Deep Learning for Natural Language Processing: A Gentle Introduction

Mihai Surdeanu and Marco A. Valenzuela-Escárcega
Department of Computer Science
University of Arizona

October 22, 2021

Contents

Preface

An obvious question that may pop up when seeing this material is: ”Why another deep learning and natural language processing book?” Several excellent ones have been published, covering both theoretical and practical aspects of deep learning and its application to language processing. However, from my experience teaching courses on natural language processing, I argue that, despite their excellent quality, most of these books do not target their most likely readers. The intended reader of this book is one who is skilled in a domain other than machine learning and natural language processing and whose work relies, at least partially, on the automated analysis of large amounts of data, especially textual data. Such experts may include social scientists, political scientists, biomedical scientists, and even computer scientists and computational linguists with limited exposure to machine learning.

Existing deep learning and natural language processing books generally fall into two camps. The first camp focuses on the theoretical foundations of deep learning. This is certainly useful to the aforementioned readers, as one should understand the theoretical aspects of a tool before using it. However, these books tend to assume the typical background of a machine learning researcher and, as a consequence, I have often seen students who do not have this background rapidly get lost in such material. To mitigate this issue, the second type of book that exists today focuses on the machine learning practitioner; that is, on how to use deep learning software, with minimal attention paid to the theoretical aspects. I argue that focusing on practical aspects is similarly necessary but not sufficient. Considering that deep learning frameworks and libraries have gotten fairly complex, the chance of misusing them due to theoretical misunderstandings is high. I have commonly seen this problem in my courses, too.

This book, therefor, aims to bridge the theoretical and practical aspects of deep learning for natural language processing. I cover the necessary theoretical background and assume minimal machine learning background from the reader. My aim is that anyone who took linear algebra and calculus courses will be able to follow the theoretical material. To address practical aspects, this book includes pseudo code for the simpler algorithms discussed and actual Python code for the more complicated architectures. The code should be understandable by anyone who has taken a Python programming course. After reading this book, I expect that the reader will have the necessary foundation to immediately begin building real-world, practical natural language processing systems, and to expand their knowledge by reading research publications on these topics.

Acknowledgments

Your acknowledgments: TODO: Thank you to all the people who helped!

Mihai and Marco

1 Introduction

Machine learning (ML) has become a pervasive part of our lives. For example, Pedro Domingos, a machine learning faculty member at University of Washington, discusses a typical day in the life of a 21st century person, showing how she is accompanied by machine learning applications throughout the day from early in the morning (e.g., waking up to music that the machine matched to her preferences) to late at night (e.g., taking a drug designed by a biomedical researcher with the help of a robot scientist) [Domingos 2015].

Natural language processing (NLP) is an important subfield of ML. As an example of its usefulness, consider that PubMed, a repository of biomedical publications built by the National Institutes of Health,[1] has indexed more than one million research publications *per year* since 2010 [Vardakas et al. 2015]. Clearly, no human reader (or team of readers) can process so much material. We need machines to help us manage this vast amount of knowledge. As one example out of many, an inter-disciplinary collaboration that included our research team showed that machine reading discovers an order of magnitude more protein signaling pathways[2] in biomedical literature than exist today in humanly-curated knowledge bases [Valenzuela-Escárcega et al. 2018]. Only 60 to 80% of these automatically-discovered biomedical interactions are correct (a good motivation for *not* letting the machines work alone!). But, without NLP, all of these would remain "undiscovered public knowledge" [Swanson 1986], limiting our ability to understand important diseases such as cancer. Other important and more common applications of NLP include web search, machine translation, and speech recognition, all of which have had a major impact in almost everyone's life.

Since approximately 2014, the "deep learning tsunami" has hit the field of NLP [Manning 2015] to the point that, today, a majority of NLP publications use deep learning. For example, the percentage of deep learning publications at four top NLP conferences has increased from under 40% in 2012 to 70% in 2017 [Young et al. 2018]. There is good reason for this domination: deep learning systems are relatively easy to build (due to their modularity), and they perform better than many other ML

[1] https://www.ncbi.nlm.nih.gov/pubmed/

[2] Protein signaling pathways "govern basic activities of cells and coordinate multiple-cell actions". Errors in these pathways "may cause diseases such as cancer". See: https://en.wikipedia.org/wiki/Cell_signaling

methods.[3] For example, the site nlpprogress.com, which keeps track of state-of-the-art results in many NLP tasks, is dominated by results of deep learning approaches.

This book explains deep learning methods for NLP, aiming to cover both theoretical aspects (e.g., how do neural networks learn?) and practical ones (e.g., how do I build one for language applications?).

The goal of the book is to do this while assuming minimal technical background from the reader. The theoretical material in the book should be completely accessible to the reader who took linear algebra, calculus, and introduction to probability theory courses, or who is willing to do some independent work to catch up. From linear algebra, the most complicated notion used is matrix multiplication. From calculus, we use differentiation and partial differentiation. From probability theory, we use conditional probabilities and independent events. The code examples should be understandable to the reader who took a Python programming course.

Starting nearly from scratch aims to address the background of what we think will be the typical reader of this book: an expert in a discipline other than ML and NLP, but who needs ML and NLP for her job. There are many examples of such disciplines: the social scientist who needs to mine social media data, the political scientist who needs to process transcripts of political discourse, the business analyst who has to parse company financial reports at scale, the biomedical researcher who needs to extract cell signaling mechanisms from publications, etc. Further, we hope this book will also be useful to computer scientists and computational linguists who need to catch up with the deep learning wave. In general, this book aims to mitigate the impostor syndrome [Dickerson 2019] that affects many of us in this era of rapid change in the field of machine learning and artificial intelligence (this author certainly has suffered and still suffers from it![4]).

1.1 What this Book Covers

This book interleaves chapters that discuss the theoretical aspects of deep learning for NLP with chapters that focus on implementing the previously discussed theory. For the implementation chapters we will use PyTorch, a deep learning library that is well suited for NLP applications.[5]

Chapter 2 begins the theory thread of the book by attempting to convince the reader that machine learning is easy. We will use a children's book to introduce key ML concepts, including our first learning algorithm. From this example, we will start building several basic neural networks. In the same chapter, we will formalize the

[3] However, they are not perfect. See Section 1.3 for a discussion.

[4] Even the best of us suffer from it. Please see Kevin Knight's description of his personal experience involving tears (not of joy) in the introduction of this tutorial [Knight 2009].

[5] https://pytorch.org

perceptron algorithm, the simplest neural network. In Chapter 3, we will transform the perceptron into a logistic regression network, another simple neural network that is surprisingly effective for NLP. In Chapters 5 and 6 we will generalize these algorithms into feed forward neural networks, which operate over arbitrary combinations of artificial neurons.

The astute historian of deep learning will have observed that deep learning had an impact earlier on image processing than on NLP. For example, in 2012, researchers at University of Toronto reported a massive improvement in image classification when using deep learning [Krizhevsky et al. 2012]. However, it took more than two years to observe similar performance improvements in NLP. One explanation for this delay is that image processing starts from very low-level units of information (i.e., the pixels in the image), which are then hierarchically assembled into blocks that are more and more semantically meaningful (e.g., lines and circles, then eyes and ears, in the case of facial recognition). In contrast, NLP starts from words, which are packed with a lot more semantic information than pixels and, because of that, are harder to learn from. For example, the word *house* packs a lot of common-sense knowledge (e.g., houses generally have windows and doors and they provide shelter). Although this information is shared with other words (e.g., *building*), a learning algorithm that has seen *house* in its training data will not know how to handle the word *building* in a new text to which it is exposed after training.

Chapter 8 addresses this limitation. In it, we discuss word2vec, a method that transforms words into a numerical representation that captures (some) semantic knowledge. This technique is based on an observation that “you shall know a word by the company it keeps” [Firth 1957]; that is, it learns from the context in which words appear in large collections of texts. Under this representation, similar words such as *house* and *building* will have similar representations, which will improve the learning capability of our neural networks. An important limitation of word2vec is that it conflates all senses of a given word into a single numerical representation. That is, the word *bank* gets a single numerical representation regardless of whether its current context indicates a financial sense, e.g., *Bank of America*, or a geological one, e.g., *bank of the river*. In Chapter 10 we will introduce contextualized embeddings, i.e., numerical representations that are sensitive of the current context in which a word appears, which address this limitation. These contextualized embeddings are built using transformer networks, which rely on “attention,” a mechanism that computes the representation of a word using a weighted average of the representations of the words in its context. These weights are learned and indicate how much ”attention” each word should put on each of its neighbors (hence the name).

Chapter 12 introduces sequence models for processing text. For example, while the word *book* is syntactically ambiguous (i.e., it can be either a noun or a verb), knowing that it is preceded by the determiner *the* in a text gives strong hints that this instance of it is a noun. In this chapter, we will cover neural network architectures designed to model such sequences, including recurrent neural networks, convolutional neural networks, long short-term memory networks, and long short-term memory networks combined with conditional random fields.

Chapter 14 discusses sequence-to-sequence methods (i.e., methods tailored for NLP tasks where the input is a sequence and the output is another sequence). The most common example of such a task is machine translation; where the input is a sequence of words in one language, and the output is a sequence that captures the translation of the original text in a new language.

Chapter 15 introduces several natural language processing applications such as part-of-speech tagging, named entity recognition, syntactic parsing and discusses their implementation with neural architectures.

As previously mentioned, the theoretical discussion in these chapters is interleaved with chapters that discuss how to implement these notions in PyTorch. Chapter 4 shows an implementation of the logistic regression algorithm introduced in Chapter 3. Chapter 7 introduces an implementation of the feed forward neural network introduced in Chapters 5 and 6. Chapter 9 enhances the previous implementation of a neural network with the continuous word representations introduced in Chapter 8. Chapter 11 changes the previous implementation of feed forward neural networks to use the contextualized embeddings generated by a transformer network. Lastly, Chapter 13 implements the sequence models introduced in Chapter 12.

1.2 What this Book Does Not Cover

It is important to note that deep learning is only one of the many subfields of machine learning. In his book, Domingos provides an intuitive organization of these subfields into five "tribes" [Domingos 2015]:

Connectionists This tribe focuses on machine learning methods that (shallowly) mimic the structure of the brain. The methods described in this book fall into this tribe.

Evolutionaries The learning algorithms adopted by this group of approaches, also known as genetic algorithms, focus on the "survival of the fittest". That is, these algorithms "mutate" the "DNA" (or parameters) of the models to be learned, and preserve the generations that perform the best.

Symbolists The symbolists rely on inducing logic rules that explain the data in the task at hand. For example, a part-of-speech tagging system in this camp may learn a rule such as `if` previous word is *the*, `then` the part of the speech of the next word is noun.

Bayesians The Bayesians use probabilistic models such as Bayesian networks. All these methods are driven by Bayes' rule, which describes the probability of an event.

Analogizers The analogizers' methods are motivated by the observation that "you are what you resemble". For example, a new email is classified as spam because it uses content similar to other emails previously classified as such.

It is beyond the goal of this book to explain these other tribes in detail. Even from the connectionist tribe, we will focus mainly on methods that are relevant for language processing.[6] For a more general description of machine learning, the interested reader should look to other sources such as Domingos' book, or Hal Daumé III's excellent Course in Machine Learning.[7]

1.3 Deep Learning Is Not Perfect

While deep learning has pushed the performance of many machine learning applications beyond what we thought possible just ten years ago, it is certainly not perfect. Gary Marcus and Ernest Davis provide a thoughtful criticism of deep learning in their book, Rebooting AI [Marcus and Davis 2019]. Their key arguments are:

Deep learning is opaque While deep learning methods often learn well, it is unclear *what* is learned, i.e., what the connections between the network neurons encode. This is dangerous, as biases and bugs may exist in the models learned, and they may be discovered only too late, when these systems are deployed in important real-world applications such as diagnosing medical patients, or self-driving cars.

Deep learning is brittle It has been repeatedly shown both in the machine learning literature and in actual applications that deep learning systems (and for that matter most other machine learning approaches) have difficulty adapting to new scenarios they have not seen during training. For example, self-driving cars that were trained in regular traffic on US highways or large streets do not know how to react to unexpected scenarios such as a firetruck stopped on a highway.[8]

Deep learning has no common sense An illustrative example for this limitation is that object recognition classifiers based on deep learning tend to confuse objects when they are rotated in three-dimensional space, e.g., an overturned bus in the snow is confused with a snow plow. This happens because deep learning systems lack the common-sense knowledge that some object features are inherent

[6] Most of methods discussed in this book are certainly useful and commonly used outside of NLP as well.

[7] http://ciml.info

[8] https://www.teslarati.com/tesla-model-s-firetruck-crash-details/

properties of the category itself regardless of the object position, e.g., a school bus in the US usually has a yellow roof, while some features are just contingent associations, e.g., snow tends to be present around snow plows. (Most) humans naturally use common sense, which means that we do generalize better to novel instances, especially when they are outliers.

All the issues raised by Marcus and Davis are unsolved today. However, we will discuss some directions that begin to address them in this book. For example, in Chapter 15 we will discuss algorithms that (shallowly) learn common-sense knowledge from large collections of texts, as well as strategies to mitigate the pain in transferring deep learning models from one domain to another.

1.4 Mathematical Notations

While we try to rely on plain language as much as possible in this book, mathematical formalisms cannot (and should not) be avoided. Where mathematical notations are necessary, we rely on the following conventions:

- We use lower case characters such as x to represent scalar values, which will generally have integer or real values.
- We use bold lower case characters such as $\mathbf{x}$ to represent arrays (or vectors) of scalar values, and x_i to indicate the scalar element at position i in this vector. Unless specified otherwise, we consider all vectors to be column vectors during operations such as multiplication, even though we show them in text as horizontal. We use $[\mathbf{x};\mathbf{y}]$ to indicate vector concatenation. For example, if $\mathbf{x} = (1,2)$ and $\mathbf{y} = (3,4)$, then $[\mathbf{x};\mathbf{y}] = (1,2,3,4)$.
- We use bold upper case characters such as $\mathbf{X}$ to indicate matrices of scalar values. Similarly, x_{ij} points to the scalar element in the matrix at row i and column j. $\mathbf{x}_i$ indicates the vector corresponding to the entire row i in matrix $\mathbf{X}$.
- We collectively refer to matrices of arbitrary dimensions as *tensors*. By and large, in this book tensors will have dimension 1 (i.e., vectors) or 2 (matrices). Occasionally, we will run into tensors with 3 dimensions.

2 The Perceptron

This chapter covers the perceptron, the simplest neural network architecture. In general, neural networks are machine learning architectures loosely inspired by the structure of biological brains. The perceptron is the simplest example of such architectures: it contains a single artificial neuron.

The perceptron will form the building block for the more complicated architectures discussed later in the book. However, rather than starting directly with the discussion of this algorithm, we will start with something simpler: a children's book and some fundamental observations about machine learning. From these, we will formalize our first machine learning algorithm, the perceptron. In the following chapters, we will improve upon the perceptron with logistic regression (Chapter 3), and deeper feed forward neural networks (Chapter 5).

2.1 Machine Learning Is Easy

Machine learning is easy. To convince you of this, let us read a children's story [Donaldson and Scheffler 2008]. The story starts with a little monkey that lost her mom in the jungle (Figure 2.1). Luckily, the butterfly offers to help, and collects some information about the mother from the little monkey (Figure 2.2). As a result, the butterfly leads the monkey to an elephant. The monkey explains that her mom is neither gray nor big, and does not have a trunk. Instead, her mom has a "tail that coils around trees". Their journey through the jungle continues until, after many mistakes (e.g., snake, spider), the pair end up eventually finding the monkey's mom, and the family is happily reunited.

In addition to the exciting story that kept at least a toddler and this parent glued to its pages, this book introduces several fundamental observations about (machine) learning.

First, *objects are described by their properties*, also known in machine learning terminology as *features*. For example, we know that several features apply to the monkey mom: `isBig`, `hasTail`, `hasColor`, `numberOfLimbs`, etc. These features have values, which may be Boolean (true or false), a discrete value from a fixed set, or a number. For example, the values for the above features are: `false`, `true`, `brown` (out of multiple possible colors), and 4. As we will see soon, it is preferable to convert these values into numbers because most of the machine learning can be reduced to numeric operations such as additions and multiplications. For this reason, Boolean features are

Figure 2.1 A wonderful children's book that introduces the fundamentals of machine learning: Where's My Mom, by Julia Donaldson and Axel Scheffler [Donaldson and Scheffler 2008].

converted to 0 for false, and 1 for true. Features that take discrete values are converted to Boolean features by enumerating over the possible values in the set. For example, the color feature is converted into a set of Boolean features such as `hasColorBrown` with the value `true` (or 1), `hasColorRed` with the value `false` (or 0), etc.

Second, *objects are assigned a discrete label*, which the learning algorithm or *classifier* (the butterfly has this role in our story) will learn how to assign to new objects. For example, in our story we have two labels: `isMyMom` and `isNotMyMom`. When there are two labels to be assigned such as in our story, we call the problem at hand a *binary classification problem*. When there are more than two labels, the problem becomes a *classification task*. Sometimes, the labels are continuous numeric values, in which case the problem at hand is called a *regression task*. An example of such a regression problem would be learning to forecast the price of a house on the real estate market from its properties, e.g., number of bedrooms, and year it was built. However, in NLP most tasks are classification problems (we will see some simple ones in this chapter, and more complex ones starting with Chapter 12).

Little monkey: “I’ve lost my mom!”

“Hush, little monkey, don’t you cry. I’ll help you find her,” said butterfly. “Let’s have a think, How big is she?”

“She’s big!” said the monkey. “Bigger than me.”

”Bigger than you? Then I’ve seen your mom. Come, little monkey, come, come, come.”

“No, no, no! That’s an elephant.”

Figure 2.2 The butterfly tries to help the little monkey find her mom, but fails initially [Donaldson and Scheffler 2008].

Table 2.1 An example of a possible feature matrix **X** (left table) and a label vector **y** (right table) for three animals in our story: elephant, snake, and monkey.

`isBig`	`hasTail`	`hasTrunk`	`hasColorBrown`	`numberOfLimbs`
1	1	1	0	4
0	1	0	0	0
0	1	0	1	4

`Label`
`isNotMyMom`
`isNotMyMom`
`isMyMom`

To formalize what we know so far, we can organize the examples the classifier has seen (also called a training dataset) into a matrix of features **X** and a vector of labels **y**. Each example seen by the classifier takes a row in **X**, with each of the features occupying a different column. Each y_i is the label of the corresponding example $\mathbf{x}_i$. Table 2.1 shows an example of a possible matrix **X** and label vector **y** for three animals in our story.

The third observation is that a good learning algorithm *aggregates its decisions over multiple examples with different features*. In our story the butterfly learns that some features are positively associated with the mom (i.e., she is likely to have them), while some are negatively associated with her. For example, from the animals the butterfly sees in the story, it learns that the mom is likely to have a tail, fur, and four limbs, and she is not big, does not have a trunk, and her color is not gray. We will see soon that this is exactly the intuition behind the simplest neural network, the perceptron.

Lastly, learning algorithms produce incorrect classifications when not exposed to sufficient data. This situation is called *overfitting*, and it is more formally defined as the situation when an algorithm performs well in training (e.g., once the butterfly sees the snake, it will reliably classify it as not the mom when it sees in the future), but poorly on unseen data (e.g., knowing that the elephant is not the mom did not help much with the classification of the snake). To detect overfitting early, machine learning problems typically divide their data into three partitions: (a) a training partition from which the classifier learns; (b) a development partition that is used for the *internal* validation of the trained classifier, i.e., if it performs poorly on this dataset, the classifier has likely overfitted; and (c) a testing partition that is used *only* for the final, formal evaluation. Machine learning developers typically alternate between training (on the training partition) and validating what is being learned (on the development partition) until acceptable performance is observed. Once this is reached, the resulting classifier is evaluated (ideally once) on the testing partition.

2.2 Use Case: Text Classification

In the remaining of this chapter, we will begin to leave the story of the little monkey behind us, and change to a related NLP problem, text classification, in which a classifier is trained to assign a label to a text. This is an important and common NLP task. For example, email providers use binary text classification to classify emails into spam or not. Data mining companies use multiclass classification to detect how customers feel about a product, e.g., like, dislike, or neutral. Search engines use multiclass classification to detect the language a document is written in before processing it.

Throughout the next few chapters, we will focus on text classification for simplicity. We will consider only two labels for the next few chapters, and we will generalize the algorithms discussed to multiclass classification (i.e., more than two labels) in Chapter 6. After we discuss sequence models (Chapter 12), we will introduce more complex NLP tasks such as part-of-speech tagging and syntactic parsing.

For now, we will extract simple features from the texts to be classified. That is, we will simply use the frequencies of words in a text as its features. More formally, the matrix $\mathbf{X}$, which stores the entire dataset, will have as many columns as words in the vocabulary. Each cell x_{ij} corresponds to the number of times the word at column j occurs in the example stored at row i. For example, the text *This is a great great buy* will produce a feature corresponding to the word *buy* with value 1, one for the word *great* with value 2, etc., while the features corresponding to all the other words in the vocabulary that do not occur in this document receive a value of 0. This feature design strategy is often referred to as *bag of words*, because it ignores all the syntactic structure of the text, and treats the text simply as a collection of independent words. We will revisit this simplification in Chapter 12, where we will start to model sequences of words.

Table 2.2 Example output of a hypothetical classifier on five evaluation examples and two labels: positive (+) and negative (−). The "Gold" column indicates the correct labels for the five texts; the "Predicted" column indicates the classifier's predictions.

	Gold	Predicted
1	+	+
2	+	−
3	−	−
4	−	+
5	+	+

Table 2.3 Confusion matrix showing the four possible outcomes in binary classification, where + indicates the positive label and − indicates the negative label.

	Classifier predicted +	Classifier predicted −
Gold label is +	True positive (TP)	False negative (FN)
Gold label is −	False positive (FP)	True negative (TN)

2.3 Evaluation Measures for Text Classification

The simplest evaluation measure for text classification is accuracy, defined as the proportion of evaluation examples that are correctly classified. For example, the accuracy of the hypothetical classifier shown in Table 2.2 is $3/5 = 60\%$ because the classifier was incorrect on two examples (rows 2 and 4).

Using the four possible outcomes for binary classification summarized in the matrix shown in Table 2.3, which is commonly referred to as a confusion matrix, accuracy can be more formally defined as:

$$\text{Accuracy} = \frac{TP + TN}{TP + FN + FP + TN} \tag{2.1}$$

For example, for the classifier output shown in Table 2.2, TP = 2 (rows 1 and 5), TN = 1 (row 3), FP = 1 (row 4), and FN = 1 (row 2).

While accuracy is obviously useful, it is not always informative. In problems where the two labels are heavily unbalanced, i.e., one is much more frequent than the other, and we care more about the less frequent label, a classifier that is not very useful may have a high accuracy score. For example, assume we build a classifier that identifies high-urgency Medicaid applications,[1] i.e., applications must be reviewed quickly due to the patient's medical condition. The vast majority of applications are not high-urgency, which means they can be handled through the usual review process. In this example, the positive class is assigned to the high-urgency applications. If a classifier labels all applications as negative (i.e., not high-urgency), its accuracy will be high because the TN count dominates the accuracy score. For example, say that out of 1,000 applications only 1 is positive. Our classifier's accuracy is then: $\frac{0+999}{0+1+0+999} = 0.999$, or 99.9%. This high accuracy is obviously misleading in any real-world application of the classifier.

For such unbalanced scenarios, two other scores that focus on class of interest (say, the positive class) are commonly used: precision and recall. Precision (P) is the proportion of correct positive examples out of all positives predicted by the classifier. Recall (R) is the proportion of correct positive examples out of all positive examples in the evaluation dataset. More formally:

$$P = \frac{TP}{TP+FP} \tag{2.2}$$

$$R = \frac{TP}{TP+FN} \tag{2.3}$$

For example, both the precision and recall of the above classifier are 0 because $TP = 0$ in its output. On the other hand, a classifier that predicts 2 positives, out of which only one is incorrect, will have a precision of $1/2 = 0.5$ and a recall of $1/1 = 1$, which are clearly more informative of the desired behavior.

Often it helps to summarize the performance of a classifier using a single number. The F_1 score achieves this, as the harmonic mean of precision and recall:

$$F_1 = \frac{2PR}{P+R} \tag{2.4}$$

For example, the F1 score for the previous example is: $F_1 = \frac{2\times0.5\times1}{0.5+1} = 0.66$. A reasonable question to ask here is why not use instead the simpler arithmetic mean between precision and recall ($\frac{P+R}{2}$) to generate this overall score? The reason for choosing the more complicated harmonic mean is that this formula is harder to game. For example, consider a classifier that labels everything as positive. Clearly, this would be useless in the above example of classifying high-urgency Medicaid applications. This classifier would have a recall of 1 (because it did identify all the high-urgency applications), and a precision of approximately 0 (because everything else in the set of 1,000 applications is also labeled as high-urgency). The simpler arithmetic mean

[1] Medicaid is a federal and state program in the United States that helps with medical costs for some people with limited income and resources.

of the two scores is approximately 0.5, which is an unreasonably high score for a classifier that has zero usefulness in practice. In contrast, the F1 score of this classifier is approximately 0, which is more indicative of the classifier's overall performance. In general, the F_1 score penalizes situations where the precision and recall values are far apart from each other.

A more general form of the F1 score is:

$$F_\beta = (1+\beta^2)\frac{PR}{(\beta^2 P)+R} \tag{2.5}$$

where β is a positive real value, which indicates that recall is β times more important than precision. This generalized formula allows one to compute a single overall score for situations when precision and recall are not treated equally. For example, in the high-urgency Medicaid example, we may decide that recall is more important than precision. That is, we are willing to inspect more incorrect candidates for high-urgency processing as long as we do not miss the true positives. If we set $\beta = 10$ to indicate that we value recall as being 10 times more important than precision, the classifier in the above example ($P = 0.5$ and $R = 1$) has a $F_{\beta=10}$ score of: $F_{\beta=10} = 101\frac{0.5\times 1}{(100\times 0.5)+1} = 0.99$, which is much closer to the classifier's recall value (the important measure here) than the F_1 score.

We will revisit these measures in Chapter 3, where we will generalize them to multiclass classification, i.e., to situations where the classifier must produce more than two labels, and in Chapter 4, where we will implement and evaluate multiple text classification algorithms.

2.4 The Perceptron

Now that we understand our first NLP task, text classification, let us introduce our first classification algorithm, the perceptron. The perceptron was invented by Frank Rosenblatt in 1958. Its aim is to mimic the behavior of a single neuron [Rosenblatt 1958]. Figure 2.3 shows a depiction of a biological neuron,[2] and Rosenblatt's computational simplification, the perceptron. As the figure shows, the perceptron is the simplest possible artificial neural network. We will generalize from this single-neuron architecture to networks with an arbitrary number of neurons in Chapter 5.

The perceptron has one input for each feature of an example $\mathbf{x}$, and produces an output that corresponds to the label predicted for $\mathbf{x}$. Importantly, the perceptron has a weight vector $\mathbf{w}$, with one weight w_i for each input connection i. Thus, the size of $\mathbf{w}$ is equal to the number of features, or the number of columns in $\mathbf{X}$. Further, the

[2] By BruceBlaus – Own work, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=28761830

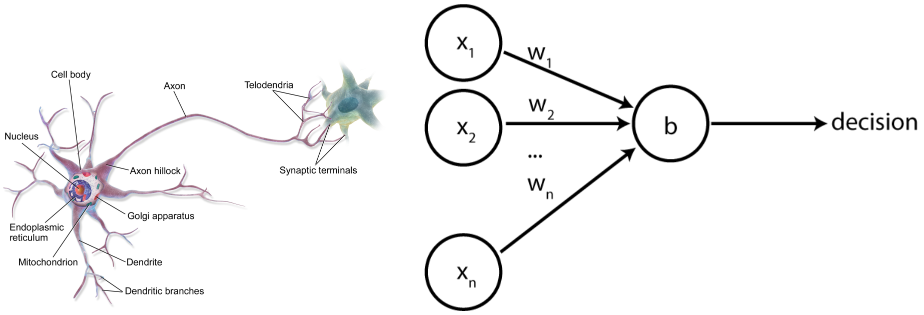


Figure 2.3 A depiction of a biological neuron, which captures input stimuli through its dendrites and produces an activation along its axon and synaptic terminals (left), and its computational simplification, the perceptron (right).

perceptron also has a bias term, b, that is scalar (we will explain why this is needed later in this section). The perceptron outputs a binary decision, let us say Yes or No (e.g., Yes, the text encoded in $\mathbf{x}$ contains a positive review for a product, or No, the review is negative), based on the decision function described in Algorithm 1. The $\mathbf{w} \cdot \mathbf{x}$ component of the decision function is called the *dot product* of the vectors $\mathbf{w}$ and $\mathbf{x}$. Formally, the dot product of two vectors $\mathbf{x}$ and $\mathbf{y}$ is defined as:

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n} x_i y_i \tag{2.6}$$

where n indicates the size of the two vectors. In words, the dot product of two vectors, $\mathbf{x}$ and $\mathbf{y}$, is found by adding (Σ), the values found by multiplying each element of $\mathbf{x}$ with the corresponding value of $\mathbf{y}$. In the case of the perceptron, the dot product of $\mathbf{x}$ and $\mathbf{w}$ is the weighted sum of the feature values in $\mathbf{x}$, where each feature value x_i is weighted by w_i. If this sum (offset by the bias term b, which we will discuss later) is positive, then the decision is Yes. If it is negative, the decision is No.

Sidebar 2.1 The dot product in linear algebra

In linear algebra, the dot product of two vectors $\mathbf{x}$ and $\mathbf{y}$ is equivalent to $\mathbf{x}^T\mathbf{y}$, where T is the transpose operation. However, in this book we rely on the dot product notation for simplicity.

Algorithm 1: The decision function of the perceptron.

```
if $\mathbf{w} \cdot \mathbf{x} + b > 0$ then
    return Yes
else
    return No
end
```

Sidebar 2.2 The sign function in the perceptron

The decision function listed in Algorithm 1 is often shown as $\text{sign}(\mathbf{w} \cdot \mathbf{x} + b)$, where the $+$ sign is used to represent one class, and the $-$ sign the other.

There is an immediate parallel between this decision function and the story of the little monkey. If we consider the Yes class to be `isMyMom`, then we would like the weights of the features that belong to the mom (e.g., `hasColorBrown`) to have positive values, so the dot product between $\mathbf{w}$ and the $\mathbf{x}$ vector corresponding to the mom turns out positive, and the features specific to other animals (e.g., `hasTrunk`) to receive negative weights, so the corresponding decision is negative. Similarly, if the task to be learned is review classification, we would like positive words (e.g., *good*, *great*) to have positive weights in $\mathbf{w}$, and negative words (e.g., *bad*, *horrible*) to have negative weights.

