Statistics and Data Science for Text Data

Connie Trojan
Supervised by Dr Jonathan Cumming

April 2021

Declaration

This piece of work is a result of my own work except where it forms an assessment based on group project work. In the case of a group project, the work has been prepared in collaboration with other members of the group. Material from the work of others not involved in the project has been acknowledged and quotations and paraphrases suitably indicated.

Contents

Chapter 1

Introduction

This report will focus on evaluating different approaches to **language modelling** and processing text data. We will start by introducing and motivating the concept of language modelling, before describing some key definitions and issues in the field. We will also take a first look at the Sherlock Holmes corpus, which will be used as a running example throughout this report.

Chapter 2 will introduce n**-gram models**, a simple class of probabilistic language model. We will train n-gram models on the Sherlock Holmes corpus and discuss some of their key weaknesses. Chapter 3 will discuss **word embeddings**, a tool that helps language models generalise to unseen data. We will introduce the core building blocks that will allow us to train word embeddings and later neural language models, before describing **skip-gram with negative sampling** and using it to train word embeddings on the Sherlock Holmes corpus. Finally, Chapter 4 will introduce **neural networks** and how they can be used for language modelling. We will discuss how the word embeddings trained in Chapter 3 can be used to improve performance, demonstrating this by training a neural language model on the Sherlock Holmes corpus.

1.1 Language Modelling

A language model aims to assign a probability to a sequence of words, based on how likely they are to be put in that order by a native speaker. It should assign a high probability to sequences that make sense and follow context-appropriate language rules. This is not a simple task - we must model the interactions of thousands of words, and what defines a 'good sentence' depends heavily on the purpose of our model: if we compare the rules and vocabularies we would need to learn for a Shakespeare model to those for a Twitter model, there will be very little overlap despite the fact that both are models for English.

Language models can be used for language generation, since we can generate sentences from the probability distribution defined by the model. Language generation can be used to generate captions for images or summaries of tables of data, and to answer questions asked by humans.

Language models are also useful in any application where we wish to process word sequences:

- A classifier that aims to identify the subject of an article should understand that while the words `white` and `house` are likely to appear individually in a piece about home

improvement, the phrase `White House` is more likely in politics.

- An automatic spellchecker needs to be able to recognise errors, but must also work out from context which possible corrections match the writer's intended meaning.

- In machine translation, we need a language model to understand which of many possible candidate translations is the best fit in the target language. While `I to myself brush the teeth` is technically a correct word-for-word translation of the French sentence `Je me brosse les dents`, a native English speaker is far more likely to say `I'm brushing my teeth`.

- In speech recognition, it is helpful to know that an English speaker is more likely to say the phrase `feel the beat from the tambourine` than the similar-sounding and grammatically correct `feel the meat on the tangerine` - assigning a higher probability to the latter would result in poor automatically generated subtitles to ABBA's 'Dancing Queen'.

1.2 Definitions

Natural language processing (NLP) is an umbrella term for the statistical or computational analysis of human language. The data type used to represent text is the **string**, which is an ordered sequence of characters.

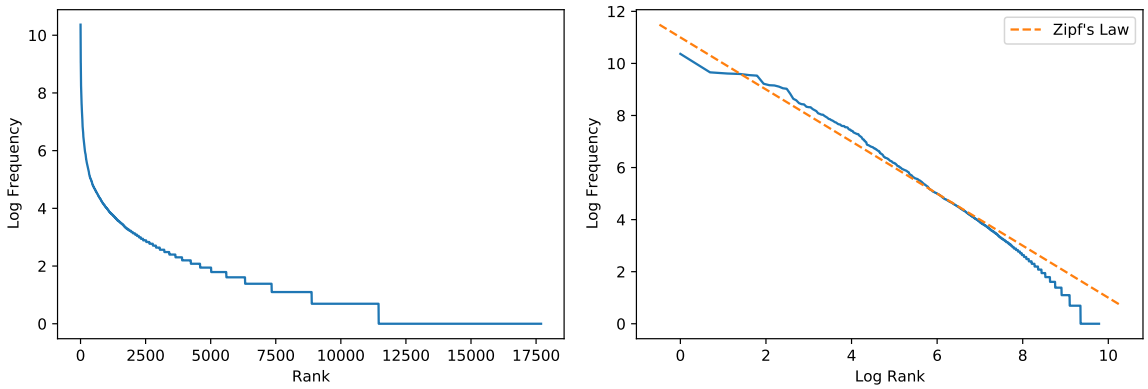
A **token** is a discrete unit of interest that can be extracted from a string - for example, a word or a character. Depending on the task we might consider punctuation like sentence end markers to be tokens, as well as emojis and special characters. The task of splitting a string into tokens is known as **tokenization**. The collection of tokens to be used in a task is called a **vocabulary**. Tokens not in the vocabulary are typically either ignored or replaced with a special 'unknown' token, `<UNK>`.

A unit of text will be referred to as a **document** - examples include a book, an article, a sentence, a review, or a tweet. A collection of documents is called a **corpus**.

The **training dataset** or **train data** is the corpus to be used for training a model. The **validation dataset** is a subset of the training data set aside and used to test a model with a view to comparing and selecting model **hyperparameters**. In contrast to the model parameters, which are treated as variables and are fitted to the training data, hyperparameters control the form the model takes and how it learns from data. For example, the number of model parameters can be a hyperparameter. The number of context words taken into account in an n-gram model (Chapter 2) is another example of a hyperparameter.

The **test dataset** is the corpus to be used for evaluating how well a model generalises to unseen data. It is important for this to be disjoint from the data used to train the model or select hyperparameters, since otherwise we will overestimate the model's true performance.

A **toy dataset** is a simple dataset that can serve as an illustrative example. Toy datasets are typically not expected to be representative of real world datasets in size or complexity.



(a) Sorted log frequencies.

(b) Log frequency against log rank.

Figure 1.1: Word frequencies in Sherlock Holmes.

1.3 Key Issues

One of the main issues in natual language processing is the vocabulary size and the frequency distribution of the words in it: most words appear only rarely, with a small number of words making up the majority of word counts in the corpus.

To give an example, we will take our corpus to be the collection of public domain **Sherlock Holmes** novels and short stories [1–8]. Chapter titles as well as front and back matter were removed, and the text was converted to lowercase. In total the corpus had around 500 000 words, with a vocabulary size of around 17 500.

The sorted log frequencies are plotted in Figure 1.1a, where we can see that the distribution is heavily skewed towards the more common words. The most frequent 1% of words in the vocabulary account for over 60% of the observed words in the corpus, while around half of words appear once or twice. 35% of the words in the vocabulary appear only once - these are known as **hapax legomena**.

In fact, word frequencies in natural languages roughly follow **Zipf's Law** [9]: if we count up the number of times each word in the vocabulary appears in the corpus and sort the list of words by their frequency, then:

$$f \propto \frac{1}{r}\,, \tag{1.1}$$

where we denote a word's frequency by f and its rank in the list by r. Taking the log of both sides, this implies that:

$$\log f = k - \log r\,,$$

for some constant k. If we plot the log ranks against the log frequencies as in Figure 1.1b, we can see that the word counts do roughly follow the predicted distribution with $k = \mathrm{e}^{11}$.

This skewed distribution means that we require a large corpus to make inferences about less common words. Even if we have access to such a corpus, the amount of data can make it difficult to fit models efficiently, since the number of computations required by many traditional statistical techniques scales rapidly with dataset size.

It is also possible for the training data for the task at hand to be limited - for example, if our corpus is the complete works of a dead author, we cannot simply go out and collect more data. It can also be slow and/or expensive to collect and label data for many applications, since we are reliant on humans to generate labels for our data. For example, in machine translation the 'labels' required are sentence translations which must be carried out by a skilled human translator. In these cases the key difficulty is in making the most of smaller datasets.

Chapter 2

N-Gram Models

A simple approach to language modelling would be to collect a large amount of text, and estimate the probability of each sentence observed by its frequency in the corpus:

$$\mathbb{P}\left(\text{Sentence}\right) = \frac{\text{Number of times sentence observed in corpus}}{\text{Total number of sentences in the corpus}}$$

However, the number of possible sentences is enormous - far larger than the number of different sentences in any text corpus we could feasibly gather. This estimate would award a probability of zero to any sentence not seen in the training corpus, giving us no useful information about the relative likelihood of previously unseen sentences.

Since our goal is to train a useful probabilistic language model, we must make some simplifying assumptions. This chapter will focus on n-gram models [10], where we assume that the probability distribution of a word in any given position in a sentence depends only on the $n-1$ previous words in that sentence.

2.1 Bag-of-Words

The simplest n-gram model is the **unigram** [10], where we simply take $n = 1$ and assume that words are decided independently of each other according to a fixed probability distribution. This model is also known as the **bag-of-words** model, since the order of words in a sentence is not taken into account when calculating its probability.

Let $\boldsymbol{w}$ be an instance of $\boldsymbol{W}$, a random variable consisting of a length-m sequence of words drawn from a vocabulary V. Denote by p_i the probability of the i^{th} word in the vocabulary appearing in any given location in $\boldsymbol{W}$, and by n_i the number of times the i^{th} word appears in $\boldsymbol{W}$. Then, under the unigram assumption:

$$\mathbb{P}\left(\boldsymbol{W} = \boldsymbol{w}\right) \overset{independence}{=} \prod_{i=1}^{m} \mathbb{P}\left(W_i = w_i\right) = \prod_{j=1}^{|V|} p_j^{n_j} \ . \tag{2.1}$$

Since the word order is not taken into account in this model, documents can simply be represented by a list of word frequencies. The list $\boldsymbol{X}$ of word counts for a document of length

m can be seen as having a multinomial distribution, $\boldsymbol{X} \sim \text{Multinomial}(m, p_1, ..., p_{|V|})$.

Although the independence assumption made for a unigram model is too strong to model any dependency between words, it is simple to train and it is informative enough to form the basis for a surprisingly effective text classifier.

2.1.1 Naive Bayes Classifier

A common task in natural language processing is classification - for example, automatically deciding if incoming emails are spam, determining whether a review is positive or negative, or assigning a topic to a document. One simple classifier that can be used for text is the **Naive Bayes classifier** [10].

We assume that, given the document class C_k (one of K possible classes), the words in a document are generated by a unigram model, i.e. $\boldsymbol{X}|C_k \sim \text{Multinomial}\big(m, p_1^{(k)}, ..., p_{|V|}^{(k)}\big)$, where $\boldsymbol{X}$ is the list of word counts (with x_i the number of times word i appears) and the $p_i^{(k)}$ describe the word distribution of class C_k. The vocabulary V is considered to be shared between all classes. Then:

$$\mathbb{P}\left(\boldsymbol{X} = \boldsymbol{x}|C_k\right) = \frac{m!}{\prod_{i=1}^{|V|} x_i!} \prod_{i=1}^{|V|} (p_i^{(k)})^{x_i} \ . \tag{2.2}$$

Although word order, sentences, and grammar are ignored by this assumption, it works well enough for classification tasks where we expect the vocabulary to differ between classes, since the presence or absence of certain words is very informative - if we see the word `"titration"` in a Wikipedia article it is reasonable to assume that the topic is chemistry, and observing the word `"excellent"` in a film review typically indicates that the reviewer enjoyed the film.

We first use maximum likelihood estimation to estimate the $\boldsymbol{p}^{(k)}$:

$$\hat{p}_i^{(k)} = \frac{1}{M^{(k)}} \sum_{D_k^{(j)} \in C_k} \text{Count}(w_i, D_k^{(j)}) \ , \tag{2.3}$$

where $M^{(k)}$ is the total number of words in all documents $D_k^{(j)}$ observed in class C_k, and $\text{Count}(w_i, D_k^{(j)})$ is the number of times word w_i is observed in document $D_k^{(j)}$. Then we can use Bayes' theorem to obtain a posterior distribution:

$$\mathbb{P}\left(C_k|\boldsymbol{X} = \boldsymbol{x}\right) = \frac{\mathbb{P}\left(\boldsymbol{X} = \boldsymbol{x}|C_k\right)\mathbb{P}\left(C_k\right)}{\mathbb{P}\left(\boldsymbol{X} = \boldsymbol{x}\right)} = \frac{\mathbb{P}\left(\boldsymbol{X} = \boldsymbol{x}|C_k\right)\mathbb{P}\left(C_k\right)}{\sum_{k=1}^{K} \mathbb{P}\left(\boldsymbol{X} = \boldsymbol{x}|C_k\right)\mathbb{P}\left(C_k\right)}$$

$$= \frac{\frac{m!}{\prod_{i=1}^{|V|} x_i!}\mathbb{P}\left(C_k\right)\prod_{i=1}^{|V|}(p_i^{(k)})^{x_i}}{\sum_{k=1}^{K}\frac{m!}{\prod_{i=1}^{|V|} x_i!}\prod_{i=1}^{|V|}(p_i^{(k)})^{x_i}\mathbb{P}\left(C_k\right)} = \frac{\mathbb{P}\left(C_k\right)\prod_{i=1}^{|V|}(p_i^{(k)})^{x_i}}{\sum_{k=1}^{K}\mathbb{P}\left(C_k\right)\prod_{i=1}^{|V|}(p_i^{(k)})^{x_i}}$$

$$\propto \mathbb{P}\left(C_k\right)\prod_{i=1}^{|V|}(p_i^{(k)})^{x_i} \overset{estimate}{\approx} p^{(k)}\prod_{i=1}^{|V|}(\hat{p}_i^{(k)})^{x_i} \ .$$

The prior distribution $p^{(k)}$ can be uniform, or estimated from the relative frequencies of the classes in the training corpus.

On observing a list of word counts, we predict the document class as the maximum a posteriori estimate:

$$\hat{C}_{\text{MAP}} = \underset{k}{\text{argmax}}\ p^{(k)} \prod_{i=1}^{|V|} (\hat{p}_i^{(k)})^{x_i}\,. \tag{2.4}$$

We will now examine the binary case $k \in \{0, 1\}$ in more detail. It is easy to derive the decision rule for naive Bayes in this case: given a new observation $\boldsymbol{x}^*$, we predict that $y^* = 1$ if $\hat{\text{P}}(y^* = 1|\boldsymbol{x}^*) > \hat{\text{P}}(y^* = 0|\boldsymbol{x}^*)$. Taking the log of both sides, we see that $\hat{C}_{\text{MAP}} = 1$ if:

$$\log \hat{\text{P}}(y^* = 1|\boldsymbol{x}^*) - \log \hat{\text{P}}(y^* = 0|\boldsymbol{x}^*) > 0$$

Since $\hat{\text{P}}(y^* = 1|\boldsymbol{x}^*) = 1 - \log \hat{\text{P}}(y^* = 0|\boldsymbol{x}^*)$, the left hand side is the log odds of $y^* = 1$. Substituting in Eq. 2.4, we find that:

$$\text{logit}(\hat{\text{P}}(y^* = 1|\boldsymbol{x}^*)) = \log p^{(1)} + \sum_{j=1}^{|V|} x_j^* \log \hat{p}_j^{(1)} - \log p^{(0)} - \sum_{j=1}^{|V|} x_j^* \log \hat{p}_j^{(0)}$$

$$= \log \left(\frac{p^{(1)}}{p^{(0)}} \right) + \sum_{j=1}^{|V|} \log \left(\frac{\hat{p}_j^{(1)}}{\hat{p}_j^{(0)}} \right) x_j^*\,. \tag{2.5}$$

This is a linear function of the word counts x_j, so we can inspect the coefficient $\log \left(\frac{\hat{p}_j^{(1)}}{\hat{p}_j^{(0)}} \right)$ to see the impact of observing word w_j on the classifier's decision. For example, if it is positive then each observation of word w_j weights the decision in favour of predicting $y^* = 1$. We will see some examples on real data in the next section.

Naive Bayes is considered to be a **generative** classifier, since it involves building a model of how the data is generated: we assume that for each document, the class is first drawn from a categorical distribution, and the word counts drawn from the corresponding multinomial distribution.

In the next chapter, we will see an example of a **discriminative** classifier: the logistic regression, which is considered to be the discriminative counterpart to naive Bayes [11]. This classifier does not rely on assumptions about the generating distribution of the data, seeking instead to compute $\mathbb{P}(C_k|\boldsymbol{X})$ directly by optimising for the weights in Eq. 2.5 that give the best classification performance. Logistic regression is often more effective than naive Bayes since it does not assume that features are independent.

2.1.2 Naive Bayes for Sentiment Analysis

Sentiment analysis is a classification task in natural language processing where the aim is to determine if a text is positive or negative. We will be looking at the IMDb film review dataset [12], which consists of 50 000 film reviews labelled with 0 or 1 for negative and positive reviews respectively. The dataset is split equally into reviews to be used for training and testing models. The plain text reviews for each dataset were retrieved from TensorFlow Datasets [13].

A naive approach to the task at hand is to simply look at the word counts in each review. Looking at Figure 2.1 it is relatively easy to identify which word cloud corresponds to positive or negative reviews, validating the assumption that we can perform the classification based on word counts alone. Surprisingly, however, the word `good` is just as common in negative reviews as in positive ones - to understand why, it is necessary to take a look at some of the contexts in which it appears:

```
"the acting was good but that cannot save lack of story"
"ruining actor's like Christopher Walken's good name"
```

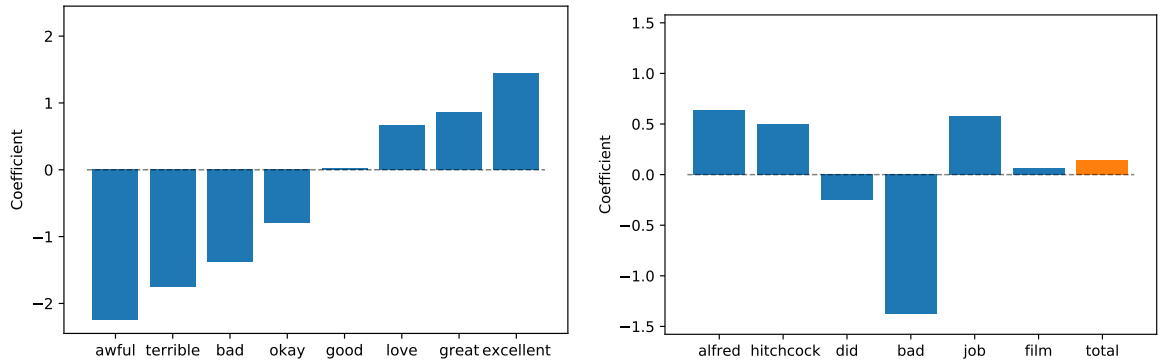
Often, reviewers will complement aspects of the film that they liked even if their overall opinion is negative - since the unigram assumption takes each word out of context, the classifier may struggle with this kind of review.

Some data cleaning was necessary before the model was trained - HTML tags like `<br />` were removed, as well as numbers and special characters. The reviews were also converted to lowercase, so that capitalised words at the start of each sentence were not considered separately from their lowercase versions. Very common words like `and` and `the` were also removed since they are equally common in both classes and hence did not provide much useful information for classification. Such words are known as **stopwords**, and they were removed using the standard stopword list for English provided by the Scikit-learn Python package [14]. Note that the definition of a stopword may vary depending on the application - one could argue that words like `film` and `movie` count as stopwords in this case. They did indeed have to be removed to generate the wordclouds in Figure 2.1 since they would

(a) Positive reviews

(b) Negative reviews

Figure 2.1: Word clouds for the training reviews.



(a) The coefficients for some adjectives.

(b) Decision-making for a sentence.

Figure 2.2

otherwise dominate the figure. However, they were kept in the training data since they have slightly different connotations.

To limit the number of parameters required by the model, the vocabulary was limited to the 5000 most common words in the training dataset after stop word removal. The Scikit-learn implementation of naive Bayes with a uniform prior distribution was used.

The trained model had an accuracy of 85.7% on the training data, and 83.4% on the test data. This is very good, confirming that word counts are an informative summary of the documents for this problem.

Since naive Bayes is a linear classifier, we can use Eq. 2.5 to calculate the impact of individual words on the final decision. Figure 2.2a shows some examples: the stronger the word, the larger the magnitude of its coefficient and its impact on the final decision. For example, the word `good` was not very informative in this application and as such has a coefficient close to zero, while the word `awful` is strongly negative. We can use this representation to see how the classifier makes its decisions for each review. For example, take the sentence `"Alfred Hitchcock did a bad job on this film."`. After preprocessing and stopword removal, this becomes `"alfred hitchcock did bad job film"`. The classifier's weights for each word are shown in Figure 2.2b, along with the calculated final decision that the review is positive, obtained by summing the coefficients for each word.

Note that the weights for `Alfred` and `Hitchcock` are both positive, with similar magnitudes: this is because the classifier has observed that reviews for Alfred Hitchcock's films are likely to be positive. However, because of the assumption that words occur independently of each other, it has not taken into account the fact that the words `Alfred` and `Hitchcock` are likely to occur together: 40% of reviews containing the word `Alfred` also contain `Hitchcock`. This causes the classifier to overestimate the probability that reviews mentioning his full name are positive - if we instead input the sentence `"Hitchcock did a bad job on this film"`, it is correctly classified as negative.

This issue can be avoided by directly optimising for the weights that give the best prediction accuracy without making this independence assumption: we will see in Section 3.2 that this is the case with the logistic regression classifier.

2.2 Higher Order N-Grams

Clearly, the bag-of-words model is not a realistic model for language since in practice words are not independent of each other. For example, in the sentence `"The monkeys really enjoy writing on their typewriters"`, we must look five words into the past to capture the dependence of `"their"` on the plural `"monkeys"`.

