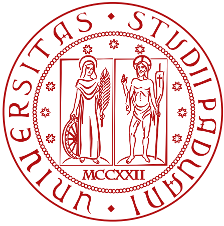
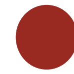


UNIVERSITÀ DEGLI STUDI DI PADOVA



DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

MASTER THESIS IN COMPUTER ENGINEERING

Analysis of a word embedding model: exploring the expressivity of word projections

MASTER CANDIDATE

Matteo Ruta

Student ID 2124456

SUPERVISOR

Prof. Giorgio Satta

University of Padova

ACADEMIC YEAR
2024/2025

Abstract

Representing words in a numerically meaningful way has always been an important goal in Natural Language Processing. In this work, we investigate the capabilities of novel embedding techniques. We present a third-order word embedding model and analyse its performance. To understand the true potential of embeddings in meaning representations of the words, we studied the underlying assumption of previous versions of our model developed in other works, and propose here new theoretical results and practical solutions to improve our embeddings.

Sommario

Rappresentare le parole sotto forma di numeri è sempre stato un obiettivo importante nel Natural Language Processing. In questo lavoro, indaghiamo le capacità di nuove tecniche di embedding. Presentiamo un modello di word embedding di terzo ordine e ne analizziamo le prestazioni. Per comprendere il vero potenziale degli embedding nel codificare il significato delle parole, abbiamo studiato i presupposti alla base delle precedenti versioni del nostro modello (sviluppate in altri lavori) e proponiamo qui nuovi risultati teorici e soluzioni pratiche per migliorare i nostri embedding.

Contents

List of Figures **xi**

List of Tables **xiii**

List of Code Snippets **xvii**

1 Introduction **1**
- 1.1 NLP 2
- 1.2 Useful concepts from ML 2
  - 1.2.1 Loss function 3
  - 1.2.2 Contrastive Learning 4

2 Static Word Embeddings **7**
- 2.1 Definition 7
- 2.2 Word2vec 10

3 Third-order Embeddings **13**
- 3.1 Motivation 13
  - 3.1.1 Second-order embeddings limitations 14
  - 3.1.2 Sense representation techniques 15
- 3.2 Projection-based embedding 16
  - 3.2.1 Contexts separation 19
- 3.3 Previous versions 20
  - 3.3.1 Version A 20
  - 3.3.2 Version B 21

4 Model Specification **23**
- 4.1 Loss description 23
- 4.2 Version B problem: the softplus effect 24

4.3 Version A problem: the "mirror" effect 26
4.4 Final loss analysis 29

5 Implementation 33
5.1 Corpus and preprocessing 34
5.2 Train Loop and Dataset creation 37
5.2.1 Repetition in positive examples 39
5.3 Hyperparameters and versions 43

6 Result Analysis 45
6.1 Loss curves 45
6.2 Evaluation Metrics 47
6.2.1 Word Similarity 47
6.2.2 Comparison with Word2vec 49

7 Conclusions 51

References 55

List of Figures

2.1 Representation of the analogy between two pairs of words into an embedding space. 8

2.2 Example of how similar words w, w' are disposed in word2vec (for only two dimensions). The green vectors represent context embeddings of words in $C(w)$. 12

3.1 Example of the limitations of a second-order word embedding, for just two dimensions. Here, placing the target embedding for *left* is not easy, since it's a word with more than one word sense. Those senses are represented by contexts (the green vectors). . . . 15

3.2 Example of the disposition of a target word w with respect to its contexts u_1, u_2, u_3 using word2vec (in to two dimensions). 17

3.3 Example of the disposition of a target word w with respect to its context words (in green) and noise words (in orange) using a third-order projection-based loss function (in two dimensions). . 18

4.1 Loss contribution in Version A for the positive component. 26

4.2 Loss contribution in Version B for the positive component. 27

4.3 Loss contribution in Version A for the negative component. 28

4.4 Loss contribution in the final loss for the negative part. 30

4.5 Example of how our new loss induces an orientation on the line where w lies. The similar word w' embedding would never end up being like in the figure, because simply by rotating it of 180° the value of the loss decreases, because $d_w(u_1, n_i)$ changes sign, from negative to positive. 31

6.1 Loss function for our model with $\delta = 10$ and standard dataset. . . 46

6.2 Loss function for our model with $\delta = 10$ and no repeated pairs in the dataset. 46

List of Tables

2.1 Example of a term-context matrix for a toy corpus. 9

6.1 Results of word similarity test with different datasets for different versions of the embedding. 49

6.2 Results of word similarity test with different datasets for different versions of the embedding compared to word2vec. 50

List of Code Snippets

5.1 List of stopwords 34
5.2 Subsampling function 36
5.3 Main loop (training) 37
5.4 Function for context extraction for a given word in the corpus . . 38
5.5 Function for the creation of a batch of the dataset 41

Introduction

Representing meaning is a crucial topic in Natural Language Processing (NLP). In order for a text to be processed by a computer, we need to transpose it into some form of mathematical representation. Having additional information about the word's meaning and uses inside that same representation is very useful in many applications.

This work explores the potential of a new type of static word embedding. The representation of words as vectors in a finite-dimensional space constitutes a fundamental step for any modern Language Model (LM), including the increasingly popular Large Language Models (LLMs). All such systems require a basic representation method that already encapsulates general information about the meaning of words. Our approach distinguishes itself from some of the best-known systems discussed in the literature by exploiting the geometric properties of the vectors involved. Specifically, our model seeks to make use of geometric projections of certain vectors onto others in order to describe the meaning of a given word in relation to another.

This work discusses several models previously developed according to this idea, critically examining certain design choices and analysing their main characteristics, including both strengths and limitations. Building on the insights gained during this phase of study, we then proceed to describe the features of our own model, which aims to leverage the same foundations as the earlier ones in a novel way, with the goal of improving performance.

A particular focus will be placed on formulating hypotheses that account for the outcomes obtained through experimentation. In particular, we shall attempt

to understand the reasons behind the results achieved by previous work in this area, addressing possible gaps and constructing new ideas and improvements on the basis of what has already been done.

To achieve this, it is first necessary to provide a brief discussion of some theoretical foundations, which will be useful for readers seeking greater context within this field of study. In this chapter, we shall introduce the scope of Natural Language Processing (NLP), the branch of Artificial Intelligence concerned with the comprehension, interpretation, and processing of human language. In addition, we will highlight certain concepts from Machine Learning (ML) that will serve to support the description and implementation of the various models presented in the following chapters.

1.1 NLP

NLP is a broad branch of Artificial Intelligence. It encompasses numerous tasks, each related to a different use or manipulation of written texts. Among the most important in everyday life, we may mention Automatic Machine Translation, which consists of translating a text from one language into another; Text Classification, namely assigning a label to a given text depending on its content; and, of course, Language Modelling, which enables the generation of text sequences starting from an initial sequence.

At the foundation of many NLP applications lie linguistic theories. For the purposes of this work, we avoid providing a general overview of the main applications of NLP and instead focus on the knowledge necessary for understanding the models of interest to us, namely word embeddings. These were developed to represent words in numerical form, or, more generally, as objects that computers can manipulate. However, a word embedding is not a mere symbol, as we shall see in Chapter 2. It must encapsulate within itself the characteristics of words when placed in a particular context, within a specific sentence.

1.2 Useful concepts from ML

Word Sense Disambiguation (WSD) is one of the best-known tasks in NLP. It consists of determining which word sense is associated with a given word in a given text. By word sense, we mean the meaning that may be attributed

to a particular word. This varies depending on the context in which the word appears. This intuition forms the basis of the *distributional hypothesis*, one of the fundamental principles underlying many NLP applications. First introduced by Harris (Harris, 1954), it states that words occurring in similar contexts tend to have similar meanings. By context, we therefore mean the set of words occurring within a given window centred on a particular word. This word is often referred to as the target word, while the surrounding words within the context are known as context words. On the basis of this idea, a wide range of methods have been developed for representing words in numerical form, including word embeddings, which we shall discuss in Chapter 2.

NLP and ML are two closely related fields of Computer Science. Many NLP systems use ML in order to learn from text some important characteristics of language and words. We focus on just two concepts that will be crucial for our work, providing some context before their use in the next chapters.

1.2.1 Loss function

A loss function is used in mathematical optimisation algorithms to map a set of parameters into a real number, which represents the "cost" of a model with those parameters. The purpose of such functions is to obtain values for those parameters that minimise the value of the loss function, to find the model that solves the task in the "cheapest" way possible. This goal highly depends on how we formulate it in mathematical terms. We could say, indeed, that the formulation of the problem is the loss function itself in the vast majority of ML tasks.

Optimisation algorithms often rely on gradient computation of the loss function with respect to the parameters of the model. However, minimising the loss for a given training dataset may lead the parameters to adapt too much to the data and not generalise well to unseen data. This phenomenon is known as *overfitting*. A standard approach to mitigate this problem is to maintain an unseen portion of data, called a validation dataset, and compute the loss for both datasets after each epoch, which means after a complete pass over all the examples in the dataset. In this way, if the training loss and the validation loss start diverging, we can stop the training and avoid overfitting too much. This method is known as *early stopping*.

In this work, the most important part of the static word embedding models

we'll discuss will be the loss function. In order to be prepared for their analysis, we present two well-known functions that often show up in many different applications.

The sigmoid function is one of the most used functions in ML. Its expression is

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1.1}$$

It maps values from $\mathbb{R}$ into $[0, 1]$. This property is often used to convert any real quantity, which in this context is referred to as *logit*, into a probability. For this reason, the sigmoid function is a very common activation function in the final layers of Deep Learning (DL) neural networks with binary outputs or in logistic regression models.

Another function we'll use very often in future chapters is the softplus.

$$softplus(x) = ln(1 + e^x) \tag{1.2}$$

This function is very similar to the ReLU activation function, which expression is

$$ReLU(x) = max(0, x) \tag{1.3}$$

The main difference between these two functions is that ReLU is not differentiable at zero. This can cause problems for the optimisation algorithm in some specific cases. In our discussions, we'll see that the softplus played a crucial role in our analysis.

1.2.2 Contrastive Learning

Contrastive learning is a way of building and designing models that view the problem as a binary classification task. It's often used when dealing with an unlabelled dataset, like an NLP corpus. This paradigm exploits similarities between groups of examples to learn meaningful data representations. The model learns different information from two classes of examples, called positive and negative. The goal of the model is to predict to which class each example belongs.

Thanks to this implementation of the problem, the model embeds information about the data structure and patterns while trying to predict a label. In word embedding models, this is particularly useful because it helps encode semantic information about words by looking at other words that are similar (positive

examples) or very different (negative examples). We are going to expand this concept in Chapter 2, where we'll present a state-of-the-art static word embedding that leverages this approach. Our model itself is based on the contrastive learning paradigm, even though it leverages the concept in a different way from other well-known embeddings.

A key aspect of this paradigm is dataset creation. As we said, this approach is often used with unlabelled data, where we need to assemble different data points together in order to collect positive and negative examples. In our application, this step is closely related to what we started discussing in Section 1.1 and that we'll expand in Chapter 2: how words relate to their *context* and how we use it, according to the distributional hypothesis, to extract information about word sense in a specific portion of the corpus.

Static Word Embeddings

In this section, we describe the concept of word embedding and its main purpose. We briefly present some historical details behind their development and analyse a state-of-the-art model for word embedding, called word2vec. This model is very important for our work, since the core structure of our system is based on and inspired by it. Also, we'll eventually compare our model's performance with word2vec in Chapter 6, trying to understand whether our model could reach state-of-the-art performance with enough data. For this reason, it is important to introduce the basic details of its implementation. We'll also introduce the notation we are going to use later for our model description.

2.1 Definition

A word embedding is the representation of one or more words. Those representations are organised into a space (usually of many dimensions) using a given method. Many strategies for generating word embeddings have been proposed over the years, but each of them shares the same goal: to keep similar words in a given language "similar" also in the embedding.