In general, we call the aggregation of a learning algorithm or classifier and its learned parameters ($\mathbf{w}$ and b for the perceptron) a *model*. All classifiers aim to learn these parameters to optimize their predictions over the examples in the training dataset.

The key contribution of the perceptron is a simple algorithm that learns these weights (and bias term) from the given training dataset. This algorithm is summarized in Algorithm 2. Let us dissect this algorithm next. The algorithm starts by initializing the weights and bias term with 0s. Note that lines of pseudocode that assign values to a vector such as line 1 in the algorithm ($\mathbf{w} = 0$) assign this scalar value to *all* the elements of the vector. For example, the operation in line 1 initializes all the elements of the weight vector with zeros.

Lines 3 and 4 indicate that the learning algorithm may traverse the training dataset more than once. As we will see in the following example, sometimes this repeated exposure to training examples is necessary to learn meaningful weights. Informally, we say that the algorithm *converged* when there are no more changes to the weight vector (we will define convergence more formally later in this section). In practice, on real-world tasks, it is possible that true convergence is not reached, so, commonly, line

Algorithm 2: Perceptron learning algorithm.

```
1  $\mathbf{w} = 0$
2  b = 0
3  while not converged do
4      for each training example $\mathbf{x}_i$ in X do
5          d = decision($\mathbf{x}_i$, $\mathbf{w}$, b)
6          if $d == y_i$ then
7              continue
8          else if $y_i == Yes$ and $d == No$ then
9              $b = b + 1$
10             $\mathbf{w} = \mathbf{w} + \mathbf{x}_i$
11         else if $y_i == No$ and $d == Yes$ then
12             $b = b - 1$
13             $\mathbf{w} = \mathbf{w} - \mathbf{x}_i$
14     end
15 end
```

3 of the algorithm is written to limit the number of traversals of the training dataset (or *epochs*) to a fixed number.

Line 5 applies the decision function in Algorithm 1 to the current training example. Lines 6 and 7 indicate that the perceptron simply skips over training examples that it already knows how to classify, i.e., its decision d is equal to the correct label y_i. This is intuitive: if the perceptron has already learned how to classify an example, there is limited benefit in learning it again. In fact, the opposite might happen: the perceptron weights may become too tailored for the particular examples seen in the training dataset, which will cause it to overfit. Lines 8 – 10 address the situation when the correct label of the current training example $\mathbf{x}_i$ is Yes, but the prediction according to the current weights and bias is No. In this situation, we would intuitively want the weights and bias to have higher values such that the overall dot product plus the bias is more likely to be positive. To move towards this goal, the perceptron simply *adds* the feature values in $\mathbf{x}_i$ to the weight vector $\mathbf{w}$, and adds 1 to the bias. Similarly, when the perceptron makes an incorrect prediction for the label No (lines 11 – 13), it decreases the value of the weights and bias by *subtracting* $\mathbf{x}_i$ from $\mathbf{w}$, and subtracting 1 from b.

Sidebar 2.3 Error driven learning

The class of algorithms such as the perceptron that focus on "hard" examples in training, i.e., examples for which they make incorrect predictions at a given point in time, are said to perform *error driven learning*.

Figure 2.4 shows an intuitive visualization of this learning process.[3] In this figure, for simplicity, we are ignoring the bias term and assume that the perceptron decision is driven solely by the dot product $\mathbf{x} \cdot \mathbf{w}$. Figure 2.4 (a) shows the weight vector $\mathbf{w}$ in a simple two-dimensional space, which would correspond to a problem that is represented using only two features.[4] In addition of $\mathbf{w}$, the figure also shows the *decision boundary* of the perceptron as a dashed line that is perpendicular on $\mathbf{w}$. The figure indicates that all the vectors that lie on the same side of the decision boundary with $\mathbf{w}$ are assigned the label Yes, and all the vectors on the other side receive the decision No. Vectors that lie exactly on the decision boundary (i.e., their decision function has a value of 0) receive the label No according to Algorithm 1. In the transition from (a) to (b), the figure also shows that redrawing the boundary changes the decision for $\mathbf{x}$.

Why is the decision boundary line perpendicular on $\mathbf{w}$, and why are the labels so nicely assigned? To answer these questions, we need to introduce a new formula that measures the cosine of the angle between two vectors, *cos*:

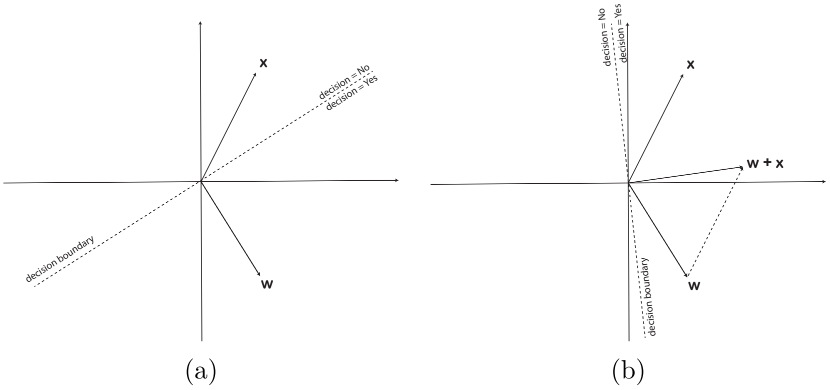
$$cos(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{||\mathbf{x}||||\mathbf{y}||} \tag{2.7}$$

where $||\mathbf{x}||$ indicates the length of vector $\mathbf{x}$, i.e., the distance between the origin and the tip of the vector's arrow, measured with a generalization of Pythagoras's theorem:[5] $||\mathbf{x}|| = \sqrt{\sum_{i=1}^{N} x_i^2}$. The cosine similarity, which ranges between -1 and 1, is widely used in the field of information retrieval to measure the similarity of two vectors [Schütze et al. 2008]. That is, two perfectly similar vectors will have an angle of 0° between them, which has the largest possible cosine value of 1. Two "opposite" vectors have

[3] This visualization was first introduced by Schütze et al. [2008].

[4] This simplification is useful for visualization, but it is highly unrealistic for real-world NLP applications, where the number of features is often proportional with the size of a language's vocabulary, i.e., it is often in the hundreds of thousands.

[5] Pythagoras's theorem states that the square of the hypothenuse, c, of a right triangle is equal to the sum of the squares of the other two sides, a and b, or, equivalently: $c = \sqrt{a + b}$. In our context, c is the length of a vector with coordinates a and b in a two-dimensional space.



(a)

(b)

Figure 2.4 Visualization of the perceptron learning algorithm: (a) incorrect classification of the vector **x** with the label Yes, for a given weight vector **w**; and (b) **x** lies on the correct side of the decision boundary after **x** is added to **w**.

an angle of 180° between them, which has a cosine of -1. We will extensively use the cosine similarity formula starting with the next chapter. But, for now, we will simply observe that the cosine similarity value has the same sign with the dot product of the two vectors (because the length of a vector is always positive). Because vectors on the same side of the decision boundary with **w** have an angle with **w** in the interval [$-90°$, $90°$], the corresponding cosine (and, thus, dot product value) will be positive, which yields a Yes decision. Similarly, vectors on the other side of the decision boundary will receive a No decision.

Sidebar 2.4 Hyper planes and perceptron convergence

In a one-dimensional feature space, the decision boundary for the perceptron is a dot. As shown in Figure 2.4, in a two-dimensional space, the decision boundary is a line. In a three-dimensional space, the decision boundary is a plane. In general, for a *n*-dimensional space, the decision boundary of the perceptron is a *hyper plane*. Classifiers such as the perceptron whose decision boundary is a hyper plane, i.e., it is driven by a linear equation in **w** (see Algorithm 1), are called *linear classifiers*.

If such a hyper plane that separates the labels of the examples in the training dataset exists, it is guaranteed that the perceptron will find it, or will find another hyper plane with similar separating properties [Block 1962, Novikoff 1963]. We say that the learning algorithm has *converged* when such a hyper plane is found, which means that all examples in the training data are correctly classified.

Table 2.4 The feature matrix **X** (left table) and label vector **y** (right table) for a review classification training dataset with three examples.

#	*good*	*excellent*	*bad*	*horrible*	*boring*
#1	1	1	1	0	0
#2	0	0	1	1	0
#3	0	0	1	0	1

Label
Positive
Negative
Negative

Table 2.5 The perceptron learning process for the dataset shown in Table 2.4, for one pass over the training data. Both **w** and b are initialized with 0s.

Example seen: #1
$\mathbf{x} \cdot \mathbf{w} + b = 0$
Decision = Negative
Update (add): $\mathbf{w} = (1, 1, 1, 0, 0)$, $b = 1$
Example seen: #2
$\mathbf{x} \cdot \mathbf{w} + b = 2$
Decision = Positive
Update (subtract): $\mathbf{w} = (1, 1, 0, -1, 0)$, $b = 0$
Example seen: #3
$\mathbf{x} \cdot \mathbf{w} + b = 0$
Decision = Negative
Update: none

Figure 2.4 (a) shows that, at that point in time, the training example **x** with label Yes lies on the incorrect side of the decision boundary. Figure 2.4 shows how the decision boundary is adjusted after **x** is added to **w** (line 10 in Algorithm 2). After this adjustment, **x** is on the correct side of the decision boundary.

To convince ourselves that the perceptron is indeed learning a meaningful decision boundary, let us go trace the learning algorithm on a slightly more realistic example. Table 2.4 shows the matrix **X** and label vector **y** for a training dataset that contains three examples for a product review classification task. In this example, we assume that our vocabulary has only the five words shown in **X**, e.g., the first data point in this dataset is a positive review that contains the words *good*, *excellent*, and *bad*.

Table 2.5 traces the learning algorithm as it iterates through the training examples. For example, because the decision function produces the incorrect decision for the first example (No), this example is added to **w**. Similarly, the second example is subtracted from **w**. The third example is correctly classified (barely), so no update is necessary. After just one pass over this training dataset, also called an *epoch*, the perceptron has converged. We will let the reader convince herself that all training examples are now correctly classified. The final weights indicate that the perceptron has learned several useful things. First, it learned that *good* and *excellent* are associated with the Yes class, and has assigned positive weights to them. Second, it learned that *bad* is not to be trusted because it appears in both positive and negative reviews, and, thus, it assigned it a weight of 0. Lastly, it learned to assign a negative weight to *horrible.* However, it is not perfect: it did not assign a non-zero weight to *boring* because of the barely correct prediction made on example #3. There are other bigger problems here. We discuss them in Section 2.7.

This example as well as Figure 2.4 seem to suggest that the perceptron learns just fine without a bias term. So why do we need it? To convince ourselves that the bias term is useful let us walk through another simple example, shown in Table 2.6. The perceptron needs four epochs, i.e., four passes over this training dataset, to converge. The final parameters are: $\mathbf{w} = (2)$ and $b = -4$. We encourage the reader to trace the learning algorithm through this dataset on her own as well. These parameters indicate that the hyper plane for this perceptron, which is a dot in this one-dimensional feature space, is at 2 (because the final inequation for the positive decision is $2x - 4 > 0$). That is, in order to receive a Yes decision, the feature of the corresponding example must have a value > 2, i.e., the review must have at least three positive words. This is intuitive, as the training dataset contains negative reviews that contain one or two positive words. What this shows is that the bias term allows the perceptron to shift its decision boundary *away* from the origin. It is easy to see that, without a bias term, the perceptron would not be able to learn anything meaningful, as the decision boundary will always be in the origin. In practice, the bias term tends to be more useful for problems that are modeled with few features. In real-world NLP tasks that are high-dimensional, learning algorithms usually find good decision boundaries even without a bias term (because there are many more options to choose from).

Sidebar 2.5 Implementations of the bias term

Some machine learning software packages implement the bias term as an additional feature in $\mathbf{x}$ that is always active, i.e., it has a value of 1 for all examples in $\mathbf{X}$. This simplifies the math a bit, i.e., instead of computing $\mathbf{x} \cdot \mathbf{w} + b$, we now have to compute just $\mathbf{x} \cdot \mathbf{w}$. It is easy to see that modeling the bias as an always-active feature has the same functionality as the explicit bias term in Algorithm 2. In this book, we will maintain an explicit bias term for clarity.

Table 2.6 The feature matrix **X** (left table) and label vector **y** (right table) for a review classification training dataset with four examples. In this example, the only feature available is the *total* number of positive words in a review.

#	Number of positive words
#1	1
#2	10
#3	2
#4	20

Label
Negative
Positive
Negative
Positive

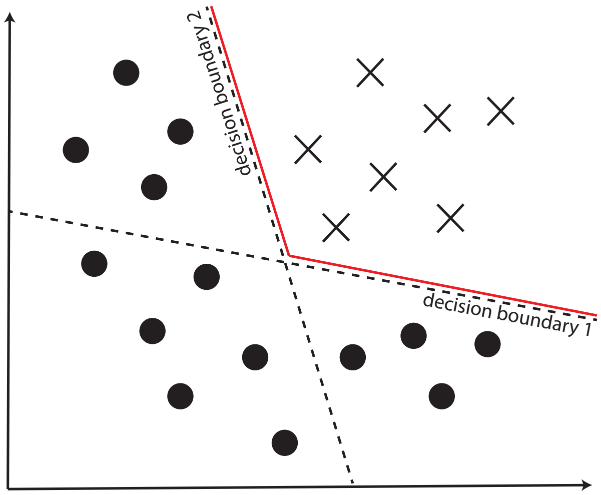


Figure 2.5 An example of a binary classification task, and a voting perceptron that aggregates two imperfect perceptrons. The voting algorithm classifies correctly all the data points by requiring two votes for the × class to yield a × decision. The decision boundary of the voting perceptron is shown in red.

2.5 Voting Perceptron

As we saw in the previous examples, the perceptron learns well, but it is not perfect. Often, a very simple strategy to improve the quality of classifier is to use an *ensemble model*. One such ensemble strategy is to *vote* between the decisions of multiple learning algorithms. For example, Figure 2.5 shows a visualization of such a voting perceptron, which aggregates two individual perceptrons by requiring that both classifiers label an example as × before issuing the × label.[6]

The figure highlights two important facts. First, the voting perceptron performs better than either individual classifier. In general, ensemble models that aggregate models that are sufficiently different from each other tend to perform better than the individual (or base) classifiers that are part of the ensemble [Dietterich 2000]. This observation holds for people too! It has been repeatedly shown that crowds reach better decisions than individuals. For example, in 1907, Sir Francis Galton has observed that while no individual could correctly guess the weight of an ox at a fair, averaging the weights predicted by *all* individuals came within a pound or two of the real weight of the animal [Young 2009]. Second, the voting perceptron is a *non-linear classifier*, i.e., its decision boundary is no longer a line (or a hyper plane in n dimensions): in Figure 2.5, the non-linear decision boundary for the voting perceptron is shown with red lines.

While the voting approach is an easy way to produce a non-linear classifier that improves over the basic perceptron, it has drawbacks. First, we need to produce several individual perceptron classifiers. This can be achieved in at least two distinct ways. For example, instead of initializing the $\mathbf{w}$ and b parameters with 0s (lines 1 and 2 in Algorithm 2), we initialize them with random numbers (typically small numbers centered around 0). For every different set of initial values in $\mathbf{w}$ and b, the resulting perceptron will end up with a different decision boundary, and, thus, a different classifier. The drawback of this strategy is that the training procedure must be repeated for each individual perceptron. A second strategy for producing multiple individual perceptron that avoids this training overhead is to keep track of all $\mathbf{w}$s and bs that are produced during the training of a single perceptron. That is, before changing the b and $\mathbf{w}$ parameters in Algorithm 2 (lines 9 and 12), we store the current values (before the change) in a list. This means that at the end of the training procedure, this list will contain as many individual perceptrons as the number of updates performed in training. We can even sort these individual classifiers by their perceived quality: the more iterations a specific b and $\mathbf{w}$ combination "survived" in training, the better the quality of this classifier is likely to be. This indicator of quality can be used to assign weights to the "votes" given to the individual classifiers, or to filter out base models of low quality (e.g., remove all classifiers that survived fewer than 10 training examples).

The second drawback of the voting perceptron is its runtime overhead at evaluation time. When the voting perceptron is applied on a new, unseen example, it must apply all its individual classifiers before voting. Thus, the voting perceptron is N times slower

[6] This example was adapted from Erwin Chan's Ling 539 course at University of Arizona.

Algorithm 3: Average perceptron learning algorithm.

```
$\mathbf{w} = 0$
b = 0
numbertotalOfUpdates = 0
$\mathbf{totalW} = 0$
totalB = 0
while *not converged* do
    for *each training example* $\mathbf{x}_i$ in $\mathbf{X}$ do
        d = decision($\mathbf{x}_i$, $\mathbf{w}$, b)
        if $d == y_i$ then
            continue
        else if $y_i == Yes$ and $d == No$ then
            $\text{numberOfUpdates} = \text{numberOfUpdates} + 1$
            $\mathbf{totalW} = \mathbf{totalW} + \mathbf{w}$
            $\text{totalB} = \text{totalB} + b$
            $\mathbf{w} = \mathbf{w} + \mathbf{x}_i$
            $b = b + 1$
        else if $y_i == No$ and $d == Yes$ then
            $\text{numberOfUpdates} = \text{numberOfUpdates} + 1$
            $\mathbf{totalW} = \mathbf{totalW} + \mathbf{w}$
            $\text{totalB} = \text{totalB} + b$
            $\mathbf{w} = \mathbf{w} - \mathbf{x}_i$
            $b = b - 1$
    end
end
$\text{avgB} = \text{totalB}/\text{numberOfUpdates}$
$\mathbf{avgW} = \mathbf{totalW}/\text{numberOfUpdates}$
```

than the individual perceptron, where N is the number of individual classifiers used. To mitigate this drawback, we will need the average perceptron, discussed next.

2.6 Average Perceptron

The average perceptron is a simplification of the latter voting perceptron discussed previously. The simplification consists in that, instead of keeping track of *all* $\mathbf{w}$ and b parameters created during the perceptron updates like the voting algorithm, these parameters are averaged into a *single* model, say **avgW** and *avgB*. This algorithm, which is summarized in Algorithm 3, has a constant runtime overhead for computing the average model, i.e., the only additional overhead compared to the regular perceptron are the additions in lines 12 − 14 and 18 − 20, and the divisions in lines 25 and 26. Further, the additional memory overhead is also constant, as it maintains a single extra weight vector (**totalW**) and a single bias term (*totalB*) during training. After training, the average perceptron uses a decision function different from the one used during training. This function has a similar shape to the one listed in Algorithm 1, but uses **avgW** and *avgB* instead.

Despite its simplicity, the average perceptron tends to perform well in practice, usually outperforming the regular perceptron, and approaching the performance of the voting perceptron. But why is the performance of the average perceptron so good? After all, it remains a linear classifier just like the regular perceptron, so it must have the same limitations. The high-level explanation is that the average perceptron does a better job than the regular perceptron at controlling for *noise*. Kahneman et al. [2021] define noise as unwanted variability in decision making. Note that noise is a common occurrence in both human and machine decisions. For example, Kahneman et al. [2021] report that judges assign more lenient sentences if the outside weather is nice, or if their favorite football team won their match the prior weekend. Clearly, these decisions should not depend on such extraneous factors.

Similarly, in the machine learning space, the regular perceptron may be exposed to such noisy, unreliable features during training. When this happens, these features will receive weight values in the perceptron model (the **w** vector) that are all over the place, sometimes positive and sometimes negative. All these values are averaged in the average vector, and, thus, the average weight value for these unreliable features will tend to be squished to (or close to) zero. The effect of this squishing is that the decision function of the average perceptron will tend to not rely on these features (because their contribution to the dot product in the decision function will be minimal). This differs from the regular perceptron, which does not benefit from this averaging process that reduces the weights of unimportant features. In general, this process of squishing the weights of features that are not important is called *regularization*. We will see other regularization strategies in Chapter 6.

2.7 Drawbacks of the Perceptron

The perceptron algorithm and its variants are simple, easy to customize for other tasks beyond text classification, and they perform fairly well (especially in the voting and average form). However, they also have important drawbacks. We discuss these drawbacks here, and we will spend a good part of this book discussing solutions that address them.

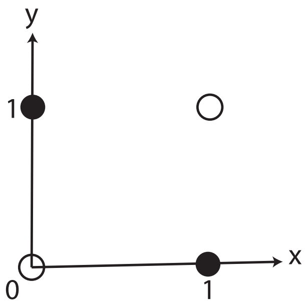


Figure 2.6 Visualization of the XOR function operating over two variables, *x* and *y*. The dark circles indicate that the XOR output is 1; the clear circles stand for 0.

The first obvious limitation of the perceptron is that, as discussed in this chapter, it is a linear classifier. Yes, the voting perceptron removes this constraint, but it comes at the cost of maintaining multiple individual perceptrons. Ideally, we would like to have the ability to learn a single classifier that captures a non-linear decision boundary. This ability is important, as many tasks require such a decision boundary. A simple example of such a task was discussed by Minsky and Papert as early as 1969: the perceptron cannot learn the XOR function [Minsky and Papert 1969]. To remind ourselves, the XOR function takes two binary variables, i.e., numbers that can take only one of two values: 0 (which stands for False) or 1 (or True), and outputs 1 when exactly one of these values is 1, and 0 otherwise. A visualization of the XOR is shown in Figure 2.6. It is immediately obvious that there is no linear decision boundary that separates the dark circles from the clear ones. More importantly in our context, language is beautiful, complex, and ambiguous, which means that, usually, we cannot model tasks that are driven by language using methods of limited power such as linear classifiers. We will address this important limitation in Chapter 5, where we will introduce neural networks that can learn non-linear decision boundaries by combining multiple layers of "neurons" into a single network.

A second more subtle but very important limitation of the perceptron is that it has no "smooth" updates during training, i.e., its updates are the same regardless of how incorrect the current model is. This is caused by the decision function of the perceptron (Algorithm 1), which relies solely on the *sign* of the dot product. That is, it does not matter how large (or small) the value of the dot product is; when the sign is incorrect, the update is the same: adding or subtracting the *entire* example $\mathbf{x_i}$ from the current weight vector (lines 10 and 13 in Algorithm 2). This causes the perceptron to be a slow learner because it jumps around good solutions. One University of Arizona student called this instability “Tony Hawk-ing the data”.[7] On data that is linearly separable, the perceptron will eventually converge [Novikoff 1963]. However, real-world datasets do not come with this guarantee of linear separation, which means that this “Tony Hawk-ing” situation may yield a perceptron that is far from acceptable. What we would like to have is a classifier that updates its model *proportionally* with the errors it makes: a small mistake causes a small update, while a large one yields a large update. This is exactly what the logistic regression does. We detail this in the next chapter.

The third drawback of the perceptron, as we covered it so far, is that it relies on hand-crafted features that must be designed and implemented by the machine learning developer. For example, in the text classification use case introduced in Section 2.2, we mentioned that we rely on features that are simply the words in each text to be classified. Unfortunately, in real-world NLP applications feature design gets complicated very quickly. For example, if the task to be learned is review classification, we should probably capture negation. Certainly the phrase *great* should be modeled differently than *not great*. Further, maybe we should investigate the syntactic structure of the text to be classified. For example, reviews typically contain multiple clauses, whose sentiment must be composed into an overall classification for the entire review. For example, the review *The wait was long, but the food was fantastic.* contains two clauses: *The wait was long* and *but the food was fantastic*, each one capturing a different sentiment, which must be assembled into an overall sentiment towards the corresponding restaurant. Further, most words in any language tend to be very infrequent [Zipf 1932], which means that a lot of the hard work we might invest in feature design might not generalize enough. That is, suppose that the reviews included in a review classification training dataset contain the word *great* but not the word *fantastic*, a fairy similar word in this context. Then, any ML algorithm that uses features that rely on explicit words will correctly learn how to associate *great* with a specific sentiment, but will not know what to do when they see the word *fantastic*. Chapter 8 addresses this limitation. We will discuss methods to transform words into a numerical representation that captures (some) semantic knowledge. Under this representation, similar words such as *great* and *fantastic* will have similar forms, which will improve the generalization capability of our ML algorithms.

Lastly, in this chapter we focused on text classification applications such as review classification that require a simple ML classifier, which produces a single binary label for an input text, e.g., positive vs. negative review. However, many NLP applications require multiclass classification (i.e., more than two labels), and, crucially, produce *structured* output. For example, a part-of-speech tagger, which identifies which words

[7] Tony Hawk is an American skateboarder, famous for his half-pipe skills. See: https://en.wikipedia.org/wiki/Tony_Hawk.

are nouns, verbs, etc., must produce the *sequence* of part of speech tags for a given sentence. Similarly, a syntactic parser identifies syntactic structures in a given sentence such as which phrase serves as subject for a given verb. These structures are typically represented as *trees*. The type of ML algorithms that produce structures rather than individual labels are said to perform *structured learning*. We will begin discussing structured learning in Chapter 12.

2.8 Historical Background

TODO: To do

2.9 References and Further Readings

TODO: To do

3 Logistic Regression

As mentioned in the previous chapter, the perceptron does not perform smooth updates during training, which may slow down learning, or cause it to miss good solutions entirely in real-world situations. In this chapter, we will discuss logistic regression (LR), a machine learning algorithm that elegantly addresses this problem.

3.1 The Logistic Regression Decision Function and Learning Algorithm

As we discussed, the lack of smooth updates in the training of the perceptron is caused by its reliance on a discrete decision function driven by the sign of the dot product. The first thing LR does is replace this decision function with a new, *continuous* function, which is:

$$\text{decision}(\mathbf{x}, \mathbf{w}, b) = \frac{1}{1 + e^{-(\mathbf{w} \cdot \mathbf{x} + b)}} \tag{3.1}$$

The $\frac{1}{1+e^{-x}}$ function is known as the logistic function, hence the name of the algorithm. The logistic function belongs to a larger class of functions called sigmoid functions because they are characterized by an S-shaped curve. Figure 3.1 shows the curve of the logistic function. In practice, the name sigmoid (or σ) is often used instead of logistic, which is why the LR decision function is often summarized as: $\sigma(\mathbf{w} \cdot \mathbf{x} + b)$. For brevity, we will use the σ notation in our formulas as well.

Figure 3.1 shows that the logistic function has values that monotonically increase from 0 to 1. We will use this property to implement a better learning algorithm, which has "soft" updates that are proportional to how incorrect the current model is. To do this, we first arbitrarily associate one of the labels to be learned with the value 1, and the other with 0. For example, for the review classification task, we (arbitrarily) map the positive label to 1, and the negative label to 0. Intuitively, we would like to learn a decision function that produces values close to 1 for the positive label, and values close to 0 for the negative one. The difference between the value produced by the decision function and the gold value for a training example will quantify the algorithm's confusion at a given stage in the learning process.

Algorithm 4 lists the LR learning process that captures the above intuitions. We will discuss later in this chapter how this algorithm was derived. For now, let us make sure that this algorithm does indeed do what we promised.

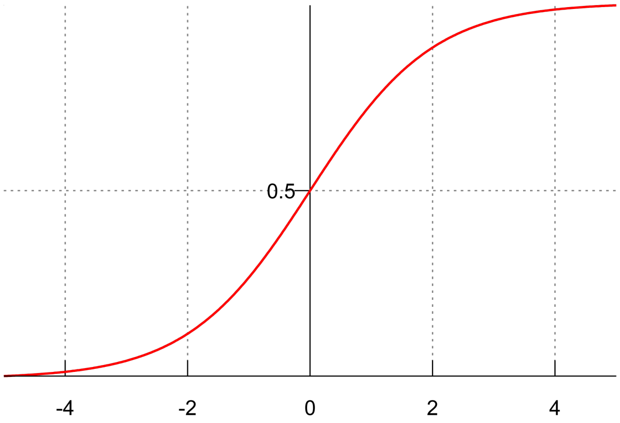


Figure 3.1 The logistic function.

Note that the only new variable in this algorithm is α, known as the learning rate. The learning rate takes a positive value that adjusts up or down the values used during the update. We will revisit this idea later on in this chapter. For now, let us assume $\alpha = 1$.

It is easy to see that, at the extreme (i.e., when the prediction is perfectly correct or incorrect), this algorithm reduces to the perceptron learning algorithm. For example, when the prediction is perfectly correct (say $y_i = 1$ for the class associated with 1), y_i is equal to d, which means that there is no weight or bias update in lines 6 and 7. This is similar to the perceptron (lines 6 and 7 in Algorithm 2). Further, when a prediction is perfectly incorrect, say, $y_i = 1$ (Yes) when $d = 0$ (No), this reduces to adding $\mathbf{x}_i$ to $\mathbf{w}$ and 1 to b (similar to the perceptron update, lines 8 – 10 in Algorithm 2). When $y_i = 0$ when $d = 1$, the algorithm reduces to subtracting $\mathbf{x}_i$ from $\mathbf{w}$ and 1 from b (similar to lines 11 – 13 in Algorithm 2).

The interesting behavior occurs in the majority of the situations when the LR decision is neither perfectly correct nor perfectly incorrect. In these situations, the LR performs a soft update that is proportional with how incorrect the current decision is, which is captured by $\mathrm{y}_i - \mathrm{d}$. That is, the more incorrect the decision is, the larger the update. This is exactly what we would like a good learning algorithm to do.

Once the algorithm finishes training, we would like to use the learned weights ($\mathbf{w}$ and b) to perform binary classification, e.g., classify a text into a positive or negative review. For this, at prediction time we will convert the LR decision into a discrete

Algorithm 4: Logistic regression learning algorithm.

```
$\mathbf{w} = 0$
$b = 0$
while not converged do
    for each training example $\mathbf{x}_i$ in X do
        $d = \text{decision}(\mathbf{x}_i, \mathbf{w}, b)$
        $\mathbf{w} = \mathbf{w} + \alpha(y_i - d)\mathbf{x}_i$
        $b = b + \alpha(y_i - d)$
    end
end
```

output using a threshold τ, commonly set to 0.5.[1] That is, if decision$(\mathbf{x}, \mathbf{w}, b) \geq 0.5$ then the algorithm outputs one class (say, positive review); otherwise it outputs the other class.

3.2 The Logistic Regression Cost Function

The next three sections of this chapter focus on deriving the LR learning algorithm shown in Algorithm 4. The reader who is averse to math, or is satisfied with the learning algorithm and the intuition behind it, may skip to Section 3.7. However, we encourage the reader to try to stay with us through this derivation. These sections introduce important concepts, i.e., cost functions and gradient descent, which are necessary for a thorough understanding of the following chapters in this book. We will provide pointers to additional reading, where more mathematical background may be needed.