To explore this further, the task of computing the joint probability of the full sequence can be broken up into the product of the conditional probabilities of each word given the previous words in the sequence. Using the probability chain rule:

$$\mathbb{P}(\boldsymbol{W} = \boldsymbol{w}) = \mathbb{P}(W_1 = w_1)\,\mathbb{P}(W_2 = w_2|W_1 = w_1)\ldots\mathbb{P}(W_i = w_i|W_{i-1} = w_{i-1}, \ldots, W_1 = w_1).$$

So that we do not have to directly estimate the probability of arbitrarily long sequences of words, we assume that the full history can be approximated by only the last $n-1$ words:

$$\mathbb{P}(W_i = w_i|W_{i-1} = w_{i-1}, \ldots, W_1 = w_1) \approx \mathbb{P}\left(W_i = w_i|W_{i-1} = w_{i-1}, \ldots, W_{i-(n-1)} = w_{i-(n-1)}\right).$$

Under this approximation, the probability of a word sequence becomes:

$$\mathbb{P}(\boldsymbol{W} = \boldsymbol{w}) \approx \prod_{i=1}^{m} \mathbb{P}\left(W_i = w_i|W_{i-1} = w_{i-1}, \ldots, W_{i-(n-1)} = w_{i-(n-1)}\right). \tag{2.6}$$

This is a Markov assumption, since we suppose that the probability distribution of the next word is determined by the $n-1$ words before it. This means that sequences generated by n-gram models are Markov chains of order $n-1$.

Note that this expression always depends on a history of length $n-1$, even for the first word in the sequence. To resolve this we insert $n-1$ extra padding tokens, `<s>`, at the beginning of the sequence. If this is done for each sentence in the dataset, we can model the probability of a word being the first word in a sentence. It will also be useful to explicitly model the probability of a word ending a sentence - for this, we also add a sentence end token, `<\s>`, to the end of each sentence in the dataset.

For example, in a bigram model, the probability of a word w being the first word of a sentence will be $\mathbb{P}(w|$`<s>`$)$, and the probability of it being the last is $\mathbb{P}($`<\s>`$|w)$, so that:

$$\mathbb{P}(\texttt{She sells shells.}) = \mathbb{P}(\texttt{she|<s>})\,\mathbb{P}(\texttt{sells|she})\,\mathbb{P}(\texttt{shells|sells})\,\mathbb{P}(\texttt{<\s>|shells}).$$

2.2.1 Maximum Likelihood Estimation

We can estimate the probability of an n-gram $w_1 w_2 \dots w_n$ by its relative frequency in the training data. This is the **maximum likelihood estimate** (MLE):

$$\hat{\mathrm{P}}_n(w_1 \dots w_n) = \frac{\mathrm{Count}(w_1 \dots w_n)}{N} \,, \tag{2.7}$$

where N is the total number of n-grams observed in the corpus. The conditional probability of observing w_n given the history $w_1 \dots w_{n-1}$ is:

$$\hat{\mathrm{P}}_n(w_n | w_1 \dots w_{n-1}) = \frac{\mathrm{Count}(w_1 \dots w_n)}{\mathrm{Count}(w_1 \dots w_{n-1})} \,. \tag{2.8}$$

The probability of an arbitrarily long sequence of words can then be estimated by substituting $\hat{\mathrm{P}}_n$ into Eq. 2.6. For example, take the toy corpus:

```
"She sells sea shells on the sea shore"
"The shells she sells are sea shells I'm sure"
```

The observed bigram counts for this corpus are shown in Table 2.1, where row i, column j contains the number of times the bigram (w_i, w_j) was observed. Using Eq. 2.8 we estimate, for example, that $\hat{\mathrm{P}}_2(\texttt{shells|sea}) = \frac{2}{3}$, and $\hat{\mathrm{P}}_2(\texttt{shore|sea}) = \frac{1}{3}$.

We can also estimate the probability of an unseen sentence:

$$\begin{aligned}\hat{\mathrm{P}}_2(\texttt{The shells on the sea shore.}) &= \hat{\mathrm{P}}_2(\texttt{the|<s>})\hat{\mathrm{P}}_2(\texttt{shells|the}) \dots \hat{\mathrm{P}}_2(\texttt{<\s>|shore}) \\ &= \frac{1}{2} \times \frac{1}{2} \times \frac{1}{3} \times 1 \times \frac{1}{2} \times \frac{1}{3} \times 1 = \frac{1}{72}\end{aligned}$$

However, this fails for some sentences since unseen bigrams are given a probability of 0:

$$\hat{\mathrm{P}}_2(\texttt{She sells shells.}) = \frac{1}{2} \times 1 \times 0 \times 0 = 0$$

This is an issue with the **sparsity** of our dataset - most of the entries in Table 2.1 are 0.

x	<s>	she	sells	sea	shells	on	the	shore	are	I'm	sure	<\s>
<s>	0	1	0	0	0	0	1	0	0	0	0	0
she	0	0	2	0	0	0	0	0	0	0	0	0
sells	0	0	0	1	0	0	0	0	1	0	0	0
sea	0	0	0	0	2	0	0	1	0	0	0	0
shells	0	1	0	0	0	1	0	0	0	1	0	0
on	0	0	0	0	0	0	1	0	0	0	0	0
the	0	0	0	1	1	0	0	0	0	0	0	0
shore	0	0	0	0	0	0	0	0	0	0	0	1
are	0	0	0	1	0	0	0	0	0	0	0	0
I'm	0	0	0	0	0	0	0	0	0	0	1	0
sure	0	0	0	0	0	0	0	0	0	0	0	1
<\s>	0	0	0	0	0	0	0	0	0	0	0	0

Table 2.1: Bigram counts for the toy corpus.

2.2.2 Data Sparsity

As n is increased, the model clearly becomes a better fit for real language, as in practice dependencies can exist between words almost arbitrarily far apart. However, as the order of the model increases, the total number $|V|^n$ of possible n-grams increases exponentially, and therefore so does the number of parameters required by the model. Here, the model parameters are the probability estimates for each n-gram.

Many n-grams will have 0 observed counts in the training data, and since nonsensical combinations of words in the vocabulary far outnumber sequences that would plausibly be used by a human, most of them will never occur in any corpus. This problem of having many more 0 counts than nonzero is known as **data sparsity**, and can cause problems with fitting n-gram models since any n-gram with a count of 0 is given 0 probability by the MLE.

The probability estimates for high order n-gram models will have higher variance [15], and more training data is required to make good estimates. For this reason, bigram and trigram models are most commonly used in practice, with higher orders used only if there is sufficient training data available.

The following sections outline some ways of dealing with the data sparsity issue and estimating probabilities for n-grams that do not appear in the training data.

2.2.3 Restricting the Vocabulary

In practice, we may need to limit the vocabulary size of the model to restrict the number of parameters required. If the model is to be applied to unseen data, it is also likely that it will come across words that do not appear in the training data, known as **out of vocabulary** (OOV) words.

In some tasks, we will have a list of all possible words that can appear - for example, a machine translation system where candidate translations can only contain words in the translation dictionary. This situation is known as a **closed vocabulary system**. In this case, we can discard words from the training data that cannot appear when the model is in use, replacing them with the unknown word token `<UNK>`, which is then treated as a normal word when training the model.

If we don't already have a fixed vocabulary for the task at hand, we can create such a vocabulary from the training data - by either choosing a fixed vocab size $|V|$ and taking the $|V|$ most common words as the vocabulary, or by fixing a minimum frequency and including only words with counts above this threshold. It is common to remove words occurring only once in the corpus - this will greatly reduce the size of the parameter space, with minimal impact on model quality. Words outside of this new vocabulary can be replaced by `<UNK>` as before and if OOV words are encountered when the model is in use, they can be dealt with in the same way.

Numbers are a potential source of a large number of unique tokens in the text - after all, there are infinitely many of them. The values of the numbers themselves will not contain much information for the purposes of language modelling, so a separate `<NUMBER>` token can be employed to replace numbers, reducing the vocabulary size further.

Notice also that the way in which numbers are used in the text can depend on their length - for example, the ideal model should decide that:

$$\mathbb{P}(\texttt{"November 1965"}) > \mathbb{P}(\texttt{"November 15.3"}),$$
and also:
$$\mathbb{P}(\texttt{"the 1965th of November"}) < \mathbb{P}(\texttt{"the 18th of November"}).$$

Replacing each digit of a number with a symbol (e.g. `"1965"` becomes `"####"`, `"15.3"` becomes `"##.#"`) would still reduce the vocabulary size without discarding much important information. This trick is useful in word models for translation [16], as it provides enough information to make decisions about word choice and order, and the original number can easily be substituted back into the final translation.

2.2.4 Smoothing

A key issue with the maximum likelihood estimates for n-gram probabilities is that a probability of 0 is given to any unseen n-gram. We want our model to make good probability estimates for previously unseen sentences, but sparse data is an issue even for low order n-grams.

A very simple solution is to add one to all counts so that no n-gram is given zero probability. This is known as **add-one** or **Laplace smoothing**, and corresponds to the Bayesian estimate derived using a uniform prior over all n-grams [9].

$$\hat{\mathrm{P}}_{\mathrm{Laplace}}(w_1 \dots w_n) = \frac{\mathrm{Count}(w_1 \dots w_n) + 1}{N + |V|^n} \tag{2.9}$$

$$\hat{\mathrm{P}}_{\mathrm{Laplace}}(w_n | w_1 \dots w_{n-1}) = \frac{\mathrm{Count}(w_1 \dots w_n) + 1}{\mathrm{Count}(w_1 \dots w_{n-1}) + |V|}$$

The addition of $|V|^n$ or $|V|$ in the denominator ensures that the result is still a valid probability distribution, since if we sum over all possible n-grams or continuations we should get 1. In practice, Laplace smoothing does not produce reasonable estimates [17] - it makes quite a large change to the estimated probabilities of the n-grams we did observe, giving far too much probability mass to unseen n-grams. This is again due to data sparsity - for a large vocabulary, plausible n-grams make up a tiny proportion of the set of all possible n-grams.

To resolve this, we simply add a number smaller than one to each count. This is known as **add-α smoothing** (or add-k/add-λ), and our estimates become:

$$\hat{\mathrm{P}}_{\text{add-}\alpha}(w_1 \dots w_n) = \frac{\mathrm{Count}(w_1 \dots w_n) + \alpha}{N + \alpha|V|^n}\,, \tag{2.10}$$

$$\hat{\mathrm{P}}_{\text{add-}\alpha}(w_n | w_1 \dots w_{n-1}) = \frac{\mathrm{Count}(w_1 \dots w_n) + \alpha}{\mathrm{Count}(w_1 \dots w_{n-1}) + \alpha|V|}\,.$$

This can be seen as a linear interpolation between the MLE and a uniform prior:

$$\hat{\mathrm{P}}_{\text{add-}\alpha}(w_1 \dots w_n) = \frac{N}{N+\alpha|V|^n} \frac{\mathrm{Count}(w_1 \dots w_n)}{N} + \left(1 - \frac{N}{N+\alpha|V|^n}\right) \frac{1}{|V|^n}$$

$$= \mu\, \hat{\mathrm{P}}_n(w_1 \dots w_n) + (1-\mu)\, \frac{1}{|V|^n}\,, \text{ where } \mu \in [0,1]. \tag{2.11}$$

A good value for α can be found by cross-validation. A common choice is 0.5, known as the **Jeffreys-Perks law** [9].

2.2.5 Evaluation

While language models can be compared by computing the probability given to a test dataset, this probability is often difficult to interpret since it depends on the size of the dataset used - larger datasets will be given a smaller probability. It will also be also very small - even a short sentence can have a probability less than 10^{-20}, and the probability of a large dataset is often small enough to be indistinguishable from zero to a computer.

For these reasons, the model **Perplexity**, $\mathrm{PP}(\boldsymbol{w})$, on dataset $\boldsymbol{w}$ is often used in practice:

$$\mathrm{PP}(\boldsymbol{w}) = (\hat{\mathrm{P}}(w_1 \dots w_N))^{-\frac{1}{N}} = \sqrt[N]{\frac{1}{\hat{\mathrm{P}}(w_1 \dots w_N)}} \in [1, \infty]\,. \tag{2.12}$$

The smaller the perplexity, the better the model - for a 'perfect' model giving the dataset a probability of 1 the perplexity will be 1, while a model that gives zero probability to the data will have infinite perplexity. The 'worst reasonable case' scenario [15] is $\mathrm{PP}(\boldsymbol{w}) = |V|$, since this is the perplexity of the uniform model $\hat{\mathrm{P}}_{\mathrm{Unif}}$ defined by giving equal probability to each word:

$$(\hat{\mathrm{P}}_{\mathrm{Unif}}(w_1 \dots w_N))^{-\frac{1}{N}} = \sqrt[N]{\prod_{i=1}^{N} \left(\frac{1}{|V|}\right)^{-1}} = \sqrt[N]{|V|^N} = |V|\,. \tag{2.13}$$

For an n-gram model, the perplexity has the form:

$$\mathrm{PP}(\boldsymbol{w}) = \left(\prod_{i=1}^{N} \hat{\mathrm{P}}(w_i | w_{i-n+1} \dots w_{i-1})\right)^{-\frac{1}{N}} \tag{2.14}$$

$$= \exp\left\{-\frac{1}{N} \sum_{i=1}^{N} \log \hat{\mathrm{P}}(w_i | w_{i-n+1} \dots w_{i-1})\right\}\,, \tag{2.15}$$

where N is the total number of words in the training corpus, including sentence end tokens. Since the individual probabilities are very small and N is typically large, the product in Eq. 2.14 will be often be rounded to zero by computers, resulting in $\mathrm{PP}(\boldsymbol{w})$ being undefined. To avoid this, the perplexity is computed in practice using the log probabilities as in Eq. 2.15.

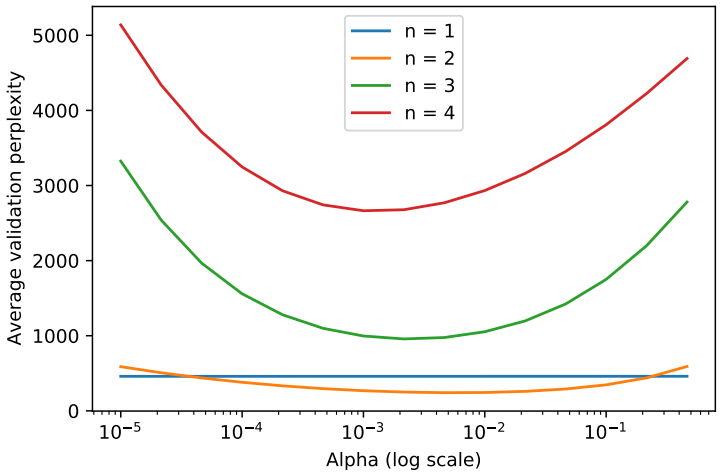


Figure 2.3: Cross-validation for α.

2.2.6 Sherlock Holmes Data

We will now apply these techniques to the Sherlock Holmes corpus introduced in Section 1.3. To simplify the task and reduce the vocabulary, all punctuation was removed, splitting contractions like "I've" and "he's" into two words instead of treating them as distinct words. Capitalisation was also removed and digits of numbers replaced with a # token. The dataset was split into test and train datasets, with a randomly sampled 75% of the sentences used for training. The vocabulary was created from the training dataset, with words mapped to the `<UNK>` token if they appeared only once.

To train the models with add-α smoothing, the best value for α for each n-gram model was found by **5-fold cross validation** [18]: the training data was split into 5 groups of equal size, and each group was in turn used as a validation dataset, with the other 4 used to create the vocabulary and train the model. The perplexity of the resulting model on the validation dataset was then computed. This was repeated for a range of values of α in a multiplicative grid of resolution $10^{-\frac{1}{3}}$, i.e. in the set $\{\alpha = 10^{-i/3} \,|\, i \in \{1, \ldots, 16\}\}$, and the value with the lowest average validation perplexity was chosen. This value was on the boundary of the grid for the unigram model, and on investigation the best value for α in this case was 0. The average validation perplexity for each choice of α and n is plotted in Figure 2.3.

The full training dataset was then used to train the language models. The perplexity on the test and train datasets for the MLE and smoothed n-gram models is reported in Table 2.2. The test perplexity for the MLE models with $n \geq 2$ was infinite, since the test dataset contained unseen n-grams which were given a probability of 0. This was not an issue for the unigram model, since unseen words were mapped to the `<UNK>` token and hence were not given 0 probability. This is likely the reason why the unigram model did not require smoothing, a technique designed to combat the issue of 0 probability n-grams.

The smoothed bigram model achieved the lowest test perplexity - while increasing n further decreased the train perplexity, the resulting model generalised poorly to the test data. This is an example of **overfitting** - the probability estimates for unseen n-grams are not very good, and for large n there was insufficient training data to ensure that the model had seen the n-grams in the test data.

n	α	Train perplexity	Test perplexity
1	0	511	474
2	0	58	-
2	0.005	81	**237**
3	0	8	-
3	0.002	29	942
4	0	**3**	-
4	0.001	14	2737

Table 2.2: Perplexities on the test and train datasets for n-gram models with add-α smoothing.

n	α	Example generated sentence
1	0	Soames the having a there but with of the were.
2	0	What are natural else will understand having his bearing expression as he muttered.
2	0.005	If there was like an authoritative calculation cliffs begged electric robberies room.
3	0	Well though somewhat crippled by occasional attacks of energy that he shall.
3	0.002	I fear fight matter short months persisted arrange groove swung grandeur...
4	0	Can you not tell when a warning is for your own sake.
4	0.001	No thank listened climbed sometimes distracting cast learns passed terrace...

Table 2.3: Examples of sentences generated by the n-gram models. Capitalisation and full stops were added by hand, with ellipses indicating a sentence too long to be displayed.

We can gain some intuition for how the models view language by using them to generate sentences. This is done by starting with $n-1$ padding tokens and sampling words from the n-gram conditional distribution until a sentence end token is generated. Table 2.3 shows some examples of generated sentences for different models. The sentences generated from MLE models appear to get better as n is increased, however on inspection it is clear that they are simply quoting parts of the training data - for example, the sentence `"Can you not tell when a warning is for your own good."` appears in the dataset. This is because many histories are observed only once or twice in the training data and hence only have one or two observed continuations.

This was not a problem with the smoothed models, however the smoothed 3- and 4-gram models typically generated worse sentences than even the unigram model, since words following an unseen history were generated from a uniform distribution. This is why the test perplexities for the smoothed 3- and 4-gram models were actually worse than that of the unigram model. Because of this issue, the smoothed 3- and 4-gram models also consistently underestimated the probability of a sentence end token and their generated sentences were very long.

Smoothing solves the issue of zero probability n-grams, but awards the same probability to each unseen n-gram - the following section describes some more sophisticated approaches that can make judgements about which unseen n-grams are most likely.

2.2.7 Interpolation and Backoff

Interpolation and backoff both aim to use information from lower order n-grams to improve the model, since probability estimates for lower order n-gram are more robust.

Interpolation (or Jelinek-Mercer smoothing) [19] constructs the **interpolated language model**, $\hat{\mathrm{P}}_{\mathrm{li}}^{(n)}$, by fitting several MLE n-gram models P_n of different orders (for example, n=1,2,3) and taking a linear combination of them:

$$\hat{\mathrm{P}}_{\mathrm{li}}^{(3)}(w_3|w_1,w_2) = \lambda_1\,\hat{\mathrm{P}}_1(w_3) + \lambda_2\,\hat{\mathrm{P}}_2(w_3|w_2) + \lambda_3\,\hat{\mathrm{P}}_3(w_3|w_1,w_2)\,, \tag{2.16}$$

$$\text{subject to: } \sum_{i=1}^{3}\lambda_i = 1\,,$$
$$0 \le \lambda_i \le 1\ \forall\, i \in \{1,2,3\}\,.$$

For general n, we can formulate the above recursively as a linear interpolation between the MLE and the order $n-1$ interpolated model:

$$\hat{\mathrm{P}}_{\mathrm{li}}^{(n)}(w_i|w_{i-n+1}...w_{i-1}) = \lambda_n\hat{\mathrm{P}}_n(w_i|w_{i-n+1}...w_{i-1}) + (1-\lambda_n)\hat{\mathrm{P}}_{\mathrm{li}}^{(n-1)}(w_i|w_{i-n+2}...w_{i-1})\,,$$

where $\lambda_i \in [0,1]\ \forall i \in \{0,\ldots,n\}$ and the recursion ends with either the unigram model $\hat{\mathrm{P}}_1$ or the uniform model $\hat{\mathrm{P}}_0$. Defined this way, we can see add-α smoothing as a special case of the interpolated language model where $\lambda_i = 0\ \forall i \notin \{0,n\}$, since it is a linear interpolation between the MLE and the uniform model.

A numerical optimisation method on a held-out corpus can be used to fit the weights λ_i - the n-gram probabilities are estimated using a subset of the training data, and the λ_i are chosen to maximise the likelihood of the remaining held-out data.

Backoff is another way of incorporating information from lower order n-gram models - if the n-gram we observe appears infrequently or not at all in the training data, we can approximate it by backing off to the $(n-1)$-gram. In order to create a valid probability distribution, we must also discount the higher order n-grams to reserve some probability mass for the backoff models.

This method is known as **Katz backoff** or Katz smoothing [20]. We rely on the discounted n-gram probability if the n-gram has been observed more than k times (typically, 0 or 1), else we back off recursively to the $(n-1)$-gram Katz model.

$$\hat{\mathrm{P}}_{\mathrm{BO}}^{(n)}(w_i|w_{i-n+1}...w_{i-1}) = \begin{cases} (1-d_n(w_{(i-n+1):(i-1)}))\,\hat{\mathrm{P}}_n(w_{(i-n+1):i}) & \text{if Count}(w_{(i-n+1):i}) > k \\ \alpha_n(w_{(i-n+1):(i-1)})\,\hat{\mathrm{P}}_{\mathrm{BO}}^{(n-1)}(w_i|w_{i-n+2}...w_{i-1}) & \text{otherwise} \end{cases} \tag{2.17}$$

Here, d represents the discounting function and α the normalising factor that distributes the remaining probability mass over the lower order n-grams. The discounting function depends on the history, $w_{i-n+1}...w_{i-1}$. This is because our confidence in the estimated n-gram probability might depend on how frequently we have observed this history - if the history appeared infrequently in the training data, we may have only seen a small fraction of the possible continuations w_i.