The reason behind this choice has a very long history and is related to the idea of the "meaning" of words. Intuitively, we would like our representations to encode some characteristics of words, not just to be "symbols". Of the many possible aspects we can think of about a word, the meaning is certainly the most general and useful. Having a representation of the meaning of a word

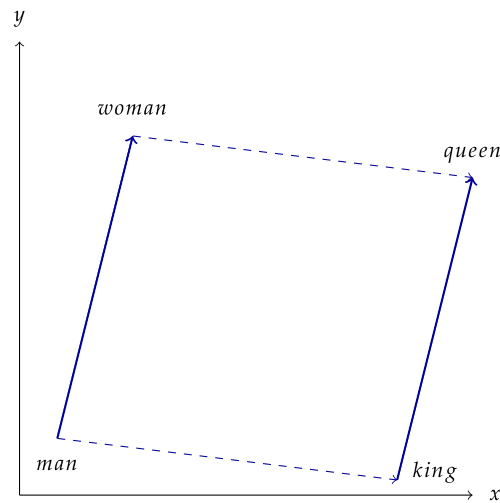


Figure 2.1: Representation of the analogy between two pairs of words into an embedding space.

would allow us to organise and analyse words directly within the encoding, without additional processes. Additionally, we could also perform operations on embeddings that would translate into operations on the meaning of such words. In Figure 2.1, we can see an example of how those operations reflect meaning between words. We can see that the difference between *woman* and *man* in the embedding space is the same vector that transforms *king* into *queen*. We would like to reach an embedding where this property applies to each analogy example.

One of the fundamental intuitions in the creation of these objects (and also in the development of NLP and linguistics itself) concerns how to extract the meaning of a word just by using the text in which that word is used. This can be done by exploiting the distributional hypothesis, which we already introduced in Section 1.1. Thanks to this assumption, we are allowed to think about meaning in a very clever way: just by looking inside a corpus, we can assess whether two words have similar meanings or not, simply by looking at their context. This idea seems incredibly powerful, but it is indeed very challenging to make it work. State-of-the-art word embedding models exploit gradient-based optimisation techniques to obtain a representation as good as possible. It is very important to understand that those embeddings are treated as points in a space. In this space, we can define a measure of the similarity between two points, which shall reflect how similar the represented words are. In many cases, very simple linear algebra tools can be used, such as the cosine similarity. The difficult part of generating

an embedding, then, is not understanding when the representations are well distributed, but moving those representations in the space in a "meaningful" way.

In the literature, many word embeddings have been proposed and studied. We can distinguish these systems into two classes based on the structure of the produced vectors. The first class is the older one, which produces sparse vectors as representations. The simplest method consists of counting, for each target or term word in the corpus, the number of times a given word appears in its context. These values can be represented in a matrix, called a *term-context matrix*, as in Figure 2.1. The representations of each term are the rows of this

	engine	fantasy	sky
car	93	3	13
movie	5	68	43
airplane	78	4	81

Table 2.1: Example of a term-context matrix for a toy corpus.

matrix. However, this kind of representation is often not good enough for real applications. A more sophisticated way of extracting sparse representations consists of using Pointwise Mutual Information (PMI) between a term and each context word. PMI is computed in the following way:

$$PMI(w, c) = log_2 \left(\frac{P(w, c)}{P(w)P(c)} \right) \tag{2.1}$$

where $P(w)$ represents the probability of a word w to be seen in the coprus. At the numerator, we have an estimation of the probability of both term and context words to appear in the same context, while at the denominator, we have the probability of both the term and context to appear together if they weren't related. Therefore, PMI measures if two events are correlated, in the sense that they tend to happen together more frequently, or not. Other variants have been developed on top of this, but they are not interesting for our work.

Sparse vector representations are often too large to handle and too weak to perform correctly in a real application. Since the birth of Deep Learning, interest has been put on another kind of word embedding, called dense embedding. This kind of representation is smaller, but not immediately interpretable as the sparse ones. This, however, is not a problem, since the information is encoded inside

each vector using an optimisation algorithm and a mathematical model (often a neural net) in order to obtain a more representative embedding space.

In the next section, we introduce one of the state-of-the-art methods among the dense word embeddings, called word2vec. This embedding is very important to understand the design process of our models.

2.2 Word2vec

Word2vec is a state-of-the-art static embedding model, introduced in this work (Mikolov et al., 2013). For each word in the training corpus, the model creates two different representations, which consist of vectors of real numbers of 300 dimensions. We'll indicate the two matrices containing all the representations as E_t and E_c. In some cases, we could also write $E = (E_t, E_c)$, referring to a particular embedding with two representations per word. Those representations are called the *target embedding* and the *context embedding*. Word2vec uses the context embedding only during training, while the target embedding is the one used in applications.

The only parameters of the model are, therefore, the vector representations of each word — the matrices E_t, E_c. In order to obtain optimal values, the authors of word2vec developed two different loss functions, called Continuous Bag-Of-Words (CBOW) and Skip-Gram with Negative Sampling (SGNS). We will not describe the first one since it is not useful for our work. The latter, instead, is one of the main inspirations for our models, and therefore it is very important to describe its main features.

The notation we introduce here will be used throughout the entire dissertation. We are going to use w to indicate a target word, or a target embedding of a given word, depending on the situation. Similarly, we are going to use u or $u_i, i \in \mathbb{Z}$ to indicate a context word, or a context embedding of a given word. Whenever these symbols are used inside a formula, they shall be interpreted as vectors. Also, target word representations w are always target embeddings in E_t. Similarly, context word representations u always belong to E_c. This explains the choice of the names of the two embeddings, which are meant to represent words with their respective roles.

SGNS leverages the principle of contrastive learning, briefly introduced in Section 1.2.2. The dataset is therefore extracted from a corpus by collecting

pairs of words, labelled as positive or negative depending on their construction. Models that build examples based on two pairs of words are called *second-order models*. Our models do not belong to this class, but they share many similarities with word2vec. Examples are built in the following way:

- **positive**: for each target word w in the corpus, let $C(w)$ be the context for that specific instance of w. Then, for each $u \in C(w)$, the pair (w, u) is a positive example.

- **negative**: for each target word w in the corpus, let n be a word not in $C(w)$, randomly selected from the corpus. Then the pair (w, n) is a negative example. We refer to n as a *noise word*, and its default representation will be in E_c.

For a given target word w, consider just one positive example (w, u). In the word2vec implementation, for each positive example, a given number $k > 1$ of negative examples is collected. The value of k reported in (Mikolov et al., 2013) is 5. We can define the loss function for a given positive example and each negative example associated with it as follows:

$$L(w, u) = -ln(\sigma(w \cdot u)) - \sum_{i=1}^{k} ln(\sigma(-w \cdot u)) \tag{2.2}$$

The interpretation of this loss function depends on the interpretation of the sigmoid function σ. As we said in chapter 1, the sigmoid function maps the set $\mathbb{R}$ into $[0, 1]$, which is very useful when we want to model probabilities. If we write

$$P(+|w, u) = \sigma(w \cdot u) \tag{2.3}$$

we can see that the model tries to classify examples between positive or negative correctly. The model classifies a generic example as positive depending on the similarity between w and u. In the case of a positive example, we would like this probability to be almost one. By the definition of probability, we have that

$$P(-|w, u) = 1 - P(+|w, u) = \sigma(-w \cdot u) \tag{2.4}$$

which is exactly what appears in the loss function. Using Equation (2.3) and

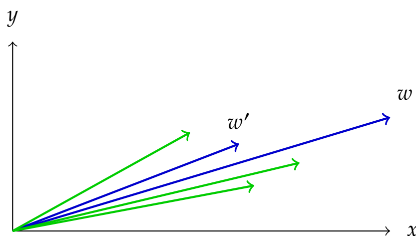


Figure 2.2: Example of how similar words w, w' are disposed in word2vec (for only two dimensions). The green vectors represent context embeddings of words in $C(w)$.

(2.4), we can write

$$L(w,u) = -ln(P(+|w,u)) - \sum_{i=1}^{k} ln(P(-|w,n_i)) = -ln(P(+|w,u)\prod_{i=1}^{k} P(-|w,n_i)) \tag{2.5}$$

We can conclude, then, that minimising this loss for any word w in the corpus is equivalent to maximising the log-likelihood of the model's predictions over the dataset. It is important to remember that we do not care about a good prediction by itself; we aim to obtain a meaningful representation of words through it.

This loss function also has an interesting geometrical interpretation. Let w be a target word with a context $C(w)$. Due to the loss function definition, the target embedding of w will be very similar to each context embedding $u \in C(w)$. Let w' be a word similar to w. By similar, we mean that the context of both w and w' contains mostly the same words (see Section 1.1). Therefore, w' would be very similar to the vast majority of words in $C(w)$ (let us suppose that there exists a $C(w')$ such that $C(w') = C(w)$). But if both w and w' are very similar to words in $C(w)$, then the similarity between the target embeddings of w and w' will also be very high. This result is important because it explains how the loss function is able to capture the main goal of every word embedding — representing similarity between words (and therefore word sense) coherently.

In this work, we are going to investigate this aspect of word embeddings in particular, trying to study a new way to encode word senses.

3

Third-order Embeddings

In this chapter, we discuss another family of word embeddings. Unlike word2vec, which only uses two words in each example (as discussed in Chapter 2), third-order embeddings try to leverage relationships between words in a more sophisticated way. Instead of simply adjusting the similarity between two words, these models use more complex geometrical tools to efficiently manage the different word senses. The increase in complexity should enhance the capabilities of the embedding, resulting in a more meaningful distribution of the vectors.

We'll focus on a particular approach to third-order embeddings, which is the key intuition behind the development of the three models we analyse in this work. In Section 3.1, we'll briefly discuss some limitations of second-order embeddings, such as word2vec, and explain why third-order embeddings could overcome those limitations. Then, in Section 3.2, we'll present the idea at the core of all the third-order embedding models we are going to analyse. Finally, in Section 3.3, we'll describe the two models that serve as the baselines for our final version.

3.1 Motivation

Some of the most popular state-of-the-art static word embeddings are second-order models. What we mean by this expression is that the examples they use at training time to update their parameters consist of just two words. One of the

main examples of these embeddings is word2vec. As we described in Chapter 2, each example processed by word2vec is composed of a target word w and another word from the corpus. Depending on the class of example we are considering, the second word can belong to the context of w or not, but the operations performed inside the loss function for both positive and negative examples always need exactly two words. Very often, those operations rely on some sort of similarity measure between the two word embeddings. A third-order embedding instead adds a level of complexity. Having three words for every example allows the model to manipulate more sophisticated characteristics of the data than just the similarity between words. Since each example provides information about three different vectors, the operations that can be performed on each example need to be complex enough to map those elements into some relevant measure for learning. Defining such a measure is not trivial, and it will probably weigh down the model due to its computational load. Why would such additional complexity and effort be required for an already complex class of models?

3.1.1 Second-order embeddings limitations

Second-order models, but also static word embeddings in general, suffer from a serious issue that affects their efficiency in semantics tasks. Representing a word with many different contexts and therefore many different word senses with a single vector leads to an inefficient optimisation of the embedding space (Camacho-Collados and Pilehvar, 2018).

