AN EXPLORATION OF THE WORD2VEC ALGORITHM: CREATING A VECTOR REPRESENTATION OF A LANGUAGE VOCABULARY THAT ENCODES MEANING AND USAGE PATTERNS IN THE VECTOR SPACE STRUCTURE

Thu Anh Le

Thesis Prepared for the Degree of

MASTER OF ARTS

UNIVERSITY OF NORTH TEXAS

May 2016

APPROVED:

William Cherry, Major Professor
Haj Ross, Minor Professor
Lior Fishman, Committee Member
Su Gao, Chairman
of the Mathematics Department
Costas Tsatsoulis, Dean
of the Toulouse Graduate School
Le, Anh Thu. *An exploration of the Word2vec algorithm: creating a vector representation of a language vocabulary that encodes meaning and usage patterns in the vector space structure*. Master of Arts (Mathematics), May 2016, 49 pp., 6 tables, 25 figures, references, 16 titles.

This thesis is an exploration and exposition of a highly efficient shallow neural network algorithm called word2vec, which was developed by T. Mikolov et al. in order to create vector representations of a language vocabulary such that information about the meaning and usage of the vocabulary words is encoded in the vector space structure. Chapter 1 introduces natural language processing, vector representations of language vocabularies, and the word2vec algorithm. Chapter 2 reviews the basic mathematical theory of deterministic convex optimization. Chapter 3 provides background on some concepts from computer science that are used in the word2vec algorithm: Huffman trees, neural networks, and binary cross-entropy. Chapter 4 provides a detailed discussion of the word2vec algorithm itself and includes a discussion of continuous bag of words, skip-gram, hierarchical softmax, and negative sampling. Finally, Chapter 5 explores some applications of vector representations: word categorization, analogy completion, and language translation assistance.
ACKNOWLEDGMENTS

I would first like to thank my advisor, Dr. Cherry, for helping me with this thesis. I would like to thank my roommate, Yvonne Chang, for her very valuable comments on this thesis. I would also love to thank my parents in Vietnam for always being great supporters. Moreover, I would like to thank Tomas Mikolov for his email clarifying the negative sampling algorithm.
# TABLE OF CONTENTS

ACKNOWLEDGMENTS iii

CHAPTER 1 NATURAL LANGUAGE PROCESSING AND VECTOR REPRESENTATIONS OF LANGUAGE VOCABULARIES 1

CHAPTER 2 DETERMINISTIC CONVEX OPTIMIZATION 6

CHAPTER 3 SOME CONCEPTS FROM COMPUTER SCIENCE 12
   3.1. Huffman Trees 12
   3.2. Neural Networks 17
   3.3. Binary Cross-entropy 20
   3.4. Learning the Weights in a Neural Network 23
      3.4.1. Learning the Weights in Theory 23
      3.4.2. Learning the Weights in Practice 25
   3.5. Learning Representations 27

CHAPTER 4 THE WORD2VEC ALGORITHM 29
   4.1. CBOW Versus Skip-Gram : Predicting a Vocabulary Word from Its Context Versus Predicting the Context from a Vocabulary Word 30
      4.1.1. Continuous Bag of Words 30
      4.1.2. Continuous Skip-Gram 33
      4.1.3. Comparison of CBOW and Skip-Gram 34
   4.2. The Output Layer(s) of the Network 34
      4.2.1. Hierarchical Softmax 34
      4.2.2. Negative Sampling 38
      4.2.3. Hierarchical Softmax versus Negative Sampling 40
   4.3. Final Output of word2vec 40

CHAPTER 5 APPLICATIONS 41
CHAPTER 1

NATURAL LANGUAGE PROCESSING AND VECTOR REPRESENTATIONS OF LANGUAGE VOCABULARIES

Natural language processing is a field of computer science that facilitates communications between computers and humans through the use of natural human language. A familiar example of modern natural language processing is Apple’s Speech Interpretation and Recognition Interface (Siri). Siri is a computer program, which was developed by Apple Inc. to understand natural language voice prompts. For example, if we request “find restaurants near me,” Siri will find multiple restaurants near our location. However, “me” and “knee” have somewhat similar pronunciations. Without any understanding of language and context, it could be difficult for a machine to distinguish between the words “me” and “knee”. But with a good statistical understanding of language, Siri can know that “restaurants near me” is a likely language fragment, whereas “restaurants near knee” is very implausible.

We measure the statistics of language using so-called \( n \)-gram statistics. An \( n \)-gram is a sequence of \( n \) contiguous items collected from a given text or speech. When \( n = 1 \), we call it a unigram; for example, “dog” and “cat” are unigrams. When \( n = 2 \), we call it a bigram or digram; for example, “good horse” and “beautiful woman” are bigrams. When \( n = 3 \), we call it a trigram; for example, “a good book” is a trigram, etc. The \( n \)-gram model is a probabilistic language model which can be used to predict words in a sequence when given the preceding words. Computers can gain a basic statistical understanding of a language by analyzing a large volume of text written in the language. By doing this, they can determine the frequencies with which all the words are used. Theoretically, they could then learn all the relative frequencies of all possible \( n \)-grams. However, as \( n \) gets larger, we run into a problem called the curse of dimensionality. The data requirements necessary to store and work with complete language statistics quickly become unmanageable.

Another area of natural language processing is automated language translation. Machine translation is the use of computer programs to translate texts from one language to
another without losing the meaning of the original texts. Since trying to hand code all the
grammars and idioms of various languages is complicated, word-to-word translation is used,
which is fairly easy, but does not result in natural sounding text. However, after the word-
to-word translation, the text can be improved by using the knowledge of only one language.
For example, consider the sentence,

“This is a white house;”

and suppose we want to translate it to French. With the word-to-word translation, we will
have:

“Ceci est une blanche maison.”

Then someone with only a knowledge of French would know it should be,

“Ceci est une maison blanche”

instead. Modern language translation starts with the crude word-to-word translation then
tries to improve the crude translation based on statistical knowledge of the target language.
In the example, looking at just bigrams, one knows that “maison blanche” is a relatively
common French bigram, whereas “blanche maison” is not. Thus, the computer can correct
the bad translation by a simple permutation of the words.

In theory, one would like to have a complete statistical understanding of $n$-gram
frequencies so that one would have complete knowledge of all the conditional probabilities
for the succeeding words in a text based on the preceding words in the text. There are two
practical problems with this approach. First, the curse of dimensionality, as we mentioned,
means it is impossible to have enough memory to store that volume of relative statistical
data. Second, consider the two sentences:

“The cat is running in the bedroom;” and

“A dog is walking in the room.”

We see that these pairs of words are in the same context and semantically similar: the, a;
cat, dog; running, walking; and bedroom, room. This means that we can interchange them
to get:
“The dog is running in the bedroom;” or
“A cat is walking in the room;” or
“The dog is walking in the bedroom.”

Even with a very huge text corpus, we will not see every reasonable sentence that can be made in a language. We want the computer to learn from analyzing the training texts that if it sees the first two “dog” and “cat” sentences as well as other uses of “dog,” “cat,” “room,” “kitchen” and “bathroom,” then the latter three are also reasonable sentences.

So, rather than try to compute and store a complete joint probability distribution representing how words are used in a language, we try to develop an algorithm to estimate or predict word order probabilities. Neural networks are a type of algorithm modeled after the biology of the human brain that allow computers to deduce patterns in data. A neural network consists of an input layer, intermediate hidden layers, and an output layer. Neural networks were introduced early in the development of natural language processing as a way around the curse of dimensionality. Each word in a language vocabulary was assigned a code to be used as the input to the neural network. The network was trained to estimate the joint probability distribution of the language from which one could compute approximate $n$-gram statistics. Originally, natural language processing used symbols to represent words. For example, the word “cat” might have been represented as, say Id537, and the word “dog” might have been represented by Id789. However, in this method, the symbols were chosen arbitrarily and did not represent the relationships between the words. One can think of the intermediate layers of the neural networks as vectors. Feeding the different words of the language in as inputs results in a vector associated to each word in the language vocabulary. Thus, we associate to each word a vector in a $d$-dimensional vector space. Initially, these vectors were not of intrinsic interest but were only an intermediate tool in computing the $n$-gram statistics. However, researchers noticed that if one displayed these intermediate vectors, then words with similar meanings (or similar grammatical roles) clustered together. Figure 1.1 illustrates this idea. Here we have taken vector representations for English words and projected them to two dimensions. Notice that “man,” “woman,” “girl,” and “boy” are
similar types of words and that their associated vectors are clustered together.