Both interpolation and backoff perform much better than add-α smoothing, which is typically only competitive for very large datasets. Generally, interpolation performs better than backoff for small or sparse datasets since it produces better estimates for rare n-grams [21]. Intuitively, this is because it always uses all of the lower order probability estimates, while backoff uses only one. Because of this, interpolation gives more probability mass to rare n-grams than backoff - this can be an issue for larger datasets, where the higher order n-gram estimates are more robust and do not require as much correction. Indeed, backoff tends to outperform interpolation on large datasets, since it produces better estimates for more frequent n-grams.

2.3 Discussion

A key issue with n-gram models is that they have a very short memory - even with the methods described to make higher order models practically feasible, we can still only model dependence on a few preceding words, with trigrams most often used in practice. This is because the number of parameters required increases exponentially with the order of the model, while the data becomes increasingly sparse.

One way of tackling this sparsity issue is to capitalise on word similarities - instead of looking at each word separately, we can gain insight by also considering other words with similar meanings. In the next chapter we will look at word embeddings, which use this idea to produce lower-dimensional, more informative representations of words.

The current state-of-the-art models for language are neural networks, which can take much longer contexts into account since they use word embeddings to represent prior context. This reduces the number of parameters required and allows them to better generalise to unseen sentences. It is also possible to design neural networks which do not have to rely on the n-gram Markov assumption and can take arbitrarily long contexts into account.

Chapter 3

Word Embeddings

As we have seen, large vocabularies and their frequency distributions can cause problems with language models, due to data sparsity and the difficulty of handling rare words. It would be useful to build a model that can take advantage of word similarity and infer information about a particular word from what it knows about words with similar meanings.

For example, if we are trying to predict the next word in the sentence:

`For dinner, we will be eating ...` ,

it would be useful for our model to understand that words like `spaghetti` and `soup` are equally likely to follow since they are both food items, even if one of them did not appear in this context in the training data. To do this, we need to consider how words are represented in the model - in an n-gram model, for example, the vocabulary is seen as an unordered list of words, with each word assigned to an index in the vocabulary. This is a simple way of storing information about words, but the representation is not meaningful - each word is treated as completely distinct from all other words.

A more meaningful way of representing words is via **word embeddings** [22]: we map words to vectors in $\mathbb{R}^d$, designing the map so that words with similar meanings are mapped to word vectors that are close together. These vectors will be low dimensional compared to the vocabulary size and will also be **dense**. Typically, the **cosine similarity** metric is used to determine vector similarity: this is simply the cosine of the angle between the vectors, computed as:

$$\text{cosine}(\boldsymbol{x}, \boldsymbol{y}) = \frac{\boldsymbol{x} \cdot \boldsymbol{y}}{|\boldsymbol{x}||\boldsymbol{y}|} \,. \tag{3.1}$$

Word vectors typically use the **distributional hypothesis** [10] to define word similarity, making the assumption that words with similar meanings appear in similar contexts. Word embeddings can then be obtained by training a classifier to make predictions related to context words, such as predicting a hidden word from its context in a sentence in the training data. We expect coordinates of the resulting vectors to reflect different aspects of word meaning: for example, we might expect there to be coordinates corresponding to verb tense or noun gender. In practice, these concepts might not be represented by a single coordinate, but by directions in the vector space [23]. For example, it is often possible to find some 'past tense' vector, so that we can map present tense verbs to their past tense versions by translation, resulting in vector equations like: `eat` + `<PAST TENSE>` = `ate`.

3.1 Word2Vec

It is not immediately obvious how to train an embedding map - how would a candidate map's accuracy be evaluated for each word without already having access to a thesaurus with exhaustive entries for every word? It would be ideal to learn embeddings in an unsupervised or self-supervised manner. One way of doing this is to train a model for some classification task that is easy to evaluate and that benefits from embeddings with the desired properties. If an embedding step is incorporated in the model and the vectors are treated as trainable parameters, we can obtain the embeddings when we train the model.

The **Word2Vec** family [24] contains two such model architectures: **continuous bag-of-words** (CBOW) and **skip-gram**. The CBOW model takes a window of context words as an input and attempts to predict the hidden target word, while Skip-gram predicts context words given the target word. For example, take the sentence:

`She sells sea shells on the sea shore.`

In CBOW, given the context window `"She <?> sea shells"`, the classifier must predict the probability of observing words in place of `<?>` based on the embeddings for `she`, `sea`, and `shells`. The order of the context words is not taken into account and the average of the embedding vectors is input to a classifier. In this case, the model should give `sells` a high probability and we would expect it to consider similar words like `buys` to be likely as well.

In skip-gram, however, we input the embedding of the word `sells` to a classifier and predict the probabilities of observing possible context words - here, `she` and `shells` are examples of true context words, but other words like `he` and `apples` might also be given a high probability since they are also likely to appear near `sells`.

Both CBOW and skip-gram can be seen as classification problems, where the set of possible 'classes' is the vocabulary V. Generally, skip-gram performs better than CBOW for rare words or small datasets, but takes longer to train. We will focus on the skip-gram model and how it can be simplified to **skip-gram with negative sampling** [25] to be trained more efficiently. To do this, we will first have to introduce the **logistic regression**, a classifier that models the probability that its input belongs to a particular class.

We will start by examining the **binary logistic regression** in detail, since it is also a key building block of the neural network models we will encounter in Chapter 4. We will introduce the **stochastic gradient descent** (SGD) algorithm in the context of fitting this model - SGD is an efficient method for fitting models to large datasets, and will be used to train the skip-gram embeddings and the neural language models we will encounter later.

3.2 Logistic Regression

Logistic regression [18] is a classifier that models the probability that some observation $\boldsymbol{x}^{(i)} \in \mathbb{R}^d$ belongs to a class $k \in K$. It is the **discriminative** counterpart to naive Bayes (Section 2.1.1), so called because it models the class probability directly instead of making assumptions about the generating distribution of the data. In this section, we will focus on the **binary** case, where $K = \{0, 1\}$. This was the case in the sentiment analysis task we encountered in Section 2.1.2, where each $\boldsymbol{x}^{(i)}$ was a list of word counts, and in Section 3.4.1 we will see how logistic regression compares to naive Bayes on this task.

To model the probability that some observation $\boldsymbol{x}^{(i)}$ belongs to class 1, a linear expression $b + \boldsymbol{w}^T\boldsymbol{x}^{(i)}$ is passed into the **sigmoid** function, σ, also known as the **logistic function**:

$$\sigma(z) = \frac{1}{1 + e^{-z}}\,. \tag{3.2}$$

The graph of σ is shown in Figure 3.1. It takes inputs in $(-\infty, \infty)$ and outputs values in $(0, 1)$, allowing us to model the probabilities of an observation $\boldsymbol{x}^{(i)} \in \mathbb{R}^d$ belonging to class 0 or 1 as follows:

$$\mathbb{P}\left(y^{(i)} = 1 \,|\, \boldsymbol{x}^{(i)}\right) = \sigma(b + \boldsymbol{w}^T\boldsymbol{x}^{(i)}) = \frac{1}{1 + e^{-\,(b + \boldsymbol{w}^T\boldsymbol{x}^{(i)})}}\,, \tag{3.3}$$

$$\mathbb{P}\left(y^{(i)} = 0 \,|\, \boldsymbol{x}^{(i)}\right) = 1 - \sigma(b + \boldsymbol{w}^T\boldsymbol{x}^{(i)}) = \frac{e^{-\,(b + \boldsymbol{w}^T\boldsymbol{x}^{(i)})}}{1 + e^{-\,(b + \boldsymbol{w}^T\boldsymbol{x}^{(i)})}}\,.$$

Here, $b \in \mathbb{R}$ is usually called the **bias** term, and $\boldsymbol{w} = (w_1, \ldots, w_d)^T \in \mathbb{R}^d$ is a list of **weights** for each element $x_j^{(i)}$ of the input. We will refer to the estimated probability $\mathbb{P}\left(y^{(i)} = 1 \,|\, \boldsymbol{x}^{(i)}\right)$ as $\hat{p}_i$ for simplicity. Generally speaking, we will predict a label of 1 for observation $\boldsymbol{x}^{(i)}$ if $\hat{p}_i > 0.5$, although a different threshold can be used for more or less conservative predictions.

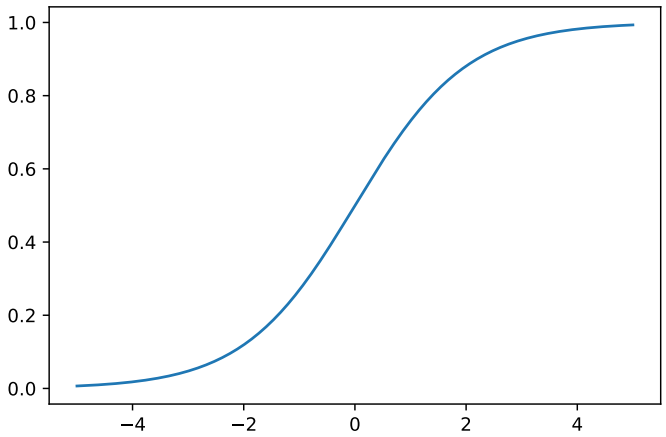


Figure 3.1: The sigmoid function.

Eq. 3.3 can be rearranged as follows to find that, like naive Bayes, the logistic regression model produces log-odds that are linear in $\boldsymbol{x}$ [26]:

$$\frac{\hat{p_i}}{1-\hat{p_i}} = e^{\,b+\boldsymbol{w}^T\boldsymbol{x}^{(i)}} \implies \log\left(\frac{\hat{p_i}}{1-\hat{p_i}}\right) = b + \boldsymbol{w}^T\boldsymbol{x}^{(i)}\,.$$

Logistic regression models are typically fitted using maximum likelihood estimation. Since there are two discrete outcomes, each observation has a Bernoulli distribution:

$$\ell(y^{(i)}, b, \boldsymbol{w}) = \mathbb{P}\left(y^{(i)}|\hat{p_i}\right) = \hat{p_i}^{y^{(i)}}(1-\hat{p_i})^{1-y^{(i)}} = \begin{cases} \hat{p_i} & \text{if} \quad y^{(i)} = 1 \\ 1-\hat{p_i} & \text{if} \quad y^{(i)} = 0 \end{cases}\,.$$

Assuming observations are independent, the log likelihood for N observations is:

$$L(b, \boldsymbol{w}) = \sum_{i=1}^{N} L(y^{(i)}, b, \boldsymbol{w}) = \sum_{i=1}^{N} y^{(i)} \log \hat{p_i} + (1-y^{(i)}) \log(1-\hat{p_i})\,. \tag{3.4}$$

Note that since the $\hat{p_i}$ are functions of the $\boldsymbol{x}^{(i)}$, the success probability is in general not constant, so fitting the model is more complicated than simply fitting a binomial distribution to the data.

As we will see in Section 3.3.1, the log likelihood function is concave, so any solution to the score equations will be its unique maximum. This can be found using the Newton-Rhapson algorithm [18]. However, this requires calculating both the gradient and Hessian matrix of the log likelihood at each iteration, which involves a calculation over every observation in the dataset. Computing and inverting the Hessian is also particularly costly when the data is high-dimensional, since its size scales quadratically with the number of features. This is problematic in natural language processing, where datasets are usually both very large and very high-dimensional.

In the next section we will look at stochastic gradient descent, an optimisation method which is computationally efficient for large datasets since it only requires an estimate of the gradient at each iteration.

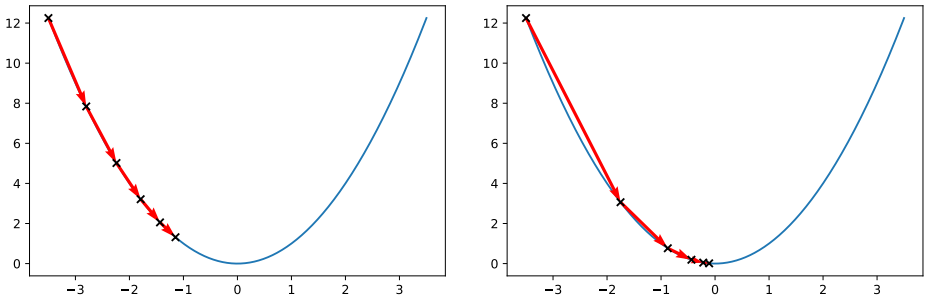
3.3 Stochastic Gradient Descent

Gradient descent [27] aims to find the parameters $\theta \in \mathbb{R}^d$ that minimise an **objective function** $L(\theta)$ by repeatedly updating θ in the opposite direction to the gradient of L. The intuition behind this is that if we visualise the loss function as a multidimensional surface, its lowest point can be found by taking successive small steps 'downhill'. Since $\nabla L(\theta)$ is the direction of steepest increase of L at our current location in $\mathbb{R}^d$, we can decrease L by taking a step in the opposite direction. Figure 3.2 gives an example of how this looks in the one-dimensional case $\theta \in \mathbb{R}$. Note the effect of changing the learning rate - if it is too small then convergence will be very slow, if too large then we will overshoot the minimum at $\theta = 0$, and in this case the algorithm will diverge if $\eta > 1$.

Algorithm 1: Gradient Descent

Require: learning rate η;
Initialise θ_0;
for $t = 1, 2, \ldots$ **do**
  $\theta_t \leftarrow \theta_{t-1} - \eta \nabla L(\theta_t)$;
end

This works very well when the function is convex since there are no suboptimal local minima to get stuck in, so with an appropriate learning rate gradient descent is guaranteed to find the global minimum (if it exists). In general, as long as the function's derivative is well-behaved and the learning rate is small enough, gradient descent will always converge to a local minimum [28].



(a) $\eta = 0.1$ (b) $\eta = 0.25$

(c) $\eta = 0.75$ (d) $\eta = 1.05$

Figure 3.2: Using gradient descent with 5 iterations to find the minimum of $y = x^2$.

Since only the first derivative of L is required, the computational cost per iteration is lower than in a second order method like Newton-Rhapson, although more iterations may be required for convergence. However, when fitting a statistical model, computing this gradient requires a calculation over the whole dataset, which can be very slow for large datasets. A more efficient method is **stochastic gradient descent** (SGD) [29], which takes a random sample of the dataset at each iteration to estimate the gradient.

We will assume that the objective function takes the form:

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} L(\theta; x^{(i)}, y^{(i)}) \,. \tag{3.5}$$

L is typically called a **loss function**, and $L(\theta; x^{(i)}, y^{(i)})$ is a measure of how far away the model's prediction is from the true label $y^{(i)}$ for observation i. Minimising an objective function of the form in Eq. 3.5 is equivalent to minimising the expected loss of a sample drawn at random from our dataset. Crucially, this should also approximately minimise the expected loss of an unseen sample drawn from the same distribution as our dataset [30].

Note that the gradient of $L(\theta)$ has the form:

$$\nabla L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta L(\theta; x^{(i)}, y^{(i)}) \,,$$

so $\nabla L(\theta)$ is the mean of the gradients of the loss function for each observation in the dataset. This means that we can cheaply compute an unbiased estimate of the full gradient by taking a random sample b of size $m < N$ from the data and calculating the sample mean of the $\nabla_\theta L(\theta; x, y)$:

$$\nabla L(\theta) \approx \frac{1}{m} \sum_{x,y \in b} \nabla_\theta L(\theta; x, y) \,.$$

Stochastic gradient descent computes this estimate instead of the true gradient to update θ at each iteration.

Algorithm 2: Stochastic Gradient Descent

Require: sample size $m < N$, step size sequence η_t;
Initialise θ_0;
for $t = 1, 2, \ldots$ **do**
  $b_t \leftarrow$ random sample from the dataset of size m;
  $g_t \leftarrow \frac{1}{m} \sum_{x,y \in b_t} \nabla_\theta L(\theta_{t-1}; x, y)$;
  $\theta_t \leftarrow \theta_{t-1} - \eta_t g_t$;
end

Note that this defines a random walk in $\mathbb{R}^d$. Subject to some regularity conditions on the loss function, SGD will converge almost surely [31] to a local minimum for step size sequences satisfying:

$$\sum_{t=1}^{\infty} \eta_t \, = \, \infty \,, \qquad \sum_{t=1}^{\infty} \eta_t^2 \, < \, \infty \,. \tag{3.6}$$

The intuition behind these requirements is that the first is necessary to ensure we reach the optimum no matter how far away the initial parameter values are, while the second is required for convergence since it implies $\lim_{t \to \infty} \eta_t = 0$.

In practice, it can be difficult to find a step size schedule satisfying Eq. 3.6 for which convergence is fast. It is convenient to instead use a small fixed step size η. Since the gradient estimate is noisy the walk is no longer guaranteed to converge to an exact local minimum, but will spend an increasing amount of time in regions where L has a small gradient [30].

We typically generate the samples in SGD by taking a random partition of the data and using each **minibatch** in the partition in turn to update θ. The process is then restarted with another random partition.

Algorithm 3: Minibatch Stochastic Gradient Descent

Require: minibatch size $m < N$, step size η, number of training epochs E;
Initialise θ_0;
$t \leftarrow 1$;
for *epoch* $e \in \{1, 2, \ldots, E\}$ **do**
  $\mathcal{B} \leftarrow$ random partition of the data into minibatches of size m;
  for *minibatch* $b \in \mathcal{B}$ **do**
    $g_t \leftarrow \frac{1}{m} \sum_{x,y \in b} \nabla_\theta L(\theta_{t-1}; x, y)$;
    $\theta_t \leftarrow \theta_{t-1} - \eta g_t$;
    $t \leftarrow t + 1$;
  end
end

The time taken for convergence is typically measured in the number of these **training epochs** (or passes through the dataset) used rather than the number of updates, since the number of calculations required for each epoch is roughly the same for any minibatch size.

Information from previous steps can be used to speed up convergence, for example by adding a **momentum** term $-\gamma\,(\boldsymbol{\theta}_{t-1} - \boldsymbol{\theta}_{t-2}),\ \gamma \in (0, 1)$ at each iteration.

3.3.1 Fitting a Logistic Regression with SGD

For notational convenience in the calculations below, we will concatenate b and $\boldsymbol{w}$ into a single column vector, $\boldsymbol{\beta} = (b, \boldsymbol{w}) \in \mathbb{R}^{d+1}$ and use X to refer to the design matrix with rows $X_i = (1, (\boldsymbol{x}^{(i)})^T) \in \mathbb{R}^{d+1}$, so that $b + \boldsymbol{w}^T \boldsymbol{x}^{(i)} = X_i\,\boldsymbol{\beta}$, and $X_{ij} = (1, (\boldsymbol{x}^{(i)})^T)_j$.

To fit a logistic regression, we will use the negative log likelihood or **log loss**. This is known more generally as the **cross-entropy loss** L_{CE} , since it is also the formula for the cross-entropy between the true probability distribution and our estimation $\hat{p}_i$ [10].

$$L_{CE}(\boldsymbol{\beta}; x^{(i)}, y^{(i)}) = -L(y^{(i)}, \boldsymbol{\beta}) = -y^{(i)} \log(\hat{p}_i) - (1 - y^{(i)}) \log(1 - \hat{p}_i)\,. \tag{3.7}$$

We will define the objective function as the average loss per sample:

$$L_{CE}(\boldsymbol{\beta}) = \frac{1}{N} \sum_{i=1}^{N} L_{CE}(\boldsymbol{\beta}; x^{(i)}, y^{(i)}) = -\frac{1}{N} L(\boldsymbol{\beta})\,. \tag{3.8}$$

Since we have simply multiplied by the constant $\frac{1}{N}$ and changed the sign to turn the maximisation problem into a minimisation problem, using SGD will be equivalent to maximising the likelihood of the dataset.