The first observation that will help us formalize the training process for LR is that the LR decision function implements a conditional probability, i.e., the probability of generating a specific label given a training example and the current weights. More formally, we can write:

$$p(y = 1|\mathbf{x}; \mathbf{w}, b) = \sigma(\mathbf{x}; \mathbf{w}, b) \tag{3.2}$$

The left term of the above equation can be read as the probability of generating a label y equal to 1, given a training example $\mathbf{x}$ and model weights $\mathbf{w}$ and b (the vertical bar "|" in the conditional probability formula should be read as "given"). Intuitively,

[1] Other values for this threshold are possible. For example, for applications where it is important to be conservative with predictions for class 1, τ would take values larger than 0.5.

this probability is an indicator of confidence (the higher the better). That is, the probability approaches 1 when the model is confident that the label for $\mathbf{x}$ is 1, and 0 when not. Similarly, the probability of y being 0 is:

$$p(y=0|\mathbf{x};\mathbf{w},b) = 1 - \sigma(\mathbf{x};\mathbf{w},b) \tag{3.3}$$

These probabilities form a probability distribution, i.e., the sum of probabilities over all possible labels equals 1. Note that while we aim to minimize the use of probability theory in this section, some of it is unavoidable. The reader who wants to brush up on probability theory may consult other material on this topic such as [Griffiths 2008].

To simplify notations, because we now know that we estimate label probabilities, we change the notation for the two probabilities to: $p(1|\mathbf{x};\mathbf{w},b)$ and $p(0|\mathbf{x};\mathbf{w},b)$. Further, when it is obvious what the model weights are, we will skip them and use simply $p(1|\mathbf{x})$ and $p(0|\mathbf{x})$. Lastly, we generalize the above two formulas to work for any of the two possible labels with the following formula:

$$p(y|\mathbf{x}) = (\sigma(\mathbf{x};\mathbf{w},b))^y(1-\sigma(\mathbf{x};\mathbf{w},b))^{1-y} \tag{3.4}$$

It is trivial to verify that this formula reduces to one of the two equations above, for $y=1$ and $y=0$.

Intuitively, we would like the LR training process to maximize the probability of the correct labels in the entire training dataset. This probability is called the *likelihood of the data* (L), and is formalized as:

$$L(\mathbf{w},b) = p(\mathbf{y}|\mathbf{X}) \tag{3.5}$$

$$= \Pi_{i=1}^{m} p(y_i|\mathbf{x}_i) \tag{3.6}$$

where $\mathbf{y}$ is the vector containing all the correct labels for all training examples, $\mathbf{X}$ is the matrix that contains the vectors of features for all training examples, and m is the total number of examples in the training dataset. Note that the derivation into the product of individual probabilities is possible because we assume that the training examples are independent of each other, and the joint probability of multiple independent events is equal to the product of individual probabilities [Griffiths 2008].

A common convention in machine learning is that instead of maximizing a function during learning, we instead aim to minimize a *cost* or *loss* function C, which captures the amount of errors in the model. By definition, C must take only positive values. That is, C will have large values when the model does not perform well, and is 0 when the learned model is perfect. We write the logistic regression cost function C in terms of likelihood L as:

$$C(\mathbf{w},b) = -\log L(\mathbf{w},b) \tag{3.7}$$

$$= -\sum_{i=1}^{m}(y_i \log \sigma(\mathbf{x}_i;\mathbf{w},b) + (1-y_i)\log(1-\sigma(\mathbf{x}_i;\mathbf{w},b))) \tag{3.8}$$

Equation 3.7 is often referred to as the *negative log likelihood* of the data. It is easy to see that C satisfies the constraints of a cost function. First, it is always positive: the logarithm of a number between 0 and 1 is negative; the negative sign in front of the sum turns the value of the sum into a positive number. Second, the cost function takes large values when the model makes many mistakes (i.e., the likelihood of the data is small), and approaches 0 when the model is correct (i.e., the likelihood approaches 1).

Thus, we can formalize the goal of the LR learning algorithm as minimizing the above cost function. Next we will discuss how we do this efficiently.

3.3 Gradient Descent

The missing component that connects the cost function just introduced with the LR training algorithm (Algorithm 4) is gradient descent. Gradient descent is an iterative method that finds the parameters that minimize a given function. In our context, we will use gradient descent to find the LR parameters ($\mathbf{w}$ and b) that minimize the cost function C.

However, for illustration purposes, let us take a step away from the LR cost function and begin with a simpler example: let us assume we would like to minimize the function $f(x) = (x+1)^2 + 1$, which is plotted in Figure 3.2. Clearly, the smallest value this function takes is 1, which is obtained when $x = -1$. Gradient descent finds this value by taking advantage of the function slope, or derivative of $f(x)$ with respect to x, i.e., $\frac{d}{dx}f(x)$. Note: if the reader needs a refresher of what function derivatives are, and how to compute them, now is a good time to do so. Any calculus textbook or even the Wikipedia page for function derivatives[2] provide sufficient information for what we need in this book.

One important observation about the slope of a function is that it indicates the function's direction of change. That is, if the derivative is negative, the function decreases; if it is positive, the function increases; and if it is zero, we have reached a local minimum or maximum for the function. Let us verify that is the case for our simple example. The derivative of our function $\frac{d}{dx}((x+1)^2+1)$ is $2(x+1)$, which has negative values when $x < -1$, positive values when $x > -1$, and is 0 when $x = -1$. Intuitively, gradient descent uses this observation to take small steps towards the function's minimum in the opposite direction indicated by the slope. More formally,

[2] https://en.wikipedia.org/wiki/Derivative

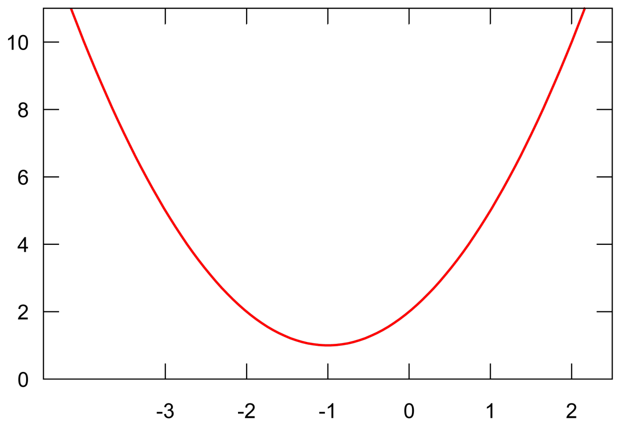


Figure 3.2 Plot of the function $f(x) = (x+1)^2 + 1$.

gradient descent starts by initializing x with some random value, e.g., $x = -3$, and then repeatedly subtracts a quantity proportional with the derivative from x, until it *converges*, i.e., it reaches a derivative of 0 (or close enough so we can declare success). That is, we repeatedly compute:

$$x = x - \alpha \frac{d}{dx} f(x) \tag{3.9}$$

until convergence.

Sidebar 3.1 Partial derivative notation

In this book we use the Leibniz notation for derivatives. That is, $\frac{d}{dx}f$ indicates the derivative of function f with respect to x, i.e., the amount of change in f in response to an infinitesimal change in x. This notation is equivalent to the Lagrange notation (sometimes attributed to Newton) of $f'(x)$.

α in the above equation is the same learning rate introduced before in this chapter. Let us set $\alpha = 0.1$ for this example. Thus, in the first gradient descent iteration, x changes to $x = -3 - 0.1 \times 2(-3+1) = -2.6$. In the second iteration, x becomes $x = -2.6 - 0.1 \times 2(-2.6+1) = -2.28$. And so on, until, after approximately 30 iterations, x approaches -1.001, a value practically identical to what we were looking for.

This simple example also highlights that the learning rate α must be positive (so we don't change the direction indicated by the slope), and small (so we do not "Tony Hawk" the data). To demonstrate the latter situation, consider the situation when $\alpha = 1$. In this case, in the first iteration x becomes 1, which means we already skipped over the value that yields the function's minimum ($x = -1$). Even worse, in the second iteration, x goes back to -3, and we are now in danger of entering an infinite loop! To mitigate this situation, α usually takes small positive values, say, between 0.00001 and 0.1. In Chapter 6 we will discuss other strategies to dynamically shrink the learning rate as the learning advances, so we further reduce our chance of missing the function's minimum.

The gradient descent algorithm generalizes to functions with multiple parameters: we simply update each parameter using its own partial derivative of the function to be minimized. For example, consider a new function that has two parameters, x_1 and x_2: $f(x_1, x_2) = (x_1 + 1)^2 + 3x_2 + 1$. For this function, in each gradient descent iteration, we perform the following updates:

$$\begin{aligned} x_1 &= x_1 - \alpha \frac{d}{dx_1} f(x_1, x_2) = x_1 - 0.1(2x_1 + 2) \\ x_2 &= x_2 - \alpha \frac{d}{dx_2} f(x_1, x_2) = x_2 - 0.1(3) \end{aligned}$$

or, in general, for a function $f(\mathbf{x})$, we update each parameter x_i using the formula:

$$x_i = x_i - \alpha \frac{d}{dx_i} f(\mathbf{x}) \tag{3.10}$$

One obvious question that should arise at this moment is why are we not simply solving the equation where the derivative equals 0, as we were taught in calculus? For instance, for the first simple example we looked at, $f(x) = (x+1)^2 + 1$, zeroing the derivative yields immediately the exact solution $x = -1$. While this approach works well for functions with a single parameter or two, it becomes prohibitively expensive for functions with four or more parameters. Machine learning in general falls in this latter camp: it is very common that the functions we aim to minimize have thousands (or even millions) of parameters. In contrast, as we will see later, gradient descent provides a solution whose runtime is linear in the number of parameters times the number of training examples.

It is important to note that gradient descent is not perfect. It does indeed work well for convex functions, i.e., functions that have exactly one minimum and are differentiable at every point such as our simple example, but it does not perform so well

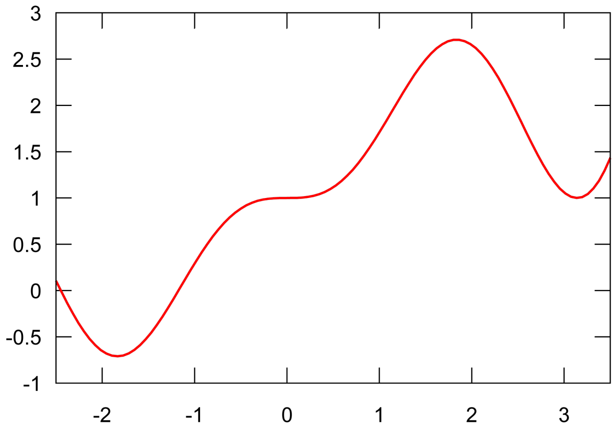


Figure 3.3 Plot of the function $f(x) = x\sin(x)^2 + 1$.

in more complex situations. Consider for example the function shown in Figure 3.3.[3] This functions has two minima (around $x = 3$ and $x = -2$). Because gradient descent is a "greedy" algorithm, i.e., it commits to a solution relying only on local knowledge without understanding the bigger picture, it may end up finding a minimum that is not the best. For example, if x is initialized with 2.5, gradient descent will follow the negative slope at that position, and end up discovering the minimum around $x = 3$, which is not the best solution. However, despite this known limitation, gradient descent works surprisingly well in practice.

Now that we have a general strategy for finding the parameters that minimize a function, let us apply it to the problem we care about in this chapter, that is, finding the parameters $\mathbf{w}$ and b that minimize the cost function $C(\mathbf{w}, b)$ (Equation 3.8). A common source of confusion here is that the parameters of C are $\mathbf{w}$ and b, not $\mathbf{x}$ and y. For a given training example, $\mathbf{x}$ and y are known and constant. That is, we know the values of the features and the label for each given example in training, and all we have to do is compute $\mathbf{w}$ and b. Thus, the training process of LR reduces to repeatedly updating each w_j in $\mathbf{w}$ and b features by the corresponding partial derivative of C:

[3] This example of a function with multiple minima taken from https://en.wikipedia.org/wiki/Derivative.

$$w_j = w_j - \alpha \frac{d}{dw_j} C(\mathbf{w}, b) \tag{3.11}$$

$$b = b - \alpha \frac{d}{db} C(\mathbf{w}, b) \tag{3.12}$$

Assuming a sufficient number of iterations, and a learning rate α that is not too large, $\mathbf{w}$ and b are guaranteed to converge to the optimal values because the logistic regression cost function is convex.[4] However, one problem with this approach is that computing the two partial derivatives requires the inspection of *all* training examples (this is what the summation in Equation 3.8 indicates), which means that the learning algorithm would have to do many passes over the training dataset before any meaningful changes are observed. Because of this, in practice, we do not compute C over the whole training data, but over a small number of examples at a time. This small group of examples is called a *mini batch*. In the simplest case, the size of the mini batch is 1, i.e., we update the $\mathbf{w}$ and b weights after seeing each individual example i, using a cost function computed for example i alone:

$$C_i(\mathbf{w}, b) = -(y_i \log \sigma(\mathbf{x}_i; \mathbf{w}, b) + (1 - y_i) \log(1 - \sigma(\mathbf{x}_i; \mathbf{w}, b))) \tag{3.13}$$

This simplified form of gradient descent is called *stochastic gradient descent* (SGD), where "stochastic" indicates that we work with a stochastic approximation (or an estimate) of C. Building from the last three equations above, we can write the logistic regression training algorithm as shown in Algorithm 5. The reader will immediately see that this formulation of the algorithm is similar to Algorithm 4, which we introduced at the beginning of this chapter. In the next section, we will demonstrate that these two algorithms are indeed equivalent, by computing the two partial derivatives $\frac{d}{dw_j} C_i(\mathbf{w}, b)$ and $\frac{d}{db} C_i(\mathbf{w}, b)$.

3.4 Deriving the Logistic Regression Update Rule

Here we will compute the partial derivative of the cost function $C_i(\mathbf{w}, b)$ of an individual example i, with respect to each feature weight w_j and bias term b. For these operations we will rely on several rules to compute the derivatives of a few necessary functions. These rules are listed in Table 3.1.

Let us start with the derivative of C with respect to one feature weight w_j:

[4] Demonstrating that the LR cost function is convex is beyond the scope of this book. The interested reader may read other materials on this topic such as http://mathgotchas.blogspot.com/2011/10/why-is-error-function-minimized-in.html.

Algorithm 5: Logistic regression learning algorithm using stochastic gradient descent.

```
w = 0
b = 0
while not converged do
    for each training example $\mathbf{x}_i$ in X do
        for each $w_j$ in w do
            $w_j = w_j - \alpha \frac{d}{dw_j} C_i(\mathbf{w}, b)$
        end
        $b = b - \alpha \frac{d}{db} C_i(\mathbf{w}, b)$
    end
end
```

$$\frac{d}{dw_j} C_i(\mathbf{w}, b) = \frac{d}{dw_j}(-y_i \log \sigma(\mathbf{x}_i; \mathbf{w}, b) - (1 - y_i) \log(1 - \sigma(\mathbf{x}_i; \mathbf{w}, b)))$$

Let us use σ_i to denote $\sigma(\mathbf{x}_i; \mathbf{w}, b)$ below, for simplicity:

$$= \frac{d}{dw_j}(-y_i \log \sigma_i - (1 - y_i) \log(1 - \sigma_i))$$

Pulling out the y_i constants and then applying the chain rule on the two logarithms:

$$= -y_i \frac{d}{d\sigma_i} \log \sigma_i \frac{d}{dw_j} \sigma_i - (1 - y_i) \frac{d}{d(1 - \sigma_i)} \log(1 - \sigma_i) \frac{d}{dw_j}(1 - \sigma_i)$$

After applying the derivative of the logarithm:

$$= -y_i \frac{1}{\sigma_i} \frac{d}{dw_j} \sigma_i - (1 - y_i) \frac{1}{1 - \sigma_i} \frac{d}{dw_j}(1 - \sigma_i)$$

After applying the chain rule on $\frac{d}{dw_j}(1-\sigma_i)$:

$$\begin{aligned}
&= -y_i \frac{1}{\sigma_i} \frac{d}{dw_j}\sigma_i + (1-y_i)\frac{1}{1-\sigma_i}\frac{d}{dw_j}\sigma_i \\
&= (-y_i \frac{1}{\sigma_i} + (1-y_i)\frac{1}{1-\sigma_i})\frac{d}{dw_j}\sigma_i \\
&= \frac{-y_i(1-\sigma_i) + (1-y_i)\sigma_i}{\sigma_i(1-\sigma_i)}\frac{d}{dw_j}\sigma_i \\
&= \frac{\sigma_i - y_i}{\sigma_i(1-\sigma_i)}\frac{d}{dw_j}\sigma_i
\end{aligned}$$

After applying the chain rule on σ_i:

$$= \frac{\sigma_i - y_i}{\sigma_i(1-\sigma_i)} \frac{d}{d(\mathbf{w}\cdot\mathbf{x}_i + b)}\sigma_i \frac{d}{dw_j}(\mathbf{w}\cdot\mathbf{x}_i + b)$$

After the derivative of the logistic function and then canceling numerator and denominator:

$$\begin{aligned}
&= \frac{\sigma_i - y_i}{\sigma_i(1-\sigma_i)}\sigma_i(1-\sigma_i)\frac{d}{dw_j}(\mathbf{w}\cdot\mathbf{x}_i + b) \\
&= (\sigma_i - y_i)\frac{d}{dw_j}(\mathbf{w}\cdot\mathbf{x}_i + b)
\end{aligned}$$

Lastly, after applying the derivative of the dot product:

$$= (\sigma_i - y_i)x_{ij} \tag{3.14}$$

where x_{ij} is the value of feature j in the feature vector $\mathbf{x_i}$.

Following a similar process, we can compute the derivative of C_i with respect to the bias term as:

$$\frac{d}{db}C_i(\mathbf{w},b) = \frac{d}{db}(-y_i \log\sigma(\mathbf{x}_i;\mathbf{w},b) - (1-y_i)\log(1-\sigma(\mathbf{x}_i;\mathbf{w},b))) = \sigma_i - y_i \tag{3.15}$$

Knowing that σ_i is equivalent with decision($\mathbf{x}_i$, $\mathbf{w}$, b), one can immediately see that applying Equation 3.15 in line 8 of Algorithm 5 transforms the update of the bias into the form used in Algorithm 4 (line 7). Similarly, replacing the partial derivative in line 6 of Algorithm 5 with its explicit form from Equation 3.14 yields an update equivalent with the weight update used in Algorithm 4. The superficial difference between the two algorithms is that Algorithm 5 updates each feature weight w_j explicitly, whereas Algorithm 4 updates *all* weights at once by updating the entire vector $\mathbf{w}$. Needless to say, these two forms are equivalent. We prefer the explicit description in Algorithm 5 for clarity. But, in practice, one is more likely to implement Algorithm 4 because vector operations are efficiently implemented in most machine learning software libraries.

Table 3.1 Rules of computation for a few functions necessary to derive the logistic regression update rules. In these formulas, f and g are functions, a and b are constants, x is a variable.

Description	Formula
Chain rule	$\frac{d}{dx}f(g(x)) = \frac{d}{dg(x)}f(g(x))\frac{d}{dx}g(x)$
Derivative of summation	$\frac{d}{dx}(af(x)+bg(x))) = a\frac{d}{dx}f(x)+b\frac{d}{dx}g(x)$
Derivative of natural logarithm	$\frac{d}{dx}\log(x) = \frac{1}{x}$
Derivative of logistic	$\frac{d}{dx}\sigma(x) = \frac{d}{dx}(\frac{1}{1+e^{-x}}) = -\frac{1}{(1+e^{-x})^2}(-e^{-x}) = \sigma(x)(1-\sigma(x))$
Derivative of dot product between vectors **x** and **a** with respect to x_i	$\frac{d}{dx_i}(\mathbf{x}\cdot\mathbf{a}) = a_i$

3.5 From Binary to Multiclass Classification

So far, we have discussed binary logistic regression, where we learned a classifier for two labels (1 and 0), where the probability of predicting label 1 is computed as: $p(1|\mathbf{x};\mathbf{w},b) = \sigma(\mathbf{x};\mathbf{w},b)$ and probability of label 0 is: $p(0|\mathbf{x};\mathbf{w},b) = 1 - p(1|\mathbf{x};\mathbf{w},b) = 1-\sigma(\mathbf{x};\mathbf{w},b)$. However, there are many text classification problems where two labels are not sufficient. For example, we might decide to implement a movie review classifier that produces five labels, to capture ratings on a five-star scale. To accommodate this class of problems, we need to generalize the binary LR algorithm to multiclass scenarios, where the labels to be learned may take values from 1 to k, where k is the number of classes to be learned, e.g., 5 in the previous example.

Figure 3.4 provides a graphical explanation of the multiclass LR. The key observation is that now, instead of maintaining a single weight vector $\mathbf{w}$ and bias b, we maintain one such vector and bias term *for each* class to be learned. This complicates our notations a bit: instead of using a single index to identify positions in an input vector $\mathbf{x}$ or in $\mathbf{w}$, we now have to maintain two. That is, we will use $\mathbf{w}_i$ to indicate the weight vector for class i, w_{ij} to point to the weight of the edge that connects the input x_j to the class i, and b_i to indicate the bias term for class i. The output of each "neuron" i in the figure produces a score for label i, defined as the sum between the

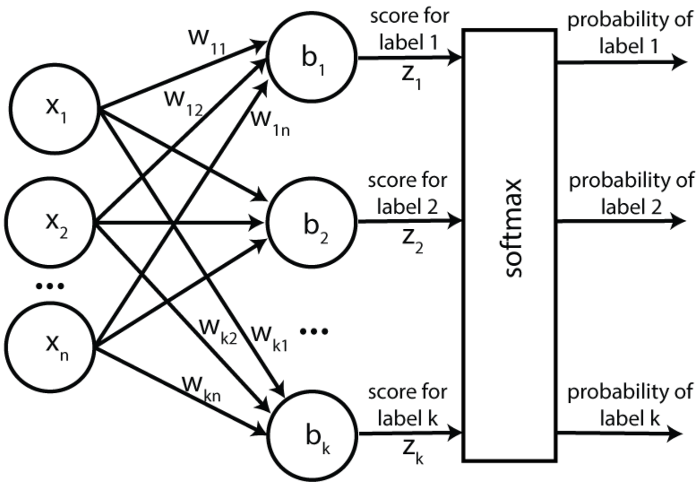


Figure 3.4 Multiclass logistic regression.

bias term of class i and the dot product of the weight vector for class i and the input vector. More formally, if we use z_i to indicate the score for label i, then: $z_i = \mathbf{w}_i \cdot \mathbf{x} + b_i$.

Note that these scores are not probabilities: they are not bounded between 0 and 1, and they will not sum up to 1. To turn them into probabilities, we are introducing a new function, called softmax, which produces probability values for the k classes. For each class i, softmax defines the corresponding probability as:

$$p(y = i|\mathbf{x};\mathbf{W},\mathbf{b}) = p(i|\mathbf{x};\mathbf{W},\mathbf{b}) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} = \frac{e^{\mathbf{w}_i \cdot \mathbf{x} + b_i}}{\sum_{j=1}^{k} e^{\mathbf{w}_j \cdot \mathbf{x} + b_j}} \tag{3.16}$$

where the $\mathbf{W}$ matrix stores all $\mathbf{w}$ weight vectors, i.e., row i in $\mathbf{W}$ stores the weight vector $\mathbf{w}_i$ for class i, and the $\mathbf{b}$ vector stores all bias values, i.e., b_i is the bias term for class i.

Clearly, the softmax function produces probabilities: (a) the exponent function used guarantees that the softmax values are positives, and (b) the denominator, which sums over all the k classes guarantees that the resulting values are between 0 and 1, and sum up to 1. Further, with just a bit of math, we can show that the softmax for two classes reduces to the logistic function. Using the softmax formula, the probability of class 1 in a two-class LR (using labels 1 and 0) is:

$$\begin{aligned} p(1|\mathbf{x};\mathbf{W},\mathbf{b}) &= \frac{e^{\mathbf{w}_1\cdot\mathbf{x}+b_1}}{e^{\mathbf{w}_0\cdot\mathbf{x}+b_0}+e^{\mathbf{w}_1\cdot\mathbf{x}+b_1}} = \frac{1}{\frac{e^{\mathbf{w}_0\cdot\mathbf{x}+b_0}}{e^{\mathbf{w}_1\cdot\mathbf{x}+b_1}}+1} \\ &= \frac{1}{e^{-((\mathbf{w}_1-\mathbf{w}_0)\cdot\mathbf{x}+(b_1-b_0))}+1} \end{aligned} \tag{3.17}$$

Using a similar derivation, which we leave as an at-home exercise to the curious reader, the probability of class 0 is:

$$\begin{aligned} p(0|\mathbf{x};\mathbf{W},\mathbf{b}) &= \frac{e^{\mathbf{w}_0\cdot\mathbf{x}+b_0}}{e^{\mathbf{w}_0\cdot\mathbf{x}+b_0}+e^{\mathbf{w}_1\cdot\mathbf{x}+b_1}} = \frac{e^{-((\mathbf{w}_1-\mathbf{w}_0)\cdot\mathbf{x}+(b_1-b_0))}}{e^{-((\mathbf{w}_1-\mathbf{w}_0)\cdot\mathbf{x}+(b_1-b_0))}+1} \\ &= 1-p(1|\mathbf{x};\mathbf{W},\mathbf{b}) \end{aligned} \tag{3.18}$$

From these two equations, we can immediately see that two the formulations of binary LR, i.e., logistic vs. softmax, are equivalent when we set the parameters of the logistic to be equal to the the difference between the parameters of class 1 and the parameters of class 0 in the softmax formulation, or $\mathbf{w} = \mathbf{w}_1 - \mathbf{w}_0$, and $b = b_1 - b_0$, where $\mathbf{w}$ and b are the logistic parameters in Equations 3.2 and 3.3.

The cost function for multiclass LR follows the same intuition and formalization as the one for binary LR. That is, during training we want to maximize the probabilities of the correct labels assigned to training examples, or, equivalently, we want to minimize the negative log likelihood of the data. Similarly to Equation 3.7, the cost function for multiclass LR is defined as:

$$C(\mathbf{W},\mathbf{b}) = -\log L(\mathbf{W},\mathbf{b}) = -\sum_{i=1}^{m} \log p(y_i|\mathbf{x}_i;\mathbf{W},\mathbf{b}) \tag{3.19}$$

or, for a single training example i:

$$C_i(\mathbf{W},\mathbf{b}) = -\log p(y_i|\mathbf{x}_i;\mathbf{W},\mathbf{b}) \tag{3.20}$$

where y_i is the correct label for training example i, and $\mathbf{x}_i$ is the feature vector for the same example. The probabilities in this cost function are computed using the softmax formula, as in Equation 3.16. This cost function, which generalizes the negative log likelihood cost function to multiclass classification, is called *cross entropy*. Its form for binary classification (Equation 3.8) is called *binary cross entropy*. These are probably the most commonly used cost function in NLP problems. We will see them a lot throughout the book.

The learning algorithm for multiclass LR stays almost the same as Algorithm 5, with small changes to account for the different cost function and the larger number of parameters, i.e., we now update a matrix $\mathbf{W}$ instead of a single vector $\mathbf{w}$, and a vector $\mathbf{b}$ instead of the scalar b. The adjusted algorithm is shown in Algorithm 6. We leave the computation of the derivatives used in Algorithm 6 as an at-home exercise

Algorithm 6: Learning algorithm for multiclass logistic regression.

```
$\mathbf{W} = 0$
$\mathbf{b} = 0$
while not converged do
    for each training example $\mathbf{x}_i$ in $\mathbf{X}$ do
        for each $w_{jk}$ in $\mathbf{W}$ do
            $w_{jk} = w_{jk} - \alpha \frac{d}{dw_{jk}} C_i(\mathbf{W}, \mathbf{b})$
        end
        for each $b_j$ in $\mathbf{b}$ do
            $b_j = b_j - \alpha \frac{d}{db_j} C_i(\mathbf{W}, \mathbf{b})$
        end
    end
end
```

Table 3.2 Example of a confusion matrix for three-class classification. The dataset contains 1,000 data points, with 2 data points in class $C1$, 100 in class $C2$, and 898 in class $C3$.

	Classifier predicted $C1$	Classifier predicted $C2$	Classifier predicted $C3$
Gold label is $C1$	1	1	0
Gold label is $C2$	10	80	10
Gold label is $C3$	1	7	890

for the interested reader. However, as we will see in the next chapter, we can now rely on automatic differentiation libraries such as PyTorch to compute these derivatives for us, so this exercise is not strictly needed to implement multiclass LR.

3.6 Evaluation Measures for Multiclass Text Classification

Now that we generalized our classifier to operate over an arbitrary number of classes, it is time to generalize the evaluation measures introduced in Section 2.3 to multiclass problems as well. Throughout this section, we will use as a walkthrough example a three-class version of the Medicaid application classification problem from Section 2.3. In this version, our classifier has to assign each application to one of three classes, where classes $C1$ and $C2$ indicate the high- and medium-priority applications, and class $C3$ indicate regular applications that do not need to be rushed through the system. Same as before, most applications fall under class $C3$. Table 3.2 shows an example confusion matrix for this problem for a hypothetical three-class classifier that operates over an evaluation dataset that contains 1,000 applications.

The definition of accuracy remains essentially the same for multiclass classification, i.e., accuracy is the ratio of data points classified correctly. In general, the number of correctly classified points can be computed by summing up the counts on the diagonal of the confusion matrix. For example, for the confusion matrix shown in Table 3.2, accuracy is $\frac{1+80+890}{1,000} = \frac{971}{1,000}$.

Similarly, the definitions of precision and recall for an individual class c, remain the same:

$$P_c = \frac{TP_c}{TP_c + FP_c} \tag{3.21}$$

$$R_c = \frac{TP_c}{TP_c + FN_c} \tag{3.22}$$

where TP_c indicate the number of true positives for class c, FP_c indicate the number of positives for class c, and FN_c indicate the number of false negatives for the same class. However, because we now have more than two rows and two columns in the confusion matrix, we have to do a bit more additional math to compute the FP_c and FN_c counts. In general, the number of false positives for a class c is equal to the sum of the counts in the column corresponding to class c, excluding the element on the diagonal. The number of of false negatives for a class c is equal to the sum of the counts in the corresponding row, excluding the element on the diagonal. For example, for class $C2$ in the table, the number of true positives is $TP_{C2} = 80$, the number of false positives is $FP_{C2} = 1+7 = 8$, and the number of false negatives is $FN_{C2} = 10+10 = 20$. Thus, the precision and recall for class $C2$ are: $P_{C2} = \frac{80}{80+8} = 0.91$, and $R_{C2} = \frac{80}{80+20} = 0.80$. We leave it as an at-home exercise to show that $P_{C1} = 0.08$, $R_{C1} = 0.5$, $P_{C3} = 0.99$, and $R_{C3} = 0.99$. From these values, one can trivially compute the respective F scores per class.