![Figure 1.1. Word Relationships](image)

Mikolov and his team at Google Inc. wanted to focus on the creation of these vector representations. They came up with an algorithm, which is called word2vec, that could be trained on huge data and produce vectors of larger dimension (300 to 500 dimensional) than was practical using prior models and in a reasonable amount of time. In addition to the clustering phenomenon, they also observed that not only were the vectors representing “king” and “queen” close to each other, but also that the displacement vector between “king” and “man” was similar to the displacement vector between “queen” and “woman,” which is illustrated in Figure 1.2. That led them to discover that if they subtracted the vector representing “man” from the vector representing “king” and then added the vector representing “woman,” then the nearest vector representative was the vector representing “queen.”

The fact that the vector space structure of these representation vectors encodes some meaning of the language means that these vectors themselves might be useful in natural language processing, independent of the $n$-gram statistics. In particular, the Mikolov team demonstrated how these vectors could be useful for word categorization [8], for automated completion and exploration of analogies [6], and to help in extending language translation dictionaries [7].
The purpose of my thesis is to explore the work of T. Mikolov et al. [7], [8], and [6], which describes an algorithm called word2vec. In my thesis, I will discuss vector representations of language vocabularies, the word2vec algorithm, and some of its applications to natural language processing. Chapter 2 will introduce some mathematical theory related to the convergence of the word2vec algorithm. Chapter 3 will explain concepts from computer science used in word2vec such as Huffman trees, neural networks, and binary cross-entropy. Chapter 4 will discuss the word2vec algorithm itself and includes a discussion of continuous bag of words, skip-gram, hierarchical softmax, and negative sampling. Finally, Chapter 5 will explore some applications of word2vec.
In this chapter, we will review the basic mathematical theory of deterministic convex optimization, following [2, §5.2]. This theory underlies why one can expect the word2vec algorithm to converge.

The goal of this chapter is to find the global minimum for a convex function. First consider the one-dimensional case. Let \( C(z) \) be a convex function of a real variable \( z \). The basic idea is that if \( C'(z) \) is negative then the minimum lies to the right of \( z \), and if \( C'(z) \) is positive then the minimum lies to the left of \( z \), as illustrated in Figure 2.1. If now \( z \) indicates a point in \( \mathbb{R}^d \), then \( -\nabla C(z) \) points in a direction where \( C \) is decreasing fastest, so \( -\nabla C(z) \) points roughly toward the location of the minimum. But, this only tells us which direction to move to find the minimum. This is referred to as gradient descent.

The hard part is deciding how far to move. Figure 2.2 illustrates that if we move with too large of a step, then we might move past the minimum point, and if we are unlucky, we could wind up in an infinite cycle that never converges to the minimum. Figure 2.2 also shows that if move with too small of a step, then we can appear to converge in such a way that \( C \) is decreasing at each step, but we nevertheless never get near the minimum. Choosing

\[
\begin{align*}
C'(z) < 0 & \quad \text{at some point} \\
C'(z) = 0 & \quad \text{at the minimum} \\
C'(z) > 0 & \quad \text{at a point to the right}
\end{align*}
\]

**Figure 2.1.** Convex function
a reasonable step-size is referred to as “step-size selection,” which is what we will explain in this chapter.

In the general theory of non-linear optimization, one also needs to worry that even after choosing a good step-size, one might converge to a local minimum that is not a global minimum, as illustrated in Figure 2.3. There are various techniques to deal with this problem, for example, the introduction of a stochastic term in the step-size as discussed in [2, §5.3]. However, since we will only be concerned with convex functions, which have a unique minimum point, we will not go into those details here.
Our first proposition will tell us that if we are not yet at the minimum, then there is some step-size such that our function will definitely decrease.

**Proposition 2.1 (Step-size Existence [2, Prop. 5.2.1]).** Assume \( C : \mathbb{R}^d \rightarrow \mathbb{R} \) has continuous second partial derivatives. Let \( \nabla C \) be the gradient of \( C \). Let \( z \in \mathbb{R}^d \) be such that \( \nabla C(z) \neq 0 \). Then, there exists \( \eta > 0 \) such that \( C(z - \eta \nabla C(z)) < C(z) \).

**Remark.** We denote the function to be minimized by \( C \) because we think of it as a “cost” to be minimized. The constant \( \eta \) is called the step-size. We already know we should move in the opposite direction of the gradient, but this tells us how large of a step we should take as a multiple of the size of the gradient in order that the function definitely decrease.

**Proof.** Let \( H \) be Hessian of \( C \), which is the \( d \times d \)-matrix of the second partial derivatives of \( C \). Then Taylor’s theorem with error term says there is some \( \tilde{z} \) lying on the line segment from \( z \) to \( z - \eta \nabla C(z) \) such that

\[
C(z - \eta \nabla C(z)) = C(z) + \nabla C(z) \cdot (-\eta \nabla C(z)) + \frac{1}{2}(-\eta \nabla C(z))^T H(\tilde{z})(-\eta \nabla C(z))
\]

\[
= C(z) - \eta \|\nabla C(z)\|^2 + \frac{\eta^2}{2}(\nabla C(z))^T H(\tilde{z})\nabla C(z). \quad (1)
\]

The second term represents a definite decrease, so we need to choose \( \eta \) small enough that the third term does not cancel this decrease. Because the entries of \( H \) are continuous and therefore bounded on the line segment connecting \( z \) and \( z - \nabla C(z) \), we know there is a constant \( K \) such that provided we choose \( \eta < 1 \), each entry of \( H(\tilde{z}) \) will be bounded above by \( K \). This means

\[
|((\nabla C(z))^T H(\tilde{z})\nabla C(z))| \leq dK \|\nabla C(z)\|^2.
\]

Thus, as long as \( \eta < \min \{1, \frac{2}{dK}\} \), then the function goes down. \( \square \)

Proposition 2.1 tells us that we can select a sufficiently small step-size so that our function will definitely decrease and that if we move by that amount, we will definitely get closer to the minimum. It even gives us an estimate for how large the step-size can be, in terms of the entries in the Hessian of \( C \) and the dimension \( d \). However, we saw in Figure 2.2...
that if we choose the step-size too small, then we might also not approach the minimum.

We now proceed to show that there is an interval of possible step-sizes that are neither too large nor too small.

**Definition 2.2.** Let $0 < \alpha < \beta < 1$. An $\eta > 0$ is called a *properly chosen step-size* relative to $\alpha$ and $\beta$ at the point $z \in \mathbb{R}^d$ if

$$C(z - \eta \nabla C(z)) \leq C(z) - \alpha \eta \|\nabla C(z)\|^2$$

and

$$\nabla C(z - \eta \nabla C(z)) \cdot (-\nabla C(z)) \geq -\beta \|\nabla C(z)\|^2.$$  

**Theorem 2.3** (Existence of Properly Chosen Step-size [2, Th. 5.2.2]). Assume $C$ has continuous second partial derivatives and is bounded below on $\mathbb{R}^d$. Given $z \in \mathbb{R}^d$, then there exists $0 < \eta_{\min} < \eta_{\max}$ such that all $\eta \in (\eta_{\min}, \eta_{\max})$ are properly chosen step-sizes relative to $\alpha$ and $\beta$ at the point $z$.