Let's consider the word *left* as an example to better understand this problem. A typical sentence in which the word *left* may be used is "When you arrive at the station, turn *left* and you'll see the bar". In this sentence, we have a given word sense for *left*, which the model shall extract from the context. But we could also have another sentence including *left*, like "Accidentally, I *left* my phone at home". Here, the word sense of *left* is completely different. Using a static word embedding like word2vec, the representation of the word *left* will be global, in the sense that it will consider all the contexts of all the occurrences of *left* in the corpus. This means that the model doesn't distinguish between two different word senses of the same target word; it just tries to build a representation that fits the most in the embedding space. Remembering what we discussed about how target embedding vectors are placed in Section 2.2, the resulting representation

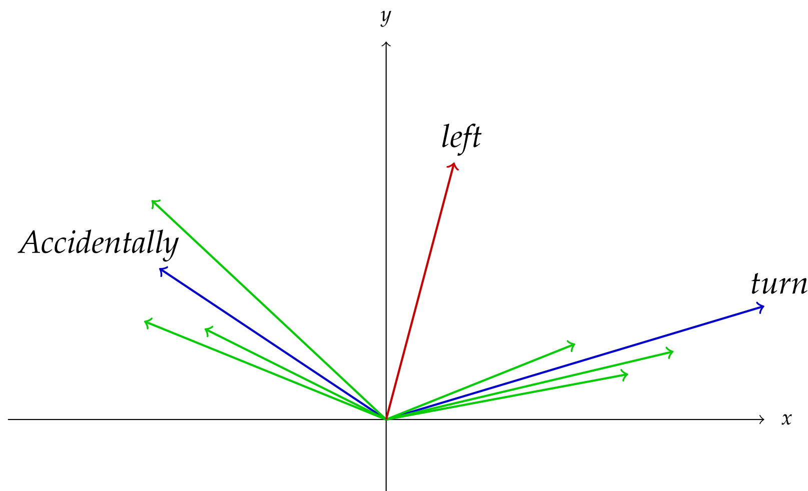


Figure 3.1: Example of the limitations of a second-order word embedding, for just two dimensions. Here, placing the target embedding for *left* is not easy, since it's a word with more than one word sense. Those senses are represented by contexts (the green vectors).

of *left* will be as similar as possible to all the context embeddings from both the first sentence and the second sentence. This has consequences when measuring the similarity between target embeddings of words in different contexts of *left*, like *turn* and *Accidentally*. The situation that could happen is represented in Figure 3.1. In this scenario, the model cannot find a perfect embedding for *left*. The embeddings of the two different contexts are too different, which implies that no single target embedding could be similar to both. This reflects the different semantic roles of the word *left* in different instances of the corpus. This causes the model to perform badly in semantics-related tasks, since increasing the similarity of a word with its context words for one of its word senses may decrease the similarity with other context words for other word senses.

3.1.2 Sense representation techniques

One of the goals of this work is to understand if third-order models may produce embeddings with better semantic understanding. Comparing words not just with respect to their similarity, but with novel metrics built on purpose for balancing different word senses into a single vector representation is something that would solve a very hard problem in a relatively easy way. Many other solutions have been proposed over the years to represent more word senses at once. The vast majority of those increase the amount of representa-

tion needed for each word, depending on that word's senses. Those kinds of embeddings are often called sense representations. Many different methods have been proposed in the literature, either unsupervised, like context-group discrimination (Schütze, 1998), or knowledge-based, which relies on external resources like WordNet (Miller, 1995). All of these methods increase the size of our embeddings, causing the model to require more resources not only during training. It is necessary to say that all of those methods, including static word embeddings, have been outperformed by contextual word embedding systems, where each word embedding is a function of the entire input sequence, allowing for more efficient semantic representation (e.g. polysemy) (Liu et al., 2020). Some of the most popular are ELMo (Peters et al., 2018) and BERT (Devlin et al., 2019), which have been the subject of many works and publications.

In this work, we investigated the capabilities of a third-order model to exploit geometric properties of embeddings to encode more semantic information and word senses. Opting for a static embedding when other state-of-the-art technologies have moved in another direction is still useful, since the majority of contextual word embeddings are obtained by exploiting deep learning networks, like LSTM (Long Short-Term Memory) and Transformers. To sum up, the advantages of a third-order static word embedding over other models could be:

- to get a better encoding of word senses in a global embedding, exploiting more complex relations among word representations;
- to keep the embedding space simple to obtain in terms of computation, without requiring a high-consuming model like a transformer.

3.2 Projection-based embedding

We already mentioned in section 3.1.2 the fact that our model uses geometrical properties of word representations as instruments to compare word senses in a more detailed way than the second-order models do. We now describe the concept at the very core of all three versions of the model we are going to discuss later (both in section 3.3 and Chapter 4). This idea is strictly related to the intuition behind word2vec, so we'll often compare the two scenarios.

Let w be a target word and $u_1, ..., u_n$ some context words for w. Suppose to have a certain word embedding for w in the target embedding space E_t. In

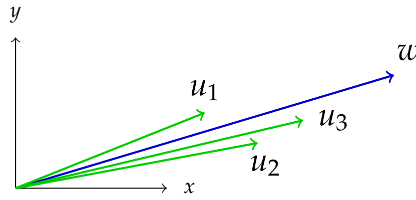


Figure 3.2: Example of the disposition of a target word w with respect to its contexts u_1, u_2, u_3 using word2vec (in to two dimensions).

other static embedding models, like word2vec, what the models aim to do is place the representation of w somewhere in the space such that all the context words' representations are *close* to it. What we mean by this is that the cosine similarity between each context word $u_i, i \in [1, n]$ and w shall be maximised (see section 2.2). What should happen is represented in Figure 3.2. Suppose now that we had more than one context for w. Instead of trying to obtain a representation almost similar to every context word, we exploit the geometric information encoded in the vectors. For a given u_i, consider the projection of the point u_i on the direction of w. We define the component of a vector $u \in \mathbb{R}^d$ with respect to another vector $w \in \mathbb{R}^d$ as the algebraic measure of the oriented segment obtained by projecting the u on the direction of w

$$p_w(u) = \frac{w \cdot u}{\|w\|} = \|u\| \cos(\widehat{wu}) \tag{3.1}$$

It is important to notice that this measure can be positive or negative depending on the angle between w and u. We can compute the distance between the projected points by leveraging the component measures:

$$d_w(u_i, u_j) = p_w(u_i) - p_w(u_j) = \frac{w \cdot u_i - w \cdot u_j}{\|w\|} \tag{3.2}$$

Our idea consists of choosing a direction for w such that for each context $C = \{u_1, ..., u_n\}$ of w, all the projections of the context words in C fall into an interval as small as possible. In other words, we would like to minimise the value $d_w(u_i, u_j)$ for each pair $(u_i, u_j) \in C, i \neq j$. As you can see in Figure 3.3, this architecture allows the model to encode the semantic properties of words in a completely different way. The context vectors $u_1, ..., u_n \in E_c$ are freer to move in their embedding space since the embedding of w can adjust itself better than in second-order embeddings. We would like to stress that

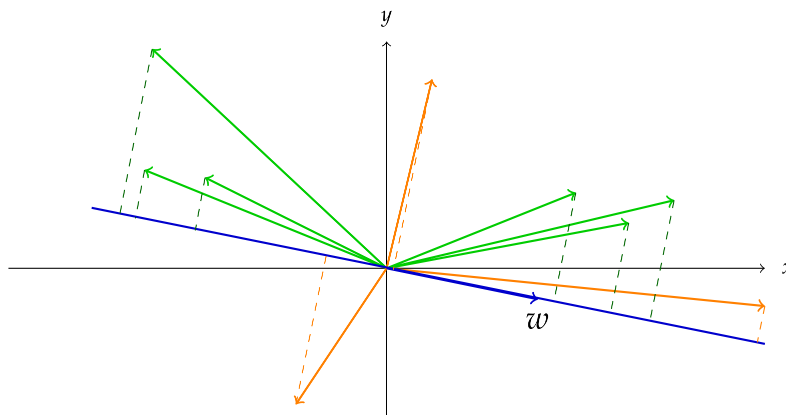


Figure 3.3: Example of the disposition of a target word w with respect to its context words (in green) and noise words (in orange) using a third-order projection-based loss function (in two dimensions).

this solution is a third-order embedding model, since the model needs vectors w, u_1, u_2 to minimise the quantity $d_w(u_1, u_2)$. From now on, we will refer to a triplet $(w, u_1, u_2), w \in E_t, u_1, u_2 \in E_c$ where u_1, u_2 belongs to the *same* context of w as a positive example. We can think of these types of examples like the equivalent of (w, u) pairs in word2vec (see again Section 2.2).

But that's not the whole story. Our model leverages, as word2vec does, the contrastive learning paradigm, which we discussed in Section 1.2.2. So, as we defined positive examples for our new model, we need to define negative examples. This can be done in the following way.

Let w be a target word, and C one of its contexts. Let $u_1 \in C$ (the motivation of the index on u will become clearer later). Let n be another word, not belonging to C, which we'll call a noise word. Referring to Figure 3.3, in order not to mix contexts with noise words, we would like our noise projections to be very far from context projections. In other words, we would like to maximise $d_w(u_1, n)$ for each pair $(u_1, n), u_1 \in C, n \notin C$. The equilibrium we want our embedding to reach is therefore dependent on how those very different vectors appear when projected on a specific direction, which encodes a specific target word. We define our negative examples to be $(w, u_1, n), w \in E_t, u_1, n \in E_c$ where w is a target word, u_1 is a context word, and n is a noise word. We can think of these types of examples like the equivalent of (w, n) pairs in word2vec.

The word n is randomly extracted from the corpus. depending on its frequency. The probability of each word in the corpus to be chosen as noise, given

a certain w and a certain context C of w is defined as

$$P[n = w] = \frac{f_w^{0.75}}{\sum_v f_v^{0.75}} \tag{3.3}$$

3.2.1 Contexts separation

In Figure 3.3 we showed a simplified case that well represents what our models aim to do. Just looking at the figure, we can notice that, for a fixed target word w:

- for any context C of w, all context words $u_i \in C$ represented in E_c, when projected on w, fall into an interval along the direction of w such that each other $u_j \in C$ falls in the same interval
- for any context C of w, all context words $u_i \in C$ represented in E_c, when projected on w, fall into an interval along the direction of w such that each other $n_j \notin C$ randomly selected from the corpus falls outside that interval

This means that our embedding has encoded word senses in its representations in a strong way. Those intervals we identified on the direction of w, which contain all and only the projections for words in a certain context, *identify* those very contexts. We can therefore identify a region in the space E_c which contains all the context words in a context C for a given w. We'll refer to those regions in the future as *corridors*, since they are induced in the space by the intervals on w. Those corridors are defined with respect to each target word. In an ideal case, those corridors converge into affine subspaces of E_c when the interval on w converges into a point (and therefore we have $d_w(u_i, u_j) = 0, u_i, u_j \in C$). Those affine spaces are orthogonal to w.

In this case, given a target word w, let w' be a similar word to w, which means that w and w' could appear very often in the same context. This, due to the distributional hypothesis which we presented in Chapter 1, is equivalent to saying that those two words have similar meanings. Therefore, following the behaviour described in Section 3.2, the direction of w' will end up being almost the same as that of w. This is because the model will try to place w' as close to an orthogonal vector to the corridors as possible. Even if in a real-case scenario those corridors are not subspaces, and therefore the concept of orthogonality is not well defined, for intervals that are small enough, the first could be approximated with the second, leading to a situation where all similar words *in a specific context* want to be orthogonal to the respective corridor. Therefore, words with

similar contexts will end up being similar in the embedding space E_t, as in every word embedding. But this method also guarantees a more robust disposition of context embeddings in E_c, which could help distinguish word senses and perform better in semantic tasks.

3.3 Previous versions

We can now start talking about the implementation of what has been discussed in Section 3.1. In this section, we present two different models, both created by different authors, which shared our same intuition and that became the baseline for our work. For this reason, we'll often refer to them as different *versions* of the same model. The purpose of this dissertation is to analyse those models and compare them with a third version, which we proposed to try to beat the performances of these two. Our final version has been designed in order to solve some issues of the first two, which we'll discuss in Chapter 4. It is important to notice that all three versions share some common characteristics, which we summarise here:

- all three versions are third-order static word embeddings
- all three versions exploit two different embedding spaces, E_t, E_c. As in word2vec, E_c is only used during training (see Section 2.2)
- all three models use the contrastive learning approach. The definition of positive and negative examples is the same as presented in Section 3.2
- all three versions are built to obtain the behaviour described in Section 3.1 or very similar. This is done using a specific loss function, which varies from version to version
- all three versions are trained on the same dataset

We'll refer to the models we're presenting as Version A and Version B. Version A was developed before Version B and was used as a baseline by the authors of Version B as well.