The important discussion for multiclass classification is how to average these sets of precision/recall scores into single values that will give us a quick understanding of the classifier's performance? There are two strategies to this end, both with advantages and disadvantages:

Macro averaging: Under this strategy we simply average all the individual precision/recall scores into a single value. For example, for the above example, the macro precision score over all three classes is: macro $P = \frac{P_{C1}+P_{C2}+P_{C3}}{3} = \frac{0.08+0.91+0.99}{3} = 0.66$. Similarly, the macro recall score is: macro $R = \frac{R_{C1}+R_{C2}+R_{C3}}{3} = \frac{0.50+0.80+0.99}{3} = 0.76$. The macro F_1 score is the harmonic mean of the macro precision and recall scores.

As discussed in Section 2.3, in many NLP tasks the labels are highly unbalanced, and we commonly care less about the most frequent label. For example, here we may want to measure the performance of our classifier on classes $C1$ and $C2$, which require rushed processing in the Medicaid system. In such scenarios, the macro precision and recall scores exclude the frequent class, e.g., $C3$ in our case. Thus, the macro precision becomes: macro $P = \frac{P_{C1}+P_{C2}}{2} = \frac{0.08+0.91}{2} = 0.50$, which is more indicative of the fact that our classifier does not perform too well on the two important classes in this example.

The advantage of the macro scores is that they treat all the classes we are interested in as equal contributors to the overall score. But, depending on the task, this may also be a disadvantage. For example, in the above example, the latter macro precision score of 0.50 hides the fact that our classifier performs reasonably well on the $C2$ class ($P_{C2} = 0.91$), which is 100 times more frequent than $C1$ in the data!

Micro averaging: This strategy addresses the above disadvantage of macro averaging, by computing overall precision, recall, and F scores where each class contributes proportionally with its frequency in the data. In particular, rather than averaging the individual precision/recall scores, we compute them using the class counts directly. For example, the micro precision and recall scores for the two classes of interest in the above example, $C1$ and $C2$, are:

$$\text{micro } P = \frac{TP_{C1}+TP_{C2}}{TP_{C1}+TP_{C2}+FP_{C1}+FP_{C2}} = \frac{1+80}{1+80+11+8} = 0.81 \tag{3.23}$$

$$\text{micro } R = \frac{TP_{C1}+TP_{C2}}{TP_{C1}+TP_{C2}+FN_{C1}+FN_{C2}} = \frac{1+80}{1+80+1+20} = 0.79 \tag{3.24}$$

Similar to macro averaging, the micro F_1 score is computed as the harmonic mean of the micro precision and recall scores.

Note that in this example, the micro scores are considerably higher than the corresponding macro scores because: (a) the classifier's performance on the more frequent $C2$ class is higher than the performance on class $C1$, and (b) micro averaging assigns more importance to the frequent classes, which, in this case, raises the micro precision and recall scores. The decision of which averaging strategy to use is problem specific, and depends on the answer to the question: should all classes be treated equally during scoring, or should they be weighted by their frequency in the data? In the former case, the appropriate averaging is macro; in the latter, micro.

3.7 Drawbacks of Logistic Regression

The logistic regression algorithm solves the lack of smooth updates in the perceptron algorithm through its improved update functions on its parameters. This seemingly

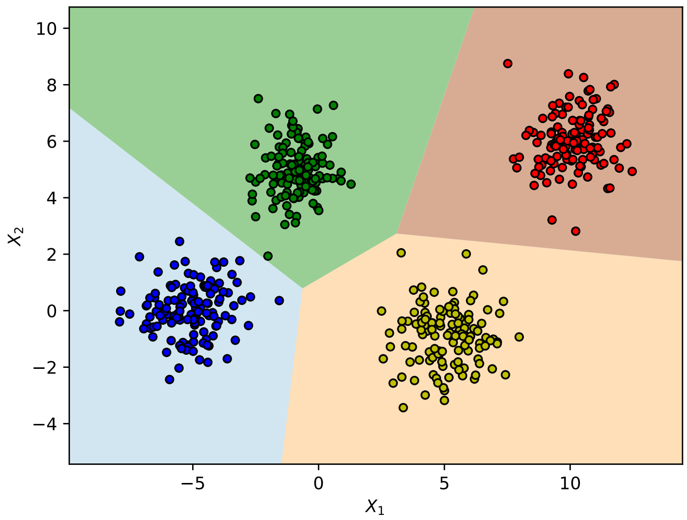


Figure 3.5 Example of a two-dimensional decision boundary for a 4-class logistic regression classifier. This figure was generated by Clayton Morrison and is reproduced with permission.

small change has an important practical impact: in most NLP applications, logistic regression tends to outperform the perceptron.

However, the other drawbacks observed with the perceptron still hold. Binary logistic regression is also a linear classifier because its decision boundary remains a hyperplane. It is tempting to say that the above statement is not correct because the logistic is clearly a non-linear function. However, the linearity of the binary LR classifier is easy to prove with just a bit of math. Remember that the decision function for the binary LR is: if $\frac{1}{1+e^{-(\mathbf{w}\cdot\mathbf{x}+b)}} \geq 0.5$ we assign one label, and if $\frac{1}{1+e^{-(\mathbf{w}\cdot\mathbf{x}+b)}} < 0.5$ we assign the other label. Thus, the decision boundary is defined by the equation $\frac{1}{1+e^{-(\mathbf{w}\cdot\mathbf{x}+b)}} = 0.5$. From this we can easily derive that $e^{-(\mathbf{w}\cdot\mathbf{x}+b)} = 1$, and $-(\mathbf{w}\cdot\mathbf{x}+b) = 0$, where the latter is a linear function on the parameters $\mathbf{w}$ and b. This observation generalizes to the multiclass logistic regression introduced in Section 3.5. In the multiclass scenario, the decision boundary between all classes consists of multiple intersecting segments, each of which are fragments of a hyperplane. Figure 3.5 shows an example of such a decision boundary for a 4-class problem, where each data point is described by two features: x_1 and x_2. Clearly, forcing these segments to be linear reduces what the multiclass logistic regression can learn.

Further, similar to the perceptron, the LR covered so far relies on hand-crafted features, which, as discussed in the previous chapter, may be cumbersome to generate and may generalize poorly. Lastly, logistic regression also focuses on individual predictions rather than structured learning. We will address all these limitations in the following chapters. We will start by introducing non-linear classifiers in Chapter 5.

3.8 Historical Background

TODO: To do

3.9 References and Further Readings

TODO: To do

4 Implementing a Review Classifier Using Logistic Regression

5 Feed Forward Neural Networks

So far we have explored classifiers with decision boundaries that are linear, or, in the case of the multiclass logistic regression, a combination of linear segments. In this chapter, we will expand what we have learned so far to classifiers that are capable of learning non-linear decision boundaries. The classifiers that we will discuss here are called *feed forward neural networks*, and are a generalization of both logistic regression and the perceptron. Without going into the theory behind it, it has been shown that, under certain conditions, these classifiers can approximate any function [Hornik 1991, Leshno et al. 1993]. That is, they can learn decision boundaries of any arbitrary shape. Figure 5.1 shows a very simple example of a hypothetical situation where a non-linear decision boundary is needed for a binary classifier.

The good news is that we have already introduced the building blocks of feed forward neural networks (FFNN): the individual neuron, and the stochastic gradient learning algorithm. In this chapter, we are simply combining these building blocks in slightly more complicated ways, but without changing any of the fundamental operating principles.

5.1 Architecture of Feed Forward Neural Networks

Figure 5.2 shows the general architecture of FFNNs. As seen in the figure, FFNNs combine multiple layers of individual neurons, where each neuron in a layer l is fully connected to all neurons in the next layer, $l+1$. Because of this, architectures such as the one in the figure are often referred to as *fully-connected FFNNs*. This is not the only possible architecture for FFNNs: any arbitrary connections between neurons are possible. However, because fully-connected networks are the most common FFNN architecture seen in NLP, we will focus on these in this chapter, and omit the fully-connected modifier from now on, for simplicity.

Figure 5.2 shows that the neuron layers in a FFNN are grouped into three categories. These are worth explaining in detail:

Input layer: Similar to the perceptron or logistic regression, the input layer contains a vector $\mathbf{x}$ that describes one individual data point. For example, for the review classification task, the input layer will be populated with features extracted from an individual review such as the presence (or count) of individual words. In Chapter 8

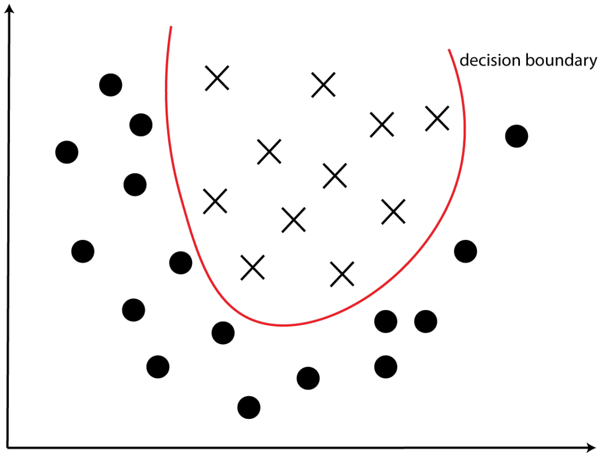


Figure 5.1 Decision boundary of a non-linear classifier.

we will switch from such hand-crafted features to numerical representations of text that capture some of the underlying semantics of language, and thus, are better for learning. Importantly, the neural network is agnostic to the way the representation of an input data point is created. All that matters for now is that each input data point is summarized with a vector of real values, $\mathbf{x}$.

Intermediate layers: Unlike the perceptron and logistic regression, FFNNs have an arbitrary number of intermediate layers. Each neuron in an intermediate layer receives as inputs the outputs of the neurons in the previous layer, and produces an output (or activation) that is sent to all the neurons in the following layer. The activation of each neuron is constructed similarly to logistic regression, as a non-linear function that operates on the dot product of weights and inputs plus the bias term. More formally, the activation a_i^l of neuron i in layer l is calculated as:

$$a_i^l = f(\sum_{j=1}^{k} w_{ij}^l a_j^{l-1} + b_i^l) = f(\mathbf{w}_i^l \cdot \mathbf{a}^{l-1} + b_i^l) = f(z_i^l) \tag{5.1}$$

where k is the total number of neurons in the previous layer $l-1$, w_{ij}^l are the weights learned by the current neuron (neuron i in layer l), a_j^{l-1} is the activation of neuron j in the previous layer, and b_i^l is the bias term of the current neuron. For simplicity, we group all the weights w_{ij}^l into the vector $\mathbf{w}_i^l$, and all activations a_j^{l-1} into the vector $\mathbf{a}^{l-1}$. Thus, the summation in the equation reduces to the dot product between the two vectors: $\mathbf{w}_i^l \cdot \mathbf{a}^{l-1}$. We further denote the sum between this dot product and the bias

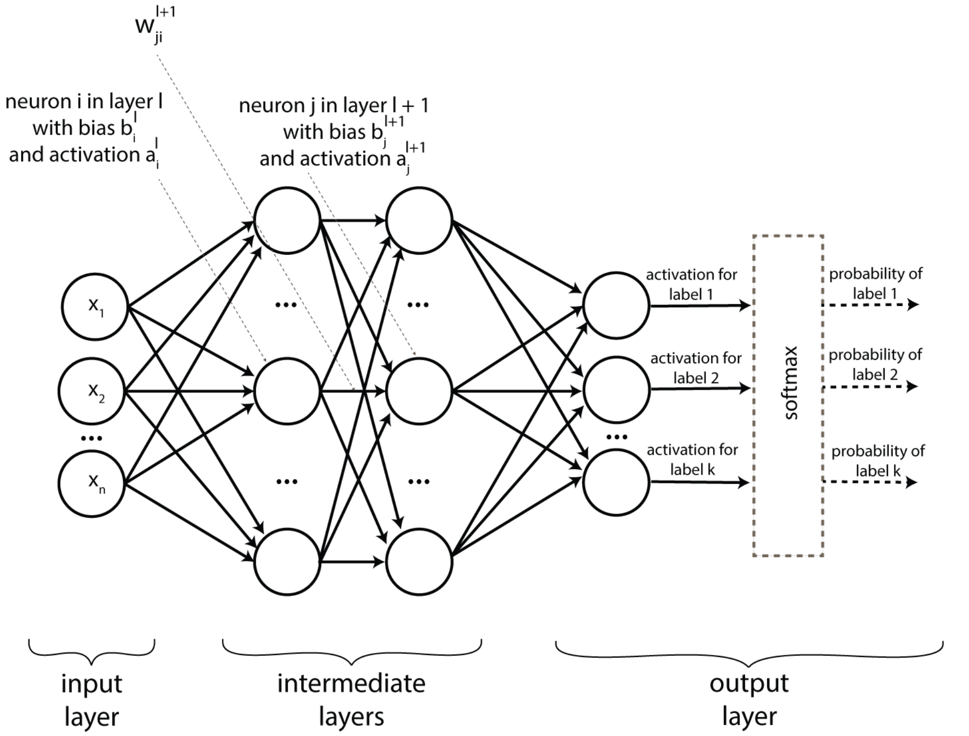


Figure 5.2 Fully-connected feed-forward neural network architecture.

term b_i^l as z_i^l. Thus, z_i^l is the output of neuron i in layer l right before the activation function f is applied.

The function f is a non-linear function that takes z_i^l as its input. For example, for the logistic regression neuron, f is the logistic function, σ. Many other non-linear functions are possible and commonly used in neural networks. We will discuss several such functions, together with their advantages and disadvantages in Chapter 6. What is important to realize at this stage is that these non-linear functions are what give neural networks the capability of learning non-linear decision boundaries. A multi-layer FFNN with linear activation functions remains a linear classifier. As a simple example, consider the neural network in Figure 5.3, which has one intermediate layer with two neurons, and a single neuron in the output layer. Let us consider that the activation function in each neuron is a "pass through" linear function $f(x) = x$. The activation of the output neuron is then computed as:

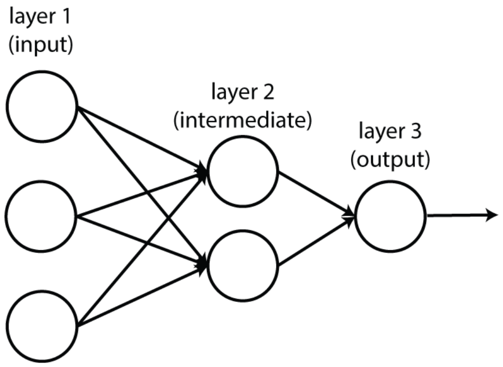


Figure 5.3 A feed-forward neural network with linear activation functions is a linear classifier.

$$\begin{aligned}
a_1^3 &= w_{11}^2 a_1^2 + w_{12}^2 a_2^2 + b_1^3 \qquad (5.2)\\
&= w_{11}^2(w_{11}^1 x_1 + w_{12}^1 x_2 + w_{13}^1 x_3 + b_1^2) + w_{12}^2(w_{21}^1 x_1 + w_{22}^1 x_2 + w_{23}^1 x_3 + b_2^2) + b_1^3\\
&= x_1(w_{11}^2 w_{11}^1 + w_{12}^2 w_{21}^1) +\\
&\quad x_2(w_{11}^2 w_{12}^1 + w_{12}^2 w_{22}^1) +\\
&\quad x_3(w_{11}^2 w_{13}^1 + w_{12}^2 w_{23}^1) +\\
&\quad w_{11}^2 b_1^2 + w_{12}^2 b_2^2 + b_1^3
\end{aligned}$$

which is a linear function on the input variables x_1, x_2, and x_3. It is easy to show that this observation generalizes to any arbitrary FFNN, as long as the neuron activation functions are linear.

Output layer: Lastly, FFNNs have an output layer that produces scores for the classes to be learned. Similar to the multiclass logistic regression, these scores can be aggregated into a probability distribution if the output layer includes a softmax function. However, the softmax function is optional (hence the dashed lines in the figure). If softmax is skipped, the class scores will not form a probability distribution, and they may or may not be bounded to the $[0,1]$ interval depending on the activation functions used in the final layer.

The architecture shown in Figure 5.2 can be reduced to most of the classifiers we introduced so far. For example:

Perceptron: The perceptron has no intermediate layers; has a single neuron in the output layer with a "pass through" activation function: $f(x) = x$; and no softmax.

Algorithm 7: Stochastic gradient descent algorithm for the training of neural networks.

```
initialize parameters in $\Theta$
while not converged do
    for each training example $\mathbf{x}_i$ in $\mathbf{X}$ do
        for each $\theta$ in $\Theta$ do
            $\theta = \theta - \alpha \frac{d}{d\theta} C_i(\Theta)$
        end
    end
end
```

Binary logistic regression: Binary LR is similar to the perceptron, with the only difference that the activation function of its output neuron is the logistic: $f = \sigma$.

Multiclass logistic regression: Multiclass LR has multiple neurons in its output layer (one per class); their activation functions are the "pass through" function, $f(x) = x$; and it has a softmax.

5.2 Learning Algorithm for Neural Networks

At a very high level, one can view a neural network as a complex machinery with many knobs, one for each neuron in the architecture. In this analogy, the learning algorithm is the operating technician whose job is to turn all the knobs to minimize the machine's output, i.e., the value of its cost function for each training example. If a neuron increases the probability of an incorrect prediction, its knob will be turned down. If a neuron increases the probability of a correct prediction, its knob will be turned up.

We will implement this learning algorithm that applies to any neural network with a generalization of the learning algorithm for multiclass logistic regression (Algorithm 6). The first key difference is that the parameters we are learning are no longer a single weight vector and a single bias term per class as in the multiclass LR. Instead, the neural network parameters contain one weight vector and one bias term for *each* neuron in an intermediate or final layer (see Figure 5.2). Because the number of these neurons may potentially be large, let's use a single variable name, Θ, to indicate the totality of parameters to be learned, i.e., all the weights and biases. We will also use θ to point to an individual parameter (i.e., one single bias term or a single weight) in Θ. Under these notations we can generalize Algorithm 6 into Algorithm 7, which applies to any neural network we will encounter in this book.[1]

Note that the key functionality remains exactly the same between Algorithms 6 and 7: in each iteration, both algorithms update their parameters by subtracting the partial derivative of the cost from their current values. As discussed, this guarantees that the cost function incrementally decreases towards some local minimum. This observation is sufficient to understand how to implement the training algorithm for a FFNN using a modern machine learning library that includes auto-differentiation such as PyTorch. Thus, the impatient reader who wants to get to programming examples as quickly as possible may skip the remainder of this chapter and jump to the next one for code examples. However, we encourage the reader to stick around for the next sections in this chapter, where we will look “under the hood” of Algorithm 7 to understand better how it operates.

5.3 The Equations of Back-propagation

The key equation in Algorithm 7 is in row 5, which requires the computation of the partial derivative of the cost function for one training example $C_i(\Theta)$ with respect to *all* parameters in the network, i.e., all edge weights and all bias terms. While this looks mathematically simple, it is not intuitive: how are we to calculate the partial derivatives for parameters associated with neurons that are not in the final layer, and, thus, do not contribute directly to the cost function computation? To achieve this, we will implement an algorithm that has two phases: a *forward* phase, and a *backward* phase. In the forward phase, the algorithm runs the neural network with its current parameters to make a prediction on the given training example i. Using this prediction, we then compute the value of the cost function for this training example, $C_i(\Theta)$. Then, in the backward phase we incrementally propagate this information backwards, i.e., from the final layer towards the first layer, to compute the updates to the parameters in each layer. Because of this, this algorithm is commonly called *back-propagation*, or, for people in a hurry, *backprop*.

Let us formalize this informal description. To do this, we need to introduce a couple of new notations. First, because in this section we will use only one training example i and refer to the same training parameters Θ throughout, we will simplify $C_i(\Theta)$ to C in all the equations below. Second, and more importantly, we define the *error of neuron* i[2] in layer l as the partial derivative of the cost function with respect to the

[1] We will revise this algorithm slightly in Chapter 6.

[2] Note that we are overloading the index i here. In Algorithm 7 we used it to indicate a specific training example $\mathbf{x}_i$. Now we use it to indicate a specific neuron.

Algorithm 8: The back-propagation algorithm that computes parameter updates for a neural network.

```
1 compute the errors in the final layer $L$, $\delta_i^L$, using the cost function $C$
  (Equation 5.4)
2 backward propagate the computation of errors in all upstream layers
  (Equation 5.5)
3 compute the partial derivates of $C$ for all parameters in a layer $l$, $\frac{d}{db_i^l}C$ and $\frac{d}{dw_{ij}^l}C$,
  using the errors in the same layer, $\delta_i^l$ (Equations 5.6 and 5.7)
```

neuron's output:

$$\delta_i^l = \frac{d}{dz_i^l}C \tag{5.3}$$

where z_i^l is the output of neuron i in layer l before the activation function f is applied. Intuitively, the error of a neuron measures what impact a small change in its output z has on the cost C. Or, if we view z as a knob as in the previous analogy, the error indicates what impact turning the knob has. The error of a neuron is a critical component in backpropagation: we want to adjust the parameters of each neuron *proportionally* with the impact the neuron has on the cost function's value for this training example: the higher the impact, the bigger the adjustment. Lastly, we use the index L to indicate the *final* layer of the network, e.g., the layer right before the softmax in Figure 5.2. Thus, δ_i^L indicates the error of neuron i in the final layer.

Using these notations, we formalize the backpropagation algorithm with the three steps listed in Algorithm 8. Step 1 computes the error of neuron i in the final layer as the partial derivative of the cost with respect to the neuron's activation multiplied with the partial derivative of the activation function with respect to the neuron's output:

$$\delta_i^L = \frac{d}{da_i^L}C\frac{d}{dz_i^L}f(z_i^L) \tag{5.4}$$

This equation may appear daunting at first glance (two partial derivatives!), but it often reduces to an intuitive formula for given cost and activation functions. As a simple example, consider the case of binary classification, i.e., a single neuron in the final layer with a logistic activation, coupled with the *mean squared error* (MSE) cost function. We will discuss the MSE cost in more detail in Chapter 6. For now, it is sufficient to known that MSE is a simple cost function commonly used for binary classification: $C = (y - a_1^L)^2$, where y is the gold label for the current training example, e.g., 0 or 1 assuming a logistic activation. That is, the MSE cost simply minimizes the difference between the prediction of the network (i.e., the activation of its final neuron) and the gold label. The derivative of the MSE cost with respect to the neuron's activation is: $\frac{d}{da_1^L}C = 2(a_1^L - y)$.[3] The derivative of the logistic with respect to the neuron's output is: $\frac{d}{dz_1^L}\sigma(z_1^L) = \sigma(z_1^L)(1-\sigma(z_1^L))$ (see Table 3.1). Thus, δ_1^L in this simple example is computed as: $\delta_1^L = 2(a_1^L - y)\sigma(z_1^L)(1-\sigma(z_1^L)) = 2(\sigma(z_1^L) - y)\sigma(z_1^L)(1-\sigma(z_1^L))$. It is easy to see that this error formula follows our knob analogy: when the activation of the final neuron is close to the gold label y, which can take values of 0 or 1, the error approaches 0 because two of the terms in its product are close to 0. In contrast, the error value is largest when the classifier is "confused" between the two classes, i.e., its activation is 0.5. The same can be observed for any (differentiable) cost and activation functions (see next chapter for more examples).

Equation 5.4 is easy to prove using a direct application of the chain rule:

$$\begin{aligned}\delta_i^L &= \sum_k \frac{d}{da_k^L}C\frac{d}{dz_i^L}a_k^L \\ &= \frac{d}{da_i^L}C\frac{d}{dz_i^L}a_i^L\end{aligned}$$

where k iterates over all neurons in the last layer. Note that we need to sum over all neurons in the final layer in the first line of the proof because C theoretically depends on all activations in the final layer. However, neuron i impacts only its own activation, and, thus, we can ignore all other activations (second line of the proof).

Equation 5.4 computes the errors in the *last* layer of the network. The next back-propagation equation incrementally propagates the computation of errors into the upstream layers, i.e., the layers that are farther to the left in Figure 5.2. That is, this equation computes the errors in a layer l using the errors in the layer immediately downstream, $l+1$, as follows:

$$\delta_i^l = \sum_k \delta_k^{l+1} w_{ki}^{l+1} \frac{d}{dz_i^l} f(z_i^l) \tag{5.5}$$

where k iterates over all neurons in layer $l+1$.

We prove this equation by first applying the chain rule to introduce the outputs of the downstream layer, z_k^{l+1}, in the formula for the error of neuron i in layer l, and then taking advantage of the fact the outputs in the downstream layer $l+1$ depend

[3] This is trivially derived by applying the chain rule.

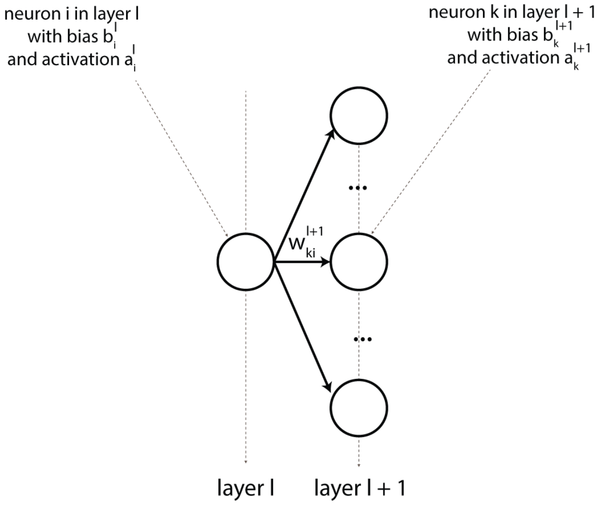


Figure 5.4 Visual helper for Equation 5.5.

on the activations in the previous layer l. More formally:

$$\begin{aligned}
\delta_i^l &= \frac{d}{dz_i^l} C \\
&= \sum_k \frac{d}{dz_k^{l+1}} C \frac{d}{dz_i^l} z_k^{l+1} \\
&= \sum_k \delta_k^{l+1} \frac{d}{dz_i^l} z_k^{l+1} \\
&= \sum_k \delta_k^{l+1} \frac{d}{dz_i^l} (\sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1}) \\
&= \sum_k \delta_k^{l+1} \frac{d}{dz_i^l} (w_{ki}^{l+1} a_i^l) \\
&= \sum_k \delta_k^{l+1} w_{ki}^{l+1} \frac{d}{dz_i^l} a_i^l \\
&= \sum_k \delta_k^{l+1} w_{ki}^{l+1} \frac{d}{dz_i^l} f(z_i^l)
\end{aligned}$$

where j iterates over all neurons in layer l. Similar to the previous proof, we need to sum over all the neurons in layer $l+1$ (second line of the proof) because the value of the cost function is impacted by all the neurons in this layer. The rest of the proof follows from the fact that $z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1}$. Figure 5.4 provides a quick visual helper to navigate the indices used in this proof.

Using Equations 5.4 and 5.5 we can compute the errors of all neurons in the network. Next we will use these errors to compute the partial derivatives for all weights and bias terms in the network, which we need for the stochastic gradient descent updates in Algorithm 7. First, we compute the partial derivative of the cost with respect to a bias term as:

$$\frac{d}{db_i^l} C = \delta_i^l \tag{5.6}$$

The proof of this equation follows similar steps with the previous two proofs, but here we iterate over neurons in the same layer l (so we can access the error of neuron i). Thus, we can ignore all neurons other than neuron i, which depends on this bias term:

$$\begin{aligned}
\frac{d}{db_i^l} C &= \sum_k \frac{d}{dz_k^l} C \frac{d}{db_i^l} z_k^l \\
&= \frac{d}{dz_i^l} C \frac{d}{db_i^l} z_i^l \\
&= \delta_i^l \frac{d}{db_i^l} z_i^l \\
&= \delta_i^l \frac{d}{db_i^l} (\sum_h w_{ih}^l a_h^{l-1} + b_i^l) \\
&= \delta_i^l
\end{aligned}$$

where k iterates over all neurons in layer l, and h iterates over the neurons in layer $l-1$.

Similarly, we compute the partial derivative of the cost with respect to the weight that connects neuron j in layer $l-1$ with neuron i in layer l, $\frac{d}{dw_{ij}^l} C$, as:

$$\frac{d}{dw_{ij}^l} C = a_j^{l-1} \delta_i^l \tag{5.7}$$

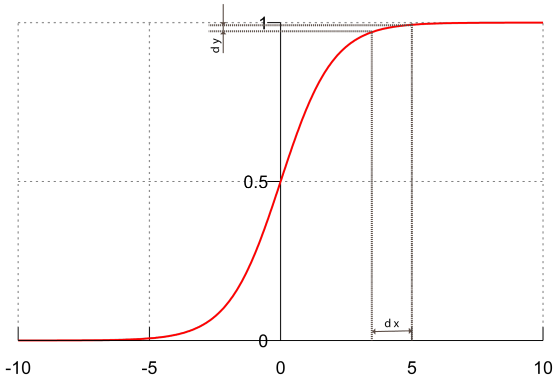


Figure 5.5 Visualization of the vanishing gradient problem for the logistic function: changes in x yield smaller and smaller changes in y at the two ends of the function, which means that $\frac{d}{dx}\sigma$ approaches zero in the two extremes.

The proof of this equation follows the same structure as the proof above:

$$\begin{aligned}\frac{d}{dw_{ij}^{l}}C &= \sum_{k}\frac{d}{dz_{k}^{l}}C\frac{d}{dw_{ij}^{l}}z_{k}^{l} \\ &= \frac{d}{dz_{i}^{l}}C\frac{d}{dw_{ij}^{l}}z_{i}^{l} \\ &= \delta_{i}^{l}\frac{d}{dw_{ij}^{l}}z_{i}^{l} \\ &= \delta_{i}^{l}\frac{d}{dw_{ij}^{l}}(\sum_{h}w_{ih}^{l}a_{h}^{l-1}+b_{i}^{l}) \\ &= \delta_{i}^{l}a_{j}^{l-1}\end{aligned}$$

where k iterates over all neurons in layer l, and h iterates over the neurons in layer $l-1$.