**Proof.** If $\nabla C(z) = 0$, there is nothing to prove, so assume that $\nabla C(z) \neq 0$. Let $L$ be a lower-bound for $C$. Let $K$ be as in the proof of Proposition 2.1. If

$$\eta < \min \left\{ 1, \frac{2(1 - \alpha)}{dK} \right\},$$

then it follows from (1) that

$$L \leq C(z - \eta \nabla C(z)) \leq C(z) - \alpha \eta \|\nabla C(z)\|^2,$$

and so (2) is satisfied for all sufficiently small $\eta$. Since $\|\nabla C(z)\|^2 > 0$, the right-hand side of (2) tends to $-\infty$ as $\eta \to \infty$. Hence, there is some $\eta^* > 0$ such that

$$C(z - \eta^* \nabla C(z)) = C(z) - \alpha \eta^* \|\nabla C(z)\|^2.$$  

By the Mean Value Theorem, $\exists \eta^{**} \in (0, \eta^*)$ such that

$$C(z - \eta^* \nabla C(z)) - C(z) = \nabla C(z - \eta^{**} \nabla C(z)) \cdot (-\eta^* \nabla C(z))$$

$$= -\eta^* \cdot \nabla C(z - \eta^{**} \nabla C(z)) \cdot \nabla C(z).$$

9
By (4) and (5), we have
\[-\alpha \eta^* \|\nabla C(z)\|^2 = -\eta^* \nabla C(z - \eta^{**} \nabla C(z)) \cdot \nabla C(z) .\]
Hence,
\[\nabla C(z - \eta^{**} \nabla C(z)) \cdot (-\nabla C(z)) = -\alpha \|\nabla C(z)\|^2 \geq -\beta \|\nabla C(z)\|^2\]
since \(\alpha < \beta\). By continuity, this last inequality continues to hold with \(\eta^{**}\) replaced by any \(\eta\) in an interval around \(\eta^{**}\). \(\square\)

Unfortunately, the proof of Theorem 2.3 does not provide an estimate for the interval \((\eta_{\min}, \eta_{\max})\). However, the fact that there is a whole interval of good step-sizes gives one confidence that in practice one can choose step-sizes that will result in convergence to the minimum.

We conclude this chapter with a theoretical result that says we can choose a sequence of step-sizes that will result in convergence to the minimum.

**Theorem 2.4 (Descent Convergence [2, Th. 5.2.3]).** Assume \(C\) has continuous second partial derivatives and is bounded below on \(\mathbb{R}^d\). Let \(0 < \alpha < \beta < 1\). Inductively define a sequence \(z(1), z(2), \ldots z(t), \ldots\) such that for each \(t\),
\[z(t + 1) = z(t) - \eta(t) \nabla C(z(t)),\]
where \(\eta(t)\) is a properly chosen step-size relative to \(\alpha\) and \(\beta\) at the point \(z(t)\). Then either \(|\nabla C(z(t))| \rightarrow 0\) when \(t \rightarrow \infty\) or \(z(t)\) is unbounded.

**Proof.** Observe that
\[C(z(n)) - C(z(0)) = \sum_{t=0}^{n-1} [C(z(t + 1)) - C(z(t))] \leq \sum_{t=0}^{n-1} -\alpha \eta(t) ||\nabla C(z(t))||^2 ,\]
where the inequality on the right is (2). Since the left-hand side is bounded below as \(n \rightarrow \infty\),
\[\sum_{t=0}^{\infty} \eta(t) ||\nabla C(z(t))||^2 < +\infty,\]
which implies
\[\lim_{t \rightarrow \infty} \eta(t) ||\nabla C(z(t))||^2 \rightarrow 0.\]
If \( z(t) \) is bounded, then by the continuity of the second partial derivatives of \( C \) and the Mean Value Theorem applied to \( \nabla C \), \( \exists K \) such that

\[
||\nabla C(z(t + 1)) - \nabla C(z(t))|| \leq K||z(t + 1) - z(t)|| = K\eta(t)||\nabla C(z(t))||.
\]

Now,

\[
[\nabla C(z(t + 1)) - \nabla C(z(t))] \cdot (-\nabla C(z(t))) = \nabla C(z(t + 1)) \cdot (-\nabla C(z(t))) + ||\nabla C(z(t))||^2
\]

\[
\geq (1 - \beta)||\nabla C(z(t))||^2
\]

by (3). But by Cauchy-Schwarz, we have

\[
(1 - \beta)||\nabla C(z(t))||^2 \leq ||[\nabla C(z(t + 1)) - \nabla C(z(t))] \cdot (-\nabla C(z(t)))||
\]

\[
\leq ||[\nabla C(z(t + 1)) - \nabla C(z(t))]| | \cdot ||\nabla C(z(t))||
\]

\[
\leq K\eta(t)||\nabla C(z(t))||^2 \to 0.
\]

So, \( (1 - \beta)||\nabla C(z(t))||^2 \to 0 \). So, \( ||\nabla C(z(t))|| \to 0 \) since \( \beta < 1 \).
CHAPTER 3

SOME CONCEPTS FROM COMPUTER SCIENCE

In this chapter, we will discuss some computer science concepts used in word2vec. First, we will talk about Huffman trees, which will be used when word2vec is run with the hierarchical softmax option. Second, word2vec is a neural network, so we will provide a basic introduction to neural networks. Third, we will discuss the so-called binomial cross entropy function, which is the most appropriate objective function to use in the word2vec neural network. Here we will make a connection to the general theory of convex optimization discussed in the previous chapter and discuss in detail how neural networks are trained.

3.1. Huffman Trees

Definition 3.1. A graph (resp. directed graph) $G = (V, E)$ consists of two sets $V$ and $E$. The set $V$, called the vertex set, is a finite non-empty set. The set $E$, called the edge set, is a set of pairs (resp. ordered pairs) of elements from $V$. We also often refer to the vertices as the nodes of the graph.

For example in Figure 3.1, $a_1, a_2, a_3, a_4, a_5, j_1, j_2, j_3, j_4$ are vertices or nodes and the edges are $(a_1, j_1), (a_1, j_2), (a_2, j_1), (a_2, j_2), (a_3, j_2), (a_3, j_3), (a_4, j_3), (a_4, j_4), (a_5, j_3), (a_5, j_4)$.

Definition 3.2. A tree is a directed graph such that $|E| = |V| - 1$ and such that every vertex is contained at least one edge.

![Figure 3.1. A Graph](image_url)
Definition 3.3. If \((v, w)\) is an edge of a tree, we call \(v\) a parent of \(w\) and \(w\) a child of \(v\).

Definition 3.4. A root node of a tree is a node without parents.

Definition 3.5. If \(v\) and \(w\) are nodes of a tree, then \(v\) is called an ancestor of \(w\) if 
\[ \exists(v, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n), (v_n, w) \in E. \]
The set of edges \((v, v_1), \ldots, (v_n, w)\) is called a directed path from \(v\) to \(w\).

Definition 3.6. A parent node of a tree is a node that has at least one child.

Definition 3.7. A leaf node of a tree is a node without children.

Definition 3.8. A binary tree is a tree in which nodes have at most two children.

Definition 3.9. A Huffman tree is a binary tree with a unique root node such that every node except the root node has exactly one parent, and such that edges are labeled with “0” or “1” so that each edge leaving a node has a different label.

Proposition 3.10. In a Huffman tree, there is a unique path from the root node to each leaf node.

Proof. The path is determined by ending with the leaf node and then going backward to each unique parent until the root node is reached.

As a consequence, each leaf node can be assigned a unique binary code by reading the labels on the edges of the path from the root to the leaf.

Figure 3.2 is one example of Huffman tree. This example is taken from a paragraph of Dr. Seuss [12]. Here the leaf nodes of our Huffman tree are those words which appear in the text more than once. The code associated to the frequent word “not” is “10,” whereas the code associated to the less frequent word “with” is “1100.” Huffman trees were first used for data compression [15].