3.3.1 Version A

Version A was first presented in this thesis work (Galvan, 2021) and aims to be an implementation of the concepts presented in section 3.1 as directly as possible. The model is strongly based on the SGNS objective of word2vec (see section 2.2), but with samples made of three words. As in SGNS, the authors of

this model defined the loss function to exploit predicted probabilities for each example to be positive or negative. In other words, the model itself assigns a value in the range $[0, 1]$ to each example, which represents the probability of that example being positive. This probability is defined as

$$P(+|w, u_1, u_2) = 2\sigma(-|d_w(u_1, u_2)|) \tag{3.4}$$

In order for this value to be a probability, it should be true that

$$P(+|w, u_1, u_2) = 1 - P(-|w, u_1, u_2) \tag{3.5}$$

from which we derive the following expression for negative examples

$$P(-|w, u_1, n) = 1 - 2\sigma(-|d_w(u_1, n)|) = 2\sigma(|d_w(u_1, n)|) - 1 \tag{3.6}$$

Those probabilities are used in the loss function to determine how well the model performed on a given example. The definition of the loss function for a given positive example (w, u_1, u_2) and k negative examples $(w, u_1, n_i), i = 1, ..., k$ is

$$L(w, u_1, u_2, n_1, ..., n_k) = -ln(P(+|w, u_1, u_2)) - \sum_{i=1}^{k} ln(P(-|w, u_1, n_i)) \tag{3.7}$$

We could simply refer to this value as L_w for short. The model is trained in order to minimise the average of L_w over the whole dataset. The idea behind this loss function is, exactly as for SGNS, maximising the likelihood of the training dataset. The way we define those probabilities is therefore crucial, and is the main difference between Version A and Version B.

3.3.2 Version B

The second version of this model, presented in (Akman, 2022), made minor adjustments to the loss function with respect to Version A. Except for this change, the two models are identical. The change consists of defining *scores* that represent how well a given example is classified as positive or negative. We present the definition of the scores. The introduction of such scores removes probabilities

from the loss. Those values are defined as

$$sc^{(+)}(w, u_1, u_2) = 2\sigma(softplus(d_w(u_1, u_2))) \tag{3.8}$$

$$sc^{(-)}(w, u_1, n) = 2\sigma(softplus(d_w(u_1, n))) - 1 \tag{3.9}$$

It's easy to see that the relation (3.5) is no longer valid in this scenario. As for Version A, the loss for a given positive example (w, u_1, u_2) and k negative examples $(w, u_1, n_i), i = 1, ..., k$ becomes

$$L(w, u_1, u_2, n_1, ..., n_k) = -ln(sc^{(+)}(w, u_1, u_2)) - \sum_{i=1}^{k} ln(sc^{(-)}(w, u_1, n_i)) \tag{3.10}$$

The major change in this version is, for sure, the introduction of the softplus instead of the module, which led to very different experimental results and theoretical considerations. We presented the softplus function in Chapter 1.

The fact that this loss uses no longer probabilities, but arbitrary scores instead, makes it difficult to compare with word2vec. In this sense, this is an atypical loss function, because it's not SGNS, but it seems like it is. We are not fully aware of the exact intuition of the authors that led them to this formulation of the loss, but we can assume that they view these scores as an approximation of the true probabilities. The authors provided many experimental results supporting their change, but were not able to completely motivate them, leaving it as an open problem. We'll discuss this topic in our analysis in Chapter 4.

4

Model Specification

In this chapter, we are finally able to define the final version of the model. This is the only one out of three that was completely designed and tested by us for this work. The definition comes with the most important discussions on previous versions' distinctions and flaws. This will allow us to justify our choices and explain clearly the reason behind the final formulation of our model, highlighting the major differences between all three versions.

In section 4.1, we report the new loss. In sections 4.2 and 4.3, we analyse the previous version of the model and criticise some aspects of both. For each topic we'll present, we'll propose a solution implementable with a loss function modification. We'll cover how all these changes converge into the final result in section 4.4.

4.1 Loss description

We start by presenting the loss function for our model. We are following the notation introduced in Chapter 2 and expanded in Chapter 3.

Given a target word w, let (w, u_1, u_2) be a positive example and $(w, u_1, n_i), i = 1, \ldots, k$ one of the k negative examples in the dataset associated with that specific positive example. Then, the loss function for those examples is

$$L(w, u_1, u_2, n_1, \ldots, n_k) = |d_w(u_1, u_2)| + \sum_{i=1}^{k} softplus(-d_w(u_1, n_i)) \tag{4.1}$$

The loss is composed of two parts, depending on which type of example it involves. We refer to these as the 'positive' and 'negative' components, depending on the class of examples they correspond to. We could think of those parts as the equivalent of probabilities and scores for Version A and B, respectively, as we showed in section 3.3. Here, the positive component minimises the quantity $d_w(u_1, u_2)$ in absolute value, thereby aggregating the projections of the context words. The negative component, on the other hand, pushes apart the projections of u_1 and n_i. Notice that in the negative component, we are not using the absolute value of the distance: instead, we impose an order along the direction of w, requiring that context words always precede noise words. We will elaborate on this point in the next section.

The main difference from the losses in Version A and Version B is that in our later version, we completely removed probabilities (or even scores, which, as we said in section 3.3, can be considered approximations of probabilities). This choice reflects our decision to radically change the paradigm of our loss. Experiments and results on the previous models often led to inconsistent outcomes: neither we nor the authors of those models were able to fully understand some of the behaviours exhibited by the resulting embeddings. The more we investigated the earlier versions, the more our intuition diverged from the steps followed by those authors.

In the next sections, we will discuss these problems in detail, providing for each one both an introduction and a motivation. Problems are presented and discussed in chronological order of analysis, as we believe this structure will give the reader a clearer understanding of our design process.

4.2 Version B problem: the softplus effect

One of our main goals in the design process was to understand the effects of the softplus introduced in Version B. Experimental measurements confirm that the best-performing model is Version B, not only from a loss perspective. This version makes the validation loss converge better than in Version A and also provides more satisfying results in other measurements (see Chapters 4 and 5). Both the authors of Version B and ourselves believe that these improvements can be attributed to the softplus, even if the reason was not completely clear. We further investigated this aspect, trying to answer the following questions:

- why did the introduction of the softplus lead to better convergence of the loss?
- why did it improve the validation metrics?

These questions do not have a trivial answer. This is because the very purpose of our loss function (which remains the same throughout all versions of the model) does not align well with the implementation presented in Version B. Additionally, as we mentioned in section 3.3, the introduction of the softplus transformed probabilities into scores, in the sense that the equation (3.5) is no longer valid, and the model does not respect the SGNS definition anymore. This note alone was enough to intrigue and motivate us to find a better interpretation of these results. We now clarify why the Version B implementation presents important conceptual errors.

As we previously said, we would like the loss to bring context projections close together. This objective is carried out by the positive component of the loss. In Version A, this idea was translated into a high penalty corresponding to a low value of P^+ for a positive example. In other words, if the model encounters a positive example in the training set and predicts a very low probability for it being positive, the loss becomes very high. In Figure 4.1, we see the contribution of the positive component of the loss for a given example, as a function of $d_w(u, u')$ (the distance between the two projections of u and u' on w).

As we can see, the positive contribution for one positive example in Version A is symmetric with respect to the x-axis. This means that what the model actually minimises is not the distance $d_w(u_1, u_2)$ itself, but its absolute value $|d_w(u_1, u_2)|$.

Equation (3.8) does not enforce the same behaviour. As shown in Figure 4.2, the contribution to the loss depends on the sign of $d_w(u_1, u_2)$. In other words, the order in which we pick the projections matters in the Version B implementation, although this was never required by the model objectives. We argue that this issue penalises the embedding.

However, as we already said, Version B outperformed Version A in both loss and evaluation metrics. One possible reason, which can be seen in Figure 4.2, lies in the fact that the contribution of positive examples to the loss is always negative. As seen in Chapter 1, a loss function should have its minimum value at zero, which represents the optimal state of the parameters for a given training dataset. If the loss is allowed to reach negative values, then the optimisation algorithm does not stop at zero but tries to push the parameters further down. Thus, Version B's loss is not correctly defined. More specifically, the loss values

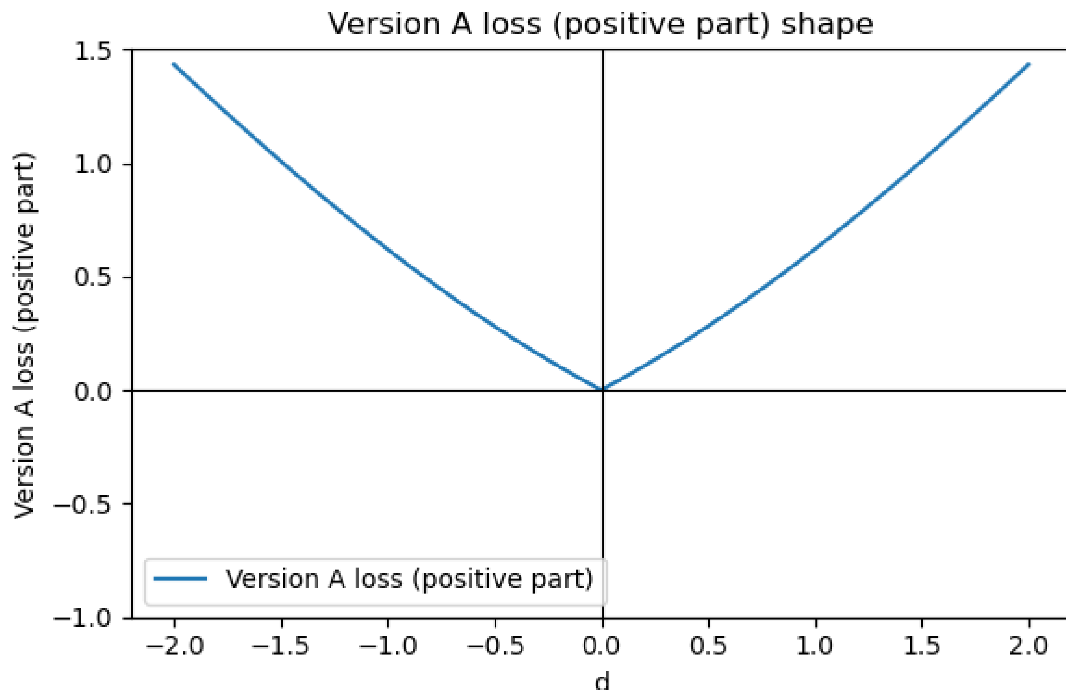


Figure 4.1: Loss contribution in Version A for the positive component.

obtained with Version B will be much lower than those of Version A due to the negative contribution of positive examples. This provides an answer to the first question posed at the beginning of this section. In the end, because of the poor definition of the Version B loss, the two models' losses are not comparable.

Interestingly, other test measurements yield higher results for Version B than for Version A. We will discuss these measurements in Chapter 6. For now, it's sufficient to know that Version B performances were significantly higher than Version A when dealing with lexical semantics tasks.

4.3 Version A problem: the "mirror" effect

Even though Version A seems conceptually very good, a more accurate analysis led us to criticise this approach as well. Here, the problem lies in the negative part of the loss. As reported in Equation (3.9), it is clear that the loss increases when the distance $d_w(u_1, n_i)$ becomes small. In this way, the embeddings are discouraged from distributing themselves such that noise words overlap with the context words of a certain target (in terms of projections).

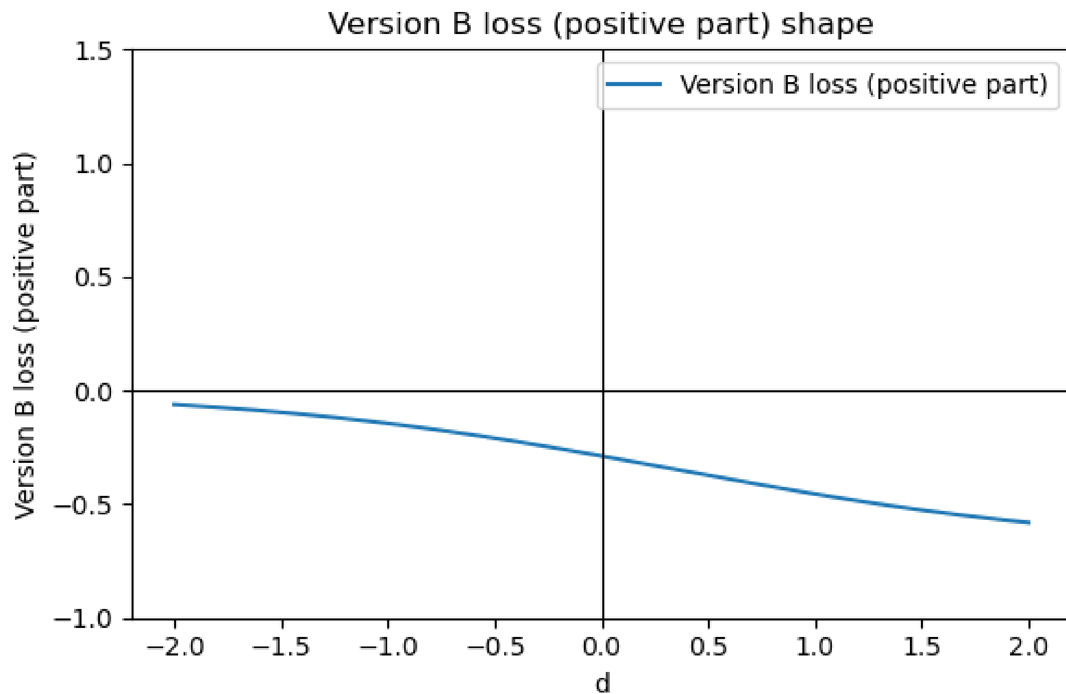


Figure 4.2: Loss contribution in Version B for the positive component.