The formula for the log likelihood of each observation can be rearranged as follows:

$$
\begin{aligned}
L(y_i, \boldsymbol{\beta}) &\overset{3.4}{=} y_i \log(\hat{p}_i) + (1 - y_i) \log(1 - \hat{p}_i) \\
&= y_i \log \frac{1}{1 + e^{-X_i \boldsymbol{\beta}}} + (1 - y_i) \log \frac{e^{-X_i \boldsymbol{\beta}}}{1 + e^{-X_i \boldsymbol{\beta}}} \\
&= -y_i \log(1 + e^{-X_i \boldsymbol{\beta}}) + (1 - y_i) \left(\log(e^{-X_i \boldsymbol{\beta}}) - \log(1 + e^{-X_i \boldsymbol{\beta}}) \right) \\
&= (1 - y_i) \log(e^{-X_i \boldsymbol{\beta}}) - \log(1 + e^{-X_i \boldsymbol{\beta}}) \\
&= -(1 - y_i) X_i \boldsymbol{\beta} - \log(1 + e^{-X_i \boldsymbol{\beta}}) \,.
\end{aligned}
$$

Taking the partial derivative of $-L(y_i)$ with respect to β_j ,

$$
\begin{aligned}
-\frac{\partial}{\partial \beta_j} L(y_i) &= X_{ij} \, (1 - y_i) - X_{ij} \, \frac{e^{-X_i \boldsymbol{\beta}}}{1 + e^{-X_i \boldsymbol{\beta}}} = X_{ij} \, (1 - y_i - (1 - \hat{p}_i)) \\
&= X_{ij} \, (\hat{p}_i - y_i) \,. \qquad (3.9)
\end{aligned}
$$

Hence the gradient of $L_{CE}(\boldsymbol{\beta})$ is:

$$
\nabla L_{CE}(\boldsymbol{\beta}) = \frac{1}{N} \sum_{i=1}^{N} (\hat{p}_i - y_i) \, X_i^T \,. \qquad (3.10)
$$

To prove that L_{CE} is a convex function, it is enough to show that the Hessian matrix $\nabla^2 L_{CE}(\boldsymbol{\beta})$ is positive semidefinite. To calculate the Hessian, we must partially differentiate once more:

$$
\begin{aligned}
-\frac{\partial^2}{\partial \beta_j \partial \beta_k} L(y_i) &\overset{3.9}{=} \frac{\partial}{\partial \beta_k} X_{ij} \left(\frac{1}{1 + e^{X_i \boldsymbol{\beta}}} - y_i \right) = X_{ij} \, X_{ik} \, \frac{e^{X_i \boldsymbol{\beta}}}{(1 + e^{X_i \boldsymbol{\beta}})^2} \\
&= X_{ij} \, X_{ik} \, \frac{1}{1 + e^{X_i \boldsymbol{\beta}}} \, \frac{e^{X_i \boldsymbol{\beta}}}{1 + e^{X_i \boldsymbol{\beta}}} = X_{ij} \, X_{ik} \, \hat{p}_i \, (1 - \hat{p}_i) \\
&= \hat{p}_i \, (1 - \hat{p}_i) \, \left[\, X_i X_i^T \, \right]_{jk} \,.
\end{aligned}
$$

Hence, the Hessian matrix is:

$$
\nabla^2 L_{CE}(\boldsymbol{\beta}) = \frac{1}{N} \sum_{i=1}^{N} \hat{p}_i \, (1 - \hat{p}_i) \, X_i^T X_i \,. \qquad (3.11)
$$

Note that $\forall \boldsymbol{\beta} \in \mathbb{R}^{d+1}$, $0 \leq \hat{p}_i \leq 1$, so $0 \leq \hat{p}_i\,(1 - \hat{p}_i)$. Also, since $\forall\, \boldsymbol{a} \in \mathbb{R}^{d+1}$ we have $\boldsymbol{a}^T X_i^T X_i \boldsymbol{a} = (\boldsymbol{a}^T X_i^T)(\boldsymbol{a}^T X_i^T)^T = (\boldsymbol{a}^T X_i^T)^2 \geq 0$, each matrix $X_i^T X_i$ is positive semidefinite. Hence $\nabla^2 L_{CE}(\boldsymbol{\beta})$ is also positive semidefinite $\forall \boldsymbol{\beta} \in \mathbb{R}^{d+1}$ (since by Eq. 3.11 it is a sum of positive semidefinite matrices), and the loss function is convex as claimed. Note that this also implies that the log likelihood is concave, since $L_{CE}(\boldsymbol{\beta}) = -\frac{1}{N} L(\boldsymbol{\beta})$. This means that the log likelihood can have at most one stationary point and logistic regression will have a unique MLE for $\boldsymbol{\beta}$ (as long as the MLE exists), as claimed.

3.3.2 Toy Example

A toy dataset was generated by sampling 1000 $x^{(i)}$ from Uniform(0,1). The $y^{(i)}$ were then drawn from a Bernoulli distribution with parameter $\sigma(1 + x^{(i)})$, so that the 'true' logistic regression parameter is $\boldsymbol{\beta} = (1, 1)^T$. Figure 3.3 shows a scatter plot of this dataset alongside the true underlying logistic model.

Since the parameter space is two-dimensional, we can easily visualise the loss function with a contour plot, as seen in Figure 3.4 where the path taken by SGD is also plotted. Note the effect of changing the minibatch size m - with smaller samples the gradient estimates have a higher variance, but many more iterations can be done in the same amount of time. This high variance can be an advantage when the loss function is not convex - the more erratic behaviour of the walk can help it to escape suboptimal local minima and saddle points [32]. In practice, good values for m are typically in the range 32-256 - large minibatch sizes are practical if parallel computing is available, since the calculations for each member of the minibatch can be carried out independently of each other. Having m be a power of two can improve runtime [33] since this allows the minibatch to fit neatly into computer memory.

In this example, the loss function's slope at the starting point is a lot steeper in the β_2 direction, which is why SGD moves sharply upwards to start with, before slowly moving across to the best value for β_1. This is because β_2 is the coefficient for x and β_1 is the bias term - starting from $\beta_1 = \beta_2 = 0$, changing β_2 has a far bigger impact on the classifier's performance than changing β_1, since if $\beta_2 = 0$ the classifier's output is a constant.

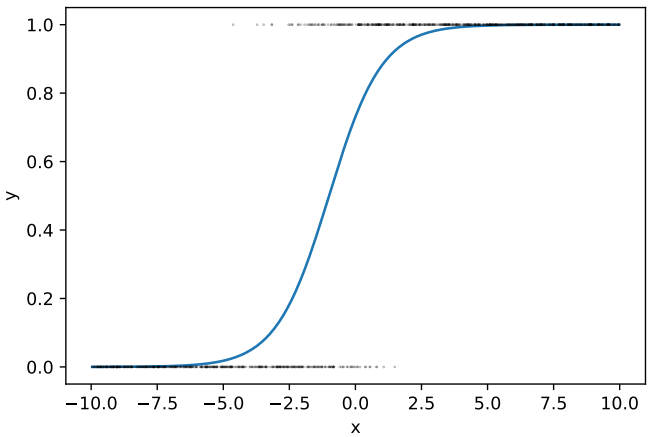
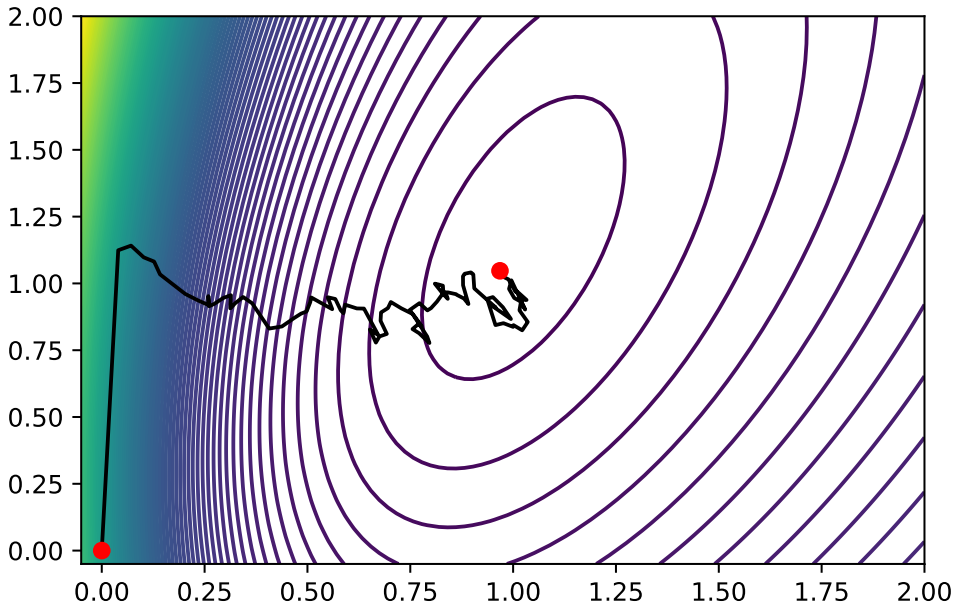
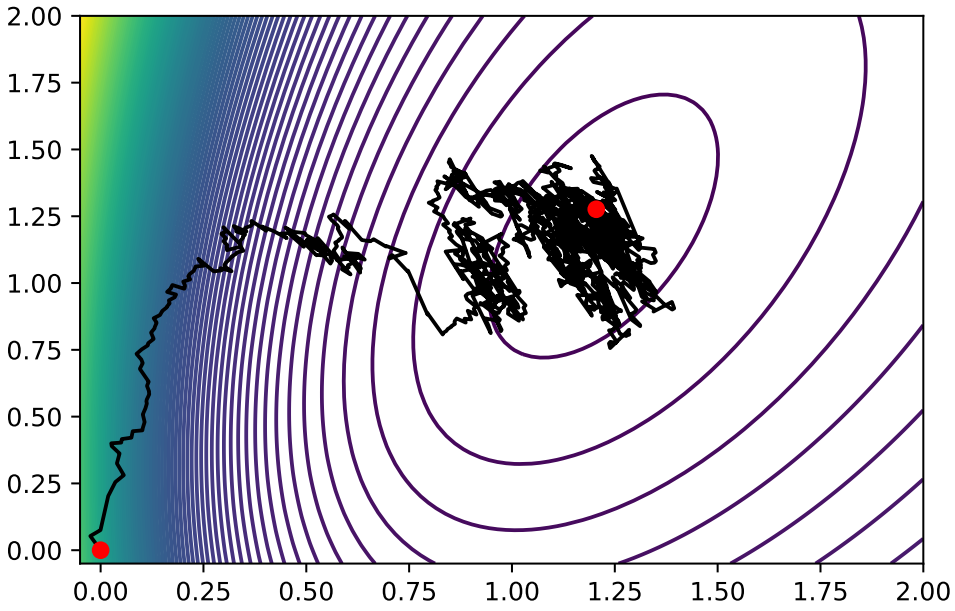


Figure 3.3: Scatter plot of simulated data, with $\sigma(1 + x^{(i)})$ plotted in blue.



(a) $m = 50$



(b) $m = 1$

Figure 3.4: Contour plot of the log loss for the toy example, with the path taken by SGD over 5 training epochs plotted in black. Start and endpoints marked in red.

3.4 Logistic Regression for Sentiment Analysis

We will revisit the sentiment analysis task seen in Section 2.1.2, training a logistic regression classifier to distinguish between positive and negative film reviews given a list of word counts for each review. The dataset was preprocessed in the same way as before, so that the logistic regression was trained and tested on the same data as the naive Bayes classifier.

The model was fitted with SGD initialised at $\boldsymbol{\beta} = \mathbf{0}$, using a minibatch size of 50 (0.27% of the training data). The learning rate was chosen by randomly splitting the training data into new training and validation datasets, with 25% of the original training data used for validation. The learning rate with the lowest validation loss after 50 epochs of training was found by a **grid search** [33] on a multiplicative grid of resolution $10^{-\frac{1}{3}}$: values for the learning rate η in the set $\{\eta = 10^{-i/3} \,|\, i \in \{0, \dots, 10\}\}$ were tested. The best learning rate $\eta = 0.02$ was in the middle of this grid.

This learning rate was then used to train a model for 50 epochs on the full training dataset. Figure 3.5 shows how the loss and accuracy on the test and train datasets changes during training. The loss and accuracy improved rapidly to start with, before levelling off as the MLE was approached. Logistic regression achieved a better accuracy than naive Bayes (Section 2.1.2) on both the test and train datasets - training accuracy was 91.6% (naive Bayes: 85.7%), and test accuracy converged to 87.5% (naive Bayes: 83.3%). The next section will discuss some reasons for this difference in performance. The fitted coefficients for some words are shown in Figure 3.6c - we can see that the model has correctly determined the strength of their positive or negative connotations.

Note that the loss on the training dataset did not converge in the 50 epochs, so we did not reach the MLE of $\boldsymbol{\beta}$ for the training data. If we continue training for longer, the loss on the test dataset actually starts increasing - this is an indicator that the model is starting to overfit to the training data. Fixing the number of training epochs and using a validation dataset to choose the learning rate helped prevent overfitting by restricting how far SGD could move from the starting point. Since the starting point was at the origin, this indicates that the true 'best' value for $\boldsymbol{\beta}$ is closer to the origin than the MLE for the training data. This is the idea behind **regularised logistic regression** [10], which prevents overfitting by penalising the norm of $\boldsymbol{\beta}$ in the loss function. For example, in L_2 regularised logistic regression the loss function is:

$$\frac{1}{N}\sum_{i=1}^{N} L_{CE}(\boldsymbol{\beta}; x^{(i)}, y^{(i)}) + \gamma ||\boldsymbol{\beta}||_2^2 \,, \tag{3.12}$$

where $||\boldsymbol{\beta}||_2^2$ is the squared Euclidean norm of $\boldsymbol{\beta}$, $||\boldsymbol{\beta}||_2^2 = \sum_j \beta_j^2$, and $\gamma \in \mathbb{R}_{>0}$ is fixed in advance. Since training aims to minimise this loss function, this penalty restricts how far the fitted solution will be from the origin. It has a Bayesian interpretation as a Gaussian prior with mean 0 on the β_j.

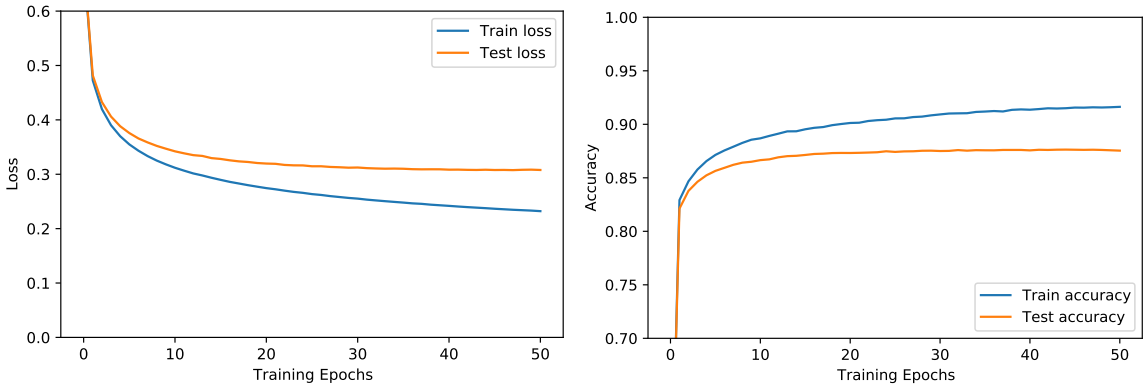
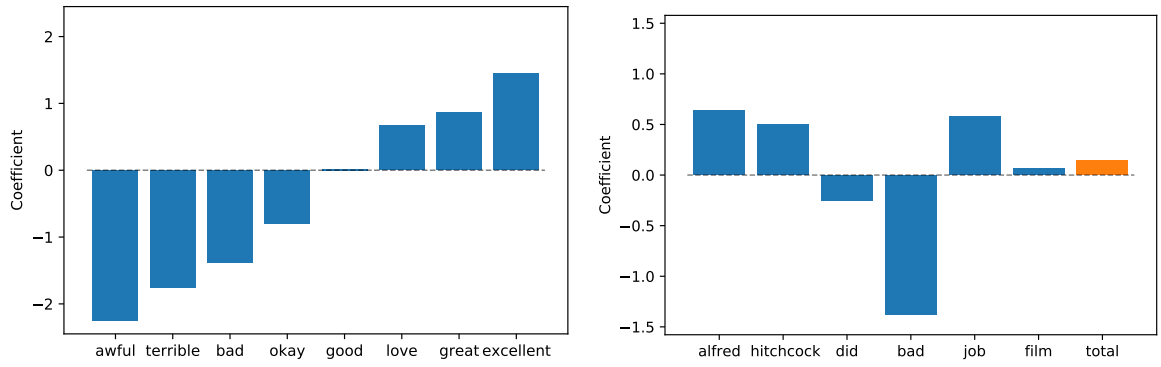
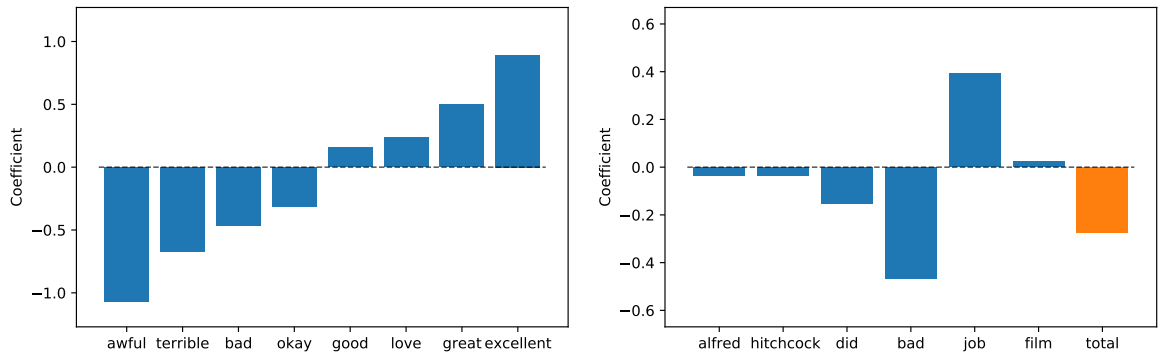


Figure 3.5: Loss and accuracy of logistic regression on the IMDb dataset.



(a) Naive Bayes coefficients.

(b) Naive Bayes decision-making.



(c) Logistic regression coefficients.

(d) Logistic regression decision-making.

Figure 3.6: Comparing logistic regression with naive Bayes.

3.4.1 Comparing Logistic Regression to Naive Bayes

We will now compare the fitted logistic regression model with the naive Bayes classifier from Section 2.1.2. Recall that both classifiers produce log odds that are linear in $\boldsymbol{x}$, so that for some observation $\boldsymbol{x}^* \in \mathbb{R}^d$ the predicted log odds of $y^* = 1$ are:

$$\text{naive Bayes: } \log\left(\frac{p^{(1)}}{p^{(0)}}\right) + \sum_{j=1}^{d} \log\left(\frac{\hat{p}_j^{(1)}}{\hat{p}_j^{(0)}}\right) x^*,$$

$$\text{logistic regression: } \beta_0 + \sum_{j=1}^{d} \beta_j x_j^*,$$

where the $p^{(k)}$and the $\hat{p}_j^{(k)}$ are the prior probabilities and unigram MLEs for each class as defined in Section 2.1.1, and the β_j are unconstrained real numbers. Both classifiers predict that $y^* = 1$ if the log odds above are positive, and $y^* = 0$ otherwise.

In this case, the bias term in naive Bayes was exactly 0 since a uniform prior distribution was used. Similarly, the fitted $\hat{\beta}_0$ in the logistic regression was very close to 0 ($\hat{\beta}_0 = -0.01$). This suggests that β_0 is not needed at all here: in fact, setting $\beta_0 = 0$ improved the test accuracy of the logistic regression from 87.50% to 87.52% even with no changes to the other fitted parameters. We could test this formally using the likelihood ratio test [34]: the logistic regression model with $\beta_0 = 0$ is nested in the unconstrained model, so we can fit both models with maximum likelihood estimation and compute the likelihood ratio test statistic $\Lambda = 2(L(\hat{\boldsymbol{\beta}}') - L(\hat{\boldsymbol{\beta}}))$, where $\hat{\boldsymbol{\beta}}'$ and $\hat{\boldsymbol{\beta}}$ are the MLEs of the constrained and unconstrained models respectively and L is the log likelihood function. If $\Lambda > \chi^2_{d-1,\alpha}$ this test would reject the hypothesis $\beta_0 = 0$ at significance level α. It would also be interesting to perform this test for each of the weights β_j to see if they could be set to 0 - this would determine whether the presence of the corresponding word has a significant impact on the sentiment of a review.

It is easy to see that logistic regression has a lower asymptotic error than naive Bayes, since the decision rule for naive Bayes is a special case of that of logistic regression, with restrictions placed on the parameters [11]. Indeed, logistic regression outperformed naive Bayes on this task - since it doesn't make the assumption that the features are independent, it handles correlated words better.

An example of this is the example sentence from Section 2.1.2, `"Alfred Hitchcock did a bad job on this film"`, which was misclassified by naive Bayes since the words `Alfred` and `Hitchcock` are correlated. Because they appeared more often in positive reviews, both were given large positive coefficients, as shown in Figure 3.6b. In the logistic regression, however, both have coefficients close to zero - this is to be expected since they are not actually indicative of the reviewer's opinion. Because of this, logistic regression correctly classifies the sentence, as shown in Figure 3.6d. Note that the bias term is omitted from this plot for readability since it had a negligible impact on this decision.

However, this does not mean that we should disregard naive Bayes completely - it is easier and faster to train, since we can solve for the MLE analytically. It can actually outperform logistic regression on small datasets since it converges faster to its asymptotic error, only requiring $O(\log d)$ training examples in most cases, compared to $O(d)$ for logistic regression [11]. The intuition here is that since the parameter space for naive Bayes is lower dimensional, it requires less data to fit the model well.

3.5 The Skip-Gram Model

We will now return to the task of training word embeddings as introduced in Section 3.1. We will look at the skip-gram model [24], which aims to predict context words given the embedding for a target word. We define a **context word** as any word appearing within a **context window** of length $l = 2c$ of the **target word**, where c is the number of words included in the window on either side of the target word.

It is assumed that context words appear independently of each other. As such, the training data can be represented using **skip-grams**, so called because they are n-grams that allow tokens to be skipped. We represent the $(l+1)$-gram centred on the target word with l skip-gram pairs of target and context words. For example, consider the sentence:

`She sells sea shells on the sea shore.`

If we set $c = 1$, the observed context words for `shells` are `sea` and `on`. We represent these as the skip-grams (`shells`,`sea`) and (`shells`,`on`). Table 3.1 contains a complete list of the skip-grams observed in this sentence.