We will describe an algorithm so that given a vocabulary and frequency for each word in the vocabulary, we create a Huffman tree with each word in the vocabulary as a leaf node and so that the binary codes for more frequent words are shorter than the binary codes for
Figure 3.2. A Huffman tree for the words in a paragraph from *Green Eggs and Ham* [12].

less frequent words. We will describe the algorithm to create a Huffman tree in two stages. First we will describe the initialization of the algorithm. We let $N$ be the number of words in our vocabulary. We will have $N$ leaf nodes, and we let pos1 be an index to the leaf nodes. We will also create $N - 1$ parent nodes, and pos2 and $i$ will be indices into the parent nodes. The index pos2 always points to a parent node which itself does not yet have a parent. If its frequency is less than $\infty$, then its frequency is the sum of the frequencies of the leaf nodes below it. Its frequency is never greater than parents created after it. The index $i$ points to the next parent node to be attached to the tree.

Figure 3.3 shows the flow chart to initialize our algorithm. Thus, at the end of our initialization, pos1 will point to the leaf node for the lowest frequency word and pos2 will point to the first parent node. Low frequency nodes should appear toward the bottom of the tree, so they get parents first.

Figure 3.4 shows the flow chart for the loop that inserts the edges into the Huffman tree. We start from the last word, which has the lowest frequency. As long as pos1 $\geq 0$, it
Input the text and determine the frequency of each word. Let $N$ be the number of words in the vocabulary.

For each word in the vocabulary, create a leaf node with the word and its frequency.

Sort these leaf nodes in descending order based on the frequencies. These are indexed from 0 to $N-1$. The lowest frequency node will be at the end and the biggest one will be at the beginning.

Create $N-1$ parent nodes. These are indexed from $N$ to $2N-2$. Set their initial frequencies to be $\infty$.

Let $pos1=N-1$, $pos2=N$.

**Figure 3.3. Initialization of the Huffman tree algorithm**

always points to a leaf node that has not yet been attached to the tree and which has the lowest frequency among all the leaf nodes that are still unattached. When $pos1 \geq 0$, we check whether the frequency of the word at $pos1$ is less than or equal the frequency of the parent node at $pos2$. If so, we set $min1$ to the leaf node in $pos1$ indicating that that node will be the left child of our new parent node and then set $pos1$ to the next unattached word. If not, we will use the parent node at $pos2$ as our left child. We again check whether the new $pos1$ is greater than or equal 0. If not, $min2$ will be set to $pos2$ indicating the right child of the new parent node will be the parent node at $pos2$ and then we move $pos2$ to the next parent node. If so, we will check whether the frequency of $pos1$ is greater than or equal the frequency of $pos2$. If so, $min2$ is set to $pos1$ and $pos1$ will move to the next unattached word. The new parent node is the node pointed to by $i$. Its “0” child is the node pointed to by $min1$ and its “1” child is the node pointed by $min2$. We continue running the loop until there are no more unattached nodes.

We will explain later how the Huffman tree fits into word2vec. The point is that
In the actual word2vec code, the frequencies above are compared with a strict inequality. I chose to use ≤ instead, but this does not lead to any essential difference.
common words have few ancestors and short codes, whereas infrequent words have more ancestors and longer codes.

3.2. Neural Networks

Neural networks [14], or artificial neural networks, are statistical learning models inspired by the biology of the human brain. Our brain contains billions of nerve cells called neurons which receive signals from dendrites—the cell’s inputs—and send out electrical information through axons—the cell’s outputs. Neurons do not work alone but are densely interconnected in a complex and parallel way. This web of connections allows us to think, feel, and communicate. A neural network [16] is created similarly to our brain as a series of interconnected neurons. In computer science, a neural network consists of groups of artificial neurons grouped into various layers. The first layer of neurons is referred to as the input layer, and these neurons are activated or not depending on the input to the network. The final layer of neurons in the network is called the output layer and produces the output of the network. In between the input layer and output layer, there can be some additional number of layers of neurons, referred to as hidden layers, which are activated or not based on the outputs of the previous layer, and the output of each hidden layer is passed on as inputs to the next layer.

There are two important types of artificial neurons: percentrons and sigmoid neurons.

Perceptron: In the human brain, biologists believe that a neuron is either activated or not. The perceptron is an artificial neuron that replicates this binary behavior. It works by taking several binary inputs \(x_1, x_2, \ldots, x_n\) acting like the dendrites attached to a real neuron and produces a single binary output similar to whether the axon of a real neuron fires or not. To compute the output, Rosenblatt [10] formulated a simple rule using weights \(w_1, w_2, \ldots, w_n\), which are real numbers. The output is determined from \(\sum w_jx_j\) by the rule:

\[
\text{output} = \begin{cases} 
0, & \text{if } \sum w_jx_j \leq \text{threshold} \\
1, & \text{if } \sum w_jx_j > \text{threshold}
\end{cases}
\]

for some threshold value.
The fact that the output of a perceptron can only be 0 or 1 has a disadvantage: small changes in the weights do not change the output. Therefore, we will discuss another type of artificial neuron called a sigmoid neuron, which is better adapted to machine learning.

**Sigmoid neurons** are similar to perceptrons but improved, so small changes in the weights cause small changes in the output. This allows the network to learn by slowly adjusting the weights in order to improve the output. Sigmoid neurons have inputs: \( x_1, x_2, \ldots, x_n \) which can take any value from 0 to 1 (for perceptrons, these inputs can only be either 0 or 1), and each input has weights: \( w_1, w_2, \ldots, w_n \), which are again real numbers as in the case of perceptrons. Also, the output is in the range \([0,1]\). We use \( \sigma \) to denote the sigmoid function, which is defined as

\[
\sigma(z) = \frac{1}{1 + e^{-z}} \quad \text{where} \quad z = \sum w_j x_j. \tag{6}
\]

Sigmoid neurons are similar to perceptrons, in that when \( z \) is large, \( e^{-z} \approx 0 \), so \( \sigma(z) \approx 1 \); and when \( z \) is very negative, \( e^{-z} \approx +\infty \), so \( \sigma(z) \approx 0 \). This is approximately the same behavior as perceptrons for \( z \) very large or very negative. But for \( z \) near 0, the output of a sigmoid neuron lies between 0 and 1. This is illustrated by the graph of the sigmoid function in Figure 3.5. Since \( \sigma \) is smooth, small changes in the weights, \( \Delta w_j \), will result in small changes in the output, \( \Delta \text{output} \), from the neuron. The first order approximation of

![Figure 3.5. Sigmoid function](image)
\[ \Delta \text{output} \approx \sum_j \frac{\partial \text{output}}{\partial w_j} \Delta w_j, \tag{7} \]

where the sum is over all the weights.

In a typical neural network as illustrated in Figure 3.6, neurons are organized into layers: input, hidden, and output. Working with a neural network typically proceeds in three phases: the training phase, the testing phase, and the application phase. During the training phase, data with expected outputs will be input to the neural network and the weights of the network will be learned so as to best approximate the known desired outputs. After finishing the training data, independent input data with known expected outputs will be used to test whether the network works well without adjusting the weights. If the network
passes the testing phase, then it is ready to be applied to real data where the outputs are not known in advance.

3.3. Binary Cross-entropy

Entropy is a concept that originated in physics and measures how orderly or chaotic a system is. Large entropy means the system is chaotic or random, and small entropy indicates the system is orderly, highly structured, or has a lot of symmetry. Entropy is also used to measure information content [13]. For example, if we have a binary bit string consisting of 0’s and 1’s, then the Shannon entropy of the stream is given by

\[ E(p) = -[p \log p + (1 - p) \log(1 - p)] \]

where \( p \) is the probability that a bit in the stream is 0, so that \( 1 - p \) is then the probability that a bit in the stream is 1.

Figure 3.7 illustrates the graph of the entropy function, and we can see that entropy is maximized when \( p = 1/2 \), indicating completely random data. Shannon entropy is a quite useful concept in the theory of data compression in that the larger the entropy of a set of data is, the less it can be compressed. For example, instead of completely random data, suppose that \( p \) is near 1, so that it ends up being much more likely that a 0 will be in the data set than a 1. That might mean that you would want to re-code strings so that strings
with a lot of 0’s, which occur frequently, get short codes and strings with a lot of 1’s, which occur rarely, get longer codes. For example, consider the encoding illustrated in Table 3.8. Although the codes for inputs containing many 1’s are much longer, because they occur rarely, when \( p \) is large, the short codes for inputs containing many 0’s help compress the data. In this way, one can also view entropy as a measure of how much space is required to represent data after it is compressed.