We already discussed in section 3.2 how, given a target word and its context, the only way for a similar word to position itself in the embedding space is to align with the direction of the target, or at least with a very close one. This direction minimises the difference between projections of the context words on the target (ideally, those projections should converge to zero, though this never happens in practice). At the same time, the same direction should maximise the distances between pairs of noise–context words, as observed in the negative examples of the training dataset. The problem with the Version A implementation is that maximising $|d_w(u_1, n_i)|$ is not sufficient to obtain a good embedding.

As shown in Figure 4.3, the loss contribution is the same for $d_w(u_1, n_i)$ regardless of its sign. This means that the noise projections are pushed away from the context regardless of the side of the line on which they lie relative to the context. This can lead to what we call the "mirror" effect. Since the direction that minimises the loss is unique, a target word with a similar context (and therefore, we assume, “similar” noise words) will lie along it, but nothing in this loss function determines the *orientation* of the vector on that line. More specifically, this means that for our loss, an embedding could be placed with the same orientation as the first target word, or opposite, without any relevant

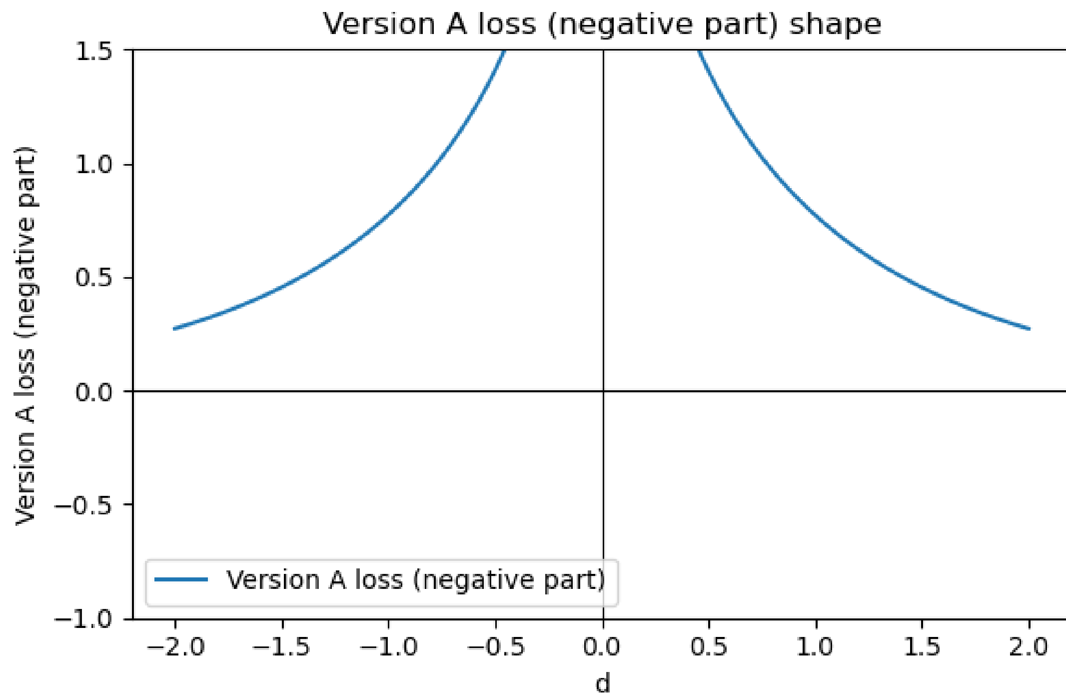


Figure 4.3: Loss contribution in Version A for the negative component.

preferences.

This situation may lead to similar words being encoded correctly — when their angle is between $[-\pi, \pi]$, resulting in high cosine similarity — or very poorly. In fact, if a word that should be similar to a certain target ends up having an embedding that is almost opposite to the first one, the cosine similarity will also be nearly opposite. The correct embedding thus becomes "mirrored," which explains the name we chose.

To solve this problem, we decided to completely change our approach from Versions A and B, and instead tried to integrate what we learned from them into a new loss function built from scratch. Since SGNS does not seem sufficient to train such a third-order model, we abandoned probability modelling and adopted a more "naive" formulation of our objective, as expressed in Equation (4.1).

4.4 Final loss analysis

Given what we have discussed so far, we can now explain the components of our final loss. We designed this loss by addressing all the previously identified issues while paying close attention to performance. More specifically, we built a loss function such that:

- the theoretical intuition behind the model is correctly translated into a meaningful and well-defined loss function,
- the "mirror" effect is handled robustly.

As mentioned in the previous section, we decided to adopt a different approach from earlier versions, leaving behind the probability formulation in favour of a more "naive" implementation. This decision arose because SGNS proved difficult to express in a coherent mathematical way, and since the straightforward formulation (Version A) failed, we chose to redesign the loss differently. At the same time, we avoided exploring many micro-variations of Version A (as the authors of Version B did) because our goal was to obtain something more concretely supported by theory.

Our loss is composed of two parts, as in the previous versions. These two parts are called positive and negative, playing the same role as in Versions A and B. We report here the two components from Equation 4.1 in a more readable way:

$$L_w^{(+)}(u_1, u_2) = |d_w(u_1, u_2)| \tag{4.2}$$

$$L_w^{(-)}(u_1, n_i) = softplus(-d_w(u_1, n_i)) \tag{4.3}$$

Let us now discuss both parts separately. The positive component is more similar to Version A than to Version B. In fact, the loss arranges the embeddings such that the absolute value of $d_w(u_1, u_2)$ is minimised across all target words w, where u_1 and u_2 are words that appear in the same context of w. It is easy to see that when $d_w(u_1, u_2) \to 0$, the loss does the same. Conversely, if $|d_w(u_1, u_2)|$ grows, the loss increases. With this positive part, we intended to solve the softplus issue in Version B, where the implementation was incorrect and produced inconsistent results.

The negative component requires more discussion. Figure 4.4 reports the contribution of a single negative example (w, u_1, n_i) to the loss with respect to $d_w(u_1, n_i)$.

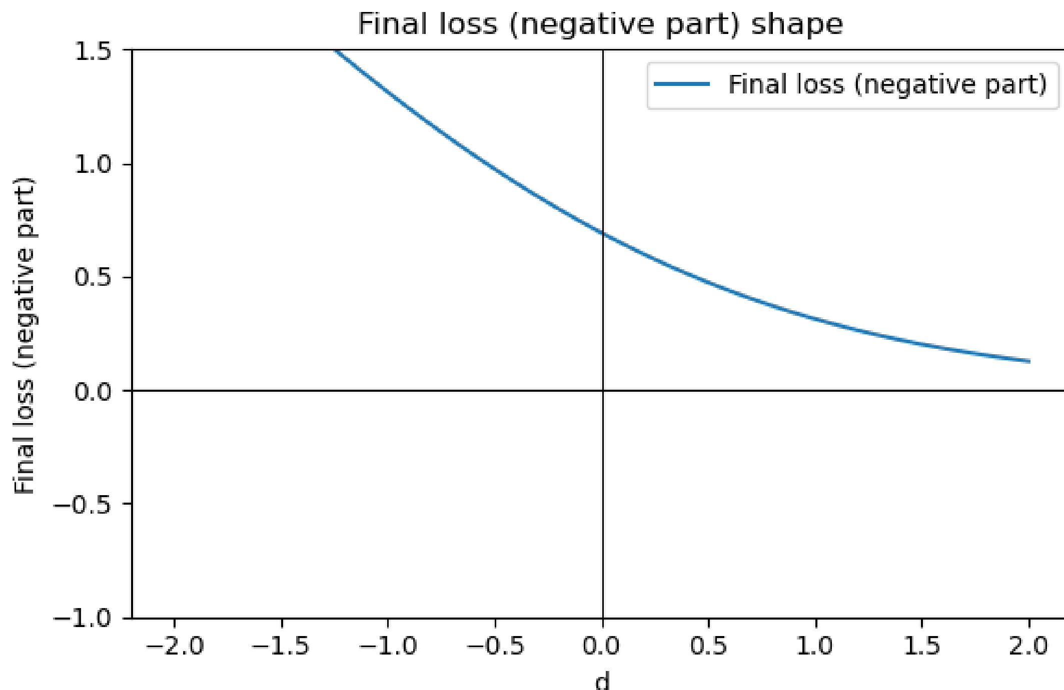


Figure 4.4: Loss contribution in the final loss for the negative part.

Thanks to this implementation, our model penalises a very specific situation. Let w be some target word and (w, u_1, n_i) some negative example for w. The loss function wants to maximise $d_w(u_1, n_i)$, pushing n_i farther from w than u_1 in w direction. Given the three projections of w, u_1, n_i, the following expression is valid (*Chasles' relation*)

$$d_w(u_1, n_i) = d_w(w, n_i) - d_w(w, u_1) \tag{4.4}$$

Then maximising $d_w(u_1, n_i)$ could be done by increasing $d_w(w, n_i)$ - so pushing away the projections of n_i from w - and/or decreasing $d_w(w, u_1)$ - therefore moving projections of u_1 closer to w. This enforces an orientation in the arrangement of embeddings, which helps the model address the "mirror" issue. Given a target word w with a certain direction, a similar word w' cannot be encoded as an almost opposite vector, since in that case the context–noise order would be inverted and the loss would not be minimised. In that case, simply by rotating w' embedding by 180° (to align it with the first target), the loss would be reduced, as only the target embedding changes. Other context words of the second target may slightly deviate from the direction, but since the two words are assumed to

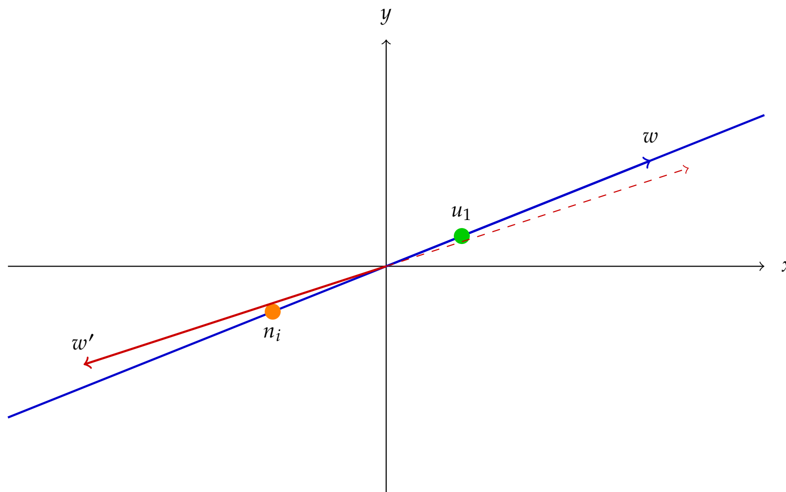


Figure 4.5: Example of how our new loss induces an orientation on the line where w lies. The similar word w' embedding would never end up being like in the figure, because simply by rotating it of 180° the value of the loss decreases, because $d_w(u_1, n_i)$ changes sign, from negative to positive.

be similar, we implicitly assume their optimal directions are nearly the same. As discussed in section 4.3, the only remaining challenge was choosing the correct orientation for the representation of w' (a similar word) — precisely the reason motivating our new loss. Looking at Figure 4.5, we can see that our loss function aims to *clearly divide* noise words projections from w along the direction of w, placing context words projections in the middle. Even when this solution is not feasible, the model tries its best to move all the projections in this certain order, handling the mirror effect more efficiently.