Equations 5.4 to 5.7 provide a formal framework to update the parameters of any neural network (weights and biases). They also highlight several important observations:

1. Implementing a basic feed forward neural network is not that complicated. Equations 5.4 to 5.7 rely on only two derivatives: the derivative of the activation function f, and the derivative of the cost function. In theory, these could be hard-coded for the typical activation and cost functions to be supported. The rest of the mathematical operations needed to implement back-propagation are just additions and multiplications. However, in practice, there are some additional

issues that need to be addressed for a successful neural network implementation. We will discuss these issues in Chapter 6.

2. Back-propagation is slow. As shown in the equations, updating the network parameters requires a considerable number of multiplications. For real-world neural networks that contain millions of parameters this becomes a significant part of the training runtime. In the next chapters we will discuss multiple strategies for speeding up the training process such as batching multiple training examples and multiple operationes together (e.g., updating all bias terms in a layer with a single vector operation rather than several scalar updates as in the equations). When these tensor operations are moved onto a graphics processing unit (GPU), which has hardware support for parallel tensor operations, they can be executed much faster.
3. Depending on the activation function, its partial derivative with respect to model parameters may be too small, which slows down the learning process. This happens because the equations to compute the errors in the network layers (Equations 5.4 and 5.5) both depend on this derivative. Multiplying this derivative repeatedly, as required by the recursive process described in the two equations, may have the unintended side effect that some errors will approach zero, which, in turn, means that the network parameters will not be updated in a meaningful way. This phenomenon is commonly called the "vanishing gradient problem." Figure 5.5 shows a visualization of this phenomenon for the logistic activation function. For this reason, other activations that are more robust to this problem are commonly used in deep learning. We will discuss some these in Chapter 6.

5.4 Drawbacks of Neural Networks (So Far)

In this chapter we generalized logistic regression into multi-layered neural networks, which can learn nonlinear functions. This is a major advantage over LR, but it can be also be a drawback: because of their flexibility, neural networks can "hallucinate" classifiers that fit the training data well, but fail to generalize to previously unseen data [Domingos 2015]. This process is called *overfitting*. We will discuss multiple strategies to mitigate overfitting in Chapter 6.

In addition to overfitting, the training process of neural networks may suffer from other problems. We discussed the vanishing gradient problem in the previous section. Another problem commonly observed when training neural networks is the tendency to "Tony Hawk" the data, which slows down convergence, or prevents it all together. Chapter 6 discusses optimization algorithms that reduce this phenomenon.

Further, similar to the perceptron and LR, the neural networks covered so far continue to rely on hand-crafted features. We will address this limitation in Chapter 8.

Lastly, feed forward neural networks focus on individual predictions rather than structured learning (i.e., where multiple predictions such as the part-of-speech in a sentence are jointly generated). We will start introducing structured prediction using neural networks in Chapter 12. This will open the door to other important NLP applications such as part-of-speech tagging, named entity recognition, and syntactic parsing.

5.5 Historical Background

TODO: To do

5.6 References and Further Readings

TODO: To do

6 Best Practices in Deep Learning

The previous chapter introduced feed forward neural networks and demonstrated that, theoretically, implementing the training procedure for an arbitrary FFNN is relatively simple: Algorithm 7 describes the learning algorithm that relies on stochastic gradient descent, and Algorithm 8 explains how the actual parameter updates are computed using back-propagation. Unfortunately, as described in Section 5.4, neural networks trained this way will suffer from several problems such as stability of the training process, i.e., slow convergence due to parameters jumping around a good minimum, and overfitting. In this chapter we will describe several practical solutions that mitigate these problems. Note that most of these solutions are implemented in modern deep learning libraries such as PyTorch. We will see them in action in the next chapter.

6.1 Mini-batching

Algorithm 7 updates the network parameters after each *individual* training example is seen. This means that the network changes its parameters at the fastest rate possible, with gradients that may have high variance (due to training examples that may be very different). These rapid changes may cause the resulting network to exhibit large differences in behavior (i.e., the network makes different predictions in response to the same inputs) in short time. If you wish, stochastic gradient descent is a training process that just had a triple espresso. Being highly caffeinated has several advantages and disadvantages. The pluses of this strategy are:

1. In some cases, stochastic gradient descent converges to a good outcome more quickly due to the rapid parameter updates. This usually happens on easier problems, where the cost function has a minimum that is easy to find and yields a good solution.
2. Stochastic gradient descent has the capacity to "jump out" of local minima encountered during training due to the high variance in the gradients corresponding to different training examples. That is, similar to the function shown earlier in Figure 3.3, the cost functions used by neural networks are not necessarily convex. At some point in the training process, i.e., when only a subset of the training examples have been seen, the learning process may converge to a poor minimum,

Algorithm 9: Batch gradient descent algorithm.

```
initialize parameters in $\Theta$
while not converged do
    for each $\theta$ in $\Theta$ do
        $\text{grad}_\theta = 0$
    end
    for each training example $\mathbf{x}_i$ in $\mathbf{X}$ do
        for each $\theta$ in $\Theta$ do
            $\text{grad}_\theta = \text{grad}_\theta + \frac{d}{d\theta}C_i(\Theta)$
        end
    end
    for each $\theta$ in $\Theta$ do
        $\theta = \theta - \alpha\frac{\text{grad}_\theta}{|\mathbf{X}|}$
    end
end
```

e.g., similar to the one in the right part of the function shown in Figure 3.3. However, the following parameter updates, which can be drastically different from the previous ones that led to the suboptimal solution, increase the probability that the neural network leaves this local minimum and continues training.

3. Last but not least, stochastic gradient descent is easy to implement and has minimal memory requirements, i.e., only one training example has to be kept in memory at a time.

The drawbacks of stochastic gradient descent are:

1. The “jumping out” of suboptimal solutions advantage often translates into the disadvantage of slower convergence because the network “jumps around” good solutions rather than settling on one.
2. Stochastic gradient descent is computationally expensive due to the frequent updates of the network parameters. Further, these parameter updates are hard to parallelize due to the sequential traversal of the training examples.

The opposite of stochastic gradient descent is *batch gradient descent*, which updates the network parameters only after *all* the training examples have been seen. That is, batch gradient descent still computes the parameter gradients after each training example is processed, but updates them only at the end of each epoch with the average of all previously-computed gradients. The process is summarized in Algorithm 9. In the algorithm, the variables grad_θ keep track of the sum of the partial derivatives of C with respect to θ for each training example i, and $|\mathbf{X}|$ indicates the size of the training dataset $\mathbf{X}$. If stochastic gradient descent is a highly-caffeinated training process, batch gradient descent had a calming beverage such as chamomile tea. The advantages and disadvantages of this sedated training algorithm are opposite those of stochastic gradient descent. That is, its main advantages are:

1. The average gradients used to update the network parameters tend to be more stable than the individual gradients used in stochastic gradient descent, and this often leads to convergence to better (local) minima on some problems.
2. Batch gradient descent is more computationally efficient because of the fewer updates of the network parameters. Further, batching is better suited for parallel implementations. That is, the for loop in line 6 of Algorithm 9 can theoretically be executed in parallel because the individual gradients are only used at the end of the loop.

And its disadvantages are:

1. Batch gradient descent may prematurely converge to a less-than-ideal solution because its ability to "jump out" of an undesired local minimum is reduced.
2. Despite the computational efficiency within an individual epoch, batch gradient descent may take longer to train (i.e., more epochs) because the network parameters are updated only once per epoch.
3. The implementation of batch gradient descent is more complicated than that of stochastic gradient descent because it needs to keep track of the sum of all gradients for each network parameter throughout an epoch.

These two extreme strategies suggest that a middle ground may be the best practical solution. This middle ground is called *mini-batch gradient descent*. Similar to batch gradient descent, the mini-batch variant updates the network parameters only after a batch is completed, but its batches are smaller. For example, typical mini-batch sizes for many NLP problems are 32 or 64 training examples. The mini-batch gradient descent algorithm is described in Algorithm 10. Similar to Algorithm 9, the variables grad_θ keep track of the sum of the partial derivatives of C with respect to θ for each training example i in a given mini-batch $\mathbf{M}$. They are reset at the start of each mini-batch (lines 4 – 6), and are used to update the parameter values after each mini-batch completes (lines 12 – 14). $|\mathbf{M}|$ indicates the number of training examples in the mini-batch $\mathbf{M}$.

Algorithm 10: Mini-batch gradient descent algorithm.

```
initialize parameters in $\Theta$
while not converged do
    for each mini-batch $\mathbf{M}$ sampled from $\mathbf{X}$ do
        for each $\theta$ in $\Theta$ do
            $\text{grad}_\theta = 0$
        end
        for each training example $\mathbf{x}_i$ in $\mathbf{M}$ do
            for each $\theta$ in $\Theta$ do
                $\text{grad}_\theta = \text{grad}_\theta + \frac{d}{d\theta} C_i(\Theta)$
            end
        end
        for each $\theta$ in $\Theta$ do
            $\theta = \theta - \alpha \frac{\text{grad}_\theta}{|\mathbf{M}|}$
        end
    end
end
```

On the spectrum of caffeinated beverages, mini-batch gradient descent consumed a green tea, a beverage that provides just enough energy without the jitters that may be associated with large espressos. More formally, the advantages of mini-batch gradient descent combine the best traits of the previous two training algorithms:

1. Mini-batch gradient descent tends to robustly identify good local minima because it reduces the “jumping around” disadvantage of stochastic gradient descent, while keeping some of its “jumping out” of undesired minima advantage.
2. Mini-batch gradient descent allows for efficient, parallel implementation within an individual mini-batch. We will show how this is done in PyTorch in the next chapter.

The main disadvantages of mini-batch gradient descent are:

1. Similar to batch gradient descent, its implementation is somewhat more complicated than that of stochastic gradient descent due to the additional bookkeeping necessary for each mini-batch.
2. More importantly, mini-batch gradient descent introduces a new *hyper parameter*, i.e., a variable that needs to be tuned outside of the actual training process:

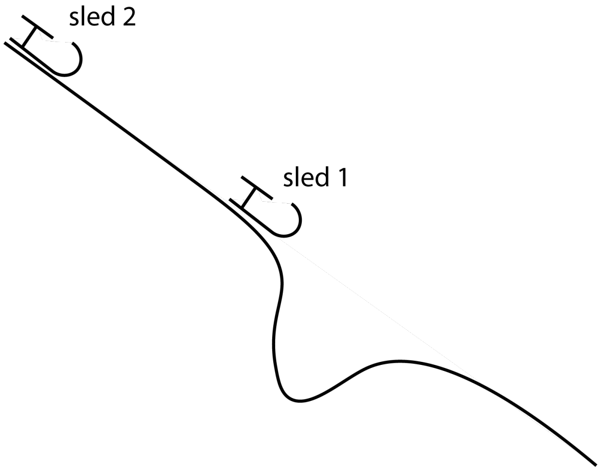


Figure 6.1 Illustration of momentum: sled 1 is more likely to get stuck in the ravine than sled 2, which starts farther up the hill, and carries momentum when it enters the ravine.

the size of the mini-batch. Unfortunately, the size of the mini-batch tends to be specific to each task and dataset. Thus, the developer must search for the best mini-batch size through an iterative trial-and-error *tuning process*, where various sizes are used during training, and the performance of the resulting model is evaluated on a separate tuning (or development) partition of the data.

In the next section we describe other optimization algorithms that further increase the stability of the training process.

6.2 Other Optimization Algorithms

Beyond mini-batching, several improvements have been proposed to increase the robustness of gradient descent algorithms. The first one we will discuss is *momentum* [Qian 1999]. Figure 6.1 provides a simple real-world analogy for it: imagine two sleds going down a hill, and about to encounter a ravine. Sled 1 starts right before the ravine, whereas sled 2 starts further up the hill. Clearly, sled 1 is more likely to get stuck in the ravine than sled 2, which carries more speed (or momentum) as it enters the ravine, and is more likely to escape it. Sled 1 is the equivalent of the previous mini-batch gradient algorithm, which is more likely to get stuck in a local minimum (the ravine). Algorithm 10 shows that for each mini-batch, i.e., step down the hill, we initialize each gradient, i.e., the speed at this moment, (line 5) with zero. That is, we forget about the speed we had previously, and compute the current speed simply based on the slope under our sled at this time. Gradient descent with momentum fixes this by initializing the gradients with a fraction of the final gradients computed for the previous mini-batch. That is, at time t, i.e., when the tth mini-batch is processed, line 5 in Algorithm 10 changes to:

$$\text{grad}_{\theta}^{t} = \gamma \, \text{grad}_{\theta}^{t-1} \tag{6.1}$$

where $\text{grad}_{\theta}^{t-1}$ is the gradient for θ computed for the previous mini-batch, and γ is a hyper parameter with values between 0 and 1,[1] which indicates how much of the previous momentum we want to preserve.

A variant of momentum, called *Nesterov momentum* [Nesterov 1983], builds upon this intuition by also changing line 9 of Algorithm 10. In particular, Nesterov momentum does not compute the partial derivative of the cost function, $\frac{d}{d\theta}C_i$, using the actual parameters in Θ. Instead, this algorithm subtracts the momentum, i.e., a fraction of $\text{grad}_{\theta}^{t-1}$, from each parameter θ when computing C_i. The intuition behind this operation is that this allows the algorithm to "peak into the future," by using values that estimate the parameter values at time $t+1$. This is possible because we know through the combination of the momentum initialization (discussed in the previous paragraph) and the actual update operation (line 13 in Algorithm 10) that the value of each parameter θ at the end of this mini-batch will be computed by subtracting a fraction of its corresponding momentum from the old value of θ.[2] Thus, Nesterov momentum is informed by both the past (through the momentum initialization) and the future (through the modified parameter values when computing the cost function). Empirically, it has been shown that this brings more stability to the training process [Dean et al. 2012].

Another complication of gradient descent is identifying a good learning rate, i.e., an appropriate value for the hyper parameter α in line 13 of Algorithm 10. Any deep learning practitioner will quickly learn that the performance of most deep learning models depends heavily on the learning rate used. In the opinion of the authors, the learning rate is the most important hyper parameter to be tuned. A value that is too big will yield faster training, but may cause the training process to "jump" over good minima. On the other hand, a value that is too small may cause training to be too slow, and to risk getting stuck in a suboptimal minimum (i.e., the ravine in Figure 6.1). Further, the learning rate value should be adjusted based on the timeline of training. Earlier in the process, a larger training rate helps approaching a good solution more quickly from the randomly chosen starting point, but later in the course a smaller value is generally preferable to avoid jumping out of good minima. Lastly, different features likely require different learning rates. That is, using a single learning

[1] Common values for γ are around 0.9.

[2] We will, of course, also subtract the other $\frac{d}{d\theta}C_i$ computed for this mini-batch, but these are unknown at this time.

rate may cause the frequent features, i.e., features commonly observed with non-zero values in training examples, to dominate in the learned model because their associated parameters, i.e., the edges connecting them to the output neurons, will be updated more frequently. Thus, ideally, we would like to perform larger updates for parameters associated with less frequent features (which may still contain useful signal!) so they have a say in the final model.

The solution to all of the above problems is to use *adaptive learning rates*, i.e., have a distinct learning rate for each parameter θ in the network, and allow these values to change over time. A common strategy is to have each learning rate be inversely proportional to the square root of the sum of the squares of the gradients observed for this parameter in each mini-batch up to the current one. That is, if we denote the sum of squares of the gradients for parameter θ as G_θ, then line 13 in Algorithm 10 becomes:

$$\theta = \theta - \frac{\alpha}{\sqrt{G_\theta + \varepsilon}} \frac{\text{grad}_\theta}{|\mathbf{M}|} \tag{6.2}$$

where ε is a small constant to avoid division by zero. There are several important observations about this seemingly simple change:

- Because G_θ is distinct for each θ, this formula yields different learning rates for different parameters.
- The summation of squares in G_θ guarantees that G_θ monotonically grows over time, regardless of the sign of the gradients. Thus, this formula captures our temporal intuition: learning rates will be larger in the beginning of the training process (when G_θ is small), and smaller later (when it is larger).
- Similarly, G_θ guarantees that parameters associated with frequent features will get smaller updates, while parameters associated with infrequent features will receive larger ones. This is because the gradients of parameters associated with frequent features will more often have non-zero values, which will lead to larger values for G_θ.

Most modern variants of gradient descent incorporate some form of momentum and adaptive learning rates. For example:

- The AdaGrad algorithm uses Nesterov momentum and the adaptive learning described above [Duchi et al. 2011].
- AdaDelta [Zeiler 2012] and RMSProp[3] address the fact that AdaGrad's learning rates are continuously diminishing due to the ever-increasing G_θ. They both

[3] RMSProp was proposed by Geoff Hinton but never formally published.

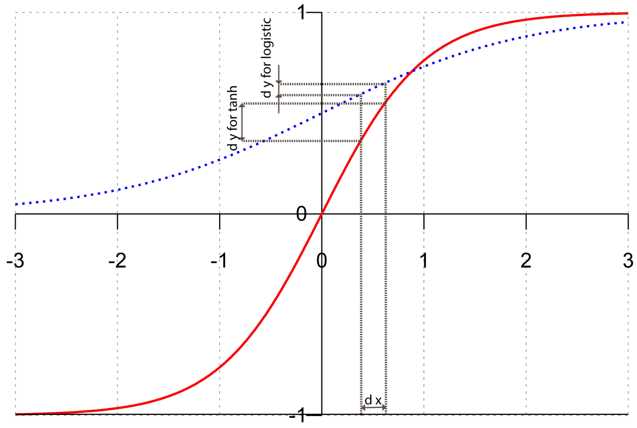


Figure 6.2 Comparison of the tanh (red continuous line) and logistic (blue dashed line) functions. The derivative of the tanh is larger than the derivative of the logistic for input values around zero.

achieve this by changing the formula of G_θ to iterate only over the most recent few gradients. Further, instead of using a straight summation, these algorithms use a *decaying average*, where the contribution of older gradients decreases over time.[4]

- Adaptive Moment Estimation (Adam) [Kingma and Ba 2015] builds on the previous two algorithms by also applying the same decaying average idea to the actual gradient values. That is, in Equation 6.2, Adam does not use the actual $\frac{\text{grad}_\theta}{|M|}$ value computed for this mini-batch, but a decaying average of the past few gradients.

All these algorithms are supported by most deep learning libraries such as PyTorch. A superficial analysis of NLP publications performed by the authors suggests that Adam and RMSProp seem to be the ones most commonly used for NLP at the time this book was written.

6.3 Other Activation Functions

In the previous chapter, we mentioned that one important drawback of the logistic function (and its multiclass equivalent, the softmax) is the vanishing gradient problem. This is caused by the fact that the derivative of the logistic tends to be small. When

[4] For the exact math behind this change, and a more expanded discussion of optimization algorithms, we recommend Sebastian Ruder's excellent blog post, available here: https://ruder.io/optimizing-gradient-descent/.

several such derivatives are multiplied during back-propagation, the resulting value may be too close to zero to impact the network weights in a meaningful way. One solution to the vanishing gradient problem is to use other activation functions that have larger gradient values. One such activation function is the hyperbolic tangent function, or tanh:

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{6.3}$$

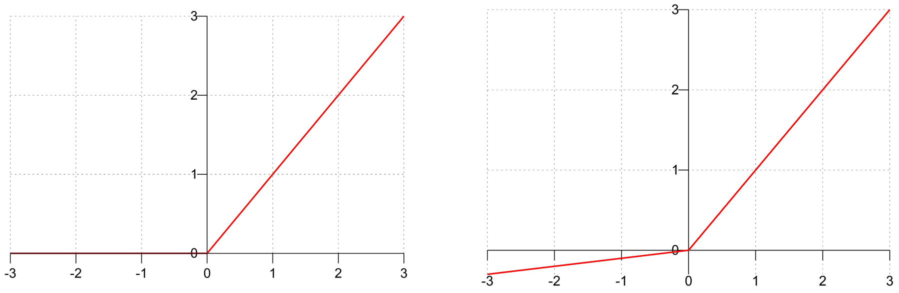
Figure 6.2 shows a plot of the tanh function overlaid over the plot of the logistic function. The figure highlights the key advantage of the tanh function: the slope of the tanh is steeper than the logistic's, and, thus, its derivative has larger values than the derivative of the logistic for most input values. This is the key reason why tanh suffers less from the vanishing gradient problem, and why it is usually preferred over the logistic. However, as shown in the figure, for extreme input values (very large or very small), tanh also exhibits saturated gradients, i.e., partial derivatives with very small values. One activation function that avoids this problem is the rectified linear unit, or ReLU:

$$ReLU(x) = \max(0, x) = \begin{cases} 0 & x < 0 \\ x & \text{otherwise} \end{cases} \tag{6.4}$$

Figure 6.3 (a) shows a plot of this function.[5] The figure shows that for positive input values ReLU grows at a constant rate, without exhibiting the saturated gradients of the tanh of logistic functions, both of which taper at 1. In practice, this means that the training process for networks that rely on ReLU activation functions tends to converge faster than those that use tanh or logistic activations. As an empirical rule of thumb, ReLU tends to learn faster than tanh, which usually converges more quickly than logistic.

A second, more subtle advantage of ReLU is caused by the fact that, as the figure shows, the function's value is 0 for all negative input values. This means that all neurons with ReLU activations whose dot product of input values and weights is negative become inactive, i.e., their output is 0. Glorot et al. [2011] observed that "after uniform initialization of the weights, around 50% of the hidden units' continuous output values are real zeros" in such networks. This percentage increases when regularization is used (see Chapter 6.5). But why would sparse representations of neural networks be preferred? It turns out that sparsity has several advantages [Glorot et al. 2011]:

[5] Figure 6.3 shows that ReLU and its variants are not differentiable for $x = 0$. For this input value, we typically set the derivative of ReLU to an arbitrary value. e.g., 1 or 0.5.



(a) (b)

Figure 6.3 The ReLU (a) and Leaky ReLU (b) activation functions.

- It allows a network to learn more flexible representations. That is, varying the number of neurons with non-zero activations in each layer allows better control of the actual dimensionality assigned to each layer (rather than relying on the hard-coded structure).
- It yields better explainability or better "information disentangling." That is, in the usual, dense networks where most activations are non-zero it is hard to understand what the underlying reason for a given output is because all neurons contributed something to the final activation. Further, such networks tend to be more sensitive to small changes in inputs. Both these drawbacks are mitigated by sparse networks. They can explain their outputs easier because a smaller number of neurons contributes to the final activations. Further, some small input changes are dampened by the inactive neurons and, thus, do not affect the network's outputs.
- Sparse representations are more likely to be linearly separable, which means it is easier and cheaper to learn a good classifier.

A third advantage of ReLU is its computational simplicity. Unlike logistic and tanh, which require exponential operations, ReLU relies solely on the much simpler max function.

One drawback of ReLU's hard saturation at 0 is an extreme form of vanishing gradient commonly called the "dying ReLU." That is, not surprisingly, the derivative of an inactive ReLU neuron is always 0, which means that weights immediately downstream of it are not updated during back-propagation (see Equation 5.7). The solutions to this problem are variants of ReLU that keep the general ReLU behavior, but avoid the hard saturation at 0. One such function is Leaky ReLU:

Table 6.1 Three cost functions commonly used in NLP tasks. m indicates the number of data points in the training dataset (or the mini-batch, in the case of mini-batch gradient descent). y_i is the correct (or gold) label for example i.

Name	Type of classification	Formula
Mean squared error	Binary	$\frac{1}{m}\sum_{i=1}^{m}(a_1^L - y_i)^2$
Binary cross entropy	Binary	$-\sum_{i=1}^{m}(y_i \log p(1\|\mathbf{x}_i;\mathbf{w},b) + (1-y_i)\log p(0\|\mathbf{x}_i;\mathbf{w},b))$
		$= -\sum_{i=1}^{m}(y_i \log a_1^L + (1-y_i)\log(1-a_1^L))$
Cross entropy	Multiclass	$-\sum_{i=1}^{m}\log p(y_i\|\mathbf{x}_i;\mathbf{W},\mathbf{b})$

$$LeakyReLU(x) = \begin{cases} \alpha x & x < 0 \\ x & \text{otherwise} \end{cases} \tag{6.5}$$

where α typically takes values around 0.01. Figure 6.3 (b) shows a plot of Leaky ReLU.

All in all, this discussion suggests that, while some recommendations can be made, there is no universal answer to the question: which activation function is best? Many activation functions have been proposed (we recommend you check your favorite deep learning library's documentation for the list of supported activation functions). All activation functions try to balance multiple desired, but sometimes conflicting properties such as mitigating vanishing gradient and producing sparse representations. Which one works best for you is likely to depend on the problem and data you work on at the moment. Be prepared to try several.

6.4 Cost Functions

So far, we have seen three cost functions in the book: cross entropy and binary cross entropy (Chapter 3) and, very briefly, the mean squared error (MSE) cost (Chapter 5). These are probably the most common cost functions in NLP, so it is worth discussing them slightly more formally.

Recall that a cost function must have several properties: (a) it should take only positive values, (b) it should measure the distance between the classifier predictions and the corresponding correct (or gold) labels, i.e., the higher the value of the cost function the more incorrect the underlying classifier is; and (c) it should be differentiable, so we can "plug" it in some form of the gradient descent algorithm. All three cost functions, listed in Table 6.1, have these properties. Let us convince ourselves that this is indeed the case.

First, the mean squared error (MSE) as shown here is designed for binary classification. That is, the underlying network has a single final neuron, whose activation (a_1^L)

is typically produced with either a logistic function or a hyperbolic tangent function. In the former case, the value of a_1^L should approach 1 for the positive class, and 0 for the negative class. In the latter situation, the positive class has label 1, and the negative one -1. Regardless of the choice of activation function, MSE always takes positive values due to the square in its formula. Also, MSE explicitly measures the distance between the classifier prediction and the gold label, and this distance is differentiable. MSE is easy to explain and is trivial to implement, but it has one major disadvantage: it may lead to slow learning. To understand this disadvantage let's revisit Equation 5.4 in the context of MSE. For a binary classifier with a single final neuron, the error in the final layer is: $\delta_1^L = \frac{d}{da_1^L} C \frac{d}{dz_1^L} f(z_1^L)$. For a single training example, this error becomes: $\delta_1^L = 2(a_1^L - y)\frac{d}{dz_1^L} f(z_1^L)$, where y is the gold label for the corresponding example. Thus, δ_1^L depends on the derivative of the activation function, which becomes vanishingly small at the two ends of the function, for both the logistic and tanh functions. Because the weight and bias updates in any neural network (see Equations 5.6 and 5.7) depend on δ_1^L, a neural network trained using the MSE cost is likely to experience learning slowdown.

The binary cross entropy addresses this limitation. Before we explain how, let us convince ourselves that the binary cross entropy is a proper cost function. Because the logarithms in its formula take probabilities as parameters, we are restricted here to logistic activation functions for a_1^L. Thus, the summation in the formula is always smaller or equal to 0 (the natural logarithm of a number smaller than 1 is negative), and the resulting overall value is larger or equal to 0. Further, binary cross entropy measures the quality of the classifier. For example, a good classifier that produces an a_1^L approaching 1 for a positive label ($y = 1$) will have a binary entropy cost of $-\log a_1^L \approx -\log 1 = 0$. At the opposite extreme, a really bad classifier that produces a_1^L approaching 1 for a negative label ($y = 0$) will have a binary entropy cost of $-\log(1 - a_1^L) \approx -\log 0 = \infty$.

To understand why the binary cross entropy reduces the learning slowdown, let us derive δ_1^L in this context, for a single training example with gold label y:[6]

[6] We recommend that the user verifies this derivation using the information in Table 3.1.

$$
\begin{aligned}
\delta_1^L &= \frac{d}{da_1^L} C \frac{d}{dz_1^L} f(z_1^L) \\
&= \frac{d}{da_1^L}(-y \log a_1^L - (1-y)\log(1-a_1^L)) \frac{d}{dz_1^L}\sigma(z_1^L) \\
&= (-\frac{y}{a_1^L} + \frac{1-y}{1-a_1^L}) \frac{d}{dz_1^L}\sigma(z_1^L) \\
&= \frac{a_1^L - y}{a_1^L(1-a_1^L)} \frac{d}{dz_1^L}\sigma(z_1^L) \\
&= \frac{a_1^L - y}{\sigma(z_1^L)(1-\sigma(z_1^L))} \frac{d}{dz_1^L}\sigma(z_1^L) \\
&= \frac{a_1^L - y}{\sigma(z_1^L)(1-\sigma(z_1^L))} \sigma(z_1^L)(1-\sigma(z_1^L)) \\
&= a_1^L - y
\end{aligned}
$$

Thus, surprisingly, δ_1^L for binary cross entropy does *not* depend on the derivative of the activation function! Because of this, the binary cross entropy cost function is more resilient to learning slowdown, which makes it the most common choice for binary classification problems implemented with networks that have logistic activation in the output layer.

The cross entropy cost[7] (last row in Table 6.1) is simply a generalization of binary cross entropy to multiclass classification. Recall from the previous chapter that networks designed for multiclass classification have one neuron dedicated to each class in the output layer, and these neurons are usually followed by a softmax layer such that the final outputs form a probability distribution (see Figure 5.2). For such architectures the cross entropy cost maximizes the probability of the gold label for each training example i: $p(y_i|\mathbf{x}_i; \mathbf{W}, \mathbf{b})$. Because the denominator in the softmax formula iterates over the other activations (see Equation 3.16), maximizing the probability of the gold label for a given training example has the desired side effect of also minimizing the probabilities of all other (incorrect) labels for the same data point. Cross entropy is the preferred cost function for multiclass classification in most NLP tasks for the same reason binary cross entropy is the favored cost function for binary classification.

6.5 Regularization

In three of the four previous sections in this chapter we discussed techniques to improve the stability of the training process for neural networks (e.g., mini-batching,

[7] Goodfellow et al. [2016] (Chapter 5.5) point out that calling this cost function "cross entropy" is a misnomer, as the actual cross entropy formula is more complex. However, minimizing the actual cross entropy is equivalent to minimizing the formula in Table 6.1. For this reason, this abuse of terminology is widely spread. We will continue to use it throughout this book.

improved optimizers, and better cost functions). Regularization is a fourth common technique used for this purpose. Recall from Chapter 2 that regularization is a family of techniques that control for the noise that is potentially present in the training data. Implementation-wise, regularization methods control for undesired fluctuations in parameter (i.e., weights or bias terms) values that may occur when the training process is not stable due to exposure to noise.