What can happen in practice though is that one does not know the true probability \( p \) that a digit in the text will be zero, but rather one only has an estimate of the true probability. We will let \( q \) denote the estimated probability. We then consider something called the cross entropy or binary cross entropy. The binary cross entropy function is defined as

\[
C(q) = - [p \log q + (1 - p) \log(1 - q)].
\]

As a function of \( q \), the graph of \( C(q) \) is illustrated in Figure 3.9. We can see that the sum in brackets is negative because both of the logarithms are of numbers from 0 to 1, and there is a negative sign in the front of the sum. Therefore, \( C(q) \) is non-negative. You can see that \( C(q) \) is minimized when \( q = p \), in which case we recover the entropy for the true probability \( p \). In general, the cross-entropy measures how good a data compression scheme designed for

<table>
<thead>
<tr>
<th>Input Data</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>10</td>
</tr>
<tr>
<td>010</td>
<td>11</td>
</tr>
<tr>
<td>011</td>
<td>111</td>
</tr>
<tr>
<td>100</td>
<td>110</td>
</tr>
<tr>
<td>101</td>
<td>1110</td>
</tr>
<tr>
<td>110</td>
<td>1111</td>
</tr>
<tr>
<td>111</td>
<td>11111</td>
</tr>
</tbody>
</table>

Table 3.8. Encoding Illustration
data where the estimated probability of a 0 is $q$ works on data where the true probability of a 0 is $p$. If the estimated probability matches the true probability, then we recover data compression which matches the entropy, which is the best possible result. But if we estimate the probability incorrectly, the $C(q)$ is larger than the entropy of the incoming data stream, which expresses the fact that our data compression scheme is not compressing the data as efficiently as possible.

None of the above discussion is particularly relevant for our application. For us, we will use the binary cross entropy function as a way to measure error. Rather than thinking of probabilities, we will consider a desired output $a \in [0,1]$ and an obtained output $y$, also in the range $[0,1]$. We use

$$C(y) = -[a \log y + (1 - a) \log(1 - y)]$$

as a measure of how far our output $y$ differs from our desired output $a$. The interpretation in terms of cross-entropy will not be important for us. Rather, we choose this measure because of some convenient properties of its graph. Observe that

$$C'(y) = -\left( \frac{a}{y} - \frac{1-a}{1-y} \right) = -\frac{a - y}{y(1 - y)}$$
and

\[ C''(y) = -\left[ -\frac{a}{y^2} - \frac{1-a}{(1-y)^2} \right] = \frac{a}{y^2} + \frac{1-a}{(1-y)^2} > 0. \]

We thus see that \( C \) has a unique global minimum at \( y = a \) and that \( C(y) \) is strictly convex. Moreover, the derivative formula above leads to some convenient cancellation when we combine \( C \) with the sigmoid function \( \sigma \). Namely, by the chain rule,

\[ \frac{\partial}{\partial z} C(\sigma(z)) = -\frac{a - \sigma(z)}{\sigma(z)(1 - \sigma(z))} \sigma'(z). \]

Using the definition of the sigmoid function, we have

\[ \sigma(z) = \frac{1}{1 + e^{-z}}, \]

and then

\[ \sigma'(z) = \frac{e^{-z}}{(1 - e^{-z})^2} = \frac{1}{1 + e^{-z}} \frac{e^{-z}}{1 - e^{-z}} = \sigma(z)(1 - \sigma(z)). \]

Thus,

\[ \frac{\partial}{\partial z} C(\sigma(z)) = -\frac{a - \sigma(z)}{\sigma(z)(1 - \sigma(z))} \sigma(z)(1 - \sigma(z))] = -[a - \sigma(z)], \tag{8} \]

and

\[ \frac{\partial^2}{\partial z^2} C(\sigma(z)) = \sigma'(z) = \sigma(z)(1 - \sigma(z)) > 0. \tag{9} \]

The very simple derivative formula (8) allows quick training of our network and simple programming. The second derivative formula (9) shows that \( C(\sigma(z)) \) is a convex function of \( z \). Viewed another way, the derivative of \( \sigma \) is quite small when \( |z| \) is large, and so the appearance of cancelling terms in the formula for \( C'(y) \), which corresponds to the steepness of the graph of \( C(y) \) when \( y \) is near 0 or 1, prevents a “slow-down” effect that would otherwise be caused by the sigmoid function.

3.4. Learning the Weights in a Neural Network

3.4.1. Learning the Weights in Theory

Suppose you have a one-layer neural network which takes \( m \) input values \( x_1, x_2, \ldots, x_m \) to \( n \) output values \( y_1, y_2, \ldots, y_n \) using \( mn \) weight values \( w_{1,1}, \ldots, w_{1,n}, \ldots, w_{m,n} \). If the training data has \( \ell \) input lines, as in Table 3.10, the neural network computes the output values
$y_{\ell,j}$ in terms of the inputs $x_{l,i}$ and the weights $w_{i,j}$ with the formula

$$y_{\ell,j} = \sigma\left(\sum_i w_{i,j} x_{l,i}\right),$$

(10)

where $\sigma$ is a non-linear function which we will take as the sigmoid function. Suppose we have known desired outputs $a_1, a_2, \ldots, a_n$, and we want to learn weights so that the computed output will be close or equal to the desired outputs. In Section 3.3, we mentioned we will use the binary cross entropy cost function to measure errors. Thus, we have the cost function

$$C(w_{1,1}, w_{1,2}, \ldots, w_{1,n}, \ldots, w_{m,1}, \ldots, w_{m,n}) = \sum_\ell \sum_j -[a_{\ell,j} \log(y_{\ell,j} + (1 - a_{\ell,j}) \log(1 - y_{\ell,j})].$$

We can see that $C$ is a non-negative function. It is convex by (9) and therefore has a unique minimum. The network is not well-trained if $C$ is large. In terms of optimization, the $x$'s and $a$'s are fixed and $w$'s are the variables. So the aim for our training algorithm is to find a set of weights $w_{i,j}$ so that the cost function $C$ is minimized.

The way we train a one-layer neural network in theory is we initialize the weights $w_{i,j}$ to random values. We then compute all the outputs according to equation (10), and we then use gradient descent as discussed in Chapter 2 to adjust the weights to improve the cost. Our function was chosen so that it will have a simple gradient after applying the chain rule, namely

$$\frac{\partial C}{\partial w_{i,j}} = \sum_\ell \frac{\partial C}{\partial y_{\ell,j}} \frac{\partial y_{\ell,j}}{\partial w_{i,j}} = \sum_\ell [a_{\ell,j} - y_{\ell,j}] x_{l,i}.$$

(11)
As we learned in Chapter 2, to move toward the minimum value of $C$, we should adjust the weights $w_{i,j}$ by an amount.

$$
\Delta w_{i,j} = \eta \sum_\ell (a_{\ell,j} - y_{\ell,j}) x_{\ell,i},\tag{12}
$$

where $\eta$ is the step-size discussed in Chapter 2. Proposition 2.2 and Theorem 2.3 proved that we can always choose a proper step-size $\eta$ so that we can find a sequence of weights converging toward a solution that minimizes $C$.

3.4.2. Learning the Weights in Practice

We know in theory that there exists a proper step-size that will allow the weights in our simple one layer neural network to converge toward the minimum of the cost function $C$. But in practice, it is not so easy to compute the proper step-size. In practice, we initialize the step-size $\eta$ to an arbitrary value and decrease it at a regular rate as we proceed through our iteration. At first, the step-size might be too big, and our initial adjustments of the weights may not improve the cost function. But, as the step-size moves into the interval of properly chosen step-sizes, the weights will be adjusted in such a way that the cost decreases on each iteration. If the step-size is decreased too rapidly, we might end up in a situation where our weights appear to converge, even though they are not converging toward the minimum, as in Figure 2.3. The hope is that if the step-size decrease is gradual enough, then we will have gotten our weights close to their minimum values before the step-size becomes too small.

