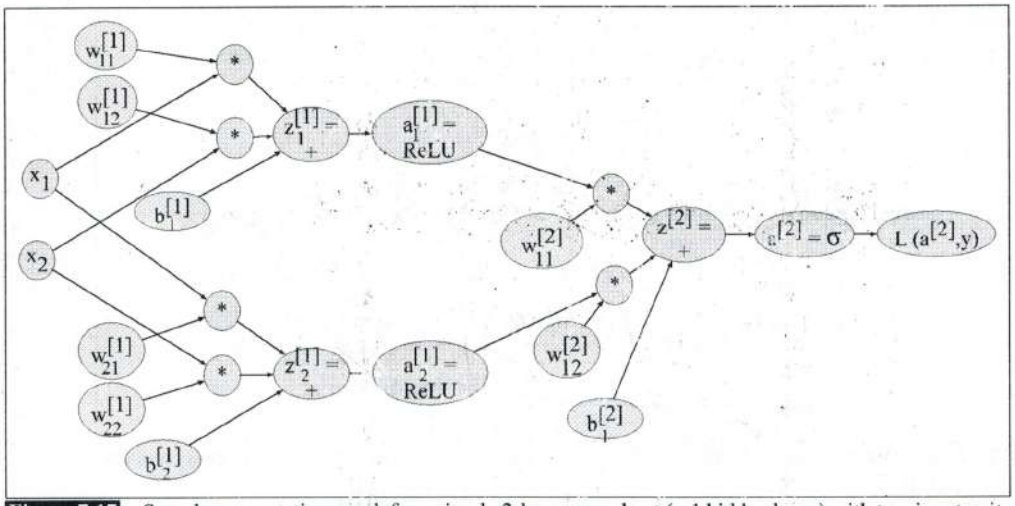


Figure 7.17 Sample computation graph for a simple 2-layer neural net (= 1 hidden layer) with two input units and 2 hidden units. We've adjusted the notation a bit to avoid long equations in the nodes by just mentioning the function that is being computed, and the resulting variable name. Thus the * to 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 $b_i^{[1]}$).

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:

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z} \tag{7.41}$$

So let's first compute $\frac{\partial L}{\partial a^{[2]}}$, taking the derivative of Eq. 7.37, repeated here:

$$\begin{aligned} L_{CE}(a^{[2]},y) &= -\left[y \log a^{[2]} + (1-y)\log(1-a^{[2]})\right] \\ \frac{\partial L}{\partial a^{[2]}} &= -\left(\left(y\frac{\partial \log(a^{[2]})}{\partial a^{[2]}}\right) + (1-y)\frac{\partial \log(1-a^{[2]})}{\partial a^{[2]}}\right) \\ &= -\left(\left(y\frac{1}{a^{[2]}}\right) + (1-y)\frac{1}{1-a^{[2]}}(-1)\right) \\ &= -\left(\frac{y}{a^{[2]}} + \frac{y-1}{1-a^{[2]}}\right) \end{aligned} \tag{7.42}$$

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:

$$\begin{aligned} \frac{\partial L}{\partial z} &= \frac{\partial L}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z} \\ &= -\left(\frac{y}{a^{[2]}} + \frac{y-1}{1-a^{[2]}}\right) a^{[2]}(1-a^{[2]}) \\ &= a^{[2]} - y \end{aligned} \tag{7.43}$$

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.

7.6.5 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.

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). Tuning

hyperparameter

of **hyperparameters** is also important. The parameters of a neural network are the weights W and biases 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.

7.7 Training the neural language model

Now that we've seen how to train a generic neural net, let's talk about the architecture for training a neural language model, setting the parameters $\theta = \mathbf{E}, \mathbf{W}, \mathbf{U}, \mathbf{b}$.

freeze

For some tasks, it's ok to **freeze** the embedding layer $\mathbf{E}$ with initial word2vec values. Freezing means we use word2vec or some other pretraining algorithm to compute the initial embedding matrix $\mathbf{E}$, and then hold it constant while we only modify $\mathbf{W}$, $\mathbf{U}$, and $\mathbf{b}$, i.e., we don't update $\mathbf{E}$ during language model training. However, often we'd like to learn the embeddings simultaneously with training the network. This is useful when the task the network is designed for (like sentiment classification, translation, or parsing) places strong constraints on what makes a good representation for words.