It is worth mentioning the case where the distance $d_w(u_1, n_i)$ for a negative example is close to zero. In older models, like Version A, there was a strong theoretical mechanism to handle this situation. As Figure 4.3 shows, for values close to 0, the contribution of that example diverges to infinity, exactly as expected: the goal was simply to push noise projections away from the context, regardless of direction. By contrast, in our formulation, what matters is pushing the noise projection *before* context projections along the direction of w (see again Figure 4.5). For this reason, our contribution could not have asymptotes (or even local maxima) at 0, as this would make gradient-based optimisation intractable. Instead, our loss progressively moves noise projections away. Although this penalty is less aggressive than in Version A, where, as we just said, the slope at close to zero distances is very high, we preferred to cover the mirror effect

in a more robust way. We thought this tradeoff — sacrificing some optimisation strength in exchange for better handling of the "mirror" effect — could be beneficial for validation metrics and overall performance of the model.

However, in order to find the perfect calibration for this tradeoff, we introduced a new hyperparameter in our loss function. The value of this parameter can affect how high the value of $d_w(u_1, n_i)$ must be to get a loss close to zero. This new parameter, called δ, affects only the negative part of the loss function, which becomes

$$L_w^{(-)}(u_1, n_i) = softplus(-d_w(u_1, n_i) + \delta) \tag{4.5}$$

This parameter shifts the negative loss function part $L_w^{(-)}$ horizontally. When $\delta > 0$, the minimum value of $d_w(u_1, n_i)$ such that $L_w^{(-)}$ is very close to zero grows. Increasing this value may reduce the freedom of the model too much, increasing the loss value and decreasing the performance.

We'll show some results on this variant of the model in Chapter 6. We could see the loss function presented since now as a particular case of this new one, with $\delta = 0$. So we'll consider these two variants as almost equivalent in terms of theoretical discussions. When no value of δ is specified, it's considered to be 0. See also section 5.3 for a more detailed summary of the values of each hyperparameter.

5 Implementation

In this chapter, we expose the details of the implementation of our model, from the dataset creation to the training and validation steps. All the topics discussed here are focused on the practical aspects of this work and could be presented along with code snippets from our source. In section 5.1, we'll describe all the steps of preprocessing we adopted for our corpus. In section 5.2, we uncover the implementation of the training loop, focusing also on the dataset creation part, which deserves a more accurate description. Finally, in section 5.3 we summarise all the hyperparameters involved in our model and report their value.

We wrote our code in Python, mostly relying on Pytorch library ("Pytorch library", n.d.). Pytorch is one of the most popular machine learning frameworks in Python that provides a lot of useful instruments in many fields of artificial intelligence. It handles very important aspects of the computation, like automatic differentiation, efficient dataset structures and defines an appropriate object for matrix operations and feature representation called tensors. Those objects are very similar to NumPy arrays, but they are built to be more efficient with specific hardware like CUDA-capable Nvidia GPU.

We chose this framework due to our personal familiarity with its syntax, but this work could be easily replicable with other very popular open-source libraries like TensorFlow ("Tensorflow library", n.d.).

5.1 Corpus and preprocessing

Our model has been trained on a dataset built using the principles of contrastive learning. In order to perform such a task, it's necessary to choose a starting corpus. Since our work was not only a design task, but also an analysis and comparison study, we maintained the choice of the authors of versions A and B. All those models have been trained on text8 (Mahoney, 2011), a portion of the first 10^8 bytes of English Wikipedia. It contains 17005207 tokens with 253854 types. After all the preprocessing operations are done, we are left with 3084085 tokens for the train split, 340664 tokens for the validation split.

Our preprocessing includes many steps, which we list below:

- punctuation conversion
- low-frequency words and stopwords
- word tokenisation
- subsampling

At the very end of the pipeline, we split the data into train and validation corpus, which will be used to generate datasets for the respective portions. We'll expand each of these steps where needed, adding snippets of code for the relative implementations.

The first step is useful to clean the text (which was previously lower-cased). Each instance of punctuaction is converted into a specific token. Also, the second step is performed for the same reason. It consists of removing all words which appear less than a given number of times, which in our case was fixed to 5. After that, we removed from the corpus all the occurrences of words in a given list, which we assembled by collecting words with little information.

```
STOPWORDS = set([
    "a", "about", "above", "after", "again", "against", "all",
    "also", "altough", "am", "an", "and", "any", "are", "aren't",
    "as", "at",
    "b", "be", "because", "been", "before", "being", "below",
    "between", "both", "but", "by",
    "c", "can", "can't", "cannot", "could", "couldn't",
    "d", "de", "did", "didn't", "do", "does", "doesn't",
    "doing", "don't", "down", "during",
    "e", "each", "either", "even",
    "f", "few", "for", "from", "further",
```

```
12      "g",
13      "h", "had", "hadn't", "has", "hasn't", "have", "haven't",
14      "having", "he", "he'd", "he'll", "he's", "her", "here", "here's",
15      "hers", "herself", "him", "himself", "his", "how",
16      "how's", "however",
17      "i", "i'd", "i'll", "i'm", "i've", "if", "ii", "in", "into",
18      "is", "isn't", "it", "it's", "its", "itself",
19      "j", "just",
20      "k",
21      "l", "like",
22      "m", "many", "may", "me", "more", "most", "much", "must",
23      "my", "myself",
24      "n", "nd", "neither", "no", "nor", "not", "now",
25      "o", "of", "off", "on", "once", "only", "or", "other",
26      "our", "ours", "ourselves", "out", "over", "own",
27      "p",
28      "q",
29      "r", "rd",
30      "s", "same", "shall", "she", "she'd", "she'll", "she's",
31      "should", "shouldn't", "so", "some", "such",
32      "t", "th", "than", "that", "that's", "the", "their", "theirs",
33      "them", "themselves", "then", "there", "there's", "these",
34      "they", "they'd", "they'll", "they're", "they've", "this",
35      "those", "though", "through", "to", "too",
36      "u", "under", "until", "up", "us",
37      "v", "very",
38      "w", "was", "wasn't", "we", "we'd", "we'll", "we're", "we've",
39      "were", "weren't", "what", "what's", "when", "when's", "where",
40      "where's", "which", "while", "who", "who's", "whom", "why",
41      "why's", "will", "with", "won't", "would", "wouldn't",
42      "x",
43      "y", "you", "you'd", "you'll", "you're", "you've", "your",
44      "yours", "yourself", "yourselves",
45      "z",
46      "zero", "one", "two", "three", "four", "five", "six",
47      "seven", "eight", "nine", "ten", "eleven", "twelve"
48  ])
```

Code 5.1: List of stopwords

In Code 2.1 are reported all the stopwords we removed from the corpus. We chose very general-purpose words that are used in a lot of situations, and that, for this reason, carry a lot less information than more specific words.

As the third step, we proceeded to split the whole text file into tokens. We opted for a word tokenisation instead of a subword tokenisation. This is because our goal was to create particular word embeddings, rather than trying to build an optimal embedding framework.

As the fourth and last step, we performed subsampling as reported in the original word2vec paper (Mikolov et al., 2013). This procedure consists of removing some words randomly, computing a specific probability for each word. More specifically, what we compute is the probability for a given word to be removed from the corpus depending on its relative frequency.

```
1    def _subsampling(self, idx_words):
2        """
3        Applies subsampling to the list of word indices to reduce the
     frequency of very common words.
4        Returns a filtered list of word indices.
5        """
6        resulting_words = []
7
8        # dict word_idx:occurences
9        word_counts = Counter(idx_words)
10
11       # dict word_idx:rel_freq
12       freqs = {word: count / len(idx_words) for word, count in
   word_counts.items()}
13
14       # dict word_idx:prob_dropping
15       p_drop = {word: 1 - np.sqrt(self.subsampling_threshold /
   freqs[word]) for word in word_counts}
16
17       # list of words after subsampling
18       resulting_words = [word for word in idx_words if random.
   random() < (1 - p_drop[word])]
19
20       return resulting_words
```

Code 5.2: Subsampling function

The probability $P(w)$ of a given word being removed is

$$P(w) = 1 - \sqrt{\frac{t}{f(w)}} \tag{5.1}$$

where t is a threshold set by the designers to $1e-5$ and $f(w)$ is the relative

frequency of word w in the coprus. We reported our implementation of the formula in Code 2.2.

5.2 Train Loop and Dataset creation

We built the dataset after all the preprocessing on the corpus was completed. We split the corpus into two parts, used respectively for training and validation purposes. We used a validation ratio of 0.1, which means that one in ten parts of the preprocessed corpus was reserved for validation measurements. This is a very common choice for many machine learning and deep learning models.

Given a corpus split, we build the dataset in batches during the train loop. This is done in order to reduce the computational load. We go through the training split in batches, a portion of examples for which we compute the loss and then update the parameters of the model. The base algorithm for this approach is called Stochastic Gradient Descent (SGD). In our training loop, we use a more sophisticated version, called Adam (Kingma and Ba, 2017), which also exploits gradient momentum. Adam is arguably the most used optimisation algorithm for modern deep learning applications, like Transformers (Pan and Li, 2023). We chose it over other algorithms like SGD itself due to the fact that Adam is more robust to bad hyperparameter initialisation. Here we report the main loop implementation, which is very standard.

```
1  for e in range(epochs):
2      curr_batch = 0
3      epoch_loss = 0.0
4
5      ############ For each batch ############
6      for targets, contexts1, contexts2, noise1, noise2 in
   train_dataloader:
7          steps, curr_batch = steps + 1, curr_batch + 1
8
9          # get embeddings
10         target_embeds = self.model.get_target_embed(self._to_tensor(
   targets))
11         context_embeds1 = self.model.get_context_embed(self.
   _to_tensor(contexts1))
12         context_embeds2 = self.model.get_context_embed(self.
   _to_tensor(contexts2))
13         noise_embeds = self.model.get_context_embed(self._to_tensor(
```

```
noise1))
14          noise_embeds2 = self.model.get_context_embed(self._to_tensor(
noise2)) if noise2 is not None else None
15
16          # compute loss and update parameters
17          loss, _, _, _, _, _ = self.criterion(target_embeds,
context_embeds1, context_embeds2, noise_embeds, noise_embeds2)
18          if self.regularizer is not None:
19              loss += self.regularizer(self._to_tensor(contexts1), self
._to_tensor(contexts2), self._to_tensor(noise1), self._to_tensor(
noise2))
20          epoch_loss += loss.item()
21
22          optimizer.zero_grad()
23          loss.backward()
24          optimizer.step()
```

Code 5.3: Main loop (training)

Regarding the dataset, each batch consists of pairs *(word, context)*, extracted from the corresponding corpus split. The source used to retrieve the context from a given word at position i in the corpus is reported in Code 2.3. This function chooses a number uniformly at random between 2 and the maximum window size, which is a hyperparameter of the model. This number becomes the radius of the context R, meaning that any word $[i - R, i + R]$ is considered a context word for that instance of w. We also implemented the option to use a dynamic window approach, where the size of the context is chosen depending on the number of instances of the word itself.

From there, each of these pairs is transformed into a corresponding list of positive and negative examples. For the negative ones, a random word is extracted from the corpus using a sampling table built previously. It uses relative frequencies to determine the probability of a word being used as noise. Regarding the positive ones, it's worth spending some time on them.

```
1 def _get_context(self, words, idx, win_size):
2     """
3     Returns the list of context word indices for a given word at
position idx.
4     If win_size < 0, uses the number of unique context words to
dynamically set the window size.
5     """
6     if win_size < 0:
```

```
7            ctx_count = len(self.idx_to_unique_words_in_context[words[idx
        ]])
8            R = (3 - min(int(ctx_count / 150), 2)) * 2
9        else:
10           R = np.random.randint(2, max(win_size + 1, 3))
11
12       start = idx - R if (idx - R) > 0 else 0
13       stop = idx + R
14       context_words = words[start : idx] + words[idx + 1 : stop + 1]
15
16       return list(set(context_words))
```

Code 5.4: Function for context extraction for a given word in the corpus

5.2.1 Repetition in positive examples

Previous versions of the model built positive examples by retrieving all possible pairs of context words for a given target and its context. This led them to have $n(n-1)$ positive examples for each word w, where n is the size of the context of w. In this way, the same pair is considered twice during training, but in reverse order. We will refer to this type of dataset as *standard dataset*, since it's the same method used in the baseline works. In Version B, the authors claimed that the softplus introduction would have compensated for this fact. We report their claim below.