When working with the abstract theory of neural networks, we imagine processing the entire training file, computing the cost function based on the entire training file and then repeating this during each iteration of the gradient descent optimization algorithm. In practice, we iterate the gradient descent algorithm and read through the training data simultaneously. That is, rather than considering the cost function of the entire training data

$$
C = \sum_\ell \sum_j -[a_{\ell,j} \log(y_{\ell,j}) + (1 - a_{\ell,j}) \log(1 - y_{\ell,j})],
$$

we consider instead just the cost function from the current input-output pair in the training
\[ C_\ell = \sum_j -[a_{\ell,j} \log(y_{\ell,j}) + (1 - a_{\ell,j}) \log(1 - y_{\ell,j})]. \]

So,
\[ \Delta w_{i,j} = \eta (a_{\ell,j} - y_{\ell,j}) x_{\ell,i}. \quad (13) \]

Thus, rather than updating the weights after passing through the whole training data, we update the weights with each new input-output pair in the training file. This comes with the trade-off that the cost function we are trying to minimize now changes as we move through the training data. Let us examine what this means if we encounter inconsistent training data, as in Table 3.11. In the table on the left, inconsistent data occur non-randomly in the training file while in the table on the right, inconsistent data occur randomly in the training file. If we process the whole input file at once either way, the weights will adjust so that the computed output is near 1/2. But if we process one line at a time, for data in the right-hand table, we will end up with weights that give output near 1/2, while for data in the left-hand table, we will end up with weights that give output near 1. If the pattern on the left persists, we will not get good convergence of our network as we continue to train.

<table>
<thead>
<tr>
<th>(x_1)</th>
<th>(x_2)</th>
<th>output</th>
<th>(x_1)</th>
<th>(x_2)</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.11. Inconsistent data
In sum, the network will only converge reasonably under the assumption that any randomly selected part of the training file is representative of the full set of training data.

3.5. Learning Representations

Prior to the 1980’s, neural networks were regarded as we have discussed them so far: weights are learned so that the networks compute outputs from given inputs based on some training data. The state of the intermediate hidden layers were not thought to be of interest. In an influential paper [11], Rumelhart, Hinton, and Williams discussed a multi-layer neural network used to compute relationships among individuals in a family tree. The input layer of their network consisted of one neuron for each individual in the family tree. The first hidden layer after the input layer consisted of six neurons. After training the network, one could input an individual by activating one of the neurons in the input layer. This would then result in various levels of activation of the six neurons in the first hidden layer. Rumelhart, Hinton, and Williams realized that the activation states of these six neurons could be thought of as associating a six-dimensional vector to each of the individuals in the family tree. This led to a slight change in perspective that would have significant consequences. One then realized that one does not need the original input layer of the network to denote the individuals one wants to input into the network. One can encode each individual as a six dimensional vector, thus only needing six input neurons rather than one for each possible individual. One might originally view these initial vector representations for the individuals as fixed. But notice that the weights \( w_{i,j} \) and the inputs \( x_{\ell,i} \) occur in equation (10) in a completely symmetric way. That means we can also view the \( x_{\ell,i} \) as variable rather than fixed and deduce that we can also lower the cost by adjusting the inputs \( x_{\ell,i} \) according to the formula

\[
\Delta x_{\ell,i} = \eta \sum_j (a_{\ell,j} - y_{\ell,j}) w_{i,j},
\]

which parallels formula (13).

By the time the work of Bengio et al. [1] came along in the early 2000’s, this perspective was well-entrenched, and their neural probabilistic language model was designed so that the neural network simultaneously learned its weights and the input representations.
As discussed in Chapter 1, researchers discovered that these learned input representations already carried significant information about the semantic and syntactic properties of the words in the language and were therefore of independent interest. Word2vec is a simple neural network designed to produce good vector representations for the language vocabulary, that we will explore in detail in the next chapter.
At its heart, word2vec [7], [6], and [8], is a simple neural network with three layers: input, hidden, and output. The basic goal of word2vec is to read in a large volume of text in one particular language and embed each vocabulary word as a vector in a vector space in such a way that the mathematical operation of vector addition has some connection to the meanings of the words. The output of word2vec as a program is a list of vectors, but the output of the neural network within word2vec will be different, which we will discuss later in this chapter.

As discussed in Chapter 3, a typical neural network uses a gradient descent algorithm to adjust the weights that determine how each layer transforms into the next. Most neural networks have weights in multiple layers and use a technique called back-propagation [9] to adjust the weights. The word2vec network is unusually simple and only has weights from the hidden to output layer [5], so back-propagation is not needed. In a typical neural network, the inputs in the training data come from the real world and are fixed. Unlike most neural networks, the input vectors in word2vec are initialized to random vectors and adjusted together with the weights when the training data is processed. Thus, word2vec’s input vectors are treated much like the weights, as we discussed in Section 3.5.

Word2vec is not an example of “deep learning.” A deep neural network has many layers. Multi-layer neural networks take longer to train on the same set of data than shallow networks. The word2vec philosophy is to use fast simple algorithms and train good-sized vectors on really huge data sets. Simple models trained on huge data seem to end up knowing more than complex models trained on smaller data. Since the network models in word2vec are so simple, they have a minimal amount of non-linearity in them. That may be why the linear structure of the vector representatives produced by the word2vec algorithm has so much meaning. However, Mikolov et al. [8, §3] observed that the vector representatives produced by even highly non-linear models seem to converge to representations with linear
Word2vec can be run using one of two models: continuous bag of words or continuous skip-gram. In the first section of this chapter, we will explain these two models and how they affect the transition from the input layer to the hidden layer of the network. From the hidden layer to the output layer, the user can choose one of two algorithms or both: hierarchical softmax and negative sampling. If the user chooses to simultaneously use both hierarchical softmax and negative sampling, then there are two distinct output layers to the neural network. We will explain these two algorithms in the second section of this chapter.

4.1. CBOW Versus Skip-Gram: Predicting a Vocabulary Word from Its Context Versus Predicting the Context from a Vocabulary Word

These two models describe the two ways word2vec can train the word vectors. The philosophy is that in continuous bag of words (or CBOW) we know something about a word only from its context, while in skip-gram, we know something about the context of a word, knowing just the word itself. This concept was hypothesized by Harris in 1954 as, “words in the same context tend to have the same semantic meanings.” This hypothesis is known as the Distributional Hypothesis [3].

4.1.1. Continuous Bag of Words

CBOW, which is an acronym for continuous bag of words, works under the principle that if I know words that frequently occur nearby a given word, then I know something about the word in the middle. The idea is the following: suppose that I have a text, consisting of words $t_1, t_2, t_3, \ldots$ Now consider the $i^{\text{th}}$ word $t_i$. Let $s$ denote a window size and then consider $t_{i-s}, t_{i-s+1}, \ldots, t_{i-1}, t_{i+1}, \ldots, t_{i+s}$. This collection of words, ignoring the order they come in, is the “bag of words” associated to the $i^{\text{th}}$ word $t_i$. This approach is called continuous bag of words because the bag changes continuously as we read through the text by dropping two words out of the bag ($t_{i-s}$ and $t_{i+s}$) and adding two words into the bag ($t_i$ and $t_{i+s+1}$).

In word2vec, we associate a $d$-dimensional vector to each vocabulary word. Initially, these vectors are initialized simply to be random vectors whose components are random
numbers in the interval \([-\frac{1}{2d}, \frac{1}{2d}]\), but our goal is to adjust them in such a way that the meaning of the words is reflected in the mathematics of the vectors. For example, if we consider the sentence: “The cow jumped over the moon.” If the current word is “over” and the window size is two, then \{cow, jumped, the, moon\} forms the context of the word “over.” In CBOW mode, we will use the sum of the vectors associated to the words in this context to try to predict something about the center word “over,” as we will explain when we get to the output layers. If we let \(v_i\) denote the vector associated to the word \(t_i\), then we can efficiently represent the bag of words associated to \(t_i\) with \(v_{i-s} + \cdots + v_{i-1} + v_{i+1} + \cdots + v_{i+s}\) where \(s\) is the chosen window size. Conceptually, we use the average of the vectors associated to each word in the bag, but word2vec just uses the sum rather than dividing by the size of the bag, since the effect is the same. Thus, in CBOW, the input layer of the network consists of the components of the vectors associated to the words in the bag, and the output of the hidden layer is simply the components of the sum of those vectors.