The task of predicting context words can be seen as a classification task, where the model must output the probability of each possible context word in the vocabulary given the target word. We define this probability distribution with the **softmax** function, $\text{softmax} : \mathbb{R}^K \to \mathbb{R}^K$, a multi-dimensional generalisation to the sigmoid (logistic) function:

$$\text{softmax}(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}\,. \tag{3.13}$$

The softmax function produces a valid probability distribution: $\text{softmax}(\boldsymbol{z})_i \in (0,1)\ \forall i$ and $\sum_{i=1}^{K} \text{softmax}(\boldsymbol{z})_i = 1$. It can be used to generalise the binary logistic regression to K classes, defining the probabilities for a multinomial logistic regression as:

$$\mathbb{P}\left(y^{(i)} = k\right) = \text{softmax}(\boldsymbol{b} + W^T \boldsymbol{x}^{(i)})_k = \frac{e^{\,b_k + W_k \cdot \boldsymbol{x}^{(i)}}}{\sum_{j=1}^{K} e^{\,b_j + W_j \cdot \boldsymbol{x}^{(i)}}} \quad \text{for} \quad k \in \{1, ..., K\}\,, \tag{3.14}$$

where $W \in \mathbb{R}^{d \times K}$ is a matrix of weights with columns $W_j \in \mathbb{R}^d$, and $\boldsymbol{b} \in \mathbb{R}^K$ is a vector of biases. This actually overparametrises the distribution [33], since only $K-1$ outputs are required to compute $\mathbb{P}\left(y^{(i)} = K\right)$ as $1 - \sum_{k=1}^{K-1} \mathbb{P}\left(y^{(i)} = k\right)$. This can be resolved by setting b_K and W_K to 0 [18], but in practice the overparametrisation is not a big issue for word

Target word	Observed skip-grams
she	(she, sells)
sells	(sells, she), (sells, sea)
sea	(sea, sells), (sea, shells), (sea, the), (sea, shore)
shells	(shells, sea), (shells, on)
on	(on, shells), (on, the)
the	(the, on), (the, sea)
shore	(shore, the)

Table 3.1: Skip-grams observed in the example sentence

embeddings or neural networks. In these contexts, the formulation in Eq. 3.14 is generally preferred since it is simpler to implement. In this case, it will allow us to interpret the weights as a second set of word embeddings.

In the skip-gram model, the set of possible classes is simply the vocabulary V. We feed the target embedding $\boldsymbol{v}_{w_i}$ for word w_i into a multinomial logistic regression with $|V|$ classes and a bias vector of 0. Its j^{th} output is the estimated probability of word w_j appearing as a context word for w_i, $\hat{\mathrm{P}}(w_j|w_i)$. Since each possible context word in the vocabulary has its own weight vector in the logistic regression, it is more intuitive to think of these as another set of word embeddings, the **context embeddings**, so that:

$$\hat{\mathrm{P}}(w_j|w_i) = \mathrm{softmax}(C^T \boldsymbol{v}_{w_i})_j = \frac{e^{\boldsymbol{c}_{w_j} \cdot \boldsymbol{v}_{w_i}}}{\sum_{w \in V} e^{\boldsymbol{c}_w \cdot \boldsymbol{v}_{w_i}}}\,, \tag{3.15}$$

where C is a matrix whose j^{th} column contains $\boldsymbol{c}_{w_j}$, the context embedding for word w_j. The context embeddings can be discarded after training, although in practice they are often added or concatenated to the target embeddings instead to improve their quality.

The training objective for skip-gram is to find the set of target and context embeddings that maximise the average log probability of the observed context words in the dataset:

$$\frac{1}{N} \sum_{i=1}^{N} \sum_{-c \leq j \leq c \,:\, j \neq 0} \log \hat{\mathrm{P}}(w_{i+j}|w_i)\,. \tag{3.16}$$

Note that this cannot have a unique maximum - applying a rotation to all of the vectors will preserve the dot products in Eq 3.15 and hence the log likelihood. Although there is no unique set of 'best' word embeddings, we can still find a good set of vectors by optimising the negative log likelihood with SGD. The gradient of the loss function can be calculated using the **backpropogation** algorithm, which we will discuss in more detail in the context of neural network models in Section 4.2.2.

One key issue with this approach is that calculating a softmax over the full vocabulary is computationally expensive for large vocabularies, since the computation cost scales linearly with the vocabulary size. One alternative is to use the **hierarchical softmax** [25], a computationally efficient approximation to the full softmax using a binary tree, requiring only around $\log_2 |V|$ calculations. As we will see in the next section, another approach is to simplify the model so that it describes a binary classification problem instead.

3.5.1 Skip-Gram with Negative Sampling

Since the context word prediction task is just a means to achieve our real aim of obtaining word embeddings, we are free to simplify the model further. The idea behind **negative sampling** [25] is to instead train a binary classifier to distinguish between real context words and randomly selected noise words.

Training data for our model is created by treating words observed as context words of the target as **positive samples**, and randomly generating **negative samples** from the rest of the vocabulary. For each positive sample (w, w^+), we generate some fixed number k of negative samples (w, w_j^-). In the example sentence `"She sells sea shells on the sea shore."`,

(`shells`,`she`) and (`shells`,`shore`) are possible negative samples for the observed skip-gram (`shells`,`sea`) since `she` and `shore` were not observed within our context window for `shells`.

The negative samples are generated according to their unigram frequency: noise word w_j is sampled with probability $\mathrm{P}(w_j) \propto \mathrm{Count}(w_j)^\alpha$ for some constant α, with $\alpha = \frac{3}{4}$ most commonly used in practice. This approach outperforms using the unigram or uniform distributions [25], since it redistributes probability mass towards rarer words. This is desirable since it increases the amount of data available to train the context embeddings for rare words, while still taking word frequency into account.

Using the logistic function, the probability of w_j being a real context word for w_i will be defined in terms of their target and context word embeddings as:

$$\hat{\mathrm{P}}(w_j|w_i) = \sigma(\boldsymbol{v}_{w_i} \cdot \boldsymbol{c}_{w_j})\,.$$

This will be high if the angle between the target vector for w_i and the context vector for w_j is small. Making the assumption that context words occur independently of each other, the log likelihood for one positive observation and k negative is:

$$\log\Big(\hat{\mathrm{P}}(w^+|w)\prod_{j=1}^{k}(1-\hat{\mathrm{P}}(w_j^-|w))\Big) = \log\sigma(\boldsymbol{c}_{w^+}\cdot\boldsymbol{v}_w) + \sum_{j=1}^{k}\log\sigma(-\boldsymbol{c}_{w_j^-}\cdot\boldsymbol{v}_w).$$

The total log likelihood for the dataset is obtained by summing this expression over each positive sample (w, w^+) in the set of observed context words for w, for each word in the vocabulary V:

$$\sum_{w\in V}\sum_{w^+}\Bigg(\log\sigma(\boldsymbol{c}_{w^+}\cdot\boldsymbol{v}_w) + \sum_{j=1}^{k}\log\sigma(-\boldsymbol{c}_{w_j^-}\cdot\boldsymbol{v}_w)\Bigg).$$

Like the log likelihood for the original skip-gram model, this is preserved under rotation of the word vectors and therefore does not have a unique maximum. However, we can still train the word embeddings by optimising with SGD.

3.5.2 Hyperparameters and Training

Generally, increasing the dimension of embeddings improves their quality, although more training data and computation time is required to train good high-dimensional vectors. Mikolov et al. [25] recommend using a context window of length $l = 10$ for skip-gram, and taking 2-5 negative samples for large datasets, with 5-20 necessary for smaller ones. The intuition behind this rule of thumb is that increasing k increases the number of training samples, which can improve the embeddings. This is important for small datasets, where we need to make the most of what data we have, while for larger datasets this is less of a concern than the increased computation cost of training. Increasing l only improves quality up to a point, since more distant words are less informative. In fact, the size of l has an effect on the type of word similarities picked up by the model [35]: If l is small the embeddings will primarily reflect syntactic similarities (e.g. grouping nouns or verbs), while with a large l words will tend to be grouped by topic.

Target word	Most similar to:
you	ye, yourselves
good	kind, bad
take	bring, give
took	drew, received
say	saying, bet
said	answered, cried
chin	brows, chest
brother	father, son
sister	mother, wife
coat	overcoat, waistcoat
revolver	pistol, butt

Table 3.2: Examples of word similarities found by the model.

We will once again use the Sherlock Holmes corpus to train the model. Since this dataset is relatively small (typical datasets for this task can have millions or even billions of words), a small embedding dimension $d = 100$ was chosen, with $l = 10$ and $k = 20$. Any words with less than 2 occurrences in the dataset were discarded, since this greatly reduced the vocabulary size while having only a small impact on the training data. The Gensim library [36] was used to train the model, since this ran far more efficiently than an implementation from scratch - Gensim uses techniques like parallel processing to speed up computations.

Note that unlike the loss function for logistic regression, the loss for skip-gram is not convex so there is no unique minimum. This means that the resulting vectors vary a lot depending on the random initialisation, and we will even see different results starting from the same initial values, since SGD is a random walk. The resulting word embeddings can be evaluated by examining how useful they are in the target task, for example by using them to initialise the embedding layer of a neural language model, as we will see in Section 4.3.1. It is also possible to directly evaluate how well the embeddings capture word similarities by comparing the cosine similarities between pairs of vectors to a database of similarity scores for the corresponding words [15].

On inspection, the model did well at grouping words with similar meanings together - some examples are listed in Table 3.2. The model also successfully grouped words by syntactic meaning - nouns, adjectives, and verbs were usually grouped together, with verbs even grouped by tense as well as meaning. This worked best for more common words - due to the small amounts of training data for rarer words, their embeddings cannot move far from their random initialisation.

The word vectors can be visualised more easily by projecting them into two dimensions. Using principal component analysis (PCA) to identify the principal directions of variation, the resulting projection preserves some of the structure of the vector space, including some clusters of similar words. Figure 3.7 shows the embeddings for 100 common words from the dataset, normalised and then projected into two dimensions with PCA. It is interesting to note that while the individual embeddings and word associations varied each time the model was fitted, the arrangement of these clusters in the PCA projection was very similar each time - for example: crime words clustered at the bottom and family members in the centre.

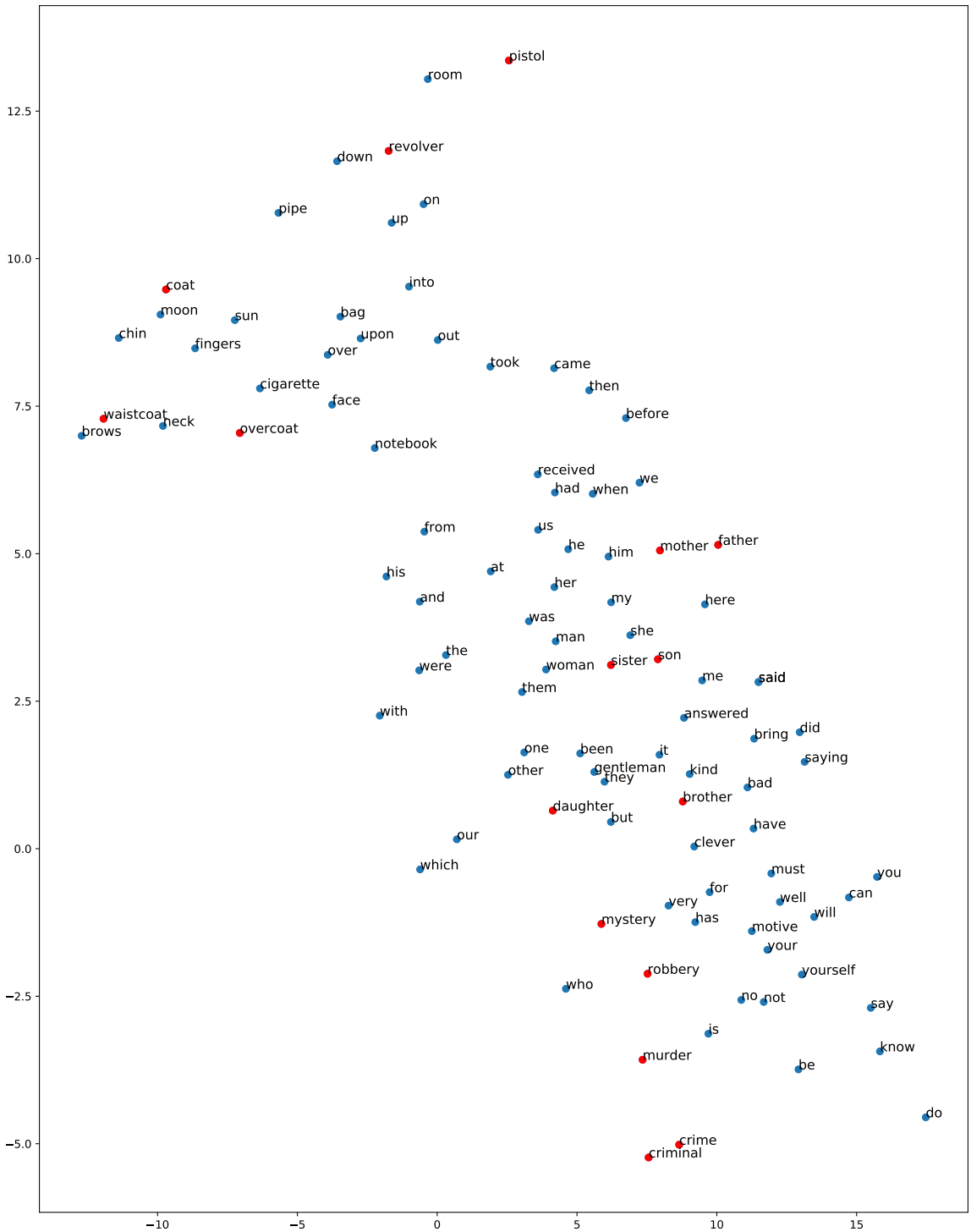


Figure 3.7: The embeddings for 100 words from Sherlock Holmes, projected into 2D. Some clusters of words with high cosine similarity are highlighted in red.

3.6 Discussion

More training data would be required to unlock the full potential of word embeddings - state-of-the-art word vectors are trained on datasets with over a billion words and even follow logical semantic rules [23] . For example, using their vector representations, performing the calculation `king` - `man` + `woman` results in a vector close to `queen`. Due to the computational expense of training embeddings on such large datasets, it is common to use pre-trained word vectors. These can either be used out of the box in a language model, or fine-tuned on the desired dataset first. Using a larger dataset to help with initialising word vectors is a form of **transfer learning** [37], and can be a good strategy for improving model performance with limited training data since it makes it possible to obtain good representations of rare words. Fine-tuning can be seen as a Bayesian approach, since the pre-trained embeddings represent a prior belief on how words should be represented in the model, which is then updated after observing the target dataset and desired model.

Word embeddings for different languages often share a similar structure - Mikolov et al. [38] showed that embeddings trained with Word2Vec for different languages have a similar geometric structure, and it is even possible to learn a linear projection between the embedding spaces that allows for translation of words. The matrix for this projection is trained using SGD and a list of translations for some common words. The resulting projection was surprisingly effective at translating words, even allowing for the detection of some errors in a translation dictionary.

One issue with Word2Vec is that it does not account for the meanings of words being context-dependent, since each word is assumed to have a fixed meaning. For example, `stick` can be either a noun or a verb and means something very different in each case. Word2Vec embeddings are an example of **static** embeddings, since the embedding for each word is fixed, and must capture every possible sense of the word. An alternative is to use **contextual** word vectors like those computed by BERT (Bidirectional Encoder Representations from Transformers) [39], which aim to capture what words mean in a particular context and can consider both the left and right context at the same time. These embeddings are pre-trained on very large corpora, and can be fine-tuned on the desired dataset.

While vector relationships like the above `"man is to king as woman is to queen"` reflect an analogous relationship between the meanings of the words they represent, the same is not true of all such learned relationships. In practice, the semantics of word vectors often reflect biases present in the training data such as gender stereotypes [40]: for example, the same model computes that `"man is to surgeon as woman is to nurse"`. It is possible to correct this bias - Bolukbasi et al. [40] suggested a method for debiasing word embeddings by learning a subspace of the embedding space that captures the bias. Their method preserves analogies for gender specific words like `gramdmother` and `grandfather`, while removing inappropriate gender analogies.

In the next chapter, we will introduce neural network language models, which typically include an embedding layer as their first layer. These embeddings can be trained as part of the network, but using pre-trained word vectors to initialise them can often improve performance, and using fixed pre-trained embeddings can significantly cut down computation time.

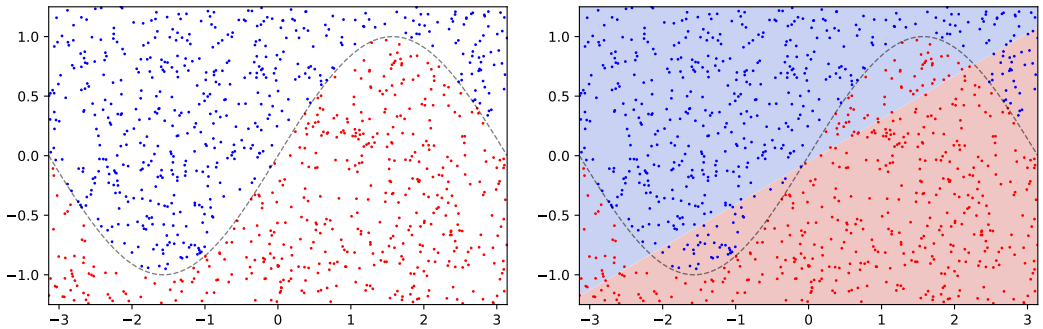
Chapter 4

Neural Models

Neural networks [33] are a class of models inspired by the workings of the brain. As the name suggests, they consist of a network of artificial **neurons**, which are simple computational units that each take a vector of input values and produce a scalar output which is passed on to other neurons in the network.

We will focus on using neural networks for classification, since language modelling can be seen as a classification problem. Given some sequence of words $\boldsymbol{w} = w_1, \ldots, w_{t-1}$ the aim is to estimate the probability that some word w_i will appear next in the sequence. In this case, the set of possible 'classes' for the input data $\boldsymbol{w}$ is the vocabulary V, and the statement "$\boldsymbol{w} \in$ class w_i" is equivalent to "w_i is the next word in the sequence $w_1, \ldots, w_{t-1}$" .

We have examined two classifiers so far - logistic regression (Sections 3.2 and 3.5) and naive Bayes (Section 2.1.1). These are both examples of **linear classifiers**, because they make decisions based on a linear function of their inputs. As we have seen, the binary logistic regression will predict a label of $y^* = 1$ if $\beta_0 + \sum_{j=1}^{d} \beta_j x_j^* > 0$ and 0 if $\beta_0 + \sum_{j=1}^{d} \beta_j x_j^* < 0$. Hence its **decision boundary** is the hyperplane $\beta_0 + \sum_{j=1}^{d} \beta_j x_j^* = 0$. Because of this, logistic regression will do well when the data is **linearly separable**, ie. when it is possible to find a hyperplane that divides the data into the two classes. This is not always the case - for example, in Figure 4.1a we see an example of a two-dimensional toy dataset that is split into two classes (represented in red and blue) by the curve $x_2 = \sin x_1$. If we fit a logistic regression to this dataset its decision boundary is the straight line shown in Figure 4.1b, which does a poor job of separating the classes.



(a) Scatter plot of the toy dataset. (b) Decision boundary of logistic regression.

Figure 4.1: Toy dataset divided into classes by a curve.

While it is often possible to transform the input variables for logistic regression and other linear classifiers in such a way that they become linearly separable, this transformation must be chosen manually based on prior knowledge about the classification task. The key advantage of neural networks is that they automatically learn useful representations of the raw input data, and can learn to approximate a wide range of functions. This is very advantageous in the context of language modelling, a complex classification task where it is important to represent the input words in a way that reflects their meanings and similarities.

In this chapter we will first introduce feedforward neural networks for classification and discuss how to train them, as well as how to resolve some of the difficulties this task presents. We will see how to train a neural network classifier for language modelling, using the concept of word embeddings introduced in Chapter 3.

4.1 Neural Units

Neural units or **neurons** [10] are the building blocks of the neural network. Each neuron has a similar structure to a logistic regression (Section 3.2): given inputs $x_1, \ldots, x_h$ the unit computes a linear combination $b + \sum_{i=1}^{h} w_i x_i$, where the weights w_i and bias b are parameters to be learned.

This linear expression is passed into a nonlinear **activation function** $a : \mathbb{R} \to \mathbb{R}$, so that the output of the neuron is $a(b + \sum_{i=1}^{h} w_i x_i)$. Representing the inputs and weights as vectors in $\mathbb{R}^h$, we can write this as $a(b + \boldsymbol{w}^T \boldsymbol{x})$. Each neuron in the network is typically represented as a node in a directed graph, as shown in Figure 4.2.

While the parameters b and $\boldsymbol{w}$ will be learned during training, the activation function a for each neuron is a **hyperparameter**, since it is determined in advance. In principle, a can be any function, although the **sigmoid** (Eq. 4.1), **tanh** (Eq. 4.2) and **rectified linear** (Eq. 4.3) functions are by far the most common choices.

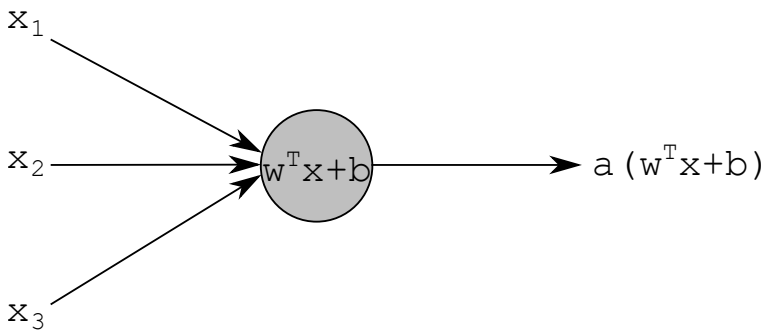
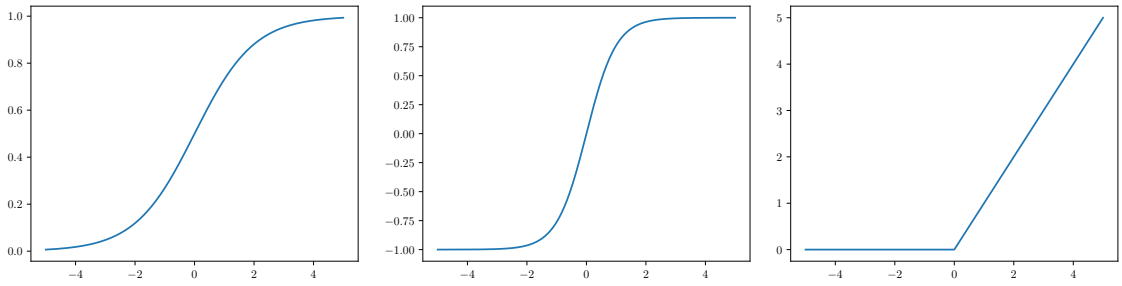


Figure 4.2: The computation carried out by a single neuron.