Consider the case of a pair (u_1, u_2), where u_1, u_2 are context words for w. Such a pair will be associated with a distance $d_w(u_1, u_2)$ between projections as explained in Chapter 4. When the model considers the inverse pair, which we define as (u_2, u_1), the value of the distance becomes

$$d_w(u_2, u_1) = -d_w(u_1, u_2) \tag{5.2}$$

Let's say, without loss of generality, that the pair (u_1, u_2) corresponds to a positive distance. Then, after the softplus, its value will remain almost the same, since it falls in the positive semiaxis. The inverse pair, instead, will be mapped to a value very close to zero. Version B authors claimed that in this way, the redundant positive example (the one associated with the inverse pair) would not have been considered by the loss, since the softplus would have "turned off" that example.

This claim is simply wrong. Even if the sofplus maps a certain point to a score of 0, the loss won't be exactly 0 as the Figure 4.2 shows. But even if

we had such a case, having a null loss implies that the model tends to stay in that situation, not that the corresponding example is ignored. The optimisation algorithm is applied to the whole loss and tends to minimise the whole loss, so every considered point could be influenced, even if its contribution at a given time is null.

Understanding this phenomenon changed how we approached the dataset creation process up to then. We opted for a more coherent version of the system, where the loss didn't have to count redundant positive examples. We thought it might have improved the performance by adjusting the ratio between positive and negative examples. We opted for a radical solution, which is removing the redundant examples from the dataset. By doing just this, however, we faced another non-trivial problem. Consider Equation (4.1) as an example. This equation for pair (u_2, u_1) becomes

$$L(w, u_2, u_1) = |d_w(u_2, u_1)| + \sum_{i=1}^{k} softplus(-d_w(u_2, n_i)) \tag{5.3}$$

Simply removing the pair (u_2, u_1) from the dataset is not possible. Before is needed to define how to handle all the negative examples associated with (w, u_2). We could say that negative examples are dependent on the *ordering* of context word pairs, since they rely on exactly one of them.

We propose two solutions we thought about. We only implemented one of them, but we think it could be useful for the reader to understand some other possible approaches:

- **Ignore negative examples associated with pair** (w, u_2, u_1): in this way, the ratio among positive and negative examples remains the same (the hyperparameter k). On the other hand, this solution would be strongly dependent on the order in which context words are retrieved. Let u_1 be the first context word in a context $C = u_1, ..., u_n$ of n words. Since we have k negative examples for each positive example, the total number of negative examples involving a certain context word u_i depends on the number of pairs where u_i is the first element. Therefore, for word u_1 we'll have nk negative examples, for word u_2 only $(n-1)k$, and so on. Finally, the last word u_n would have no negative examples, since there will be no pair $(u_n, u_i), i \neq n$. For this reason, we decided to discard this approach

- **Add negative examples of** (w, u_2, u_1) **to the ones of** (w, u_1, u_2): this maintains the number of negative examples per word the same (all the negative examples for u_1 plus all negative examples for u_2). This changes the meaning of the hyperparameter k, which now represents the number of negative examples per pair *(target, context)* in a positive example.

We'll refer to datasets obtained with the process we just described as *dataset with no repeated pairs*. We'll present some results obtained with this new dataset generation method in Chapter 6.

The creation of the dataset is performed by exploiting the Pytorch built-in classes for data manipulation, but with some customisation. We report the function used to extract from a batch of *(target, context)* the corresponding dataset batch.

```
def collate(self, batch):
    """
    Given a batch (list of word indices), returns positive and
    negative training examples.
    It was created to process a batch, now is used passing the whole
    corpus
    Returns: target_word_list, context_words1, context_words2,
    noise_words, noise_words2
    """
    target_word_list = []  # list containing target words
    context_words1 = []  # list of words in the context, second order
    context_words2 = []  # list of words in the context, third order
    noise_words1 = []
    noise_words2 = []

    # ---- start batch job ----
    for ii in range(len(batch)):
        target = batch[ii][0]
        context = batch[ii][1]

        # TARGERT WORDS -----------------
        # get number of contiguous target words in the list
        target_word_rep = len(context) * (len(context) - 1)
        if self.with_no_rep:
            target_word_rep = int(target_word_rep / 2)

        # repeat target
        target_word_list.extend([target] * target_word_rep)

        for i in range(len(context)):
            # CONTEXT WORDS 1 ---------------
            # get number of contiguous context words (the first) in
    the list
            context_word1_rep = len(context) - 1
            if self.with_no_rep:
```

```
32                  context_word1_rep -= i
33
34              # repeat context word (the first)
35              if context_word1_rep != 0:
36                  context_words1.extend([context[i]] *
context_word1_rep)
37
38                  start_range = i + 1 if self.with_no_rep else 0
39                  for j in range(start_range, len(context)):
40                      # CONTEXT WORDS 2
41                      # add j-th context words (the second) in the list
 (if different from the first)
42                      if i != j:
43                          context_words2.append(context[j])
44
45      # ---- end batch job ----
46
47      noise_words1 = np.random.choice(self.sample_table, size=(len(
target_word_list), self.neg_samp_per_word))  # list of negative
samples for each context word
48      noise_words2 = None
49      if self.with_no_rep:
50          noise_words2 = np.random.choice(self.sample_table, size=(len(
target_word_list), self.neg_samp_per_word))
51      for i in range(len(context_words1)):
52          for j in range(self.neg_samp_per_word):
53              while (context_words1[i] == noise_words1[i][j]): # if a
noise word is the same with the context
54                  noise_words1[i][j] = self.sample_table[random.randint
(0, len(self.sample_table) - 1)] # force it to be different than
corresponding context sample
55
56              # same of before for the respective context_words2 and
the respective neg word
57              if self.with_no_rep:
58                  while (context_words2[i] == noise_words2[i][j]):
59                      noise_words2[i][j] = self.sample_table[random.
randint(0, len(self.sample_table) - 1)]
60
61      return target_word_list, context_words1, context_words2,
noise_words1, noise_words2
```

Code 5.5: Function for the creation of a batch of the dataset

5.3 Hyperparameters and versions

At this point, we have described the technical details of our implementations. In this section, we describe further all the hyperparameters involved in our model. Then we present some variations we tested for our model.

This is a list of all the hyperparameters of the model. For each of them, we briefly describe its role and the value we assigned it:

- *minimum number of word frequency*: the minimum number of tokens of the same word written in the corpus. If a word has fewer instances in the corpus than this value, it's removed from the corpus in the first step of preprocessing. See Section 5.1
- *subsampling threshold t*: threshold value in the subsampling formula presented in Section 5.1. The value is the same as that assigned by the authors of word2vec (Mikolov et al., 2013), which is $1e^{-}5$
- *embedding dimension*: the number of components of our embedding vectors. This is also the dimension of the space where those embeddings are located. It has been set to 300
- *window size s*: maximum radius of a context for a given target word w. It has been set to 4
- *number of negative examples per word k*: this value changes meaning depending on how we are constructing our dataset (see Section 5.2.1). If the dataset is built in a "traditional" way, then it represents the number of negative examples per positive example. In the other case, it represents the number of negative examples *per single context word* in the respective positive example. In both cases, this value has been set to 5
- *horizontal offset δ*: only used in some variants of the final loss to force noise projections to go further from w than the context projections (see section 4.4). Formally, in the standard version of the loss is set to 0. When used, a good value we found is 10
- *validation ratio*: the percentage of the corpus that is used to generate a validation split. Has been set to 0.1
- *batch size b*: the number of target words in a batch. This means that the batch size of the final dataset is actually bigger. For each target word, the number of positive examples depends on the number of context words, and therefore on the window size. Then the effective batch size is never the same at each iteration
- *number of epochs*: number of times we loop through the whole dataset and update our parameters. It has been set to 6
- *learning rate μ*: scalar used in Adam optimization algorithm. It scales the gradient of the parameters before the update. It has been set to 0.003

6
Result Analysis

In this chapter, we present, analyse, and discuss the outcomes of experimental measurements on our model. More specifically, we investigate the capabilities of all three versions of the model, comparing their performance across different tasks and evaluation methods. Our model is introduced in two variants, depending on the dataset used: the standard version and the version without repeated pairs, as described in Section 5.2.1.

We begin in Section 6.1 by presenting the loss function values over the epochs for the final version of the model. We did not report this data for Versions A and B, as we consider it uninformative. Instead, those models are used as baselines in other evaluation metrics to highlight the capabilities of the new version.

In Section 6.2, we describe each test we performed, discuss the results, and detail the datasets employed for each task.

6.1 Loss curves

As described in Chapter 5, our model was trained for 6 epochs. At each iteration step, we recorded the loss values for both the training and validation sets. Computing the loss for both datasets is useful to identify the point at which the model begins to overfit, as discussed in Chapter 1.

We report the loss curves for our model with the hyperparameter δ set to 10. In Figure 6.1, the model was trained on the standard dataset, while in Figure 6.2 it was trained on the dataset without repeated pairs.

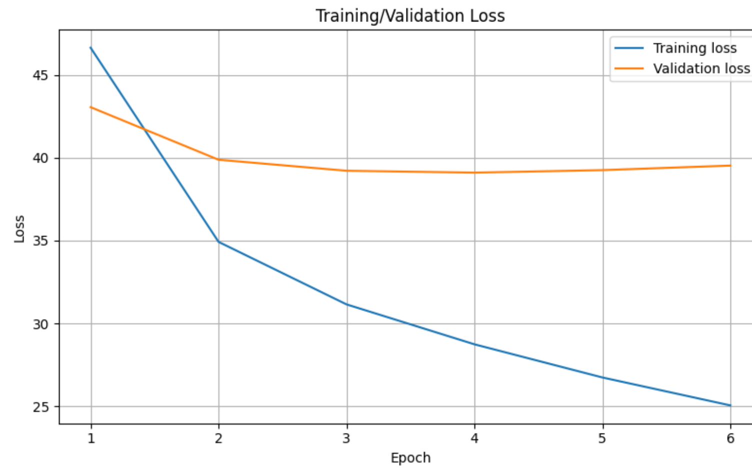


Figure 6.1: Loss function for our model with $\delta = 10$ and standard dataset.

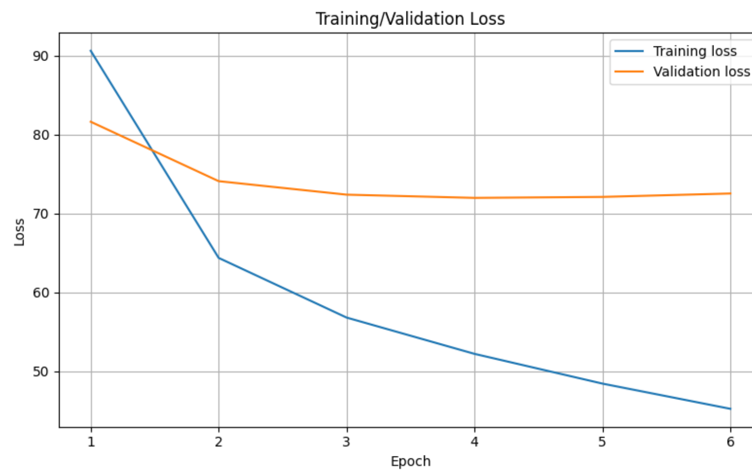


Figure 6.2: Loss function for our model with $\delta = 10$ and no repeated pairs in the dataset.

Although the absolute values of the losses differ significantly, the performance of the two variants is not substantially different. We discuss these results further in the next section.

From the figures, we observe that neither variant exhibits severe overfitting. Both reached the minimum validation loss at epoch 4. While the increase from epoch 4 to 6 is small in both cases, the divergence is still visible, so we did not extend training beyond 6 epochs.