In Chapter 2 we have seen the averaged perceptron as one simple regularization method. Moving the same intuition into the space of cost functions, regularization is implemented for neural networks by adding an additional term to the cost:

$$C_{reg}(\mathbf{W},\mathbf{b}) = C(\mathbf{W},\mathbf{b}) + \lambda R(\mathbf{W},\mathbf{b}) \tag{6.6}$$

where $C(\mathbf{W},\mathbf{b})$ is any cost function without regularization such as the ones listed in Table 6.1, R is the new regularization function, and λ is a positive number (usually a small one) that indicates how much importance to put on the regularization component of the cost. Informally, any implementation of R guarantees that minimizing R minimizes the network's parameter values.

Intuitively, there is a tug of war[8] between the C and R functions in the above equation when C_{reg} is minimized. On one hand, C needs to be minimized, which has the effect of increasing the values of certain weights and biases, e.g., to maximize the probabilities of the gold labels for the cross entropy cost. On the other hand, minimizing R has, by definition, the effect of explicitly minimizing all weight and bias values, which keeps the former component in check, and has the desired effect of "squishing" unreliable parameter values.

There are many possible implementations for the regularization function R. *L2 regularization* is probably the most common one:

$$R(\mathbf{W},\mathbf{b}) = \sum_{i=1}^{V} v_i^2 \tag{6.7}$$

where v_i iterates over all parameters in the network, i.e., edge weights and bias terms, and V is the total number of weights and biases in the network.[9] In plain language, L2 regularization is simply the sum of the squared values of the weights and biases in the given network.

So, why would adding such a regularization term to the cost function have the effect of mitigating the parameter value fluctuation? To understand this, recall that

[8] https://en.wikipedia.org/wiki/Tug_of_war

[9] The more mathematically-inclined reader may have realized that the name L2 regularization comes from the fact that this function is the square of the L2 norm of the vector containing all network parameters v_i.

gradient descent updates each network parameter v_i (again, these parameters include all weights and biases) by subtracting the partial derivative of the cost with respect to v_i, $\frac{d}{dv_i}C_{reg}(\mathbf{W},\mathbf{b})$, from the current value of v_i. For each v_i, the partial derivative of the L2 regularization, which is part of C_{reg}, is: $\frac{d}{dv_i}R(\mathbf{W},\mathbf{b}) = 2v_i$. Thus, during each back-propagation step, in addition of subtracting the partial derivative of the original C, we also subtract $2\lambda v_i$ from each parameter v_i. This value is proportional to v_i: large when v_i is large, and small otherwise. This has the nice effect of more aggressively reducing large parameter values (which may occur when training is not stable) than small ones.

Another common regularization function is L1 regularization,[10] which is simply the sum of the parameter values:

$$R(\mathbf{W},\mathbf{b}) = \sum_{i=1}^{V} v_i \tag{6.8}$$

Because the partial derivative of L1 regularization with respect to v_i is constant, the additional term introduced in $\frac{d}{dv_i}C_{reg}(\mathbf{W},\mathbf{b})$ by L1 regularization is simply the constant λ. In practice, this means that L1 regularization "squishes" the parameter values by a constant value in each back-propagation step, regardless of the original values of these parameters. The consequence of this is that L1 regularization reduces parameter values that are small more aggressively than L2, which produces sparser networks (i.e., with more edge weights reduced to 0).

Empirically, L2 regularization tends to perform better than L1 for NLP tasks. But, as mentioned, L1 produces sparser networks, which can be represented in memory more efficiently. All in all, most forms of regularization (L1, L2, and others) are trivial to implement (they are just an additional, simple term in the cost function), and they tend to be beneficial.

6.6 Dropout

Dropout can be seen as another simple regularization strategy. However, instead of changing the cost function used to train the neural network to encourage weight "squishing," dropout changes the *structure* of the network during training. That is, for each training example, dropout ignores (or "drops") network neurons, as well as their incoming and outgoing connections, with probability p. For example, if $p = 0.3$, the network will remove 30% of its neurons on average, for each training example. Figure 6.4 visualizes this process for a simple feed-forward network (Figure 6.4 (a)), and two views obtained in different dropout iterations (Figure 6.4 (b) and (c)).

[10] Similarly, L1 regularization is equivalent to the L1 norm of the vector containing all network parameters v_i. However, this parallel to linear algebra is not very important to our discussion, so we will ignore it from now on.

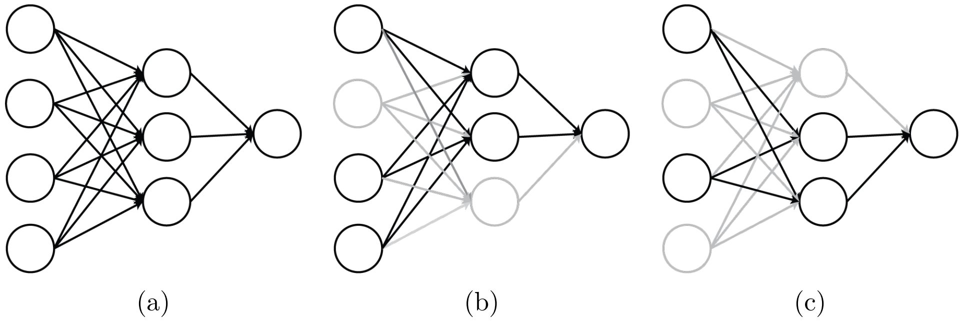


Figure 6.4 A simple neural network (a), and two views of it after dropout is applied (b and c). Greyed out nodes and edges are dropped out and, thus, ignored during the corresponding forward pass and back-propagation in (b) and (c).

An important implementation detail about dropout is that dropout is applied only during training. That is, at testing time, the entire network, e.g., the one in Figure 6.4 (a), is used. This difference has the undesired effect that the output values generated by the network during training are smaller, i.e., only $1-p$ of the corresponding values seen during testing. For example, while the full network shown in Figure 6.4 (a) has eight neurons, the one in Figure 6.4 (b) has only six. Thus, on average, we expect the output of the final neuron in Figure 6.4 (b) to be only $\frac{6}{8}$ of the output of the original network. This introduces a problem, as it is best if the networks used in training and testing are similar, otherwise the weights learned during training will not be effective. This issue is solved in deep learning libraries with one of the two following strategies. The first strategy *scales down* by p the output of each neuron during testing. The second strategy *scales up* by $\frac{1}{p}$ the output of each active neuron (i.e., not dropped out) during training. The first strategy has the advantage of a simpler training procedure (no need to worry about scaling during training), but it complicates testing. That is, the one must remember the dropout probability used during training, and scale down neuron outputs accordingly during testing. The second strategy has the opposite benefits: more complicated training, but simpler testing that requires no knowledge about how dropout was applied during training.

The hyper parameter p is chosen based on the performance of the trained network on a development dataset. In many NLP tasks, typical values for p range between 20 and 30%.

There are two explanations for why dropout is useful. The first is that dropout forces the remaining nodes in the network to take more responsibility for transforming the received inputs into the correct prediction. The other side of the coin is that because dropout uses sparser networks, it encourages the learning of sparser representations, i.e., learning smaller weights for neurons that the network learns can be safely ignored. This is exactly what regularization does as well! The second explanation is that dropout can be seen as an average of many different networks, one for each training example. For example, the training procedure may see the network in Figure 6.4 (b) for one training example, and the one in Figure 6.4 (c) for another example. But because these networks sample from the same group of neurons, the training procedure ends up aggregating all these updates into the same overall network, effectively learning an average of all the dropout views. Intuitively, this is very similar to the average perceptron we have seen in Chapter 2, with exactly the same benefits as discussed there.

6.7 Temporal Averaging

Temporal averaging is yet another simple strategy for constructing an ensemble model that builds upon the intuition behind the average perceptron. Temporal averaging works by averaging the network weights at the end of the best training epochs. More formally, temporal averaging is implemented through the following steps:

1. At the end of each training epoch, e.g., after each iteration in the outermost `while` loop in Algorithm 10, the performance of the current network is evaluated on a development dataset, and the current weights are all saved.
2. After training completes, all epochs are sorted in descending order of their development performance.
3. The weights from the top k best epochs are averaged into the final model. The hyper parameter k is empirically chosen based on the performance of the final model on the development partition. Typical values range between 3 and 5.

In other words, temporal averaging is similar to the average perceptron, but the model snapshots that are averaged are not created every time the model is updated (which would be after each training example for gradient descent!) but less frequently, e.g., at the end of an epoch. While less popular than dropout, in the experience of the authors temporal averaging is just as effective.

In the last few sections we have discussed three different regularization strategies: "traditional" regularization through the cost function (Section 6.5), dropout (Section 6.6), and the temporal averaging discussed in this section. These three techniques are largely complementary to each other, and, because of this, often combined.

6.8 Parameter Initialization and Normalization

As we discussed before, gradient descent does not find the global minimum of the cost function but rather its nearest minimum. Thus, where we start matters. In other words, it is important to initialize the parameters of the network to be trained (i.e., edge weights and bias terms) to values that increase our chances of finding a good solution. There are several rules of thumb on what one should and should not do when initializing the network parameters:

1. The parameters should be initialized randomly. It is trivial to observe that if all parameters were initialized with the same value, the network will learn the same feature for all its neurons. In contrast, initializing the parameters to different (random) values forces the network to assign different meaning to each of its neurons, which is the desired behavior.
2. The initial parameter values should not be too small. Values that are exceedingly small lead to the vanishing gradient problem, which slows down learning or stops it all together.
3. On the other hand, parameter values should not be too large either. Large parameter values yield large parameter updates during back-propagation, which causes unstable learning, or "Tony Hawking" the data, as we put it in Chapter 2. This problem is called the "exploding gradient problem." We will discuss this issue and practical solutions to mitigate it in Chapter 12.
4. The distribution of parameter values should be centered around zero because this is where the interesting things happen with most activation functions (see, for example, the tanh and ReLU activations introduced earlier in this chapter).

One very common parameter initialization strategy that follows these rules is the Glorot method [Glorot and Bengio 2010].[11] This method initializes a neuron's edge weights uniformly from the range $[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}]$, where n is the number of input edges to this neuron. For example, if layer $l-1$ in a network has 100 neurons, then the initial edge weights for neurons in layer l will be in the range $[-\frac{1}{10}, \frac{1}{10}]$. Without getting into the math, making the range of the initial weight values depend on the number of input edges makes the fluctuation (or variation) of the neuron's output activation similar to that of its inputs, which leads to more stable learning.

For the same reasons, the last three rules of thumb also apply to neuron activations throughout the entire training process, i.e., the activation a_i^l of neuron i in layer l (see Equation 5.1) should not be too small or too large, and be centered around zero. To make sure that this holds, one common strategy normalizes each neuron's activation across every individual mini-batch $\mathbf{M}$ (see Algorithm 10) such that the distribution of values for each activation fits in a small range centered around zero.[12] For obvious

[11] This method is also commonly called Xavier initialization, based on the main author's first name.

[12] For the more mathematically inclined reader, batch normalization uses a mean of 0, and a variation of 1 for each activation.

reasons, this method is called *batch normalization* or *batch norm*. In particular, for each mini-batch $\mathbf{M}$, batch normalization computes the mean μ and standard deviation σ (or, informally, how dispersed the set of values is) for all values observed for an activation a_i^l in the given mini-batch. Then, it *re-centers* a_i^l around zero by subtracting μ from it, and *re-scales* it to a standard deviation of 1 by dividing the resulting difference by σ. What is important here is that the re-centering and re-scaling operations are differentiable. In practice, batch normalization is implemented as another network component that follows each layer to be normalized in the network, and is trained using the same gradient descent algorithm used for the rest of the network.

As we will see in the next few chapters, in some situations mini-batches are not available, yet we still desire to normalize neuron activations. The normalization strategy used in such situations is called *layer normalization* [Ba et al. 2016]. As its name implies, this method normalizes all activations *in the same network layer*. That is, for each layer l, layer normalization computes the mean and standard deviation for the set of values corresponding to *all* activations a_i^l in layer l, and then re-centers and re-scales them using the same strategy as batch normalization. However, unlike batch normalization, which is applied only at training time when mini-batches are available, layer normalization applies the same procedure during both training and testing.

Both batch and layer normalization have been shown to lead to more stable and faster training, and are generally recommended. Batch normalization tends to perform better when larger mini-batches are available (due to the more robust statistics), whereas layer normalization is recommended for small mini-batches, or when batching is not possible.

7 Implementing the Review Classifier with Feed Forward Networks

8 Distributional Hypothesis and Representation Learning

As mentioned in the previous chapters, all the algorithms we covered so far rely on hand-crafted features that must be designed and implemented by the machine learning developer. This is problematic for two reasons. First, designing such features can be a complicated endeavor. For example, even for the apparently simple task of designing features for text classification questions arise quickly: How should we handle syntax? How do we model negation? Second, most words in any language tend to be very infrequent. This was formalized by Zipf [1932], who observed that if one ranks the words in a language in descending order of their frequency then the frequency of the word at rank i is $\frac{1}{i}$ times the frequency of the most frequent word. For example, the most frequent word in English is *the*. The frequency of the second most frequent word according to Zip's law is half the frequency of *the*; the frequency of the third most-frequent word is one third of the frequency of *the*, and so on.[1] In our context, this means that most words are very sparse, and our text classification algorithm trained on word-occurrence features may generalize poorly. For example, if the training data for a review classification dataset contains the word *great* but not the word *fantastic*, a learning algorithm trained on this data will not be able to properly handle reviews containing the latter word, even though there is a clear semantic similarity between the two. In the wider field of machine learning, this problem is called the "curse of dimensionality" [Bellman 1957].

In this chapter we will begin to addresses this limitation. In particular, we will discuss methods that learn numerical representations of words that capture some semantic knowledge. Under these representations, similar words such as *great* and *fantastic* will have similar forms, which will improve the generalization capability of our ML algorithms.

8.1 Traditional Distributional Representations

The methods in this section are driven by the distributional hypothesis of Harris [1954], who observed that words that occur in similar contexts tend to have similar meanings. The same idea was popularized a few years later by Firth [1957] who, perhaps more

[1] Interestingly, this law was observed to hold even for non-human languages such as dolphin whistles [Ferrer-i Cancho and McCowan 2009].

elegantly, stated that "a word is characterized by the company it keeps." It is easy to intuitively demonstrate the distributional hypothesis. For example, when presented with the phrases *bread and ...* and *bagels with ...*, many people will immediately guess from the provided context that the missing words are *butter* and *cream cheese*, respectively.

In this section, we will formalize this observation. In particular, we will associate each word in a given vocabulary with a vector, which represents the context in which the word occurs. According to the distributional hypothesis these vectors should capture the semantic meaning of words, and, thus, words that are similar should have similar vectors.

Traditionally, these vectors were built simply as co-occurrence vectors. That is, for each word w in the vocabulary, its vector counts the co-occurrence with other words in its surrounding context, where this context is defined as a window of size $[-c, +c]$ words around all instances of w in text. Here, we use negative values to indicate number of words to the left of w, and positive values to indicate number of words to the right. For example, consider the text below:

> *A bagel and cream cheese (also known as bagel with cream cheese) is a common food pairing in American cuisine. The bagel is typically sliced into two pieces, and can be served as-is or toasted.*[2]

In this text, *bagel* occurs three times. Thus, we will have three context windows, one for each mention of the word. While common values for c range from 10 to 20, let us set $c = 3$ for this simple example. Under this configuration, the three context windows for *bagel* in this text are:

- *A* bagel *and cream cheese*
- *also known as* bagel *with cream cheese*
- *American cuisine The* bagel *is typically sliced*

Note that we skipped over punctuation signs when creating these windows.[3] If we aggregate the counts of words that appear in these context windows, we obtain the following co-occurrence vector for *bagel*:

[2] Text adapted from the *Bagel and cream cheese* Wikipedia page: https://en.wikipedia.org/wiki/Bagel_and_cream_cheese.

[3] Different ways of creating these context windows are possible. For example, one may skip over words deemed to contain minimal semantic meaning such as determiners, pronouns, and prepositions. Further, these windows may be restricted to content within the same sentence. Lastly, words may be normalized in some form, e.g., through lemmatization. We did not use any of these heuristics in our example for simplicity.

A	1
also	1
American	1
and	1
as	1
cheese	2
cream	2
cuisine	1
is	1
known	1
sliced	1
The	1
typically	1
with	1

This example shows that the co-occurrence vector indeed captures meaningful contextual information: *bagel* is most strongly associated with *cream* and *cheese*, but also with other relevant context words such as *cuisine* and *sliced.* The larger the text used to compute these co-occurrence vectors is, the more meaningful these vectors become.

In practice, these co-occurrence vectors are generated from large document collections such as Wikipedia,[4] and are constructed to have size M, where M is the size of entire word vocabulary, i.e., the totality of the words observed in the underlying document collection. Note that these vectors will be sparse, i.e., they will contain many zero values, for all the words in the vocabulary that do not appear in the context of the given word. Having all co-occurrence vectors be of similar size allows us to formalize the output of this whole process into a single co-occurrence matrix $\mathbf{C}$ of dimension $M \times M$, where row i corresponds to the co-occurrence vector for word i in the vocabulary. A further important advantage of standardizing vector sizes is that we can easily perform vector operations (e.g., addition, dot product) between different co-occurrence vectors, which will become important soon.

Once we have this co-occurrence matrix, we can use it to improve our text classification algorithm. That is, instead of relying on an explicit feature matrix (see, for example, the feature matrix in Table 2.4), we can build our classifier on top of the co-occurrence vectors. A robust and effective strategy to this end is to simply average the co-occurrence vectors for the words contained in a given training example [Iyyer et al. 2015]. Take, for example, the first training example in Table 2.4: instead of training on the sparse feature vector listed in the first row in the table, we would train

[4] https://www.wikipedia.org

on a new vector that is the average of the context vectors for the three words present in the training example: *good*, *excellent*, and *bad*. This vector should be considerably less sparse than the original feature vector, which contains only three non-zero entries. The first obvious consequence of this decision is that the dimensions of the classifier's parameters change. For example, in the case of a Perceptron or a logistic regression, the dimension of the vector $\mathbf{w}$ becomes M to match the dimension of the co-occurrence vectors. The second, more important consequence, is that the parameter vector $\mathbf{w}$ becomes less sparse because it is updated with training examples that are less sparse in turn. This means that our new classifier should generalize better to other, previously unseen words. For example, we expect other words that carry positive sentiment to occur in similar contexts with *good* and *excellent*, which means that the dot product of their co-occurrence vectors with the parameter $\mathbf{w}$ is less likely to be zero.

8.2 Matrix Decompositions and Low-rank Approximations

But have we really solved the "curse of dimensionality" by using these co-occurrence vectors instead of the original lexical features? One may reasonably argue that we have essentially "passed the buck" from the explicit lexical features, which are indeed sparse, to the co-occurrence vectors, which are probably less sparse, but most likely have not eliminated the sparsity curse. This is intuitively true: consider the co-occurrence vector for the word *bagel* from our previous example. Regardless of how large the underlying document collection used to compute these vectors is and how incredible bagels are, it is very likely that the context vector for *bagel* will capture information about breakfast foods, possibly foods in general and other meal-related activities, but will not contain information about the myriad other topics that occur in these documents in bagel-free contexts.

To further mitigate the curse of dimensionality, we will have to rely on a little bit of linear algebra. Without going into mathematical details, it is possible to decompose the co-occurrence matrix $\mathbf{C}$ into a product of three matrices:

$$\mathbf{C} = \mathbf{U}\Sigma\mathbf{V}^T \tag{8.1}$$

where $\mathbf{U}$ has dimension $M \times r$, Σ is a squared matrix of dimension $r \times r$, and $\mathbf{V}^T$ has dimension $r \times M$.[5] Each of these three matrices has important properties. First, Σ is a diagonal matrix. That is, all its elements are zero with the exception of the elements on the diagonal: $\sigma_{ij} = 0$ for $i \neq j$.[6] The non-zero diagonal values, σ_{ii}, are referred to as

[5] The superscript T indicates the transpose operation. It is used here to indicate that $\mathbf{V}^T$ is computed as the transpose of another matrix $\mathbf{V}$, which has certain mathematical properties. This is less important for our discussion. But we keep the same notation as the original algorithm, for consistency.

[6] For those of us not familiar with the Greek alphabet, σ and Σ are lowercase/uppercase forms of the Greek letter sigma. We use the former to indicate elements in the latter matrix.

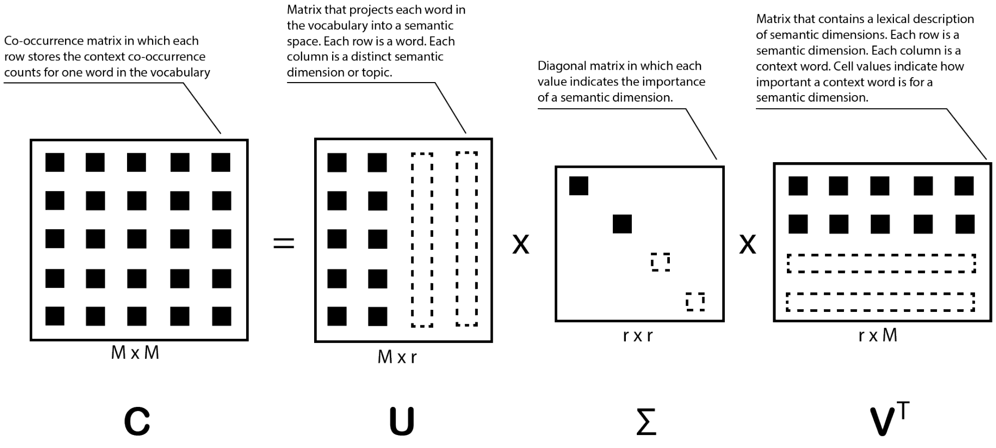

Co-occurrence matrix in which each row stores the context co-occurrence counts for one word in the vocabulary
Matrix that projects each word in the vocabulary into a semantic space. Each row is a word. Each column is a distinct semantic dimension or topic.
Diagonal matrix in which each value indicates the importance of a semantic dimension.
Matrix that contains a lexical description of semantic dimensions. Each row is a semantic dimension. Each column is a context word. Cell values indicate how important a context word is for a semantic dimension.
=
X
X
M x M
M x r
r x r
r x M
C
U
Σ
V^T

Figure 8.1 Summary of the four matrices in the singular value decomposition equation: $\mathbf{C} = \mathbf{U}\Sigma\mathbf{V}^T$. The empty rectangles with dashed lines indicate which elements are zeroed out under the low-rank approximation.

the *singular values* of $\mathbf{C}$, and, for this reason, this decomposition of $\mathbf{C}$ is called *singular value decomposition* or SVD.[7] The dimension of Σ, r, is called the *rank* of the matrix $\mathbf{C}$.[8] Importantly, as we will see in a minute, the values σ_{ii} are listed in descending order in Σ. That is, $\sigma_{ii} > \sigma_{jj}$ for $i < j$. Further, the rows in $\mathbf{U}$ are orthogonal, i.e., the dot product of any two rows in $\mathbf{U}$ is zero. Similarly, the rows in $\mathbf{V}$ (or columns in $\mathbf{V}^T$) are also orthogonal.

So, where does all this math leave us? It turns out that the output of the singular value decomposition process has important linguistic interpretations (see Figure 8.1 for a summary):

1. Each row in the matrix $\mathbf{U}$ contains the numerical representation of a single word in the vocabulary, and each column in $\mathbf{U}$ is one semantic dimension, or topic, used to describe the underlying documents that were used to construct $\mathbf{C}$. For example, if row i contains the co-occurrence vector for the word *bagel* and column j contains a topic describing foods, we would expect c_{ij} to have a high value because the food topic is an important part of the semantic description of the

[7] The general form of singular value decomposition does not require the matrix $\mathbf{C}$ to be square. For this reason, the SVD form we discuss here, which relies on a square matrix $\mathbf{C}$, is referred to as *truncated* singular value decomposition. In this book, we will omit the *truncated* modifier, for simplicity.

[8] In general, the rank of a matrix $\mathbf{C}$ is equal to the number of rows in $\mathbf{C}$ that are linearly independent of each other, i.e., they cannot be computed as a linear combination of other rows. This is not critical to our discussion.

word *bagel*. Importantly however, the SVD algorithm does not guarantee that the semantic dimensions encoded as columns in $\mathbf{U}$ are actually interpretable to human eyes. Assigning meaning to these dimensions is a post-hoc, manual process that requires the inspection of the $\mathbf{V}^T$ matrix (see third item).

2. The singular values in Σ indicate the importance of topics captured in $\mathbf{U}$. That is, if $\sigma_{ii} > \sigma_{jj}$ then topic i (i.e., the column i in $\mathbf{U}$) is more important than column j. And, since the values in Σ are listed in descending order, we can state that topic i is more important than topic j, if $i < j$. This will become important in a minute.
3. Each row i in $\mathbf{V}^T$ contains a bag-of-words description of topic i, where the value at position j in row i indicates the importance of word j to topic i. For example, if the three highest values in a given row point to the words *bagel*, *bread*, and *croissant*, one can (subjectively) interpret this topic to be about bakery products. As mentioned before, such interpretations are not always easy to make. Because the SVD algorithm is completely agnostic to linguistic interpretations, it is possible that some of the produced topics will resist an immediate interpretation. This is an unfortunate drawback we will have to live with, for the sake of mitigating the curse of dimensionality.

While the SVD process produces a new vector representation for each word in the vocabulary, i.e., row i in the matrix $\mathbf{U}$ corresponds to the new representation of word i, we are not quite done. The rank of the matrix $\mathbf{C}$, r, which also indicates the number of columns in $\mathbf{U}$, is guaranteed to be smaller than M, but it is not necessarily much smaller. We would like to produce vector representations of dimension k, where k is much smaller than M, $k \ll M$. To generate these representations, we will take advantage of the fact that, as discussed, the diagonal matrix Σ contains the topic importance values listed from largest to smallest. Thus, intuitively, if one were to remove the *last* $r-k$ topics we would not lose that much information because the top k topics that are most important to describe the content of $\mathbf{C}$ are still present. Formally, this can be done by zeroing out the last $r-k$ elements of Σ, which has the effect of ignoring the last $r-k$ columns in $\mathbf{U}$ and the last $r-k$ rows in $\mathbf{V}^T$ in the SVD multiplication. Figure 8.1 visualizes this process using empty squares and rectangles for the elements in Σ and rows/columns in $\mathbf{U}/\mathbf{V}^T$ that are zeroed out. The resulting matrix $\mathbf{C}$ that is generated when only the first k topics are used is called a *low-rank approximation* of the original matrix $\mathbf{C}$. To distinguish between the two matrices, we will use the notation $\mathbf{C}_k$ to denote the low-rank approximation matrix. There is theory that demonstrates that $\mathbf{C}_k$ is the best approximation of $\mathbf{C}$ for rank k. What this means for us is that we can use the first k columns in $\mathbf{U}$ to generate numerical representations for the words in the vocabulary that approximate as well as possible the co-occurrence counts encoded in **C**. In empirical experiments, k is typically set to values in the low hundreds, e.g., 200. This means that, once this process is complete, we have associated each word in the vocabulary with a vector of dimension $k = 200$ that is its numerical representation according to the distributional hypothesis.

8.3 Drawbacks of Representation Learning Using Low-Rank Approximation

Although this approach has been demonstrated empirically to be useful for several NLP applications including text classification and search, it has two major problems. The first is that this method, in particular the SVD component, is expensive. Without going into mathematical details, we will mention that the cost of the SVD algorithm is cubic in the dimension of **C**. Since in our case the dimension of **C** is the size of the vocabulary, M, our runtime cost is proportional to M^3. In many NLP tasks the vocabulary size is in the hundreds of thousands of words (or more!), so this is clearly a very expensive process.

The second drawback is that this approach conflates all word senses into a single numerical representation. For example, the word *bank* may mean a financial institution, or sloping land, e.g., as in *bank of the river*. But because the algorithm that generates the co-occurrence counts is not aware of the various senses of a given word, all these different semantics are conflated into a single vector. We will address the first drawback in the remaining part of this chapter, and the second in Chapter 10.

8.4 The Word2vec Algorithm

The runtime cost of learning word numerical representations has been addressed by Mikolov et al. [2013], who proposed the word2vec algorithm.[9] Similar to our previous discussion, the goal of this algorithm is to learn numerical representations that capture that distributional hypothesis. More formally, word2vec introduces a training objective that learns "word vector representations that are good at predicting the nearby words." In other words, this algorithm flips the distributional hypothesis on its head. While the original hypothesis stated that "a word is characterized by the company it keeps," i.e, a word is defined by its context, word2vec's training objective predicts the context in which a given word is likely to occur, i.e., the context is defined by the word. Mikolov et al. [2013] proposed two variants of word2vec. For simplicity, we will describe here the variant called "skip-gram," which implements the above training objective. From here on, we will refer to the skip-gram variant of word2vec simply as word2vec.

Figure 8.2 illustrates the intuition behind word2vec's training process. Visually, the algorithm matches the distribution hypothesis exactly: it makes sure that the vector

[9] The name of this algorithm is an abbreviation of "word to vector."

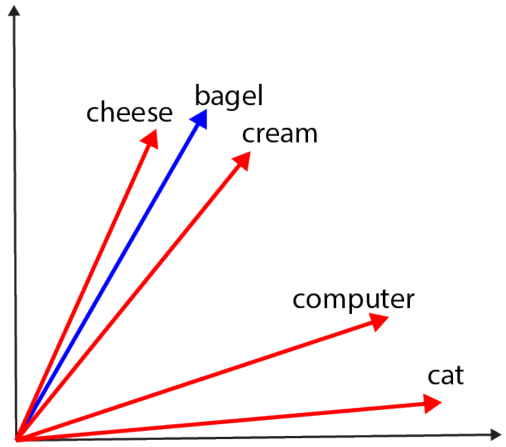


Figure 8.2 An illustration of the word2vec algorithm, the skip-gram variant, for the word *bagel* in the text: *A bagel and cream cheese (also known as bagel with cream cheese) is a common food pairing in American cuisine.* Blue indicates “input” vectors; red denotes “output” vectors. The algorithm clusters together output vectors for the words in the given context window (e.g., *cream* and *cheese*) with the corresponding input vector (*bagel*), and pushes away output vectors for words that do not appear in its proximity (e.g., *computer* and *cat*).

representation of a given word (e.g., *bagel* in the example shown in the figure) is close to those of words that appear near the given word (e.g., *cream* and *cheese*), and far from the vector representations of words that do not appear in its neighborhood (e.g., *computer* and *cat*). Importantly, to distinguish between input words and context words, the algorithm actually learns two vectors for each word in the vocabulary: one for when it serves as an input word (e.g., *bagel* in the example), and one for when it serves as a context our output word (e.g., *cheese*).