Let's see how to train the entire model including $\mathbf{E}$, i.e. to set all the parameters $\theta = \mathbf{E}, \mathbf{W}, \mathbf{U}, \mathbf{b}$. We'll do this via gradient descent (Fig. ??), using error backpropagation on the computation graph to compute the gradient. Training thus not only sets the weights $\mathbf{W}$ and $\mathbf{U}$ of the network, but also as we're predicting upcoming words, we're learning the embeddings $\mathbf{E}$ for each word that best predict upcoming words.

Fig. 7.18 shows the set up for a window size of N=3 context words. The input $\mathbf{x}$

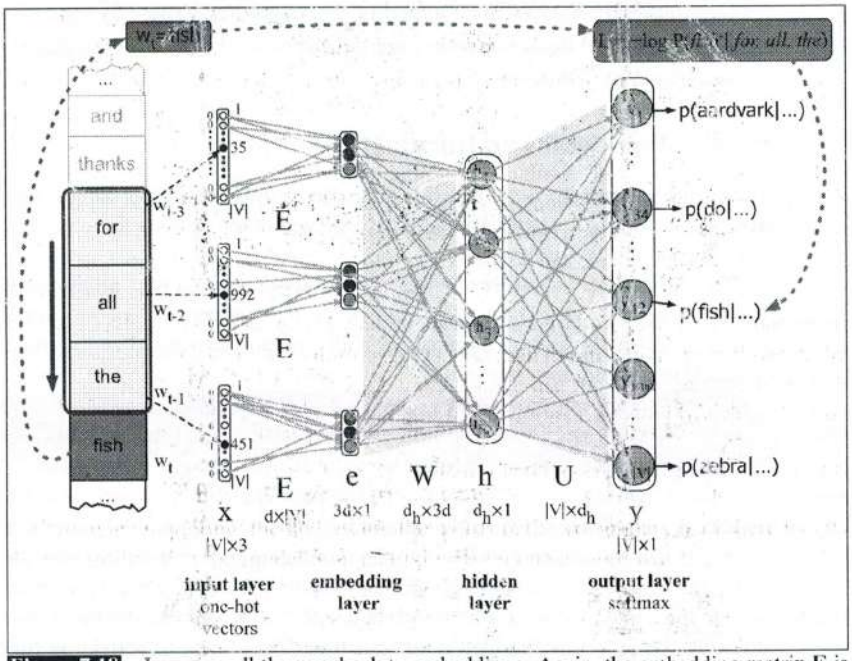


Figure 7.18 Learning all the way back to embeddings. Again, the embedding matrix **E** is shared among the 3 context words.

consists of 3 one-hot vectors, fully connected to the embedding layer via 3 instantiations of the embedding matrix E. We don't want to learn separate weight matrices for mapping each of the 3 previous words to the projection layer. We want one single embedding dictionary E that's shared among these three. That's because over time, many different words will appear as w_{t-2} or w_{t-1}, and we'd like to just represent each word with one vector, whichever context position it appears in. Recall that the embedding weight matrix E has a column for each word, each a column vector of d dimensions, and hence has dimensionality $d \times |V|$.

Generally training proceeds by taking as input a very long text, concatenating all the sentences, starting with random weights, and then iteratively moving through the text predicting each word w_t. At each word w_t, we use the cross-entropy (negative log likelihood) loss. Recall that the general form for this (repeated from Eq. 7.27 is:

$$L_{CE}(\hat{y}, y) = -\log \hat{y}_i \quad \text{(where } i \text{ is the correct class)} \tag{7.44}$$

For language modeling, the classes are the words in the vocabulary, so $\hat{y}_i$ here means the probability that the model assigns to the correct next word w_t:

$$L_{\text{CE}} = -\log p(w_t|w_{t-1}, ..., w_{t-n+1}) \tag{7.45}$$

The parameter update for stochastic gradient descent for this loss from step s to $s+1$ is then:

$$\theta^{s+1} = \theta^s - \eta \frac{\partial \left[-\log p(w_t|w_{t-1}, ..., w_{t-n+1})\right]}{\partial \theta} \tag{7.46}$$

This gradient can be computed in any standard neural network framework which will then backpropagate through $\theta = \mathbf{E}, \mathbf{W}, \mathbf{U}, \mathbf{b}$.

Training the parameters to minimize loss will result both in an algorithm for language modeling (a word predictor) but also a new set of embeddings **E** that can be used as word representations for other tasks.

7.8 Summary

- Neural networks are built out of **neural units**, originally inspired by human neurons but now simply an abstract computational device.
- 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.

Bibliographical and Historical Notes

The origins of neural networks lie in the 1940s **McCulloch-Pitts neuron** (McCulloch and Pitts, 1943), a simplified model of the human 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,

connectionist