Figure 4.1 illustrates the computation of the hidden vector in CBOW. From input layer to hidden layer, we sum the vectors from the context words to get the hidden vector. In this case, the context words are “cow,” “jumped,” “the,” and “moon,” and the sum is the hidden vector. The current word is “over,” and that will also be our target word.

Now, we will discuss the flow chart for the overall word2vec training algorithm when run in CBOW mode. We assume that we have already read through our text to determine
the words in the text and the frequency with which each word occurs. We assume we have also associated random vectors to each vocabulary word and that we have built our Huffman tree. We now read through the input text. For each word, we set the target word to the
current word, and we read the context by collecting a certain number of words before and after the current word. We sum the vectors associated to each of the context words to get the hidden vector. We pass the hidden vector and the target word into the hierarchical softmax and negative sampling routines to compute the network’s output layer(s), which we will explain in detail later in this chapter. Those output layer routines adjust their respective internal weights according to (13), and they compute an error vector to be used to adjust the vector representations according to (14):

\[ v_{\text{context word}} = v_{\text{context word}} + \text{error}. \]

Then we decrease the learning rate by

\[ \eta = \eta_{\text{initial}} \cdot \left( \frac{\text{position of current word in the input file}}{\text{number of text words in the input file}} \right). \]

In this way, the learning rate will decrease linearly until it reaches 0 at the end of the input file.

4.1.2. Continuous Skip-Gram

*Skip-gram* is in some sense the reverse of CBOW. It works under the principle that if I know a word, then I can guess what other words are likely to occur nearby. The idea is the following: suppose we have a word \( t_i \) and we want to predict the words surrounding \( t_i \). For example, we have a sentence “_____ over _____.” The current word is “over.” We want to predict something about words that are likely to fit in the blanks around the word “over.” In other words, we want to use the vector associated to the word “over” in order to predict something related to each of the context words: “cow,” “jumped,” “the,” and “moon.” In CBOW, these words are in the “bag.” But in skip-gram, they are missing words that we ask, could we predict these as likely context words knowing only the current word “over?”

In the skip-gram model, we simply transfer the vector for the current input word over to the hidden layer. This is illustrated in Figure 4.3. Figure 4.4 shows the flow chart for Skip-gram mode. We will run a loop until there are no more input words. For the hidden
layer, we will set the hidden vector to be the vector from current word in the input file. Then we run through a loop for context words. Each context word is successively set as the target word for the hierarchical softmax and/or negative sampling algorithms.

4.1.3. Comparison of CBOW and Skip-Gram

In CBOW, one vector represents all the context words in the bag. The one vector is passed to the hierarchical softmax or negative sampling computations, and the same error vector is used to adjust the vectors associated to each of the context words.

In skip-gram, we can see that hierarchical softmax and negative sampling are inside the inner loop, so these routines get run for each context word individually. This makes skip-gram slower than CBOW, by about a factor of three in practice [7, §5.3]. Thus, the skip-gram model is slower, but may produce better vectors with smaller training files, whereas CBOW is faster and may be more appropriate for learning larger training files.

4.2. The Output Layer(s) of the Network

There are two algorithms used as output layers of word2vec’s neural network: hierarchical softmax and negative sampling. Each of those take the hidden vector as an input and create outputs compared against desired outputs depending on a designated target word.

4.2.1. Hierarchical Softmax

In hierarchical softmax, the desired outputs of the neural network are the digits of the Huffman tree code of the target word. A neural network weight is associated with each parent node of the Huffman tree. The hidden vector is dotted with the weight for each node, resulting in a probability distribution over the input words. This distribution is then used to compute the negative log-likelihood of the target word, which is then backpropagated through the network to update the weights.
**Start:** Initialize and start at beginning of input & Initialize the learning rate

Get new word and set $v_{\text{hidden}}$ to the vector associated to that vocabulary word

still have more context words?

Set the target word to the next context word

Hierarchical softmax?

negative sampling?

$e_{\text{current}} += e_{\text{error}}$

**Done:** output vectors

Figure 4.4. Flow chart for Skip-gram
ancestor node of the target word with the hope that this gives the code digit\(^1\) at that point in the tree for the target word. For example, we see the location of the target word in the Huffman tree in Figure 4.5. The ancestor nodes of the target word are shaded. The

weights associated to each ancestor of the target word

Target outputs are the binary digits of the Huffman code for the target word

weights associated with these nodes will be used as the weights for the hierarchical softmax

\(^1\)In the word2vec code, the desired output is one minus the Huffman tree code. It does not matter whether one chooses the Huffman code or one minus the Huffman code as the desired output, as long as the choice is made consistently.
algorithm. The hidden vector is dotted with each of these weights, and we then compare the computed output of the sigmoid function of this dot product with the Huffman tree code for the target word at that point in the tree. This is shown in Figure 4.6.

Figure 4.7 is the flow chart for the hierarchical softmax algorithm. We will start from the leaf node in the Huffman tree for the target word and proceed up its ancestors until we get to the root node. According to the flow chart 4.7, first we check whether the node is at the root. If yes, we are done and return the error vector to the CBOW or skip-gram algorithm. If the node is not at the root, we will continue to get the code from the parent node. The computed output is the sigmoid function of the weight attached to the Huffman node. The computed output is the sigmoid function of the weight attached to the Huffman node.
tree node dotted with the hidden vector. The partial gradient is calculated by subtracting the computed output from the desired code digit. Then we compute the error as in (14) by finding the product of the partial gradient, learning rate and weight vector. We also adjust the weight of the tree node by adding a multiple of the hidden vector as in (13). We continue running the loop until there are no more ancestor nodes.

4.2.2. Negative Sampling

For negative sampling, we associate a weight to every word in our vocabulary. The idea is that if the hidden vector and the target word are related because the hidden vector comes from a word(s) that appear near the target word in the input text, then the weight vector from the target word and the hidden vector should point in a similar direction, and hence have large dot product. Thus, we hope \( \sigma(w_{\text{target}} \cdot v_{\text{hidden}}) \approx 1 \). On the other hand, if we choose a vocabulary word at random, we expect it to be unrelated to the hidden vector, and Mikolov et al. suggested training the weights so that \( \sigma(w_{\text{random}} \cdot v_{\text{hidden}}) \approx 0 \). Figure 4.8 shows how the network output is computed with negative sampling.

![Diagram of negative sampling](image)
Figure 4.9 is the flow chart for negative sampling. First we compute the dot product of the hidden vector with the weight associated with the target word. We then compare that output with 1 and adjust the error vector and the weight for the target word according to formulas (14) and (13). Next we do the same thing for some randomly chosen words, except we adjust based on a target output of 0. Note that the random words are chosen using a probability distribution that is proportional to a power\(^2\) of the unigram frequency of the words.

\(^2\)In the word2vec code, the power is set to \(\frac{3}{4}\).
4.2.3. Hierarchical Softmax versus Negative Sampling

The hierarchical softmax algorithm incorporates some additional knowledge of the language, namely the word frequencies as represented by the Huffman tree. By the structure of the Huffman tree, as frequent words are processed, only a few weights are adjusted. This means that hierarchical softmax might not have enough weights associated to frequent words to get good representation vectors for frequent words. On the other hand, since the same weights are adjusted for the same word each time, it might converge more quickly than the more random negative sampling approach. Mikolov et al. discovered in practice that hierarchical softmax can give inferior results for frequent words but train infrequent words more quickly and with a smaller training file than negative sampling [8, §3].