More formally, the algorithm implements the distributional hypothesis as a prediction task. First, for each input word w_i in the vocabulary,[10] the algorithm identifies the context windows of size $[-c, +c]$ around all instances of w_i in some large text. This process is identical to the way we constructed the context windows at the beginning of this chapter. For example, the first context window for the word *bagel* and $c = 3$ is: *A* bagel *and cream cheese.* Second, all the context (or output) words that appear in these windows are added to the pool of words that should be predicted given w_i.

[10] In practice, the algorithm uses only the most frequent k words in the vocabulary to reduce training run times.

Then, the training process maximizes the prediction probability for each word w_j in the context of w_i. That is, the theoretical[11] cost function C for word2vec is:

$$C = -\sum_{i=1}^{M} \sum_{w_j \text{ in the context of } w_i} \log(p(w_j|w_i)) \tag{8.2}$$

where the probability $p(w_j|w_i)$ is computed using the input vector for w_i, the output vector for w_j and the softmax function introduced in Section 3.5:

$$p(w_j|w_i) = \frac{e^{\mathbf{v}^o_{w_j} \cdot \mathbf{v}^i_{w_i}}}{\sum_{k=1}^{M} e^{\mathbf{v}^o_{w_k} \cdot \mathbf{v}^i_{w_i}}} \tag{8.3}$$

where $\mathbf{v}^i$ indicates an input vector (i.e., the blue vectors in Figure 8.2), $\mathbf{v}^o$ indicates a context (or output) vector (red vectors in the figure), and the denominator in the fraction iterates over all the words in the vocabulary of size M in order to normalize the resulting probability. All $\mathbf{v}^i$ and $\mathbf{v}^o$ vectors are updated using the standard stochastic gradient descent algorithm during training, similar to the procedure described in Chapter 3. That is, each weight u from a $\mathbf{v}^i$ and $\mathbf{v}^o$ vector is updated based on its partial derivative, $\frac{d}{du}C_i$, where C_i is the loss function for input word i in the vocabulary: $C_i = -\sum_{w_j \text{ in the context of } w_i} \log(p(w_j|w_i))$.

It is important to note at this stage that the last two equations provide a formalization of the intuition shown in Figure 8.2. That is, minimizing the cost function C has the effect of maximizing the probabilities $p(w_j|w_i)$ due to the negative sign in Equation 8.2. Further, maximizing these probabilities has the consequence of bringing the output vectors of context words ($\mathbf{v}^o_{w_j}$) and the input vector for word w_i ($\mathbf{v}^i_{w_i}$) closer together because that maximizes the dot product in the numerator in Equation 8.3. Similarly, maximizing these probabilities has the effect of minimizing the denominator of the fraction in Equation 8.3, which, in turn, means that the dot products with vectors of words *not* in the context of w_i will be minimized.

A second important observation is that there is a very close parallel between this algorithm and the multi-class logistic regression algorithm introduced in Section 3.5. Similar to the multi-class LR algorithm, here we use data points described through a vector representation ($\mathbf{v}^i$ here vs. $\mathbf{x}$ in the standard LR algorithm) to predict output labels (context words vs. labels in $\mathbf{y}$ for LR). Both algorithms have the same cost function: the negative log likelihood of the training data. However, there are three critical differences between word2vec and multi-class LR:

[11] We call this cost function "theoretical" because, as we will see in a minute, this is not what is actually implemented.

Difference #1: while the formulas for the dot products in the two algorithms look similar, in LR the $\mathbf{x}$ is static, i.e., it doesn't change during training, whereas in word2vec both $\mathbf{v}^i$ and $\mathbf{v}^o$ vectors are dynamically adjusted through stochastic gradient descent. This is because the $\mathbf{x}$ vector in LR stores explicit features that describe the given training example (and thus does not change), whereas in word2vec both $\mathbf{v}^i$ and $\mathbf{v}^o$ vectors are continuously moved around in their multi-dimensional space during training to match the distributional hypothesis in the training dataset. For this reason, the word2vec algorithm is also referred to as "dynamic logistic regression."

Difference #2: the $\mathbf{x}$ vector in LR stores explicit features whereas the weights u in the $\mathbf{v}^i$ and $\mathbf{v}^o$ vectors in word2vec are simply coordinates in a multi-dimensional space. For this reason, the output of the word2vec training process is considerably less interpretable than that of LR. For example, in multi-class LR, one can inspect the largest weights in the learned vector $\mathbf{w}_c$ for class c to identify the most important features for the classification of class c. This is not possible for word2vec. Further, word2vec is arguably even less interpretable than the singular value decomposition matrix $\mathbf{U}$ in Section 8.2. There we could use the $\mathbf{V}^T$ matrix to come up with a (subjective) interpretation of each column in $\mathbf{U}$. Again, this is not possible in word2vec, where no such descriptions exist.

Difference #3: Lastly, the number of classes in a multi-class LR problem is usually much smaller than the number of context words in word2vec, which is equal to the size of the vocabulary, M. Typically the former is tens or hundreds, whereas M may be in the millions or billions. Because of this, the denominator of the conditional probability in Equation 8.3 is prohibitively expensive to calculate. Due to this, the actual word2vec algorithm does not implement the cost function in Equation 8.2 but an approximated form of it:

$$C = -\sum_{i=1}^{M}\Big(\sum_{w_j \text{ in the context of } w_i} \log(\sigma(\mathbf{v}^o_{w_j}\cdot\mathbf{v}^i_{w_i})) + \sum_{w_j \text{ not in the context of } w_i} \log(\sigma(-\mathbf{v}^o_{w_j}\cdot\mathbf{v}^i_{w_i}))\Big) \tag{8.4}$$

or, for a single input word w_i:

$$C_i = -\Big(\sum_{w_j \in P_i} \log(\sigma(\mathbf{v}^o_{w_j}\cdot\mathbf{v}^i_{w_i})) + \sum_{w_j \in N_i} \log(\sigma(-\mathbf{v}^o_{w_j}\cdot\mathbf{v}^i_{w_i}))\Big) \tag{8.5}$$

where σ is the standard sigmoid function, $\sigma(x) = \frac{1}{1+e^{-x}}$, P_i is the set of context words for the input word w_i, and N_i is the set of words *not* in the context of w_i.

This new cost function captures the same distributional hypothesis: the first sigmoid maximizes the proximity of input vectors with the output vectors of words in context, whereas the second sigmoid minimizes the proximity of input vectors to output vectors of words not in context, due to the negative sign in the sigmoid parameter: $-\mathbf{v}^o_{w_j} \cdot \mathbf{v}^i_{w_i}$. However, this cost function is much easier to compute than the first cost function in Equation 8.2 for two reasons. First, we are no longer using conditional probabilities, which are expensive to normalize. Second, the right-most term of the cost function in Equation 8.5 does not operate over all the words in the vocabulary, but over a small sample of words that do not appear in the context of w_i. These words can be selected using various heuristics. For example, one can uniformly choose words from the training dataset such that they do not appear in the context of a given input word w_i. However, this has the drawback that it will oversample very frequent words (which are more common and, thus, more likely to be selected). To control for this, the word2vec algorithm selects a non-context word w proportional to the probability $p(w) = \frac{freq(w)^{3/4}}{Z}$, where $freq(w)$ indicates the frequency of word w in the training corpus, and Z is the total number of words in this corpus. The only difference between the probability $p(w)$ and the uniform probability is the 3/4 exponent. This exponent dampens the importance of the frequency term, which has the effect that very frequent words are less likely to be oversampled.

Algorithm 11 lists the pseudocode for the complete training procedure for word2vec that incorporates the discussion above. This algorithm is another direct application of stochastic gradient descent, which is used to update both the input vectors (lines 10 – 12) and output vectors (lines 13 – 17) until convergence (or for a fixed number of epochs). In all update equations, α indicates the learning rate. At the end, the algorithm returns the average of the input and output vectors as the numeric representation of each word in the vocabulary (lines 20 – 22). Note that other ways of computing the final word numeric representations are possible, but the simple average has been observed to perform well in practice for downstream tasks [Levy et al. 2015].

In addition to the more efficient cost function, this algorithm has a second practical simplification over our initial discussion. The algorithm does not identify all context windows for each word in the vocabulary ahead of time, as we discussed when we introduced the cost function in Equation 8.2. This would require complex bookkeeping and, potentially, a considerable amount of memory. Instead, Algorithm 11 linearly scans the text (line 5), and constructs a *local* context P_i and a negative context N_i from the current context window at this position in the text (lines 7 and 8). This has several advantages. First, since only one pair of local P_i and N_i sets are kept in memory at a time, the memory requirements for this algorithm are much smaller. Second, the runtime cost of this algorithm is linear in the size of the training dataset because (a) all operations in the inner `for` loop depend on the size of the context window, which is constant (lines 6 – 17), and (b) the number of epochs used in the external `while` loop (line 4) is a small constant. This is a tremendous improvement over the runtime of the

Algorithm 11: word2vec training algorithm.

```
1  for each word $w_i$ in the vocabulary do
2      initialize $\mathbf{v}^i_{w_i}$ and $\mathbf{v}^o_{w_i}$ randomly
3  end
4  while not converged do
5      for each word position $i$ in the training dataset do
6          $w_i$ = word at position $i$
7          $P_i$ = set of words in the window $[i-c, i+c]$ around $w_i$
8          $N_i$ = sampled from the set of words not in $P_i$
9          compute cost function $C_i$ using $P_i$, $N_i$ and Equation 8.5
10         for each dimension $u$ in $\mathbf{v}^i_{w_i}$ do
11             $u = u - \alpha \frac{d}{du} C_i$
12         end
13         for each word $w_j \in P_i \cup N_i$ do
14             for each dimension $u$ in $\mathbf{v}^o_{w_j}$ do
15                 $u = u - \alpha \frac{d}{du} C_i$
16             end
17         end
18     end
19 end
20 for each word $w_i$ in the vocabulary do
21     return $(\mathbf{v}^i_{w_i} + \mathbf{v}^o_{w_i})/2$
22 end
```

SVD procedure, which is cubic in the size of the vocabulary. One potential drawback of this strategy is that the local N_i used in the algorithm may not be accurate. That is, the words sampled to be added to N_i in line 8 may actually appear in another context window for the another instance of the current word in the training dataset. However, in practice, this does not seem to be a major problem.

The vectors learned by word2vec have been shown to capture semantic information that has a similar impact on downstream applications as the vectors learned through the more expensive low-rank approximation strategy discussed earlier in this chapter [Levy et al. 2015]. We will discuss some of these applications in the following chapters. This semantic information can also be directly analyzed. For example, Mikolov et al. [2013] showed that a visualization of 1000-dimensional vec-

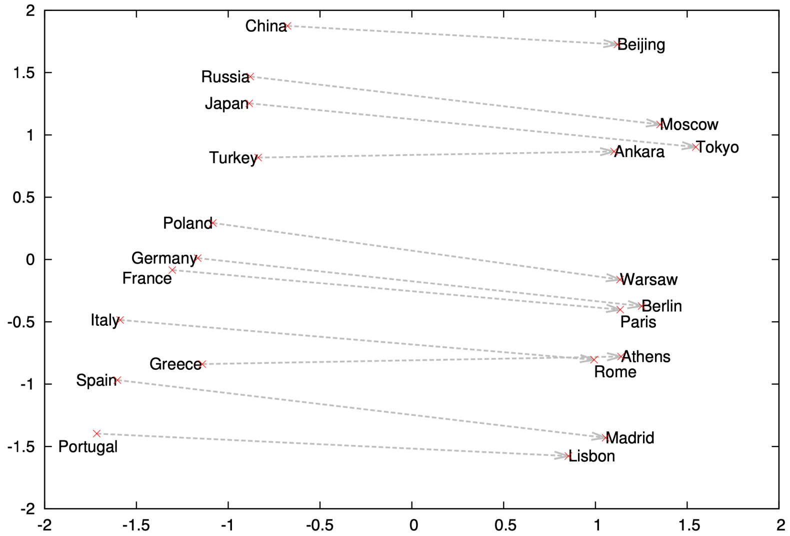


Figure 8.3 Two-dimensional projection of 1000-dimensional vectors learned by word2vec for countries and their capitals [Mikolov et al. 2013].

tors learned by word2vec surfaces interesting patterns. For example, the relation between countries and their capital cities (shown as the difference between the two respective vectors) tends to be same regardless of country and capital (Figure 8.3). That is, $\vec{China} - \vec{Beijing} \approx \vec{Portugal} - \vec{Lisbon}$, where the superscript arrow indicates the vector learned by word2vec for the corresponding word. Many other similar patterns have been observed. For example, the difference between the vectors of *king* and *man* is similar to the difference between the vectors of *queen* and *woman*: $\vec{king} - \vec{man} \approx \vec{queen} - \vec{woman}$, which suggests that this difference captures the semantic representation of a genderless monarch. In the following chapters, we will see how we use these vectors to replace the manually-designed features in our NLP applications.

8.5 Drawbacks of the Word2vec Algorithm

Word2vec has the same drawbacks as the low-rank approximation algorithm previously discussed. Both approaches produce vectors that suffer from lack of interpretability, although one could argue that word2vec's vectors are even less interpretable than the low-rank vectors in the $\mathbf{U}$ matrix, whose dimensions can be somewhat explained using the $\mathbf{V}^T$ matrix.

Further, similar to the SVD-based strategy, word2vec conflates all senses of a given word into a single numerical representation. That is, the word *bank* gets a single numerical representation regardless of whether its current context indicates a financial sense, e.g., *Bank of America*, or a geological one, e.g., *bank of the river*. In Chapter 10 we will discuss strategies to build word vector representations that are sensitive of the current context in which a word appears.

TODO: bias [Bolukbasi et al. 2016] $\vec{doctor} - \vec{man} \approx \vec{nurse} - \vec{woman}$

8.6 Historical Background

TODO: todo

8.7 References and Further Readings

TODO: todo

9 Implementing the Neural Review Classifier Using Word Embeddings

10 Contextualized Embeddings and Transformer Networks

As mentioned in the previous chapter, all the algorithms discussed there conflate all senses of a word into a single numerical representation (or embedding). For example, the word *bank* receives a single representation, regardless of its financial or geological sense. This chapter introduces a solution for this limitation, in the form of a new neural architecture called transformer networks (TNs) [Devlin et al. 2018, Vaswani et al. 2017], which learn *contextualized embeddings* of words, which, as the name indicates, change depending on the context in which the words appear.[1] That is, the word *bank* receives a different numerical representation for each of its instances in these two texts: *Bank of America* and *bank of the river* because the contexts in which they occur are different.

While transformer networks have a relatively complex structure, the intuition behind them is simple: each word receives a contextualized embedding that is a *weighted average* of some context-independent input embeddings (i.e., embeddings that are conceptually similar to the word2vec embeddings from the previous chapter).[2] Figure 10.1 visualizes this intuition. As the figure shows, the contextualized embedding of the word *bank* combines all four of the input embeddings. Because this includes the input embedding for *river*, the resulting contextualized embedding of *bank* will lean towards a semantic representation closer to geology than finance. As the figure indicates, the same process applies to all other words in the text. Importantly, the weights used for the weighted averages that generate the output embeddings are specific to each word. This allows transformer networks to generate distinct output embeddings for the different words in the text to be processed.

In reality, transformer networks consist of multiple layers, where each layer implements a weighted average as discussed above. This stacked architecture is summarized in Figure 10.2. As the figure shows, the output embeddings for layer i become the input embeddings for layer $i+1$. The number of layers typically ranges between 2 and 24. Relying on multiple layers allows transformer networks to learn more complex

[1] In this chapter we are actually discussing only the first half of the transformer networks architecture, in which these contextualized embeddings are generated. We will discuss the second part in Chapter 14.

[2] Or, rather, each word receives a weighted average of *projections* of input embeddings into a new feature space. We will deal with these details later in this chapter.

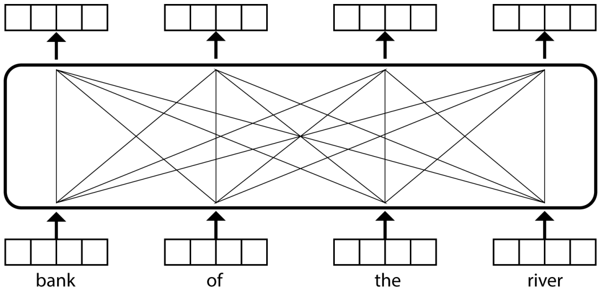


Figure 10.1 Intuition behind transformer networks: each output embedding is a weighted average of all input embeddings in the context.

functions for assembling the eventual output embeddings, which has been empirically shown to yield more meaningful representations.

In the next section we will discuss the architecture of the individual TN layer, which is the key transformer networks building block. Then, we will discuss the tokenization strategy used by transformer networks, which is different from what we discussed so far in the book. We will conclude the technical description of transformer networks in this chapter with a discussion of their training procedure.

10.1 Architecture of a Transformer Layer

Figure 10.3 shows the internal architecture of an individual TN layer. As shown, each layer implements a sequence of five operations. The first operation adds positional information to the input embedding of each word in the input text. That is, at this stage, the embedding of each word changes depending on its position in the input text. For example, each occurrence of the word *shipping* in the tongue twister *A ship shipping ship shipping shipping ships* will receive a different embedding because they occur at different positions in the text.

The second operation implements the key functionality of the TN layer summarized in Figure 10.1, i.e., generating embeddings that are a weighted average of the embeddings produced by the previous operation. In deep learning parlance this weighted average is called “self attention.”[3] The term “attention” is used to indicate that this component identifies the important parts of the data and “pays attention” to them

[3] This terminology is inspired from cognitive science and the actual cognitive attention. To the knowledge of the authors, it has not been proven that this connection between the weighted average performed by transformer networks (or other deep learning architectures) and the cognitive attention process is justified. However, because “attention” is widely used in the deep learning literature, we will continue to use this terminology throughout the book.

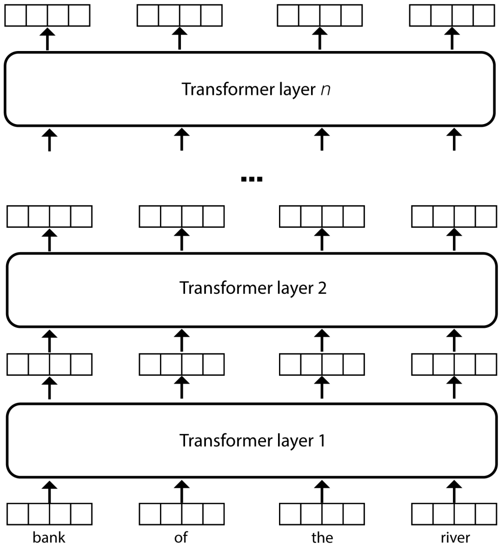


Figure 10.2 A transformer network consists of multiple layers, where each layer performs a weighted average of its input embeddings.

more (through higher weights in the weighted average). "Self" indicates that this component operates over its own input text (we will see other types of attention that operate over other texts in the following chapters).

The next three operations are not that important conceptually, but they have been shown to have a significant contribution empirically. For example, the "Add and normalize" components sum up input and output embeddings from the previous component in the pipeline (e.g., the input embeddings to the self attention layer and the corresponding output embeddings computed through the weighted average), and normalize the results to avoid values that might be too large, which may negatively impact gradient descent. The feed forward components encode each received numerical representation into a new vector, which allows the TN layer to learn more complex functions.

We detail all these operations next.

10.1.1 Positional Embeddings

Our astute reader has likely observed that the weighted average summarized in Figure 10.1 suggests a "bag-of-words" model. That is, transformer networks seem

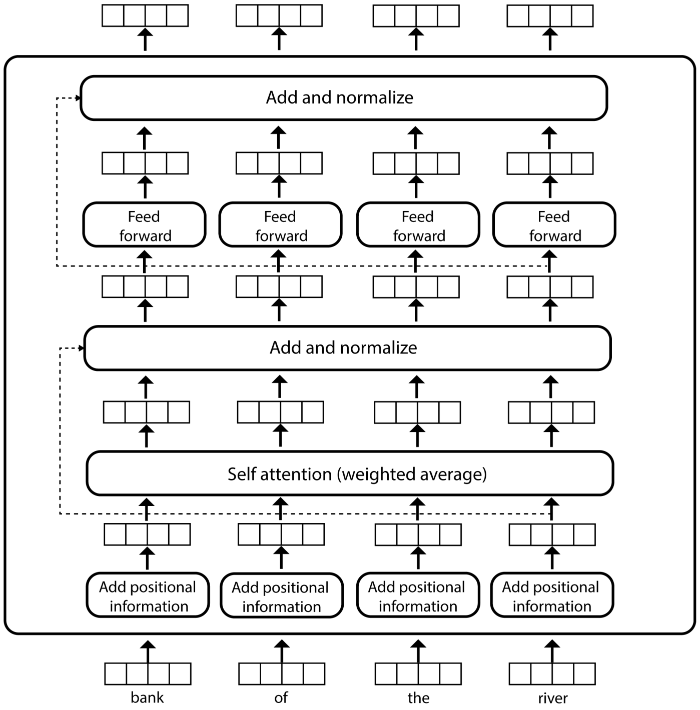

Add and normalize
Feed forward
Feed forward
Feed forward
Feed forward
Add and normalize
Self attention (weighted average)
Add positional information
Add positional information
Add positional information
Add positional information
bank
of
the
river

Figure 10.3 Architecture of an individual transformer layer.

to produce contextualized embeddings that are independent of the order of the words in the input text. This is clearly less than ideal. For example, in the text *Bank of America financed a repair of the river bank* the machine should be able to figure out that the first instance of *bank* refers to a financial institution because it is closer to the word *financed* and farther away from the word *river*. Modeling such proximity information requires keeping track of positional information, i.e., where words occur in the input texts.

Before transformer networks, neural approaches typically modeled positional information as a separate numerical representation. That is, each value i in a list of possible word positions in a text (say, 1 to 100), is associated with a numerical representation $\mathbf{p}_i$, which is initialized with random values. Then, the input embedding of word w_i becomes a concatenation of two vectors: $\mathbf{x}_i = [\mathbf{w}_i; \mathbf{p}_i]$, where $\mathbf{w}_i$ is a regular word embedding, e.g., coming from word2vec. The training process of a neural network on top of such embeddings back-propagates gradients all the way back to these vectors. Thus, during training, the network also learns numerical representations for word positions through the $\mathbf{p}$ vectors. The advantage of this strategy to model positional information is flexibility: the network learns numerical representations for position values that are customized for the task at hand. The disadvantages of this approach are some additional cost caused by the extra back-propagation operations, and the inability to handle position values not seen in training. For example, assume that all training sentences for some NLP task contain fewer than 100 words. The network trained on this data will not know what numerical representations to assign to words that occur at positions larger than 100 during evaluation.

Transformer networks address these drawbacks by using hard-coded functions to generate numerical representations of word positions. That is, for a word at position i in the input text, transformer networks generate a vector $\mathbf{p}_i$, where the value at position j in this vector is computed using a function that depends on both i and j: $f(i, j)$. The actual function f used is not that important;[4] suffice to say that the resulting vector $\mathbf{p}_i$ encodes positional information because it is unique for each word position i. Then, the input embedding $\mathbf{x}_i$ for word w_i becomes: $\mathbf{x}_i = \mathbf{w}_i + \mathbf{p}_i$.[5] This approach mitigates the two disadvantages mentioned before: the function f is hard-coded, and, thus, there is no need to learn it. f generates different values for any word position i, and, thus, works for previously unseen word positions. The transformer networks creators mention that the proposed method performs similarly in practice as the more expensive and less flexible strategy discussed in the previous paragraph.

10.1.2 Self Attention

As indicated earlier, this self attention layer is the key building block in transformer networks. For each word w_i, this layer produces an output embedding, $\mathbf{z}_i$, that is a combination of all input embeddings $\mathbf{x}$ (i.e., the embeddings produced by the previous component that infuses positional information), for all the words in the input text.

To generate the output embeddings $\mathbf{z}_i$, the self attention layer uses three vectors for each word w_i: a query vector $\mathbf{q}_i$, a key vector $\mathbf{k}_i$, and a value vector $\mathbf{v}_i$. These vectors are nothing magical: they are just arrays of real values, e.g., a $\mathbf{q}_i$ vector might look like $(0.58, -045, 0.34, \ldots)$. We will discuss later in this chapter how the values in these vectors are learned; for now let us simply assume that these three vectors exist for every word in the input text, and they are populated with some meaningful values. At a high level, the query and key vectors are used to generate unique attention weights for each pair of words w_i and w_j. That is, the dot product $\mathbf{q}_i \cdot \mathbf{k}_j$ indicates how important word w_j is for the output embedding of word w_i. The value vector $\mathbf{v}_i$ is a

[4] We refer the reader to the transformer networks paper for details on this function [Vaswani et al. 2017].

[5] Note that transformer networks sum the $\mathbf{w}_i$ and $\mathbf{p}_i$ vectors to generate $\mathbf{x}_i$, whereas the approach previously discussed concatenates them. The motivation behind this design decision is that the summation allows back-propagation to adjust both $\mathbf{w}_i$ and $\mathbf{p}_i$ vectors, similarly to the concatenation approach, but without increasing the dimension of the $\mathbf{x}_i$ vector.

projection (or transformation) of the input vector $\mathbf{x}_i$ into a new feature space. Each output embedding $\mathbf{z}_i$ will be a weighted average of these value vectors.[6]

To formalize a bit more, given the query, key, and value vectors for all words in the input text (i.e., $\mathbf{q}_i$, $\mathbf{k}_i$, $\mathbf{v}_i$ for word w_i), the self attention algorithm operates as follows:

1. For each pair of words, w_i and w_j, compute the attention weight a_{ij} using the $\mathbf{q}_i$ and $\mathbf{k}_j$ vectors. In particular:
   (a) Initialize the attention weights a_{ij} with the product of the corresponding query and key vectors: $a_{ij} = \mathbf{q}_i \cdot \mathbf{k}_j$.
   (b) Divide the above values by the square root of the length of the key vector: $a_{ij} = a_{ij}/\sqrt{|\mathbf{k}_1|}$ (all $\mathbf{k}_i$ vectors have the same size, so we arbitrarily use $\mathbf{k}_1$ here).
2. For each word w_i, apply softmax on all its attention weights, a_{ij}.
3. For each word w_i, multiply the above attention weights with the corresponding value vectors, for all words w_j. Then sum up all these weighted vectors to produce the output vector for w_i: $\mathbf{z}_i = \sum_j a_{ij} \ \mathbf{v}_j$.

Table 10.1 shows an (artificial) walkthrough example for this algorithm. Step 1(a) shows the result of the $\mathbf{q}_1 \cdot \mathbf{k}_j$ multiplications, i.e., for all combinations between the word *bank* and the words in the text (*bank* included). These attention weight values indicate that the word *bank* pays attention to itself (i.e., a_{11} is large) and the word *river* (a_{14} is also large), which disambiguates *bank* in the current context, and less attention to the more ambiguous words in the context (*of* and *the*). The next step, 1(b), divides these values by the square root of the key vector. This heuristic is necessary to mitigate the *exploding gradient* phenomenon. This phenomenon is the opposite of the vanishing gradient phenomenon that we discussed in Chapter 5. That is, large parameter values in the network such as these attention weights yield gradient values that are consequently also large during back-propagation, which cause unstable learning due to too much "jumping around" in the parameter space, or even overflow in parameter values.

Step 2 applies a softmax layer on the resulting weights, which converts them into a probability distribution. This is necessary to: (a) produce a meaningful weighted average in the next step below, and (b) to further control for large weight values. Later on in this section, we will see another component that aims to control for unreasonably large parameter values in TN. Lastly, step 3 computes the output embedding, $\mathbf{z}_1$, as a

[6] Thus, Figure 10.1, oversimplifies the self attention component when it suggests that the weighted average operates over the input vectors. We hope the reader pardons this temporary approximation, which was introduced for the sake of pedagogy.

Table 10.1 A self-attention walkthrough example for computing the contextual embedding $\mathbf{z}_1$ for the word *bank* in the text *bank of the river*.

1 (a)	Compute all the $\mathbf{q}_1 \cdot \mathbf{k}_i$ dot products for the four words in the text: $a_{11} = \mathbf{q}_1 \cdot \mathbf{k}_1 = 40$ $a_{12} = \mathbf{q}_1 \cdot \mathbf{k}_2 = 16$ $a_{13} = \mathbf{q}_1 \cdot \mathbf{k}_3 = 8$ $a_{14} = \mathbf{q}_1 \cdot \mathbf{k}_4 = 32$
1 (b)	Divide all the above values by $\sqrt{\lvert\mathbf{k}_1\rvert} = 8$: $a_{11} = 5$ $a_{12} = 2$ $a_{13} = 1$ $a_{14} = 4$
2	Apply softmax on the above 4 values: $a_{11} = 0.70$ $a_{12} = 0.03$ $a_{13} = 0.01$ $a_{14} = 0.26$
3	Compute the contextualized embedding $\mathbf{z}_1$ for *river*, as a weighted average of the value vectors: $\mathbf{z}_1 = 0.70\mathbf{v}_1 + 0.03\mathbf{v}_2 + 0.01\mathbf{v}_3 + 0.26\mathbf{v}_4$

weighted average of the $\mathbf{v}$ vectors for all the words in the context multiplied by their corresponding attention weights generated in the previous step.

The above walkthrough example shows that, given query, key, and value vectors for words in the input text, it is relatively trivial to produce contextualized output embeddings. But where do the query, key, and value vectors come from? All these vectors are generated by projecting the input embeddings $\mathbf{x}$ into a new feature space. More formally, each self attention block contains three matrices, $\mathbf{W^Q}$, $\mathbf{W^K}$, and $\mathbf{W^V}$, where $\mathbf{W^Q}$ and $\mathbf{W^K}$ have dimension $|\mathbf{x}| \times |\mathbf{k}|$, and $\mathbf{W^V}$ has dimension $|\mathbf{x}| \times |\mathbf{v}|$.[7] Each of the $\mathbf{q}_i$, $\mathbf{k}_i$, and $\mathbf{v}_i$ vectors is then computed simply as:

1. $\mathbf{q}_i = \mathbf{x}_i \times \mathbf{W^Q}$,
2. $\mathbf{k}_i = \mathbf{x}_i \times \mathbf{W^K}$, and
3. $\mathbf{v}_i = \mathbf{x}_i \times \mathbf{W^V}$.

[7] Note that the $\mathbf{q}$ and $\mathbf{k}$ vectors must have the same dimension because of the dot product in the computation of the attention weights.