(a) Sigmoid (σ) (b) Tanh (c) Rectified linear (ReLU)

Figure 4.3: The graphs of some common activation functions.

These are defined as follows:

$$\sigma : \mathbb{R} \to (0,1) \qquad \sigma(x) = \frac{1}{1+e^{-x}}\,, \tag{4.1}$$

$$\tanh : \mathbb{R} \to (-1,1) \qquad \tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}\,, \tag{4.2}$$

$$\mathrm{ReLU} : \mathbb{R} \to [0,\infty) \qquad \mathrm{ReLU}(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}\,. \tag{4.3}$$

Figure 4.3 shows the plots of these functions. Note that the sigmoid is the activation function for the binary logistic regression, so binary logistic regression can in fact be seen as a single neuron with sigmoid activation. We will see in the next section that a multinomial logistic regression can be represented with a layer of neurons.

There are no hard rules for how to choose an activation function - as is the case with many other neural network hyperparameters it is common to use a process of trial and error, comparing performance on a validation dataset. ReLU is typically a good general purpose choice: it is close to linear and hence cheaper to compute and easier to optimise with gradient-based methods [33]. The sigmoid and tanh functions have the advantage of being smoothly differentiable everywhere, and map large inputs towards the mean since they have a finite range. They are in fact closely related: $\tanh(x) = 2\sigma(2z) - 1$. However, tanh typically outperforms sigmoid since it is similar to the identity function around 0.

These units can be linked together into a network - this is done by constructing a directed graph, where each node represents a neuron and each edge represents the output of one neuron being transmitted to one of the inputs of the next. As we will see in the next section, a simple way of doing this is to connect them into an acyclic graph.

4.2 Feedforward Neural Networks

In a **feedforward neural network** or **multi-layer perceptron** (MLP) [33], neurons are arranged in layers so that neurons in each layer are connected to every neuron in the following layer. Each layer of neurons is called a **fully connected** or **dense** layer. This architecture is shown in Figure 4.4. This is known as a feedforward network since the outputs from each layer of neurons are fed forwards into the next layer, with no connections back to previous layers.

The layers of the network are grouped into three types: the **input layer**, **hidden layers** and **output layer**. Note that although the inputs to the network are included as a layer, they simply transmit the input $\boldsymbol{x} \in \mathbb{R}^{d_i}$ to the first layer of the network and have no trainable parameters. The output layer produces the output $\hat{\boldsymbol{y}} \in \mathbb{R}^{d_o}$ of the network, which should be close to the true label $\boldsymbol{y} \in \mathbb{R}^{d_o}$. The hidden layers are so called because their outputs are not used directly in evaluating the network and we do not impose any definition of what counts as a good output beyond that it should be useful for subsequent layers.

When using neural networks for classification, the true label $\boldsymbol{y}$ is typically a **one-hot vector** in $\mathbb{R}^{d_o}$, with zeros in every coordinate except the one corresponding to the true class. That is to say: if the true class of $\boldsymbol{x}^{(i)}$ is $k \in \{1, \dots, d_o\}$, then $\boldsymbol{y}_k^{(i)} = 1$ and $\boldsymbol{y}_j^{(i)} = 0 \ \forall j \in \{1, \dots, d_o\} \setminus \{k\}$. We assume that the distribution of $\boldsymbol{y}^{(i)}$ given the data $\boldsymbol{x}^{(i)}$ is multinoulli, i.e. $\boldsymbol{y}^{(i)}|\boldsymbol{x}^{(i)} \sim \text{Multinomial}(1, \boldsymbol{p})$ for some function $\boldsymbol{p} : \mathbb{R}^{d_i} \to \mathbb{R}^{d_o}$ of $\boldsymbol{x}^{(i)}$ that our neural network output will approximate by $\hat{\boldsymbol{y}}^{(i)}$. This means that the j^{th} output of the neural network should be an estimate of $\mathbb{P}\left(\{\boldsymbol{x}^{(i)} \in \text{ class } j\}\right)$.

To ensure that the output of the network is a valid probability distribution, we will use the **softmax** function defined in Section 3.5 as the activation function for the output layer:

$$\text{softmax}(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{d_o} e^{z_j}} \text{ for } 1 \leq i \leq d_o. \tag{4.4}$$

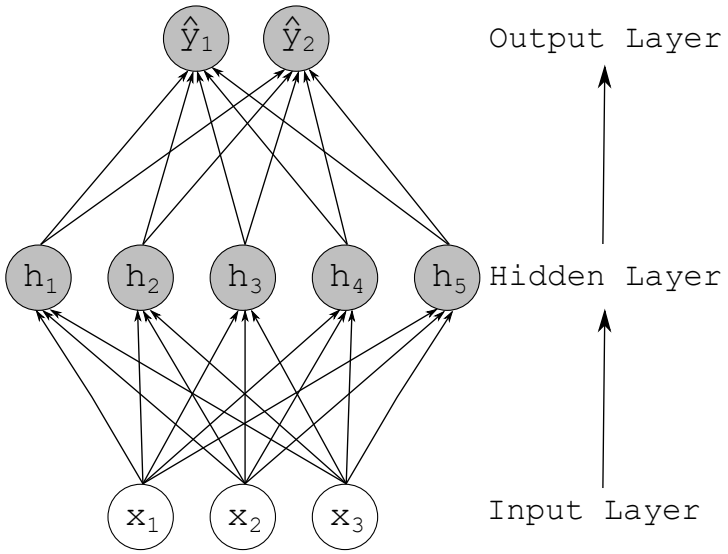


Figure 4.4: A simple neural network with one hidden layer.

The full computation or **forward pass** carried out by a neural network classifier with one hidden layer is as follows:

- **Input layer**: a vector $\boldsymbol{x} = (x_1, \ldots, x_{d_i})^T$ of fixed length d_i is fed into the network.

- **Hidden layer**: each of the d_h neurons computes a weighted sum of the x_j, so that the output of neuron k is $h_k = a(b_k + \sum_{j=1}^{d_x} w_{k,j}\, x_j)$. We can write this more compactly by representing the linear transformation as a matrix multiplication: the output of the layer is $\boldsymbol{h} = (h_1, \ldots, h_{d_h})^T = a(\boldsymbol{b} + W\boldsymbol{x})$, where $\boldsymbol{b} = (b_1, \ldots, b_{d_h})^T$, $W \in \mathbb{R}^{d_h \times d_i}$ is such that $W_{jk} = w_{j,k}$, and a is applied element-wise to the input vector.

- **Output layer**: this layer has d_o neurons, one per class. Given weight matrix $U \in \mathbb{R}^{d_o \times d_h}$ and bias vector $v \in \mathbb{R}^{d_o}$, the output of the network is $\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{v} + U\boldsymbol{h})$.

In total, this network has $(d_i + 1) \times d_h + (d_h + 1) \times d_o$ trainable parameters. Additional hidden layers h can easily be added, carrying out the same calculation with their own activation function $a^{(h)}$, weight matrix $W^{(h)}$ and bias vector $\boldsymbol{b}^{(h)}$. A classification decision can be obtained from the network by predicting the class with the highest estimated probability, $\hat{C} = \text{argmax}_k\, \hat{\boldsymbol{y}}_k$.

Note that if the activation function for each neuron is linear (for example, the identity function), the whole network can be reduced to a single linear transformation: the nonlinear activation functions are required to produce a model with a nonlinear decision boundary.

The key advantage of neural networks is that they automatically learn useful representations of the input data. The computation carried out by the output layer is in fact identical to that of a multinomial logistic regression classifier (as defined in Section 3.5), so we can think of the hidden layers as learning a representation $\boldsymbol{h}$ of the input data which can then be used as the input to a standard multinomial logistic regression [10].

In fact, neural networks with only one hidden layer can be trained to be arbitrarily good approximations to a very large family of functions, including all continuous functions on compact subsets of $\mathbb{R}^n$ [35]. However, this may require a very large hidden layer and there is no guarantee that the required model parameters are easy to find. As we will see in Section 4.2.4, neural networks with several smaller hidden layers are typically preferable in practice since they often require fewer parameters to reach the same level of accuracy.

Neural networks are usually fitted using stochastic gradient descent (Section 3.3). The next sections will introduce the loss function typically used in neural network classifiers, as well as a method for computing its gradient with respect to the neural network parameters.

4.2.1 Categorical Cross-Entropy

The loss function most often used for neural networks for classification is the **categorical cross-entropy**. This is the multi-class generalisation to the binary cross-entropy loss introduced in Section 3.3.1 for the binary logistic regression, and is equal to the negative log likelihood.

Recall that each true label $\boldsymbol{y}^{(i)}$ is a one-hot vector, i.e. if the true class of $\boldsymbol{x}^{(i)}$ is k then $\boldsymbol{y}_k^{(i)} = 1$ and $\boldsymbol{y}_j^{(i)} = 0 \; \forall j \neq k$. We assume that the distribution of $\boldsymbol{y}^{(i)}$ given the data $\boldsymbol{x}^{(i)}$ is multinoulli, i.e. $\boldsymbol{y}^{(i)}|\boldsymbol{x}^{(i)} \sim \text{Multinomial}(1, \boldsymbol{p}(\boldsymbol{x}^{(i)}))$ for some function $\boldsymbol{p}$. Then the likelihood of $\boldsymbol{y}^{(i)}$ given the neural network's estimate $\hat{\boldsymbol{y}}^{(i)}$ of $\boldsymbol{p}(\boldsymbol{x}^{(i)})$ is:

$$\ell(\boldsymbol{y}^{(i)}|\hat{\boldsymbol{y}}^{(i)}) = \boldsymbol{y}^{(i)} \cdot \hat{\boldsymbol{y}}^{(i)} = \hat{\boldsymbol{y}}_k^{(i)} \,,$$

and the categorical cross-entropy loss for one observation $(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})$ is:

$$L_{CE}(\hat{\boldsymbol{y}}^{(i)}; \boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}) = -\log(\boldsymbol{y}^{(i)} \cdot \hat{\boldsymbol{y}}^{(i)}) \,. \tag{4.5}$$

The loss for a dataset of size N is:

$$L_{CE}(\hat{\boldsymbol{y}}) = \frac{1}{N} \sum_{i=1}^{N} L_{CE}(\hat{\boldsymbol{y}}^{(i)}; \boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}) = -\frac{1}{N} \sum_{i=1}^{N} \log(\boldsymbol{y}^{(i)} \cdot \hat{\boldsymbol{y}}^{(i)}) \,.$$

4.2.2 Backpropogation

To compute the gradient of the loss function, we will use an algorithm called **backpropogation** [33]. This works by representing the network as a **computation graph** and recursively applying the chain rule to calculate the derivative with respect to each parameter.

We will start by examining how this works on a single neuron. Figure 4.5 shows a computation graph representation of the calculation carried out by a single neuron with two inputs and activation function a. The derivative of the output z can be computed with respect to w_i using the chain rule as follows:

$$\frac{\partial z}{\partial w_i} = \frac{\partial a}{\partial p} \frac{\partial p}{\partial m_i} \frac{\partial m_i}{\partial w_i} = a'(p) \cdot 1 \cdot x_i = x_i \, a'\left(b + \sum_{j=1}^{2} w_j x_j\right) . \tag{4.6}$$

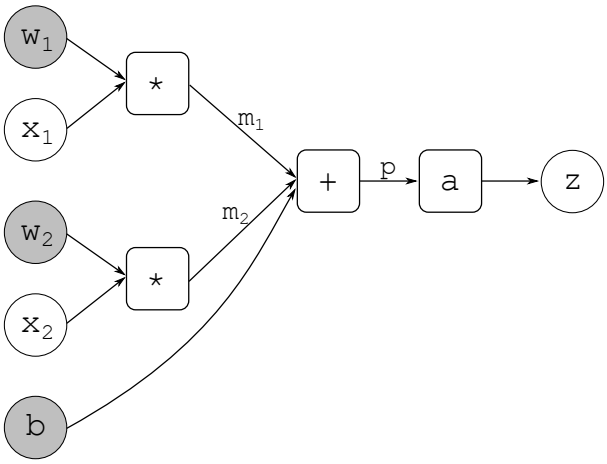


Figure 4.5: Computation graph representation of a single neuron with two inputs.

Similarly:

$$\frac{\partial z}{\partial x_i} = w_i a'\left(b + \sum_{j=1}^{2} w_j x_j\right),$$
$$\frac{\partial z}{\partial b} = a'\left(b + \sum_{j=1}^{2} w_j x_j\right).$$

The only thing missing is the derivative of the activation function. The derivatives of the sigmoid and tanh activation functions are as follows [10]:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)),$$
$$\tanh'(x) = 1 - \tanh(x)^2.$$

However, we run into a problem with ReLU since its derivative is undefined at $x = 0$. In practice this is not a big issue - it is very unlikely that the input to ReLU will be exactly 0 [41]. It is also not an issue that it is possible for the gradient at a minimum to be undefined, since when training the neural network we are only trying to approximately minimise the loss function and do not expect to reach a true minimum. As such, we can simply define the derivative as:

$$\text{ReLU}'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases},$$

where we return one of the one-sided derivatives if $x = 0$. The justification for this approach is that a computer will round small values of x to 0, so if ReLU receives an input of 0 it is far more likely to be as a result of this rounding than the input actually being exactly 0 [33].

We will now use the calculation above for individual neurons to define the derivatives of the loss function for all of the network parameters. We will first define some notation:

Let the network have n hidden layers. Let layer h have d_h neurons, activation function $a^{(h)}$, weight matrix $W^{(h)}$, and bias vector $\boldsymbol{b}^{(h)}$. Let $\boldsymbol{l}^{(h)}$ be the linear part of layer h, so that $\boldsymbol{l}^{(h)}(\boldsymbol{z}^{(h-1)}) = \boldsymbol{b}^{(h)} + W^{(h)}\boldsymbol{z}^{(h-1)}$, and $l_i^{(h)} = \sum_j W_{ij}^{(h)} z_j^{(h-1)}$. Denote the output of layer h by $\boldsymbol{z}^{(h)}$, so that $z_i^{(h)} = a^{(h)}(l_i^{(h)})$.

Define $\boldsymbol{z}^{(0)}$ to be the input to the network, $\boldsymbol{z}^{(0)} = \boldsymbol{x}$, and let W^{out} and $\boldsymbol{b}^{out}$ be the parameters of the output layer.

In the **forward pass** of the network, we compute and store the $\boldsymbol{l}^{(h)}$ and $\boldsymbol{z}^{(h)}$ for a particular input $\boldsymbol{x}$. Having done this, we can use backpropogation to compute the derivatives of L with respect to the neural network parameters.

We will first compute this for the output layer. Note that by Eq. 4.5, the loss of one observation $\boldsymbol{x}$ depends only on the k^{th} element of the output layer, where k is the true class of $\boldsymbol{x}$. Using the definition of softmax in Eq. 4.4 we have that:

$$L_{CE}(\hat{\boldsymbol{y}}; \boldsymbol{x}, \boldsymbol{y}) = -\log\big(\text{softmax}(\boldsymbol{l}^{out})_k\big) = -l_k^{out} + \log\left(\sum_j \exp(l_j^{out})\right),$$

and hence:

$$\frac{\partial L_{CE}}{\partial l_i^{out}} = -\delta_{ik} + \frac{\exp(l_i^{out})}{\sum_j \exp(l_j^{out})}\,,$$

where δ_{ik} is the Kronecker delta, $\delta_{ik} = 1$ if $i = k$ else 0.

To compute the other derivatives, the **multivariate chain rule** is required: let $\boldsymbol{x} \in \mathbb{R}^n$, $z = f(g(\boldsymbol{x})) \in \mathbb{R}$, where $g : \mathbb{R}^n \to \mathbb{R}^m$ and $f : \mathbb{R}^m \to \mathbb{R}$. If $\boldsymbol{y} = g(\boldsymbol{x})$, then:

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}\,. \tag{4.7}$$

Applying Eq. 4.7, the derivatives of the output layer with respect to the $z_i^{(n)}$ are:

$$\frac{\partial L}{\partial z_i^{(n)}} = \sum_{j=1}^{d_{out}} \frac{\partial L}{\partial l_j}\frac{\partial l_j}{\partial z_i^{(n)}} = \sum_{j=1}^{d_{out}} \frac{\partial L}{\partial l_j} W_{ji}^{out}\,,$$

and, using the univariate chain rule:

$$\frac{\partial L}{\partial W_{ij}^{out}} = \frac{\partial L}{\partial l_i} z_j^{(n)}\,,$$
$$\frac{\partial L}{\partial b_i^{out}} = \frac{\partial L}{\partial l_i}\,.$$

Note that L is a scalar function of $\boldsymbol{z}^{(n)}$ and $\boldsymbol{z}^{(n)}$ is a function of $\boldsymbol{z}^{(n-1)}$, so Eq. 4.7 can be applied to compute $\frac{\partial L}{\partial z_i^{(n-1)}}$ in terms of the $\frac{\partial L}{\partial z_j^{(n)}}$, which we calculated above. In fact, given any set of partial derivatives for the outputs of layer h we can define the partial derivatives for layer $h-1$ recursively as follows, using the computation for a single neuron in Eq. 4.6:

$$\frac{\partial L}{\partial z_i^{(h-1)}} = \sum_{j=1}^{d_h} \frac{\partial L}{\partial z_j^{(h)}}\frac{\partial z_j^{(h)}}{\partial z_i^{(h-1)}} = \sum_{j=1}^{d_h} \frac{\partial L}{\partial z_j^{(h)}} \times (a^{(h)})'\left(l_j^{(h)}\right) W_{ji}^{(h)}\,.$$

The derivatives for the parameters of layer h are:

$$\frac{\partial L}{\partial W_{ij}^{(h)}} = \frac{\partial L}{\partial z_i^{(h)}}\frac{\partial z_i^{(h)}}{\partial W_{ij}^{(h)}} = \frac{\partial L}{\partial z_i^{(h)}} \times (a^{(h)})'\left(l_i^{(h)}\right) z_j^{(h-1)}\,,$$
$$\frac{\partial L}{\partial b_i^{(h)}} = \frac{\partial L}{\partial z_i^{(h)}}\frac{\partial z_i^{(h)}}{\partial b_i^{(h)}} = \frac{\partial L}{\partial z_i^{(h)}} \times (a^{(h)})'\left(l_i^{(h)}\right)\,.$$

By computing the above for every $h \in \{1, \ldots, n\}$, we will have found the derivative of the loss function with respect to every parameter in the network.

4.2.3 Accelerating Convergence of SGD

The task of optimising neural network parameters is very challenging, since the objective function is typically non-convex and very high-dimensional. SGD can be slow to converge in these conditions: one reason for this is the difficulty in choosing a learning rate or a deterministic learning rate schedule, since different regions of the loss surface may require different learning rates. One simple solution to this is **momentum SGD** [32], where previous updates are used to accelerate convergence.

Algorithm 4: Momentum SGD

Require: learning rate η, momentum parameter $\gamma \in (0, 1)$;
Require: sequence $\boldsymbol{g}_t$ of gradient estimates;
Initialise $\boldsymbol{m}_0 = 0$;
Initialise $\boldsymbol{\theta}_0$;
for $t = 1, 2, \ldots$ **do**
  $\boldsymbol{m}_t \leftarrow \gamma \boldsymbol{m}_{t-1} + \eta \boldsymbol{g}_t$ Update velocity;
  $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \boldsymbol{m}_t$ Update $\boldsymbol{\theta}$;
end

The sequence of gradient estimates can be produced in the same way as in Section 3.3 by taking a sequence of random subsamples of the dataset. A typical choice for γ is 0.9.

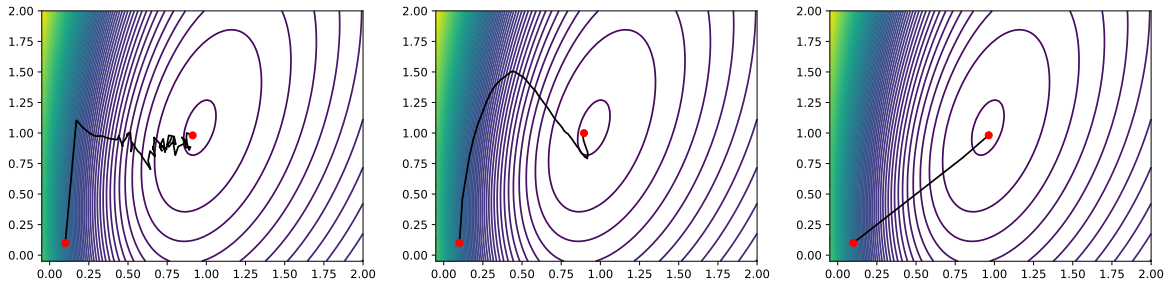
The intuition behind standard SGD is that, picturing the loss function as a surface, we are taking steps 'downhill' to minimise the loss. In momentum SGD, we allow these steps to accelerate if we spend time descending in the same direction, eventually reaching terminal velocity since $\gamma < 1$. The momentum term also helps reduce the impact of the noisy gradient estimates, since it smooths out sudden changes in direction. Note that $\boldsymbol{m}_t = \eta \sum_{s=1}^{t} \gamma^{t-s} \boldsymbol{g}_s$, so since $\gamma \in (0, 1)$, gradient estimates from previous iterations have an exponentially decaying weight in the calculation of new updates as time goes on.