In Section 6.2, we use the model version that achieved the best validation score. This approach is known as early stopping, a general machine learning technique that halts training once the training and validation losses begin to diverge excessively, as mentioned in Chapter 1.

6.2 Evaluation Metrics

To properly assess the capabilities of word embeddings, examining only the loss function is not sufficient. The practical performance of such systems depends on the tasks for which they are applied. For this reason, the literature defines a variety of evaluation methods to measure the quality of word embeddings in specific contexts.

In this work, we focus on testing the lexical semantic capabilities of our model. To this end, we present one standard evaluation method, involving various datasets. This method is intrinsic, meaning that it evaluates the embeddings in isolation. By contrast, extrinsic methods assess the performance of an end-to-end application that incorporates the embeddings.

6.2.1 Word Similarity

With this experiment, we aimed to evaluate our model's ability to capture the meaning of words—one of the most important objectives of this work. To do so, we adopted a standard strategy: using human-annotated datasets to test whether our model aligns with their ground truth. Many word similarity datasets have been proposed in the literature. Most of these are built such that each example consists of a pair of words accompanied by a real number within a fixed interval, representing the similarity between the two.

The procedure is the same for each dataset. Given a pair of words, we compute their similarity as the cosine similarity between their embeddings.

After collecting all values for the dataset, we compare them to the ground-truth scores using the Spearman correlation coefficient. This statistic measures the strength of a monotonic relationship between two variables. It is obtained by ranking all values for each variable (from 1 to n) and then computing:

$$\rho = 1 - \frac{6 \sum_{i=1}^{n} d_i}{n(n^2 - 1)} \tag{6.1}$$

where n is the number of paired values and d_i is the difference between the ranks of the two variables at position i. The coefficient ρ ranges from -1 to 1.

We used the following well-known datasets:

- **WordSim-353**: 353 word pairs from various topics, annotated by multiple individuals on a 0–10 scale. The final similarity score is the mean across annotators. This dataset, introduced in 2002 (Finkelstein et al., 2002), remains one of the most widely used benchmarks for semantic similarity.
- **RG-65**: Proposed in 1965 (Rubenstein and Goodenough, 1965), this dataset contains 65 word pairs scored on a 0–4 scale and has been extensively reused in later studies.
- **SimLex-999**: A dataset of 999 word pairs with similarity ratings on a 0–10 scale, first introduced in (Hill et al., 2014).
- **Stanford Rare Words**: A dataset focused on rare words, containing 2,034 pairs (Luong et al., 2013).

We split WordSim-353 into two subsets: 203 pairs for validation and 150 for testing. At each training iteration, we computed the Spearman correlation on the validation subset to track how the model improved (or degraded) over time.

Table 6.1 reports the Spearman correlations for three variants of our final model, differing in the hyperparameter δ (see Section 4.4). Unless otherwise specified, models were trained on the dataset version without repeated pairs (see Section 5.2.1). A label "std" on the model name indicates that the model was trained on a standard dataset instead. The labels in Table 6.1 can be read as follows:

- "WSim": WordSim dataset, split into validation or test part, as previously described. The label "WSim All" represents the performance obtained on the whole WordSim dataset without splits
- "RW": Stanford Rare Words dataset
- "RG-65": RG-65 dataset
- "SimLex": SimLex-999 dataset

Model	WSim Val	WSim Test	WSim All	RW	RG-65	SimLex
$\delta = 10$	0.70	0.48	0.65	0.25	**0.62**	0.20
$\delta = 5$	0.66	0.47	0.62	0.23	0.60	0.19
std, $\delta = 10$	0.65	**0.49**	0.62	**0.26**	0.52	**0.23**
std, Ver. A	0.07	-0.04	0.03	0.01	0.23	0.05
std, Ver. B	0.70	0.48	**0.66**	0.24	0.57	0.21
Ver. B	**0.72**	0.44	0.65	0.22	0.60	0.19

Table 6.1: Results of word similarity test with different datasets for different versions of the embedding.

The results of Version A are included only for reference: they confirm the model's inability to generate coherent semantic representations and motivated the development of Version B. Version B dramatically improved performance, as already reported by its authors.

Our models achieved performance comparable to Version B. In fact, on some datasets, our best variant ($\delta = 10$) slightly outperformed it. While the numerical differences are not large, our main goal was to reach at least the same performance as Version B while providing a stronger theoretical motivation. This foundation makes our model easier to build upon in future work.

The variant trained on the standard dataset outperformed all others on RW and SimLex-999, though it performed poorly on RG-65. Comparing the two dataset-construction methods for the same δ, it is difficult to declare one universally superior.

A final remark concerns WordSim-353. Since we split the dataset into validation and test subsets, part of it was used directly for model improvement. This is reflected in the results: for both top-performing models, the gap between the validation and test subsets is as large as 0.22. For this reason, the "WSim Test" values should be given more weight than "WSim Val" or "WSim All," as the latter are likely affected by overfitting.

6.2.2 Comparison with Word2vec

In this section, we present another experiment performed to compare our model capabilities with respect to word2vec. Word2vec original implementation is trained on 6 billion tokens (Mikolov et al., 2013). The training dataset we used for this work, instead, has merely 3084988 tokens after preprocessing. We argue

that the magnitude difference in the dataset dimension may play a key role in performance. In order to test this, we trained word2vec as described in section 2.2 on the same dataset as our models and measured the Spearman correlation for each dataset we saw before. We report those results in Table 6.2.

Model	WSim Val	WSim Test	WSim All	RW	RG-65	SimLex
$\delta = 10$	**0.70**	0.48	0.65	0.25	**0.62**	0.20
$\delta = 5$	0.66	0.47	0.62	0.23	0.60	0.19
std, $\delta = 10$	0.65	**0.49**	0.62	**0.26**	0.52	**0.23**
std, Ver. B	**0.70**	0.48	**0.66**	0.24	0.57	0.21
word2vec	0.62	0.46	0.59	0.22	0.51	0.17

Table 6.2: Results of word similarity test with different datasets for different versions of the embedding compared to word2vec.

What we can see from the table is that word2vec cannot match the performance of our models on either dataset. This seems to demonstrate the evident lack of data for our model training. For our research, this is one of the most important results. We are now able to suppose that the intuition and implementation of our model contain a reasonable improvement with respect to the state-of-the-art. Future works may try to exploit bigger datasets and document the variation of performance at different magnitudes of tokens. Anyway, for the same amount of tokens, the state-of-the-art method for static embeddings SGNS seems to perform worse in word similarity tasks than the proposed models. This is an interesting observation for us and one of the main results of this work.

Conclusions

With this work, we analysed novel word embedding systems that leverage vector projections. Our goal was to assess whether these geometric tools can provide an effective means to inspect word senses and to learn higher-quality word embeddings.

Our analysis of previous work revealed several inconsistencies between the reported results and the formal definition of the models, particularly regarding their loss functions. We therefore presented a detailed account of these incongruences and explained the flaws in both Versions A and B. Although a complete explanation of the effective behaviour of these models remains elusive, we provided clear motivations and insights into what is likely to occur during training and what is not.

Building on these analyses, we proposed a new version of the embedding model. We described its design, implementation, and experimental results, focusing on its ability to capture multiple word senses. Using standard datasets for word similarity tasks, we compared the previous models with our new version. The results show that our model performs on par with Version B while being supported by a stronger theoretical foundation. We also expanded the design space by investigating dataset creation strategies and experimenting with new approaches to manage them. Our model also outperformed word2vec for the same training data size. This enforces our theoretical results with a solid experimental baseline for future development and expansions of the model, which should be trained with an appropriate amount of data.

We would like to spend a few more words on certain aspects of our work

that remain incomplete, or at least not fully clarified. Originally, this research was built upon the work presented in Akman, 2022, which focused on Version B of the model. In that dissertation, considerable emphasis was placed on the experimental results, leaving some gaps in the explanation and interpretation of those findings. As extensively discussed throughout this work, the goal of providing an explanation for the results obtained with Version B has always been the driving force behind our research.

However, despite the progress achieved on the experimental side, the good performance of Version B, in light of the issues discussed in 4.2, remains poorly understood. Our hypotheses have led us to believe that the use of the softplus function may have induced a "spontaneous" ordering of the context and noise projections along the direction of the targets (see section 4.3), although we were never able to prove such a claim. More importantly, we have not been able to provide an explanation as to why a loss function whose behaviour differs so markedly from the ideal one is nevertheless capable of producing such results (see Figure 4.2). It would be highly valuable to finally resolve these questions, as doing so could further unlock the development of a third-order word embedding model inspired by SGNS, building upon the foundations laid by these works.

Another interesting direction for future exploration concerns the performance of the model. As mentioned earlier, in this work, we limited our experiments to datasets specific to Word Similarity tasks, but the number of possible analyses is much broader. A classic example concerns the Word Analogy tasks, briefly introduced as an example in Section 2.1. It would be interesting to assess whether our model has managed to exploit vector projections efficiently for this kind of task as well. Numerous datasets have also been proposed for such evaluations, which remain a standard benchmark for these models.

We stressed a lot the fact that projections could allow the embedding to encode more word senses with a single representation more efficiently. It would be interesting to design a specific experiment to investigate if we reached this goal. More specifically, given a word with more than one word sense, like *bank*, a future work could check if, along the direction of the word *bank*, words from different contexts displace themselves such that their projections end up in different corridors (as discussed in section 3.2.1). Building a specific dataset for such a task could lead to a depper investigation of our model capabilities and performace.

In the end, the main contribution of this work is a clearer understanding of

the role that geometric projections can play in static word embedding systems and of their potential improvement with respect to the state-of-the-art systems.

References

Akman, A. O. (2022). *Design of a third-order word embedding model using vector projections* [Master's thesis]. University of Padova.

Camacho-Collados, J., & Pilehvar, M. T. (2018). From word to sense embeddings: A survey on vector representations of meaning. https://arxiv.org/abs/1805.04032

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. https://arxiv.org/abs/1810.04805

Finkelstein, L., Gabrilovich, E., Matias, Y., Rivlin, E., Solan, Z., Wolfman, G., & Ruppin, E. (2002). Placing search in context: The concept revisited. *ACM Trans. Inf. Syst.* https://doi.org/10.1145/503104.503110

Galvan, E. (2021). *Word embedding: Progettazione e valutazione di un modello del terzo ordine* [Master's thesis]. University of Padova.

Harris, Z. S. (1954). Distributional structure. *WORD*, *10*(2-3), 146–162. https://doi.org/10.1080/00437956.1954.11659520

Hill, F., Reichart, R., & Korhonen, A. (2014). Simlex-999: Evaluating semantic models with (genuine) similarity estimation. https://arxiv.org/abs/1408.3456

Kingma, D. P., & Ba, J. (2017). Adam: A method for stochastic optimization. https://arxiv.org/abs/1412.6980

Liu, Q., Kusner, M. J., & Blunsom, P. (2020). A survey on contextual embeddings. https://arxiv.org/abs/2003.07278

Luong, M.-T., Socher, R., & Manning, C. D. (2013). Better word representations with recursive neural networks for morphology. *CoNLL*.

Mahoney, M. (2011). Large text compression benchmark. https://www.mattmahoney.net/dc/text.html

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. https://arxiv.org/abs/1310.4546

Miller, G. A. (1995). Wordnet: A lexical database for english. *Commun. ACM*, *38*(11), 39–41. https://doi.org/10.1145/219717.219748

Pan, Y., & Li, Y. (2023). Toward understanding why adam converges faster than sgd for transformers. https://arxiv.org/abs/2306.00204

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L. (2018). Deep contextualized word representations. https://arxiv.org/abs/1802.05365

Pytorch library. (n.d.). https://pytorch.org/

Rubenstein, H., & Goodenough, J. B. (1965). Contextual correlates of synonymy. *Commun. ACM*, *8*(10), 627–633. https://doi.org/10.1145/365628.365657

Schütze, H. (1998). Automatic word sense discrimination (J. Hirschberg, Ed.). *Computational Linguistics*, *24*(1), 97–123. https://aclanthology.org/J98-1004/

Tensorflow library. (n.d.). https://www.tensorflow.org/