Thus, the parameters in a self attention layer are the three matrices, $\mathbf{W^Q}$, $\mathbf{W^K}$, and $\mathbf{W^V}$. A typical configuration for the self attention layer has $|\mathbf{x}| = 512$, and $|\mathbf{k}| = |\mathbf{v}| = 64$.

10.1.3 Multiple Heads

Transformer networks further expand the above self attention layer by repeating it multiple times. A typical configuration includes eight different instances of the above algorithm. Each of these instances is called a "head." To make sure that the heads capture different information, each head receives different copies of the $\mathbf{W^Q}$, $\mathbf{W^K}$, and $\mathbf{W^V}$ matrices, which are all initialized with different values. This allows each layer to produce output embeddings $\mathbf{z}$ that operate in different feature spaces, and, hopefully, capture complementary information. Then, the actual embedding $\mathbf{z}_i$ for the word at position i is computed as the product between the concatenated embeddings produced by each head and a new "output" matrix $\mathbf{W^O}$:

$$\mathbf{z}_i = [\mathbf{z}_i^1; \mathbf{z}_i^2; \ldots \mathbf{z}_i^n] \times \mathbf{W^O} \tag{10.1}$$

where the superscript j in $\mathbf{z}_i^j$ indicates which head produced it, and n indicates the total number of heads. The dimension of the output matrix $\mathbf{W^O}$ is $n|\mathbf{v}| \times |\mathbf{x}|$, which guarantees that the output embeddings $\mathbf{z}$ have the same dimension as the input embeddings $\mathbf{x}$.

Thus, the complete list of parameters for a multi-head self attention layer includes n copies of the $\mathbf{W^Q}$, $\mathbf{W^K}$, and $\mathbf{W^V}$ matrices, and one copy of the output matrix $\mathbf{W^O}$. Under a typical configuration of $|\mathbf{x}| = 512$, $|\mathbf{k}| = |\mathbf{v}| = 64$, and $n = 8$, this means that a multi-head self attention layer contains 1,048,576 parameter weights.

10.1.4 Add and Normalize and Feed Forward Layers

The next three components, i.e., two instances of the add and normalize layer, and one feed forward layer, are not that important conceptually, but they do matter empirically.

Each of the add and normalize layers starts by summing up the input and output embeddings produced by the previous component. For example, the add and normalize layer that follows the self attention layer sums up the $\mathbf{x}_i$ and $\mathbf{z}_i$ for each word i. The motivation for this summation is to make sure that the signal from the input embeddings $\mathbf{x}$ (which we know from the Chapter 8 carry important information) does not get lost in the machinery implemented by the self attention layer. Then, the resulting embedding, $\mathbf{x}_i + \mathbf{z}_i$, is normalized using layer normalization, as discussed in Section 6.8. This latter normalization step is yet one more component that aims to mitigate the exploding gradient problem.

The feed forward layer in between the two add and normalize layers projects the output embeddings into a new feature space, similar to what the value matrix, $\mathbf{W^V}$, is doing to the input embeddings. This introduces more parameters in the transformer layer, which allow transformer networks to learn more complex functions.

10.2 Sub-word Tokenization

So far, we have used the term "word" to describe the inputs to the first transformer layer, but that is a misnomer introduced to simplify our presentation so far. In reality, transformer networks operate over subword units, i.e., their tokens may be word fragments rather than complete words. These subword units are generated automatically using the *byte pair encoding* (BPE) algorithm for word segmentation [Sennrich et al. 2015]. In a nutshell, this algorithm creates a symbol vocabulary, which keeps track of the allowed subword units. This dictionary is initialized with individual characters, including a special symbol `</w>` that indicates the end of a word. Then, it iteratively counts the frequency of symbol pairs in a large text corpus, and replaces the most frequent pair with the concatenation of the two symbols, which is also added to the symbol dictionary. Merging is not allowed across word boundaries, which means that the new symbols created during the merge operations are always subwords (up to entire words), and never include parts of different words. The size of the output symbol dictionary is equal to the size of the initial dictionary plus the number of merge operations (because each merge creates a new symbol).

For example, assume that *bank of the river* is part of the training corpus for the BPE algorithm. Then, in the first iteration, this phrase will be segmented into individual characters and end-of-word markers:

b a n k `</w>` *o f* `</w>` *t h e* `</w>` *r i v e r* `</w>`

Let's say that the most frequent sequence of symbols so far is *t h*. Then, after merging these two symbols and adding *th* the symbol vocabulary, our text becomes:

b a n k `</w>` *o f* `</w>` *th e* `</w>` *r i v e r* `</w>`

Then, if the most frequent pair now is *th e*, the text becomes:

b a n k `</w>` *o f* `</w>` *the* `</w>` *r i v e r* `</w>`

and so on, until we reach the desired size for the output symbol dictionary. A typical size for this dictionary is 50,000. Since 50,000 is clearly smaller than the size of any language's vocabulary, it is unavoidable that BPE tokenization will fragment infrequent words into one or more subword units. For example, the word *transformers* might be fragmented into *transform* and *ers*.

Because transformer networks rely on this subword tokenization, their input embeddings, i.e., the embeddings that are fed into the first layer in the architecture (Figure 10.3), no longer align with "traditional" word embedding algorithms such as word2vec, which operate over complete words. For this reason, TNs initialize the input embeddings assigned to subword units randomly, and update them together with the rest of the their parameters during training (which we will discuss in the next section).

But why go through this additional trouble of subword tokenization? There are two advantages. First, operating over subwords makes the transformer network more robust to unknown words. For example, assume that a transformer network sees the word *transformers* for the first time after training. A traditional word embedding algorithm may not know how to handle this word (or, at least, how to handle it well), but a transformer network may still be able to, if it tokenizes it into subwords that were seen in training such as *transform* and *ers*, for which it has trained input embeddings. The second advantage is saving space. That is, a word embedding algorithm that relies on complete words may quickly reach a vocabulary size in the billions. This is because vocabulary size keeps growing indefinitely as the underlying text corpus grows (Chapter 5 in [Schütze et al. 2008]). This is a problem: if we have a vocabulary of one billion words, and we use vectors of size 512 for the input embeddings, we would need $4 \times 512 \times 1,000,000,000 = 2,048$ gigabytes[8] just to store the input embeddings! In contrast, a transformer network with 50,000 subword units requires only $4 \times 512 \times 50,000 = 102.4$ megabytes for its input embeddings.

10.3 Training a Transformer Network

So far, we have introduced transformer networks, which allow us to construct contextualized embeddings for a sequence of (subword) tokens. But how do we learn their parameters, e.g., the input embeddings for the subword tokens, or the various **W** matrices in each layer? At a high level, the training process for transformer networks consists of two steps: one unsupervised procedure called *pre-training*, and a supervised one called *fine-tuning* [Devlin et al. 2018]. Devlin et al. [2018] called the transformer network that resulted from this training process BERT, from **B**idirectional **E**ncoder **R**epresentations from **T**ransformers.[9] We discuss the two training procedures next.

10.3.1 Pre-training

The pre-training procedure uses a *masked language model* (MLM) objective [Devlin et al. 2018]. That is, during pre-training we randomly mask tokens in some input text by replacing them with a special token, e.g., `[MASK]`, and ask the transformer network

[8] Assuming we use 4 bytes to store each real number in the 512-dimensional embedding vector.

[9] This started a somewhat unfortunate trend that overused Muppet names for variants of transformer network architectures.

to guess the token behind the mask. Typically, 15% of the input tokens are masked in the training texts. While this task sounds (and is!) trivial, there are several important details that need discussing:

1. First, this task is implemented as a multi-class classifier that uses as input the contextualized embeddings of the masked words and generates one of the subword tokens from the vocabulary created by the BPE algorithm. That is, imagine the transformer as the sequence of layers as summarized in Figure 10.2. Then, if we mask the word *river*, the classifier that guesses the masked word will operate on top of the contextualized embedding produced by layer n for the word *river*. The classifier itself is implemented simply as a softmax layer that produces a probability distribution over all the tokens in the BPE vocabulary, and is trained with the usual cross-entropy loss function, which maximizes the probability of the "correct" token, i.e., the token that was masked.
2. Pre-training is often referred to as an unsupervised algorithm because it does not require any annotated training data from domain experts (e.g., as we would when training our review classifier). This is not entirely correct: after all, the texts used during pre-training are written by people and, thus, provide some human supervision. However, what is important is that, for most languages, there is a plethora of texts available on the web that can be easily transformed into training data for transformer networks using the masked language model trick. For this reason, most MLM pre-training settings include billions of words. This allows transformer networks to capture many language patterns before they are trained on any specific NLP application (see the fine-tuning subsection below)!
3. Because the masked tokens can appear anywhere in a given text and, thus, there is meaningful context that can be used to guess the masked token both to the left and to the right of the mask, this pre-training procedure is called a *bidirectional* language model. This is to contrast it with traditional language modeling (LM) tasks, which normally proceed left to right. That is, a traditional LM guesses what word follows after the user types a few words, like the texting application in your phone. The pre-training procedure does not have this directionality constraint, as it can "peek" on both sides of the mask.

A second pre-training procedure that was proposed by Devlin et al. [2018] is *next sentence prediction* (NSP). This task trains transformer networks to predict which sentence follows a given sentence. Similar to the previous MLM task, this task is "unsupervised," in the sense that is relies solely on text without any additional expert annotations. However, unlike MLM, which operates over contextualized embeddings of individual tokens, this task requires a different setting, which is exemplified in

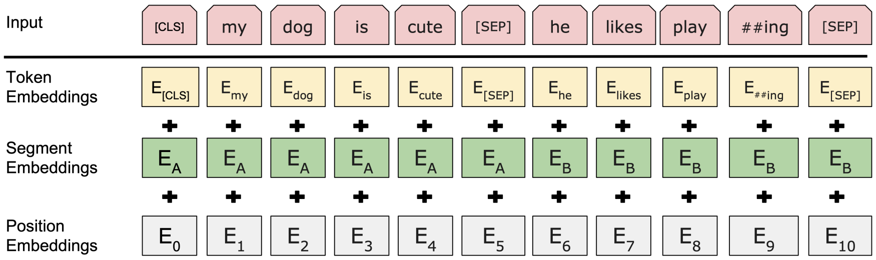


Figure 10.4 Input example for the next sentence prediction pre-training task from [Devlin et al. 2018]. `[SEP]` is a special separator token used to indicate end of sentence. The `[CLS]` token stands for *class*, and is used to train the binary classifier, which indicates if sentence B follows sentence A in text or not. The `##` marker indicates that the corresponding token is a subword token that should be appended to the token to its left.

Figure 10.4. The figure shows that, unlike the original transformer network, the input embeddings for NSP sum up three embeddings: token and position embeddings (similar to the original architecture) and a new embedding that encodes which segment the current token belongs to (A or B). More importantly, this architecture introduces the virtual `[CLS]` token, which is inserted at the beginning of the text, and whose contextualized embedding is used to train the binary NSP classifier. That is, the actual classifier is simply a sigmoid on top of the contextualized embedding for `[CLS]`. Note that the `[CLS]` token is treated like any other token in the text, i.e., its attention weights cover all the tokens in the text. Thus, the classifier that operates on top of the `[CLS]` contextualized embedding has indirect access to the entire text through its attention mechanism.

To train the classifier, NSP generates positive examples from actual sentences that follow a given sentence (as in the figure), and negative examples from sentences randomly sampled from the corpus. The proportion of negative to positive examples is 1:1.

Devlin et al. [2018] report that pre-training BERT with NSP benefits downstream tasks such as question answering. However, other works have "questioned the necessity of the NSP loss" for NLP applications [Liu et al. 2019]. That is, Liu et al. [2019] observe that training solely with the MLM objective performs just as well or better than training with both MLM and NSP on downstream tasks.

Table 10.2 Two examples of inputs for NLP applications formatted for transformer networks.

Application	Example input
Review classification	[CLS] although this was obvious ##ly a low budget production the perform ##ances and the songs in this movie are worth seeing . [SEP] one of walken 's few musical roles to date .
Question answering	[CLS] Who is the president of the united states ? [SEP] joseph rob ##inette biden jr. is an american politician who is the 46 ##th and current president of the united states .

10.3.2 Fine-tuning

The above pre-training procedures allow transformer networks to capture a variety of language patterns that are application independent. In contrast, fine-tuning trains transformer networks for specific NLP applications such as text classification, question answering, natural language inference, etc. We will discuss these applications in detail in Chapter 15. For now, we will simply mention that many of these applications can be modeled with architectures similar to the one shown in Figure 10.4, where one or more texts (separated by [SEP]) are preceded by a [CLS] token, which drives the actual classification. Table 10.2 shows a few example inputs for two NLP applications. As the table shows, many of these tasks can be reduced to classification tasks (again, on top of the [CLS] embedding) that receive as input one or more sentences. For example, for review classification, the input sentences are the actual sentences in the review, and the classification task needs to produce multiple labels such as Positive, Negative, or Neutral. For question answering, the texts that are concatenated contain the question and the sentence that potentially answers it. The classification task in this case is binary, i.e., is this a correct answer or not?

During fine-tuning, the training process receives a transformer network that was pre-trained using one of the algorithms discussed in the previous subsection. The training continues with a loss function that is specific to the task at hand. For example, for review classification, the loss will be the standard cross-entropy, which will maximize the probability of the correct review label, e.g., Positive for the review in Table 10.2.

10.4 Historical Background

TODO: To do

10.5 References and Further Readings

TODO: To do

11 Using Transformers with the Hugging Face Library

12 Sequence Models

13 Implementing Sequence Models

14 Sequence-to-sequence Methods

15 Neural Architectures for NLP Applications

A Overview of the Python Programming Language

B Character Encodings: ASCII and Unicode

Every NLP practitioner needs to understand how computers represent text. In this appendix we will discuss different representations of text and try to demystify the concepts involved. In particular, we will discuss the difference between text and bytes, what encoding/decoding means in this context, and text normalization. This is a vast topic and we will not cover it completely. We aim to explain the fundamentals, in particular the ones most needed for NLP.

B.1 How Do Computers Represent Text?

Computers represent text the same way that they represent images, sounds, video, and everything else: as numbers. *Character encodings* establish a mapping between characters and unique numbers that identify them. Note that character encodings are not exclusive to computers. For example, Morse code is a character encoding that predates computers, but that illustrates the usefulness of encoding text into a representation that can be transmitted over long distances.[1]

In the United States, the ASCII character encoding became an early standard for encoding text in computers.[2] ASCII consists of 127 characters. Of these, 96 are *printable* characters (i.e., letters, digits, punctuation, and other symbols), and the rest are *control* characters, which were not meant to be printed, but to control devices such as printers. As mentioned, each of these characters needs to be represented as a number by the computer. We list the numbers corresponding to ASCII control characters in Table B.1, and those for printable characters in Table B.2.

Computers represent numbers as sequences of ones and zeros, with each digit in the sequence being referred to as a *bit*.[3] Internally, computers manipulate groups of bits at a time. For mostly historical reasons that we will not explore here, computers group sequences of 8 bits together; this is referred to as a *byte*. Because a byte has 8 bits, it can encode $2^8 = 256$ values. Thus, since ASCII has 127 characters, any ASCII character can be stored in a byte.

[1] As electric pulses over wires.

[2] ASCII is short for American Standard Code for Information Interchange.

[3] The term bit is a portmanteau for *binary digit*.

Table B.1 ASCII control characters.

number	description	number	description
0	Null	16	Data Line Escape
1	Start of Heading	17	Device Control 1
2	Start of Text	18	Device Control 2
3	End of Text	19	Device Control 3
4	End of Transmission	20	Device Control 4
5	Enquiry	21	Negative Acknowledgment
6	Acknowledgment	22	Synchronous Idle
7	Bell	23	End of Transmit Block
8	Backspace	24	Cancel
9	Horizontal Tab	25	End of Medium
10	Line Feed	26	Substitute
11	Vertical Tab	27	Escape
12	Form Feed	28	File Separator
13	Carriage Return	29	Group Separator
14	Shift Out	30	Record Separator
15	Shift In	31	Unit Separator

As Table B.2 shows, the ASCII encoding is tailored for languages that rely on the Latin alphabet, and not for others. To begin to remedy this limitation, several standards emerged that used ASCII as a base, but that added more characters, taking advantage of the fact that ASCII only defines 127 out of 256 possible values in a byte. ISO-8859-1, commonly referred to as Latin-1, is an example of an encoding that adds characters to ASCII to support additional languages spoken in Europe and parts of Africa. Windows-1252 is another of these encodings, which extends ISO-8859-1 and was popularized by Microsoft. Other regions of the world developed their own standards to suit their needs such as the Japanese Industrial Standard (JIS) in Japan.[4]

As the Internet and the world became more connected, the need to share documents between the many regions of the world became more pressing, and the different encodings became problematic. Further, the various ASCII extensions are still woefully insufficient to encode the variety of world's alphabets. This led to a push for a universal encoding for all writing systems: in 1991, the Unicode Foundation published the first version of the Unicode standard. The Unicode standard can be thought of as a large table that assigns a unique identifier to each character, much like ASCII, but with tens of thousands of characters instead of a couple of hundred. Each of these numerical

[4] https://en.wikipedia.org/wiki/JIS_encoding

Table B.2 ASCII printable characters.

number	character	number	character	number	character	number	character
32	(space)	56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g	127	(delete)

identifiers is referred to as a *code point.*[5] Note that not all of these code points correspond to what one would typically consider a character. Some of these, called *combining characters*, must be combined with another code point to form a character. This means that there can be a sequence of code points that together form a single character. For example, diacritical marks such as accents are commonly combined with Latin letters to form characters in many European languages, e.g., á in Spanish.

Importantly, Unicode code points do not fit into a single byte the way ASCII characters do. Instead, we need a way of transforming a code point into multiple

[5] There are 143,859 characters as of Unicode 13.0. For backwards compatibility, the first 128 Unicode code points correspond to the ASCII characters.

bytes. The most convenient encoding for this is UTF-8, which encodes a code point into a sequence of bytes of variable length, between 1 to 4 bytes, depending on its value. One nice property of UTF-8 is that ASCII is a subset of UTF-8, as the first 128 Unicode code points correspond to the ASCII characters. In practice, this means that any file encoded with ASCII can be decoded with UTF-8. Further, the algorithm for the UTF-8 encoding uses as few bytes as possible to encode the code points, which helps reduce the size of the represented texts.

Lastly, the Unicode standard also establishes other character properties such as names and numeric values. For example, the character ½ has a numeric value of 0.5.

```
>>> s = '½'
>>> unicodedata.numeric(s)
0.5
```

The Unicode standard also provides rules for text normalization, collation, and even rendering.

B.2 How to Encode/Decode Characters in Python

Now that we have a better understanding of what character encodings are, and why they exist, let's see how they work in Python. We'll start with a string that has an acute accent in the first character:

```
>>> s = 'ábaco'
```

Calling the `encode` method on a string returns a `bytes` Python object, which is a sequence of integers between 0 and 255 (inclusive). Python prints `bytes` objects as strings, except that it prepends a `b` to it, and it prints bytes that do not map to ASCII characters using a backslash followed by the letter `x` and two hexadecimal digits (e.g., `\xc3` and `\xa1`):[6]

```
>>> s.encode('utf8')
b'\xc3\xa1baco'
>>> s.encode('latin1')
b'\xe1baco'
```

Note that UTF-8 requires two bytes to encode the character á, and Latin-1 requires only one (because of its extension to the ASCII standard with new characters that fit within a byte).

If you encode a string and decode it with the *same encoding*, the content will be preserved:

```
>>> s.encode('utf8').decode('utf8')
'ábaco'
```

[6] Recall that one hexadecimal digit takes values between 0 and 15, and a hexadecimal number of two digits takes values between 0 and 255, similar to a byte. For example, the hexadecimal digit `a` corresponds to the decimal number 10, and the hexadecimal number `a1` corresponds to the decimal number $10 \times 16 + 1 = 161$.

However, if you encode with one encoding and decode with another, the message gets corrupted or the decoding simply fails.

```
>>> s.encode('utf8').decode('latin1')
'Ã¡baco'
>>> s.encode('latin1').decode('utf8')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe1
in position 0: invalid continuation byte
```

To address this, the errors flag can be provided, which tells Python how to handle failures. The default value is strict (problems cause a crash), but other options include ignore (skip problem characters), and replace (replace problem characters with an unknown token).

```
>>> s.encode('latin1').decode('utf8', errors='ignore')
'baco'
>>> s.encode('latin1').decode('utf8', errors='replace')
' baco'
```

An alternative is to find out the correct encoding of the file to be processed using command-line tools such as chardet:[7]

```
$ chardet readme.txt
readme.txt: utf-8 with confidence 0.99
```

B.3 Text Normalization

Being able to compare strings is critical to many NLP applications. For this, we need to be able to compare sequences of code points and determine if they are equivalent. Unicode allows several sequences of code points to render identically, so they *appear* to be the same even though the underlying representation is not. For example, consider the character *á*, which can be represented either directly as that character, or else as *a* combined with an acute accent combining character. This is shown in the following snippet:

```
>>> print(s1)
á
>>> print(s2)
á
>>> s1 == s2
False
>>> import unicodedata
>>> len(s1)
1
>>> unicodedata.name(s1)
'LATIN SMALL LETTER A WITH ACUTE'
>>> len(s2)
2
>>> unicodedata.name(s2[0])
```

[7] https://github.com/chardet/chardet

Table B.3 The four normalization forms in Unicode.

Name	Description
Normalization Form D (NFD)	Canonical decomposition
Normalization Form C (NFC)	Canonical decomposition, followed by canonical composition
Normalization Form KD (NFKD)	Compatibility decomposition
Normalization Form KC (NFKC)	Compatibility decomposition, followed by canonical composition

```
'LATIN SMALL LETTER A'
>>> unicodedata.name(s2[1])
'COMBINING ACUTE ACCENT'
```

B.3.1 Unicode Normalization Forms

Unicode defines two types of equivalence between characters: *canonical equivalence* and *compatibility equivalence*. According to the Unicode Standard Annex 15[8]:

> *Canonical equivalence is a fundamental equivalency between characters or sequences of characters which represent the same abstract character, and which when correctly displayed should always have the same visual appearance and behavior.*
>
> ...
>
> *Compatibility equivalence is a weaker type of equivalence between characters or sequences of characters which represent the same abstract character (or sequence of abstract characters), but which may have distinct visual appearances or behaviors.*

To make use of these types of equivalence, Unicode defines four different forms of normalization, based on composition and decomposition. Essentially, composition can be thought of as replacing sequences of characters that combine with a single composite character that is visually equivalent. Decomposition performs the opposite operation: composite characters are broken down into the combining characters that could be used to create their visual equivalent, with a consistent ordering. These normalization forms are listed in Table B.3.

Normalization Form C (NFC) is the one most commonly used, at least for American, European, and Korean languages.[9] To apply this normalization in Python we can use the `unicodedata` module, as shown below.

[8] https://unicode.org/reports/tr15/

[9] https://www.w3.org/wiki/I18N/CanonicalNormalization

```
>>> import unicodedata
>>> unicodedata.normalize('NFC', s)
'ábaco'
```

Decomposition is useful when we want to strip diacritics from their characters. To do this, we first decompose the characters in a string, then remove all combining characters, and, finally, combine the remaining characters:

```
def remove_diacritics(text):
    s = unicodedata.normalize('NFD', text)
    s = ''.join(c for c in s if unicodedata.combining(c))
    return unicodedata.normalize('NFC', s)
```

Sometimes, depending on our application, canonical equivalence may not be sufficient. For example, Unicode provides several variants of the Greek letter π, e.g., GREEK SMALL LETTER PI, GREEK PI SYMBOL, DOUBLE-STRUCK SMALL PI, MATHEMATICAL BOLD SMALL PI. Sometimes these distinctions are meaningful, but sometimes we just need to know if it is the Greek letter π in any of its variants. In these situations, compatibility equivalence is more appropriate:

```
>>> c1 = unicodedata.lookup('GREEK SMALL LETTER PI')
>>> c2 = unicodedata.lookup('GREEK PI SYMBOL')
>>> c1 == c2
False
>>> nfc1 = unicodedata.normalize('NFC', c1)
>>> nfc2 = unicodedata.normalize('NFC', c2)
>>> nfc1 == nfc2
False
>>> nfkc1 = unicodedata.normalize('NFKC', c1)
>>> nfkc2 = unicodedata.normalize('NFKC', c2)
>>> nfkc1 == nfkc2
True
```

B.3.2 Case-folding

Another string equivalence that is needed by some applications is case insensitive equivalence. For example, when building a classifier over social media texts, we may prefer a case insensitive classifier, as case is inconsistently used in such informal texts. The way this is usually done is by converting all strings to lowercase before comparing them, or using them in downstream components. This process is called *case-folding*. This is sufficient when dealing with ASCII or Latin-1 characters. However, when we consider other writing systems things get more complicated. Unicode's casefolding is similar to transforming text to lowercase, but it adds some additional transformations that make the resulting strings more suitable for case insensitive analyses:

```
>>> s1 = 'groß'
>>> s2 = 'gross'
>>> s1.lower()
'groß'
>>> s1.casefold()
'gross'
```

```
>>> s1.lower() == s2.lower()
False
>>> s1.casefold() == s2.casefold()
True
```

Bibliography

J. L. Ba, J. R. Kiros, and G. E. Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450*.

R. Bellman. 1957. *Dynamic programming*. Princeton University Press.

H.-D. Block. 1962. The perceptron: A model for brain functioning. i. *Reviews of Modern Physics*, 34(1): 123.

T. Bolukbasi, K.-W. Chang, J. Y. Zou, V. Saligrama, and A. T. Kalai. 2016. Man is to computer programmer as woman is to homemaker? debiasing word embeddings. *Advances in neural information processing systems*, 29: 4349–4357.

J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, et al. 2012. Large scale distributed deep networks. In *Proceedings of the Conference on Neural Information Processing Systems*.

J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

D. Dickerson. 2019. How i overcame impostor syndrome after leaving academia. *Nature*. https://www.nature.com/articles/d41586-019-03036-y. DOI: 10.1038/d41586-019-03036-y.

T. G. Dietterich. 2000. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*, pp. 1–15. Springer.

P. Domingos. 2015. *The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World*. Basic Books.

J. Donaldson and A. Scheffler. 2008. *Where's My Mom?* Dial Books.

J. Duchi, E. Hazan, and Y. Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7).

R. Ferrer-i Cancho and B. McCowan. 2009. A law of word meaning in dolphin whistle types. *Entropy*, 11: 688–701.

J. R. Firth. 1957. A synopsis of linguistic theory, 1930-1955. *Studies in linguistic analysis*.

X. Glorot and Y. Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256. JMLR Workshop and Conference Proceedings.

X. Glorot, A. Bordes, and Y. Bengio. 2011. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315–323. JMLR Workshop and Conference Proceedings.

I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. 2016. *Deep learning*, volume 1. MIT press Cambridge.

D. Griffiths. 2008. *Head first statistics*. O'Reilly Germany.

Z. S. Harris. 1954. Distributional structure. *Word*, 10(2-3): 146–162.

K. Hornik. 1991. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2): 251–257.

M. Iyyer, V. Manjunatha, J. Boyd-Graber, and H. Daumé III. 2015. Deep unordered composition rivals syntactic methods for text classification. In *Proceedings of the 53rd annual meeting of the association for computational linguistics and the 7th international joint conference on natural language processing (volume 1: Long papers)*, pp. 1681–1691.

D. Kahneman, O. Sibony, and C. R. Sustein. 2021. *Noise: A Flaw in Human Judgment*. Little, Brown Spark; Hachette Book Group.

D. P. Kingma and J. Ba. 2015. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference for Learning Representations (ICLR)*, pp. 1–13.

K. Knight, 2009. Bayesian inference with tears. https://www.isi.edu/natural-language/people/bayes-with-tears.pdf. Accessed: 2019-10-19.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105.

M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken. 1993. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6): 861–867.

O. Levy, Y. Goldberg, and I. Dagan. 2015. Improving distributional similarity with lessons learned from word embeddings. *Transactions of the Association for Computational Linguistics*, 3: 211–225.

Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

C. D. Manning. 2015. Computational linguistics and deep learning. *Computational Linguistics*, 41(4): 701–707.

G. Marcus and E. Davis. 2019. *Rebooting AI: Building Artificial Intelligence We Can Trust*. Penguin Random House.

T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. 2013. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, eds., *Advances in Neural Information Processing Systems*, volume 26, pp. 3111–3119. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf.

M. Minsky and S. Papert. 1969. Perceptron: an introduction to computational geometry. *The MIT Press, Cambridge, expanded edition*, 19(88): 2.

Y. Nesterov. 1983. A method for unconstrained convex minimization problem with the rate of convergence $O(1/k^2)$. In *Doklady ANSSSR*, volume 269, pp. 543–547.

A. B. Novikoff. 1963. On convergence proofs for perceptrons. Technical report, Stanford Research Institute.

N. Qian. 1999. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1): 145–151.

F. Rosenblatt. 1958. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6): 386.

H. Schütze, C. D. Manning, and P. Raghavan. 2008. Introduction to information retrieval. In *Proceedings of the international communication of association for computing machinery conference*, p. 260.

R. Sennrich, B. Haddow, and A. Birch. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

D. R. Swanson. 1986. Undiscovered public knowledge. *The Library Quarterly*, pp. 103–118.

M. A. Valenzuela-Escárcega, O. Babur, G. Hahn-Powell, D. Bell, T. Hicks, E. Noriega-Atala, X. Wang, M. Surdeanu, E. Demir, and C. T. Morrison. 2018. Large-scale automated machine reading discovers new cancer driving mechanisms. *Database: The Journal of Biological Databases and Curation*. http://clulab.cs.arizona.edu/papers/escarcega2018.pdf. DOI: 10.1093/database/bay098.

K. Z. Vardakas, G. Tsopanakis, A. Poulopoulou, and M. E. Falagas. 2015. An analysis of factors contributing to pubmed's growth. *Journal of Informetrics*, 9(3): 592–617.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008.

E. Young, 2009. The real wisdom of the crowds. https://www.nationalgeographic.com/science/article/the-real-wisdom-of-the-crowds. Accessed: 2021-06-04.

T. Young, D. Hazarika, S. Poria, and E. Cambria. 2018. Recent trends in deep learning based natural language processing. *IEEE Computational intelligence magazine*, 13(3): 55–75.

M. D. Zeiler. 2012. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.

G. K. Zipf. 1932. Selected studies of the principle of relative frequency in language.

Author's Biography

Your Name

Your Name began life as a small child ...