We can speed up convergence further by adjusting the learning rate for each parameter automatically during training. One algorithm that does this is **Adaptive Moment Estimation (Adam)** [42]. Like momentum SGD, Adam uses a velocity vector $\boldsymbol{m}_t$ which is an exponential moving average of previous gradient estimates. Adam also adjusts the per parameter learning rate using estimates of the second moment of the gradient, computed as an exponential moving average of the squared gradient estimates. The algorithm for Adam is given in Algorithm 5. Note that the operations on vectors (square, square root, division etc.) are performed elementwise. The bias correction step is included because initialising the first and second moment vectors at 0 produces estimates that are biased towards 0.

Good default values for the hyperparameters are $\eta = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$, and $\epsilon = 10^{-8}$. ϵ is required to ensure that each parameter update is defined and bounded in size: it is possible for the vector $\sqrt{\hat{\boldsymbol{v}}_t}$ to have some zero entries (or arbitrarily close to 0), in which case the elementwise division by $\sqrt{\hat{\boldsymbol{v}}_t}$ is undefined (or can be arbitrarily large). Unlike SGD, where the learning rate must be carefully tuned for good results, Adam rarely requires changes to these default values [33]. This greatly reduces the time required to tune neural network hyperparameters, since it is usually not necessary to optimise a learning rate for each combination of hyperparameters.

Algorithm 5: Adam

Require: learning rate η, exponential decay rates $\beta_1, \beta_2 \in [0, 1)$;
Require: small constant ϵ;
Require: sequence $\boldsymbol{g}_t$ of gradient estimates;
Initialise: $\boldsymbol{m}_0 \leftarrow 0$, $\boldsymbol{v}_0 \leftarrow 0$;
Initialise $\boldsymbol{\theta}_0$;
for $t = 1, 2, \ldots$ **do**
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$ Update biased first moment estimate;
  $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t^2$ Update biased second moment estimate;
  $\hat{\boldsymbol{m}}_t \leftarrow \frac{\boldsymbol{m}_t}{1-\beta_1^t}$ Correct bias in first moment estimate;
  $\hat{\boldsymbol{v}}_t \leftarrow \frac{\boldsymbol{v}_t}{1-\beta_2^t}$ Correct bias in second moment estimate;
  $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta \frac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon}$ Update $\boldsymbol{\theta}$;
end



(a) SGD (b) Momentum SGD (c) Adam

Figure 4.6: Contour plot of the log loss for the toy example, with the path taken by each optimiser plotted in black. Start and endpoints marked in red.

To compare these algorithms, we will return to the toy example from Section 3.3.2, since its loss function can easily be visualised with a contour plot. Figure 4.6 shows the paths taken by SGD, momentum SGD, and Adam as they attempt to minimise this loss function. Each optimiser used a batch size of 50 and ran for 3 epochs, and in each case a gridsearch was used to find the best learning rate. Note that the 'true' value of $\boldsymbol{\theta}$ in this case is $(1, 1)^T$. As we can see, momentum SGD is a lot smoother - regular SGD zig-zags back and forth a lot as it descends the slope. However, momentum SGD also tends to overshoot the minimum slightly and has to correct course.

Adam seems to jump straight to the minimum - in fact, it converges in just 4 iterations (0.2 epochs) to a value very close to the true value of $\boldsymbol{\theta}$. This is because it scales the learning rate for each parameter - at the starting point, the loss function's slope is much steeper in the second coordinate than in the first, resulting in SGD and momentum SGD making a big jump upwards before slowly moving across to the minimum. Adam, however, scales the learning rates for each parameter so that the first step in each direction is about the same size. The per-parameter learning rates are then slowly scaled down as training continues. In this case, with a large learning rate η the first iteration of Adam brings it very close to the minimum so convergence is very fast.

4.2.4 Toy Example

We will now revisit the toy example from the introduction to this chapter, to demonstrate how neural networks can effectively classify data that is not linearly separable. The toy dataset was generated by sampling points from a two-dimensional uniform distribution and labelling them as class 0 (blue) if they fell above the line $x_2 = \sin x_1$ and class 1 (red) otherwise. This means that the two classes can be separated with 100% accuracy by the function $f : \mathbb{R}^2 \to \mathbb{R}$, where:

$$f(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \sin(x_1) \leq x_2 \\ 0 & \text{if } \sin(x_1) > x_2 \end{cases}$$

We will look at the decision boundaries of various neural networks to determine how well they can approximate f.

A selection of neural network models were trained to classify the data. Each model had $d_i = 2$ inputs and $d_o = 2$ output neurons, and had one or two hidden layers with ReLU activation. The models were built and trained using the TensorFlow [43] Python package, and Adam was used as the optimiser with its default hyperparameters. Four different model architectures were used:

(a) One hidden layer with 4 units (Figure 4.7a). Number of parameters: $3 \times 4 + 5 \times 2 = 22$

(b) Two hidden layers with 4 units each (Figure 4.7b). Number of parameters: $3 \times 4 + 5 \times 4 + 5 \times 2 = 42$

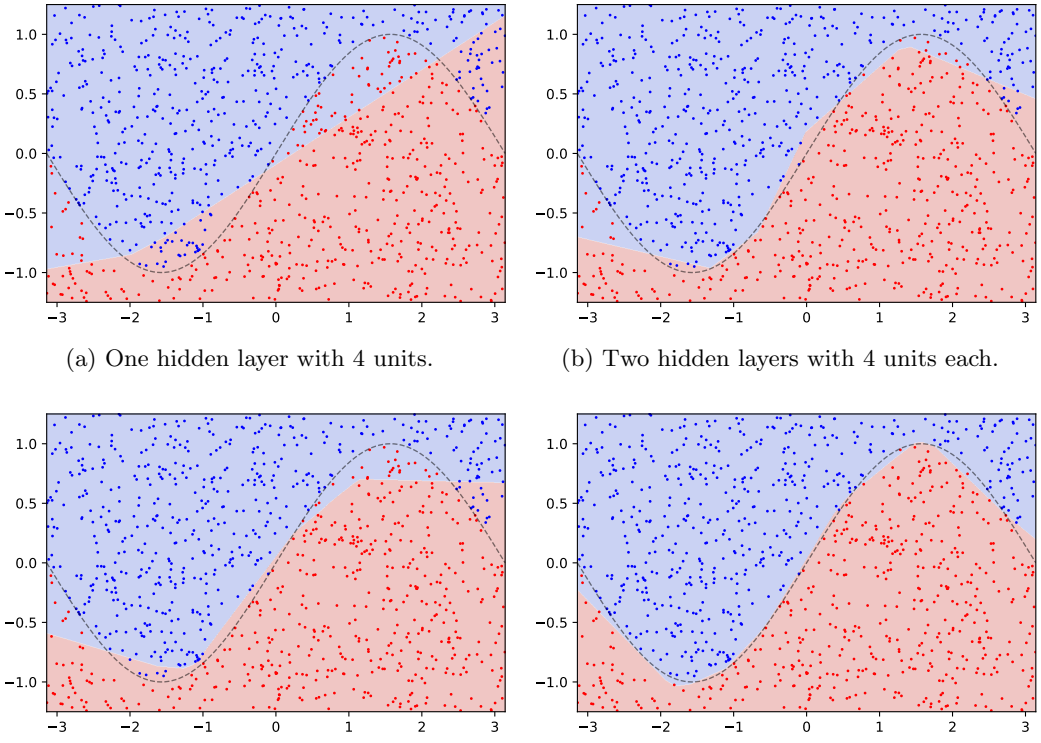
(c) One hidden layer with 16 units (Figure 4.7c). Number of parameters: $3 \times 16 + 17 \times 2 = 82$

(d) Two hidden layers with 16 units each (Figure 4.7d). Number of parameters: $3 \times 16 + 17 \times 16 + 17 \times 2 = 354$

The decision boundaries for each model are visualised in Figure 4.7, where the red and blue shaded areas correspond to the model output for points in that region, and the dashed line indicates the decision boundary of f.

Note that increasing the number of layers and increasing the number of units per layer both improve the accuracy of the model - this is to be expected since both increase the number of model parameters. However, we can also improve performance without requiring more parameters by rearranging the neurons - model (b) is a slightly better match for the desired decision boundary than model (c) despite having far fewer parameters, since it has two hidden layers instead of just one.

In practice, model architectures are chosen by a process of trial and error based on their performance on a validation dataset. While larger models can in theory do a better job of modelling the data, in practice a model that is too large will easily overfit to the training data and will struggle to generalise to unseen data.



(a) One hidden layer with 4 units.

(b) Two hidden layers with 4 units each.

(c) One hidden layer with 16 units.

(d) Two hidden layers with 16 units each.

Figure 4.7: Comparison of the decision boundaries for different neural network architectures.

4.3 Neural Language Models

We will now look at how to use neural networks as language models. We will once again use the probability chain rule to simplify this to the task of repeatedly predicting the next word w_t in the sentence given some sequence $w_{t-1}, w_{t-2}, \ldots$ of previous words:

$$\mathbb{P}(w_1, \ldots, w_t) = \mathbb{P}(w_1)\,\mathbb{P}(w_2|w_1)\ldots\mathbb{P}(w_t|w_{t-1}, \ldots, w_1)\,.$$

Note that feedforward neural networks as defined in Section 4.2 require their inputs to have a fixed length, so cannot handle an arbitrarily long list of context words. A simple solution to this is to consider only the most recent $n-1$ words when estimating the probability distribution of w_t [44]. This is also the simplification made by n-gram models, so as in Section 2.2 the probability of some sequence $w_1, \ldots, w_t$ of words will be approximated by:

$$\mathbb{P}(w_1, \ldots, w_t) \approx \prod_{i=1}^{t} \mathbb{P}\left(w_i|w_{i-1}, \ldots, w_{i-(n-1)}\right), \tag{4.8}$$

where $\mathbb{P}\left(w_i|w_{i-1}, \ldots, w_{i-(n-1)}\right)$ is estimated by the output of a neural network classifier. The key difference between the n-gram and neural approaches is that we will train the neural network using the **word embeddings** introduced in Chapter 3. This helps avoid the data sparsity issues that cause problems for n-gram models (Section 2.2.2), since the neural model will be able to take advantage of word similarities.

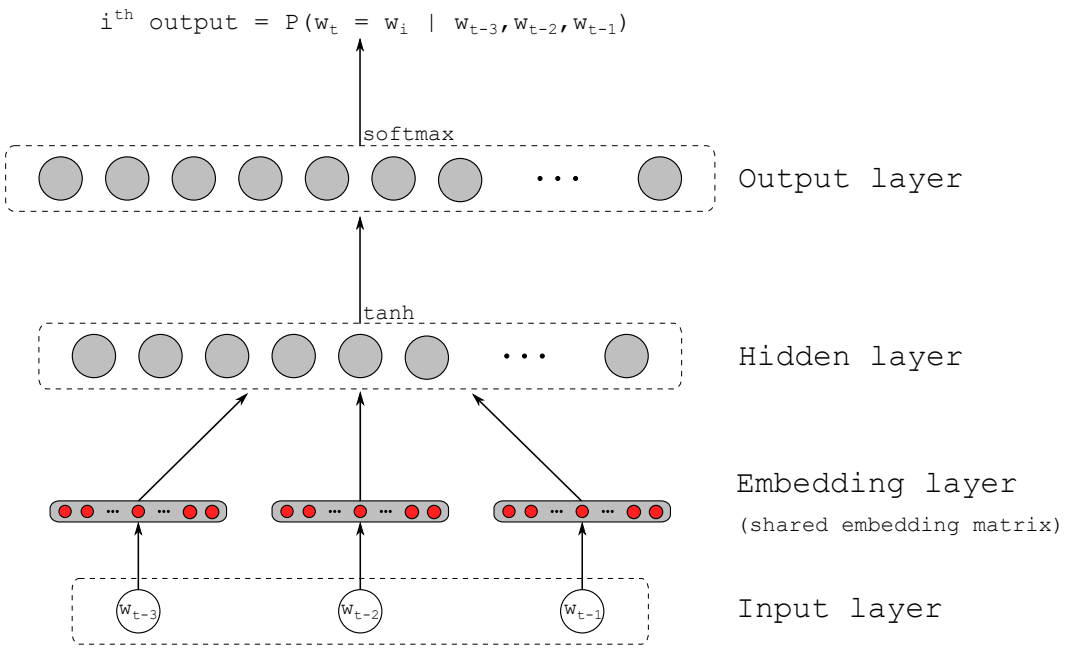


Figure 4.8: Structure of the neural language model for $n = 4$.

We can represent a set of word embeddings of dimension m with a matrix $E \in \mathbb{R}^{m \times |V|}$, where the i^{th} column of E contains the embedding $\boldsymbol{v}_{w_i} \in \mathbb{R}^m$ of the word $w_i \in V$. If we represent each word w_i as a one-hot vector in $\mathbb{R}^{|V|}$ then we can use matrix multiplication to extract the embedding of w_i from E, since $\boldsymbol{v}_{w_i} = E\,\boldsymbol{e}_i$, where $\boldsymbol{e}_i$ is the basis vector with a 1 in coordinate i.

This means that looking up an embedding is equivalent to feeding a one-hot vector into a neural network layer with a bias vector of zero and the identity function as its activation. Such a layer is called an **embedding layer**, and we can include its weight matrix E with the other trainable parameters of the network. For example, the skip-gram model described in Section 3.5 can be seen as a neural network classifier consisting of an embedding layer followed by a dense output layer with a bias vector of 0.

Putting all of this together, we can build the neural language model [44] represented in Figure 4.8, with the dense connections between layers represented as a single arrow for readability. This model performs the following computations in its forward pass:

- **Input**: $n-1$ one-hot vectors, representing the sequence $w_{t-(n-1)}, \ldots, w_{t-1}$ of context words.
- **Embedding layer**: extract the embeddings $\boldsymbol{v}_{w_{t-(n-1)}}, \ldots, \boldsymbol{v}_{w_{t-1}} \in \mathbb{R}^m$ for the $n-1$ context words from the matrix E. The number of trainable parameters in this layer is $m|V|$.
- **Flatten**: concatenate the $n-1$ embeddings into one vector, $\boldsymbol{x} \in \mathbb{R}^{m(n-1)}$.
- **Dense layer**: fully connected layer with d_h neurons and tanh activation. This layer computes $\boldsymbol{h} = \tanh(\boldsymbol{b} + W\boldsymbol{x}) \in \mathbb{R}^{d_h}$ and has $((n-1)m+1)d_h$ trainable parameters.

- **Output layer**: fully connected layer with $|V|$ neurons and softmax activation. The output is $\text{softmax}(\boldsymbol{s} + U\boldsymbol{h}) \in \mathbb{R}^{|V|}$, a probability distribution over all of the possibilities for the next token in the sequence, w_t. The i^{th} output of this layer is $\hat{\text{P}}(w_i|w_{i-n+1}, \ldots, w_{i-1})$, an estimate of the probability that $w_t = w_i$. This layer has $(d_h + 1)|V|$ trainable parameters.

Overall this model has $m|V| + ((n-1)m+1)d_h + (d_h+1)|V|$ parameters, while the equivalent n-gram model has $|V|^n$. This is linear in both n and $|V|$, resulting in a model that is less prone to overfitting than n-gram models, and can deal with much larger values for n. The embedding matrix E can be initialised randomly and trained from scratch alongside the rest of the network, however we can often improve performance by initialising E with a set of pre-trained word embeddings. These can be obtained very efficiently using the skip-gram algorithm described in Section 3.5.

Note that the loss function for this model has a close relationship with the **perplexity** metric defined in Section 2.2.5. For a dataset consisting of a sequence $w_1, \ldots, w_N$, the cross-entropy loss is:

$$L_{CE}(\boldsymbol{w}) = -\frac{1}{N}\sum_{i=1}^{N} \log \hat{\text{P}}(w_i|w_{i-n+1}...w_{i-1})$$

$$= \log\left\{\left(\prod_{i=1}^{N} \hat{\text{P}}(w_i|w_{i-n+1}...w_{i-1})\right)^{-\frac{1}{N}}\right\}$$

$$= \log(\text{PP}(\boldsymbol{w}))$$

This means that having calculated the loss L_{CE} of the model on a dataset we can simply obtain its perplexity as $\exp\{L_{CE}\}$. This is the method used to compute the perplexity of the models in the following section for comparison with the n-gram models trained in Section 2.2.6.

4.3.1 Sherlock Holmes Data

We will now train a neural language model on the Sherlock Holmes corpus, using the architecture described in the previous section. We will follow some of the hyperparameter suggestions of Bengio et al. [44], using n-grams of size $n = 5$ and setting the hidden layer size to $d_h = 100$. Bengio et al. train their word embeddings from a random initialisation, but suggest that performance might be improved with a knowledge based initialisation - as such, we will compare the performance of several different initialisation strategies:

- Random ($m = 30$), random ($m = 300$): 30- and 300-dimensional word embeddings initialised randomly.

- Sherlock ($m = 30$), Sherlock ($m = 300$): 30 and 300-dimensional word embeddings initialised with a set of embeddings trained using skip-gram with negative sampling on the Sherlock training dataset.

Model initialisation	Train perplexity	Best test perplexity	Time per epoch
Random ($m = 30$)	72	132	161s
Random ($m = 300$)	33	132	528s
Sherlock ($m = 30$)	66	126	**149s**
Sherlock ($m = 300$)	34	129	534s
Google News ($m = 300$)	**31**	**124**	521s

Table 4.1: Perplexities on the test and train datasets for the neural models.

- Google News ($m = 300$): 300-dimensional word embeddings initialised with the Google News set of pre-trained word vectors [25]. This is a set of word embeddings published by Google, trained using skip-gram on a corpus of news articles with a total of around 1 billion words. These were included to analyse the impact of using embeddings pre-trained on a larger dataset.

The corpus was preprocessed in the same way as for the n-gram models in Section 2.2.6 - sentences were padded with $n-1$ start tokens and a sentence end token, and words occurring less than twice in training data were removed from the vocabulary and replaced with `<UNK>`. The test/train split was the same as the one used to evaluate the n-gram models.

The dataset was generated by extracting all of the n-grams from the corpus and using the first $n-1$ tokens in each n-gram as the model input, with the n^{th} as the 'true' label in the classification task. All hidden layers and output layers were initialised with random parameters. The starting loss was around 9.22 for all models - this corresponds to a perplexity of around 10100, which was equal to the vocabulary size. This is exactly what one would expect from a random language model, since it corresponds to the perplexity of the uniform language model (Section 2.2.5).

The models were implemented in TensorFlow [43], and Gensim [36] was used to train or download the pre-trained word embeddings. Adam with the default hyperparameters was used for optimisation - this significantly sped up training, since all models reached their best test accuracy within 5 epochs. Other optimisers were tested for comparison, however they failed to converge even after 100 epochs of training.

Figure 4.9 shows plots of train and test loss against the number of training epochs. It also includes plots of loss against the amount of time spent training - this is the wall clock time elapsed during training, as reported by TensorFlow. The time required to pre-train the word embeddings on the training dataset was very small compared to the time spent training the full model - around 8s for $m = 30$ and 22s for $m = 300$. The perplexity reached on the training data after 5 epochs, as well as the lowest perplexity achieved on the test data, are reported for each model in Table 4.1.

All models outperformed the n-gram models in Section 2.2.6, where the lowest test perplexity of 237 was achieved by the smoothed bigram model. All models had started to overfit to the training data by the end of the 5 epochs, as evidenced by the test loss starting to rise again in the plots. Training could have been stopped automatically before this point by using a validation dataset to determine when the model started to overfit - this is known as **early stopping** [33].

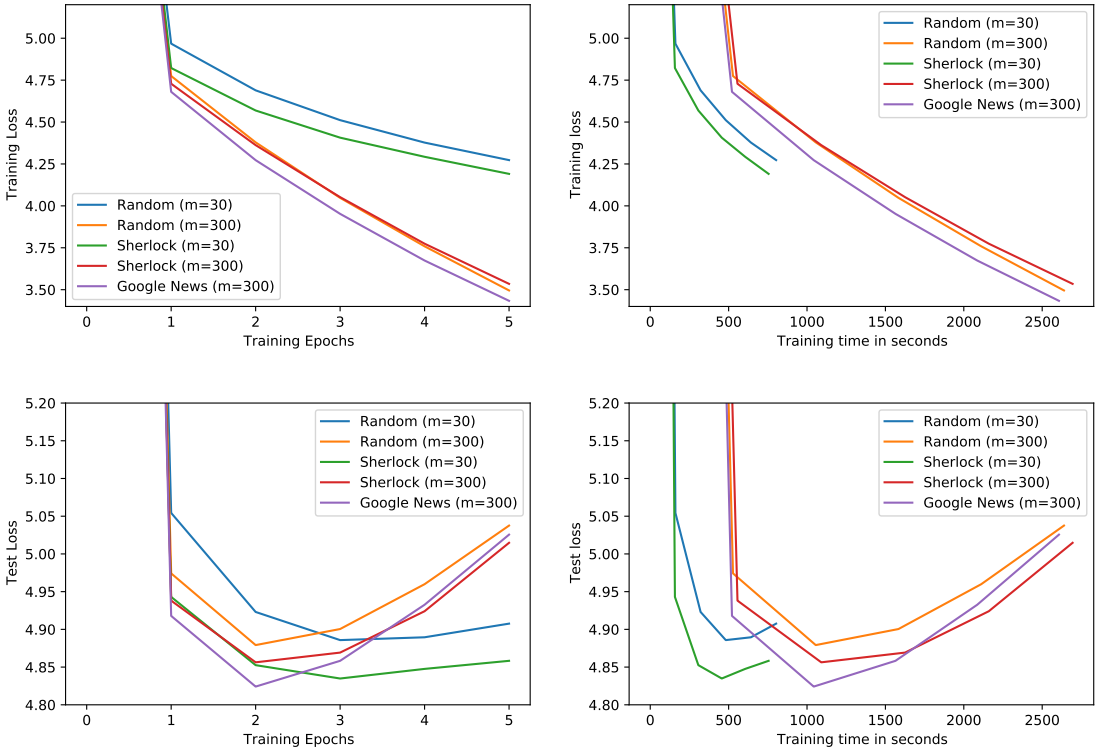


Figure 4.9: Plots showing the changes in loss on the test and train datasets during training.