4.3. Final Output of word2vec

After completing the training, we end up with vector representations of the language’s vocabulary words. The algorithm is set up so that the directions in which the vectors point converge, but not their lengths. In fact, the vectors grow longer the longer the training algorithm is run. Thus, only the directions of the vectors, not their magnitudes, are thought to have meaning. Therefore, we normalize the vectors by dividing by their magnitudes before using them. After doing this, we end up with vectors whose position in $\mathbb{R}^d$ reflects the semantics and syntax of the words.
In this chapter, we will discuss what can be done with the vector representations obtained from the word2vec algorithm. Examples of applications include word categorization, analogies, and language translation assistance.

As we remarked at the end of Chapter 4, before using the output vectors of word2vec, one should normalize them to be unit vectors. In this chapter, we therefore assume that our vectors have been normalized into unit vectors, so we have all unit vectors by the end of the word2vec algorithm. In order to determine whether two unit vectors are close to each other or far apart from each other, we will define the distance between them with the formula

$$\text{dist}(v_1, v_2) = \cos^{-1}(v_1 \cdot v_2).$$

In other words, we measure distance between unit vectors by measuring the angle between them. Thus, vectors pointing in similar directions will have distance near 0, vectors that are close to orthogonal will have distance near $\frac{\pi}{2}$, and vectors that point in nearly opposite directions will have distance near $\pi$.

5.1. Word Categorization

Word categorization is a process by which words are grouped together based on similarities in meaning and usages. For example, “king,” “queen,” “prince,” and “princess” are similar words in one group while “cat,” “dog,” “mouse,” and “bird” are in another group.

Figure 5.1 shows two dimensional projection of the vectors produced by the word2vec algorithm for a few words. We see that there are four groups: \{dog, cat, mouse, bird\}; \{teacher, school, student\}; \{king, queen, prince, princess\}; and \{father, mother, son, daughter\}. Words in each group are semantically similar words. Moreover, Table 5.2 shows the five closest words to the word “father.” According to the table, the five closest words are “son,” “grandfather” “uncle,” “mother,” and “daughter.” Moreover, all these words have
similar usage to the word “father” in English, and thus, the vector representations produced by word2vec group English words together in a meaningful way.

5.2. Analogies

Word analogy is a comparison between two pairs of words based on their meaning—semantic analogy, or their usages—syntactic analogy. For example, room, house; and college, university are semantically similar pairs because a room is a part of a house and a college is a part of a university. On the other hand, car, cars; and bicycle, bicycles are syntactically similar pairs because “cars” is the plural of “car” and “bicycles” is the plural of “bicycle.” We can use the vector representations produced by word2vec to automate the completion of
analogy application of word2vec is to find the fourth word given the first three words in an analogy pair. For example, consider: man, woman; king, _____. We need to find what is to “king” as “woman” is to “man.” The expected answer is “queen.”

We can use the vector representations produced by word2vec to help us complete an analogy. We suppose that $v_1, v_2, v_3$ are the vectors associated to our known words. In order to find the unknown fourth word, we will use vector addition and subtraction to compute

$$v_4 = v_2 - v_1 + v_3.$$ 

We then look for the closest representation vectors to $v_4$. The first five word vectors shown in Table 5.3 are closest to $v_4$. The result shows the closest vector is the vector for “queen,”

<table>
<thead>
<tr>
<th>Word</th>
<th>Distance to $v_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>queen</td>
<td>0.778711</td>
</tr>
<tr>
<td>monarch</td>
<td>0.903369</td>
</tr>
<tr>
<td>princess</td>
<td>0.939436</td>
</tr>
<tr>
<td>prince</td>
<td>1.003051</td>
</tr>
<tr>
<td>kings</td>
<td>1.019626</td>
</tr>
</tbody>
</table>

Table 5.3. Distances between words

which also completes the semantic analogy. One reason we might want a computer to do this task is that we can use this as a way to measure how well a set of vector representations represents the meanings of the words in the language. In fact, Mikolov et al. computed how well a large list of analogies were completed to compare the quality of vector representations produced by different algorithms [6, §4.3].

5.3. Language Translation Assistance

Suppose we want to translate from English to French. We can run the word2vec algorithm on large bodies of text in each language. Suppose also that we know a relatively
small but incomplete translation dictionary from English to French. This will generate vector representations of English and French vocabularies. This is illustrated in Table 5.4, where we suppose that $v_1$ is the vector for “dog” in English and $b_1$ is the vector for “chien” in French. The last line of the table illustrates that the translation of “queen” is missing from our dictionary.

As we saw before, the linear relationship among the vector representations produced by word2vec carries meaning. Thus, we can represent our dictionary as a linear transformation from English vectors to French vectors. This linear transformation can be expected to be determined by a small known dictionary. This automates the creation of a large translation dictionary. The mathematical problem of finding the linear transformation representing the dictionary amounts to finding a matrix $A$ such that

$$Av_j = b_j,$$

for all $j$ in the partial dictionary and $v_j, b_j \in \mathbb{R}^d$.

We regard the $v_j$ and $b_j$ as known and the $d^2$ entries of $A$ as unknown. Equation (15) is thus $nd$ linear equations in the $d^2$ unknown entries of $A$, where $n$ is the number of translations given by our partial dictionary. As we expect the system to be overdetermined, we look for a so-called “least squares” solution.
Definition 5.1. If $v_1, \ldots, v_n$ and $b_1, \ldots, b_n$ are given vectors in $\mathbb{R}^d$, a least-squares solution of equation (15) is a $d \times d$ matrix $\hat{A}$ such that

$$\sum_{j=1}^n ||\hat{A}v_j - b_j||^2 \leq \sum_{j=1}^n ||Av_j - b_j||^2.$$ 

for all $d \times d$ matrices $A$.

One can find $\hat{A}$ by gradient descent as discussed in Chapter 2. But one can also use sophomore linear algebra as follows. We have $Av_j = b_j \Rightarrow (Av_j)^T = (b_j)^T \Rightarrow v_j^T A^T = (b_j)^T$ for $j = 1, \ldots, n$. Let

$$V = \begin{bmatrix} v_1^T \\ v_2^T \\ \vdots \\ v_n^T \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} b_1^T \\ b_2^T \\ \vdots \\ b_n^T \end{bmatrix}.$$ 

Thus, equation (15) can be rewritten as

$$VA^T = B.$$ 

Theorem 5.2. The set of least-squares solutions of $Av_j = b_j$ for $j = 1, \ldots, n$ coincides with the nonempty set of solutions $\hat{A}$ of the normal equation

$$V^TV\hat{A}^T = V^TB,$$

which can be solved by row reduction.

Proof. This can be proven as in [4, §6.5]. □

Now, if $v$ is any vector representing an English word, we can compare $Av$ and determine which French vectors $b$ lie closest. These are potential translations for $v$. This also gives us a way to detect likely errors in large translation tables and provides alternate translations to consider beyond what might already be in a given dictionary or translation table.
APPENDIX

VECTOR REPRESENTATIONS USED IN THIS THESIS AND PRINCIPAL COMPONENT ANALYSIS
There are several implementations of the Word2Vec algorithm available for download. Work on this thesis was done using the C implementation available for download at:

https://code.google.com/archive/p/word2vec/

Also included are vector representations that were trained using a 100-billion-word training file created from the Google News dataset. These vector representations are 300-dimensional and were trained using negative sampling in the skip-gram model. All examples included in this thesis are taken from these pre-trained vectors. Other popular word2vec implementations include a Python implementation:

https://github.com/dav/word2vec

and a Java implementation:

http://deeplearning4j.org/word2vec

Principal Component Analysis is a standard method to help visualize high dimensional data. Two-dimensional pictures illustrating the vector representations used in this thesis, such as Figure 1.2, were created using the first two principal components of subsets of the Google News vectors. See [4, §7.5] for full details on Principal Component Analysis and how principal components are computed. Principal components were computed in this thesis using the PCA implementation in the Armadillo linear algebra library:

http://arma.sourceforge.net/


48