The models with higher dimensional embeddings achieved far lower perplexity on the training data, but performed similarly to the lower dimensional embeddings on the test data. The key difference is that they took far longer to train - in situations where computation power or time is limited, the best option is to use low dimensional embeddings initialised with an efficient method like skip-gram with negative sampling.

Generally, models initialised with pre-trained word vectors outperformed those with random initialisations, converging faster on the training data and generalising better to the test data. The lowest train and test perplexity was achieved by initialising the embeddings with the Google News vectors - using them allowed the model to take advantage of this much larger dataset. However, the difference was not as big as one might expect - this is likely because the Google News vectors were trained on modern American English articles, so the word meanings were slightly different and many words (such as place names and words with English spellings) were missing from the vocabulary and their embeddings had to be trained from scratch.

Despite having fewer parameters than the equivalent n-gram model and being trained on the same dataset, the neural models were more effective but took considerably longer to train. The n-gram models in Section 2.2.6 took less than a minute to train, while the neural models each took 10-45 minutes. Computing probabilities with the neural models was also much slower - it took around a minute to evaluate the loss on the test and train datasets, compared with a few seconds for the n-gram models. This is because calculating probabilities using an n-gram model simply involves looking up each n-gram probability, while a forward pass of the neural network is much more computationally expensive.

As in Section 2.2.6, we can gain some intuition for how the models view language by using them to generate sentences. One way of doing this is to start with $n-1$ padding tokens and iteratively sample the next word in the sentence from the distribution defined by the model's output given the last $n-1$ tokens. This is repeated until a sentence end token is generated. Some sentences generated by the model with Google News initialisation are:

```
"I'll be happy to read it to you and so that one has said against him."
 "I helped myself in a chair pistol lay I closed the door behind him."
```

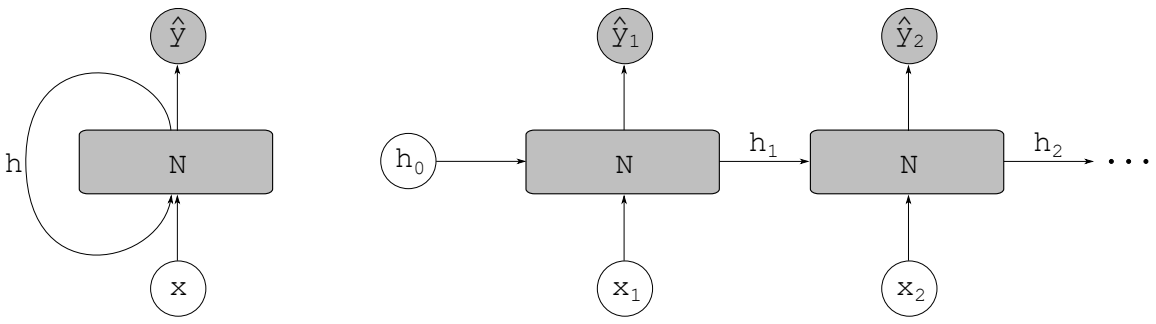
These sentences are quite close to making sense, and demonstrate some understanding of grammar rules and word meanings, although the model's memory of 4 words does not allow it to remember how it started the sentence. This technique can also be employed to use the model to complete sentences. For example, given the sentence beginning `"<s> the murderer is"` the model generated the sentence `"The murderer is against the law."`.

4.4 Recurrent Neural Networks

We have seen how we can use feedforward neural networks to incorporate word embeddings into language models, but this model suffers from the same limited context restriction as an n-gram model. However, we can modify the neural network architecture to enable it to handle arbitrarily long sequences of words.

One way of doing this is to use a **recurrent neural network** (RNN) [33]. RNNs are a family of neural networks designed to process sequences element by element - given a sequence $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}$ of input data, the $\boldsymbol{x}^{(t)}$ are fed into the network one at a time. Unlike the feedforward networks we have seen so far, RNNs allow cycles in their structure - this means that the output $\hat{\boldsymbol{y}}^{(t)}$ of the network at time t depends not only on $\boldsymbol{x}^{(t)}$ but also on the internal state of the network at time $t-1$.

We can think of this as the network continually updating and passing along a context vector $\boldsymbol{h}^{(t)}$ representing information about the sequence up to time t. The dependence of the network's output on this context vector can be represented compactly as a cycle (Figure 4.10a), but it can be visualised more intuitively by 'unrolling' the RNN as in Figure 4.10b.



(a) A simple RNN.

(b) The same RNN unrolled.

Figure 4.10: RNN architecture.

We can use this idea to define a recurrent neural language model [45]. Given a sequence of words $w_1, \ldots, w_T$, let $\boldsymbol{x}^{(t)} \in \mathbb{R}^{|V|}$ denote the one-hot vector representation of w_t. At each time step $t \in \{1, \ldots, T\}$ our aim is to estimate the probability distribution of w_{t+1} given $w_1, \ldots, w_t$. To do this, the network performs the following updates:

$$\begin{aligned}\boldsymbol{h}^{(t)} &= N(\boldsymbol{x}^{(t)}, \boldsymbol{h}^{(t-1)}) = \sigma(W\boldsymbol{x}_t + V\boldsymbol{h}^{(t-1)})\,, \\ \hat{\boldsymbol{y}}^{(t)} &= \text{softmax}(U\boldsymbol{h}^{(t)})\,.\end{aligned}$$

Here, the parameters of the network are the matrices $W \in \mathbb{R}^{d_h \times |V|}, V \in \mathbb{R}^{d_h \times d_h}$, and $U \in \mathbb{R}^{|V| \times d_h}$. Note that this network has no bias vectors - Mikolov [46] chose to omit them in order to simplify the model, having observed that including them as trainable parameters had no significant impact on performance. This is equivalent to simply fixing all of the bias parameters in the network to zero. Since $\boldsymbol{x}^{(t)}$ is a one-hot vector, the matrix multiplication $W\boldsymbol{x}^{(t)}$ simply extracts a column of W - this means that we can see W as a matrix of word embeddings, similar to the embedding layer in the feedforward neural language model in Section 4.3.

The RNN also requires an initial context vector $\boldsymbol{h}^{(0)}$ - this can be set to 0, or included as a trainable parameter. Training $\boldsymbol{h}^{(0)}$ can be beneficial when the word sequences in the training corpus are relatively short, for example if the dataset consists of tweets. This is because $\boldsymbol{h}^{(0)}$ usually only has a significant impact on the first few network outputs.

Notice that a given output $\hat{\boldsymbol{y}}^{(t)}$ can be expressed as the output of a deep feedforward network where the hidden layers share the same parameters - this is easier to see in Figure 4.10b, where we can see that t copies of the hidden layer N are required to calculate $\hat{\boldsymbol{y}}^{(t)}$. This parameter sharing allows the RNN model to have a longer memory than the feedforward model in Section 4.3, which must learn a separate set of parameters in its hidden layer for each position in the input history. It makes some intuitive sense to share parameters in this way: it is not the exact position of a word in the sequence that is important, but rather the order of the words around it. For example, the phrase `"I hated this film"` in a film review indicates a negative opinion regardless of where exactly in the review it appears.

Considering the unrolled RNN as a special feedforward network allows us to apply backpropogation (Section 4.2.2) in the same way as for a normal feedforward network. This process is known as **backpropogation through time** [35]. Since RNNs produce a sequence of outputs there are several different ways to compute the loss, depending on the purpose of the RNN. For language modelling, where the classification problem is next word prediction, there is an obvious true label to compare to each output in the sequence: the word that actually occurred next in the training data. In this case, we would use all of the outputs of the RNN to compute the loss. However, if the RNN is to be used for a classification task like sentiment analysis, then we might only use the final output of the RNN since we want a classification decision based on the text as a whole.

The language models we have seen so far have all been word-based, with the classification task being to predict the next word in a sequence. Considering words to be the building blocks of language is an intuitive approach to take, but it is not the only one. It is also possible to define **character level** language models, where we instead define the classification task to be next character prediction. This is often not a practical approach to take for n-gram based models, since next character prediction clearly has very long range dependencies. For example, to understand spelling a character based model must look as far

back as the beginning of the current word, and understanding dependencies between words requires a very long memory in characters. However, RNNs can be very effective at next character prediction since they can capture these long range dependencies [47]. It is very easy to adapt the architecture above for character level language modelling - we can simply define the vocabulary to be a set of characters, including punctuation, spaces and special characters. This approach can be used to generate computer code, for example, since this is an application where words are not the key building blocks.

4.5 Discussion

This chapter introduced neural networks, a powerful class of models that can solve many of the issues raised in previous chapters. However, in moving towards more complicated models we have sacrificed some of the qualities that make simpler models easy to understand and interpret - it is easy to interpret the parameters of logistic regression and analyse their influences on the final classification decision (see Section 3.4), but this is much harder for neural networks. It is possible to make some deductions about the influence of individual neurons by looking at their activations while processing a dataset. For example, in character level RNNs it is possible to identify neurons that are responsible for keeping track of whether speech marks or parentheses are open, since they only have high activations when processing characters inside quotes or brackets [47]. However, many of the features extracted by neural networks do not have a clear interpretation. Even if the network performs well on a test dataset, this does not guarantee that the features it is extracting are truly useful or match up to our intuition of what features are relevant for the task.

We can illustrate this issue by generating **adversarial examples**. This is a process where an existing member of the dataset that is correctly classified by the model is transformed slightly in such a way that its true label does not change, but the model misclassifies it. We encountered an example of this for the naive Bayes classifier in Section 2.1.2, where using Alfred Hitchcock's full name caused the classifier to misclassify a negative review as positive despite the fact that it didn't change the meaning of the sentence.

Adversarial examples generated for neural networks show that things can go wrong in surprising ways. Some of the clearest examples of this are in image processing - for example, it is possible to fool state-of-the-art image classifiers by changing only a single pixel of the input image [48]. This is a change that is almost imperceptible to the human eye but causes the classifier to make incorrect predictions with a high degree of confidence - this shows that the network's 'reasoning' does not match up with our intuitions for this task. This is not a purely theoretical concern - it is possible to create physical objects that are adversarial examples even when photographed from different angles [49]. It is also possible to design adversarial examples for deep natural language models: for example, Alzantot et al. [50] generated adversarial reviews for an RNN trained for sentiment analysis on the IMDb film review dataset by changing a handful of words to close synonyms.

The question of how to make neural networks robust to this kind of attack is currently an open problem - adding the adversarial examples to the training dataset can help in some cases, and improving model interpretability is a key issue in increasing trust in neural networks' decisions. This is a serious concern for real world applications where there is an incentive to mislead the model - for example, there is a huge financial incentive to fool models responsible for detecting money laundering.

Chapter 5

Conclusion

5.1 Summary

In this report, we looked at several key issues in language modelling and the effectiveness of different approaches in handling them. We started off by introducing n-gram language models. This is a simple and somewhat intuitive approach, where we estimate the likelihood of the next word in a sequence by counting the number of times it appeared in that context in the training dataset.

These models have the advantage of being easy to fit and easy to interpret, however are limited by the fact that they learn about each word independently of the others in the vocabulary. Since they are unable to take advantage of word similarities, they suffer greatly from the sparsity issue inherent in language modelling: due to the large vocabulary and Zipfian distribution of words, most n-grams are observed rarely and the majority are never seen. This restricts the length of the word history that n-gram models can take into account - in Section 2.2.6, we saw that n-grams with add-α smoothing performed very badly with histories longer than one word. While we can improve performance by using information from lower order n-grams via interpolation or backoff, in practice this only allows us to increase the word history's length to two. This is good enough for some applications - n-gram models are used in statistical machine translation [16], and we saw in Section 2.1.2 that even the unigram model can be useful in classification.

The key trick to training better language models is to learn useful representations of words - in Chapter 3, we saw that the skip-gram algorithm can do this efficiently in a self-supervised manner, successfully capturing intuitive word similarities. In Chapter 4, we saw how these embeddings can be used in neural language models to help overcome the data sparsity and improve generalisation. We saw in Section 4.3.1 that this approach significantly outperformed the n-gram models trained in Section 2.2.6, allowing us to train a model that could handle histories of length 4. We also introduced recurrent neural network models, which can use histories of arbitrary length.

We saw that neural models are more powerful than the models introduced previously, but are also more difficult to train and harder to interpret. However, it is important to avoid treating them as black boxes - they tend to pick up biases in their training data, and despite their power they are still vulnerable to surprising adversarial attacks.

5.2 Further Work

This section will discuss some possible extensions to the work in this report, as well as touching on some state-of-the-art techniques in natural language processing.

It would have been interesting to train a word-based RNN model on the Sherlock Holmes corpus to compare performance with the limited context models. However, in practice, the basic RNN model defined in Section 4.4 will still struggle to capture very long range dependencies - this is due to an issue known as the **vanishing gradients problem** [41], where the computed gradients quickly tend to zero during backpropogation so that earlier inputs have a very small impact on parameter updates.

One solution to this problem is to use the **Long Short-Term Memory** (LSTM) [15] RNN architecture - as well as passing on a hidden state at each iteration, an LSTM adds a **memory cell** $\boldsymbol{c}_t$ that serves as the model's long term memory. $\boldsymbol{c}_t$ is updated as a gated sum of $\boldsymbol{c}_{t-1}$ and an update computed from the current input $\boldsymbol{x}_t$ and previous hidden state $\boldsymbol{h}_{t-1}$. The LSTM uses several gates (which can be seen as smooth versions of logic gates like `AND` or `OR`) to determine when new information should be added to or forgotten from $\boldsymbol{c}_t$, and to decide what information should be passed on through $\boldsymbol{h}_t$.

We saw in Section 4.3.1 that the performance of the Sherlock Holmes neural models could be improved by initialising the embedding layer with pre-trained word vectors, and in particular by initialising the model with embeddings pre-trained on a larger dataset. A key issue in this fine-tuning approach is that the model can forget useful information learned during pre-training, a problem known as **catastrophic forgetting**. One solution to this issue is to adjust the learning rate for pre-trained parameters [37], since using a lower learning rate prevents them from moving too far from their initialisation. It would have been interesting to see if changing the learning rate for the pre-trained embeddings in Section 4.3.1 improved performance, particularly for the embeddings pre-trained on the larger Google News dataset.

This idea of using pre-trained parameter initialisations is the key trick behind much of the current state of the art in natural language processing, since it allows complex models trained on very large datasets to be adapted for use with smaller datasets. For example, in Section 3.6 we touched on the idea of contextual word embeddings, with BERT as an example. BERT is a very versatile tool that is designed to be fine-tuned for natural language tasks by adding a task-specific output layer - for example, BERT can be used for classification by padding the input text with a classification token and passing its contextual embedding to a classification output layer that is learned from scratch. Sun et al. [51] achieved 95.8% accuracy on the IMDb sentiment analysis dataset by fine-tuning BERT in this way, using a small learning rate to avoid catastrophic forgetting.

One model commonly considered to be the current state of the art in language modelling is GPT-3 [52], a neural model that owes much of its success to its enormous size (175 billion parameters) as well as the size and diversity of its training data. GPT-3 is capable of generating text that is almost indistinguishable from text written by humans, as well as being able to adapt to new tasks given only a handful of examples. This is known as **few-shot learning**, and the approach taken by GPT-3 actually requires no parameter updates - a very short task description, example, and prompt can simply be input to the model which then completes the prompt. Such tasks include generating an article given a title and subtitle, or using a new word in a sentence given a definition of the word.

Bibliography

[1] A. C. Doyle. *A Study in Scarlet*. Project Gutenberg, 1995. Retrieved February 2021, from `https://www.gutenberg.org/ebooks/244`.

[2] A. C. Doyle. *The Sign of the Four*. Project Gutenberg, 2000. Retrieved February 2021, from `https://www.gutenberg.org/ebooks/2097`.

[3] A. C. Doyle. *The Hound of the Baskervilles*. Project Gutenberg, 2001. Retrieved February 2021, from `https://www.gutenberg.org/ebooks/2852`.

[4] A. C. Doyle. *The Valley of Fear*. Project Gutenberg, 2002. Retrieved February 2021, from `https://www.gutenberg.org/ebooks/3776`.

[5] A. C. Doyle. *The Adventures of Sherlock Holmes*. Project Gutenberg, 1999. Retrieved February 2021, from `https://www.gutenberg.org/ebooks/1661`.

[6] A. C. Doyle. *The Return of Sherlock Holmes*. Project Gutenberg, 2006. Retrieved February 2021, from `https://www.gutenberg.org/ebooks/108`.

[7] A. C. Doyle. *The Memoirs of Sherlock Holmes*. Project Gutenberg, 1997. Retrieved February 2021, from `https://www.gutenberg.org/ebooks/834`.

[8] A. C. Doyle. *His Last Bow: An Epilogue of Sherlock Holmes*. Project Gutenberg, 2000. Retrieved February 2021, from `https://www.gutenberg.org/ebooks/2350`.

[9] C. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[10] D. Jurafsky and J. H. Martin. *Speech and Language Processing*. Prentice Hall, 3rd edition. December 2020 draft accessed at `https://web.stanford.edu/~jurafsky/slp3`.

[11] A. Ng and M. Jordan. On Discriminative vs. Generative Classifiers: A comparison of logistic regression and naive Bayes. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, volume 14. MIT Press, 2002.

[12] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

[13] TensorFlow Datasets, a collection of ready-to-use datasets. `https://www.tensorflow.org/datasets`.

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[15] J. Eisenstein. *Introduction to Natural Language Processing*. MIT Press, 2019. URL `https://github.com/jacobeisenstein/gt-nlp-class/blob/master/notes`.

[16] P. Koehn. *Statistical Machine Translation*. Cambridge University Press, 2009.

[17] W. A. Gale and K. W. Church. What's wrong with adding one? In *Corpus-Based Research into Language*, pages 189–198. Rodolpi, 1994.

[18] T. Hastie, R. J. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning*. Springer, 2nd edition, 2017.

[19] F. Jelinek and R. L. Mercer. Interpolated estimation of Markov source parameters from sparse data. In *Proceedings of the Workshop on Pattern Recognition in Practice*, pages 381–397. North Holland, 1980.

[20] S. M. Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Trans. Acoust. Speech Signal Process.*, 35:400–401, 1987.

[21] S. F. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. *Computer Speech and Language*, 13(4):359–394, 1999.

[22] N. Smith. Contextual word representations: Putting words into computers. *Communications of the ACM*, 63(6):66–74, 2020.

[23] T. Mikolov, W. Yih, and G. Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Atlanta, Georgia, 2013. Association for Computational Linguistics.

[24] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. In *ICLR*, 2013.

[25] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

[26] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning*. Springer, 2013.

[27] A. Cauchy. Méthode générale pour la résolution des systèmes d'équations simultanées. *Comptes rendus de l'Académie des Sciences*, 25:536–538, 1847.

[28] D. Bertsekas. *Nonlinear programming*. Athena Scientific, 2nd edition, 1999.

[29] H. Robbins and S. Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

[30] L. Bouttou, F. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311, 2018.

[31] L. Bottou. Online algorithms and stochastic approximations. In *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK, 1998. revised, oct 2012.

[32] S. Ruder. An overview of gradient descent optimization algorithms. 2017. arXiv preprint arXiv:1609.04747.

[33] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[34] A. Agresti. *Categorical Data Analysis*. Wiley Series in Probability and Statistics Ser. John Wiley & Sons, Incorporated, 2nd edition, 2002.

[35] Y. Goldberg. A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research*, 57, 2015.

[36] R. Řehůřek and P. Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

[37] S. Ruder, M. E. Peters, S. Swayamdipta, and T. Wolf. Transfer learning in natural language processing. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Tutorials*, pages 15–18, 2019. Expanded version: `https://ruder.io/state-of-transfer-learning-in-nlp/`.

[38] T. Mikolov, Q. Le, and I. Sutskever. Exploiting similarities among languages for machine translation. arXiv preprint arXiv:1309.4168.

[39] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, 2019. Association for Computational Linguistics.

[40] T. Bolukbasi, K.-W. Chang, J. Y. Zou, V. Saligrama, and A. T. Kalai. Man is to computer programmer as woman is to homemaker? Debiasing word embeddings. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[41] K. Cho. Natural language understanding with distributed representation, 2015. arXiv preprint arXiv:1511.07916.

[42] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2014. arXiv preprint arXiv:1412.6980.

[43] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from `https://www.tensorflow.org/`.

[44] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003.

[45] T. Mikolov, M. Karafiát, L. Burget, J. Cernocký, and S. Khudanpur. Recurrent neural network based language model. In *Proceedings of the 11th Annual Conference of the International Speech Communication Association, INTERSPEECH 2010*, volume 2, pages 1045–1048, 2010.

[46] T. Mikolov. *Statistical language models based on neural networks.* PhD thesis, Brno University of Technology, 2012.

[47] A. Karpathy. The unreasonable effectiveness of recurrent neural networks, 2015. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`.

[48] J. Su, D. Vargas, and K. Sakurai. One pixel attack for fooling deep neural networks. *IEEE Transactions on Evolutionary Computation*, 23(5):828–841, 2019.

[49] A. Athalye, L. Engstrom, A. Ilyas, and K. Kwok. Synthesizing robust adversarial examples. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 284–293, 2018.

[50] M. Alzantot, Y. Sharma, A. Elgohary, B. Ho, M. Srivastava, and K. Chang. Generating natural language adversarial examples. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2890–2896, Brussels, Belgium, 2018. Association for Computational Linguistics.

[51] C. Sun, X. Qiu, Y. Xu, and X. Huang. How to fine-tune BERT for text classification? In M. Sun, X. Huang, H. Ji, Z. Liu, and Y. Liu, editors, *Chinese Computational Linguistics*, pages 194–206. Springer, Cham, 2019.

[52] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.