FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Joonas Tapaninaho

USING NEURAL NETWORKS TO GENERATE FINNISH WORD EMBEDDING VECTORS

Bachelor's Thesis
Degree Programme in Computer Science and Engineering
April 2024

Tapaninaho J. (2024) Using Neural Networks to Generate Finnish Word Embedding Vectors. University of Oulu, Degree Programme in Computer Science and Engineering, 46 p.

ABSTRACT

This bachelor's thesis aims to find out how well word embedding vectors trained with the help of shallow neural networks are suitable for the Finnish language and how well they can recognize semantically similar words and synonyms. The word embedding vectors produced by neural networks that are the subject of comparison in the thesis were trained by using the Word2Vec and FastText algorithms. These algorithms contain small differences in the functioning of the neural networks, which can be seen when comparing the word embedding vectors. The thesis aims to accurately define structural differences and their effects on word embedding vectors, as well as the historical development of natural language processing from methods using machine learning to current neural network-based methods, which utilize deep learning. The common presupposition is that word embedding vectors trained using the FastText algorithm should better recognize semantically similar words and synonyms compared to those trained using the Word2vec algorithm. However, the thesis shows that this is not necessarily the case with the Finnish language, for example, the amount of training data and the source of it have a strong influence on the matter in addition to the dimension of the embedding vectors.

Keywords: Artificial intelligence, natural language processing, machine learning, Word2vec, FastText

Tapaninaho J. (2024) Neuroverkkojen käyttö suomenkielisten sanojen upotusvektoreiden generoinnissa. Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 46 s.

TIIVISTELMÄ

Tämä kandidaatin tutkielma pyrkii selvittämään, kuinka hyvin matalien neuroverkkojen avulla koulutetut sanojen upotusvektorit soveltuvat suomenkielelle ja kuinka hyvin niiden avulla pystytään tunnistamaan samankaltaisia sanoja sekä synonyymeja. Tutkielmassa vertailun kohteena olevat neuroverkkojen tuottamat sanojen upotusvektorit ovat koulutettu käyttäen Word2Vec- ja FastText-algoritmeja. Nämä algoritmit sisältävät pieniä funktionaalisia eroavaisuuksia neuroverkkojen toiminnassa, jotka on nähtävissä sanojen upotusvektoreita vertailtaessa. Tutkielmassa pyritään avaamaan tarkasti rakenteellisia eroja ja niiden vaikutuksia sanojen upotusvektoreihin sekä luonnollisen kielen käsittelyn historiallista kehitystä koneoppimista käyttävistä menetelmistä nykyisiin syväoppimista hyödyntäviin neuroverkkopohjaisiin menetelmiin. Ennakko-oletuksen mukaan FastText-algoritmia käyttäen koulutettujen sanojen upotusvektoreiden tulisi tunnistaa paremmin semanttisesti samankaltaisia sanoja sekä synonyymeja verrattuna Word2vec-algoritmilla koulutettuihin. Tutkielma kuitenkin osoittaa, että näin ei välttämättä ole esimerkiksi suomenkielen kohdalla, jossa opetusdatan määrällä ja lähteellä on suuri vaikutus upotusvektoreiden ulottuvuuden lisäksi.

Avainsanat: Tekoäly, luonnollisen kielen käsittely, koneoppiminen, Word2vec, FastText

TABLE OF CONTENTS

FOREWORD

My sincerest thanks to D.Sc(Tech.) Jaakko Suutala for all the help and guidance in compiling this thesis.

Oulu, April 26th, 2024

Joonas Tapaninaho

LIST OF ABBREVIATIONS AND SYMBOLS

AGI	Artificial general intelligence
AI	Artificial intelligence
CBOW	Continuous bag-of-words
DCNN	Deep convolutional neural network
DL	Deep learning
FNN	Feedforward neural network
GPT	Generative Pre-Trained Transformer
HMM	Hidden Markov model
LLM	Large language model
LM	Language model
LSTM	Long-short-term memory
ML	Machine learning
MLM	Masked language model
MT	Machine translation
NLP	Natural language processing
NNLM	Feedforward neural network language model
OOV	out-of-vocabulary
OVA	One-versus-all
PCA	Principal component analysis
PMI	Pointwise mutual information
POS	Part-of-speech
RNN	Recurrent neural network
SGD	Stochastic gradient descent
SGNS	Skip-gram architecture with negative sampling
SVM	Support vector machine
W2v	Word2vec

$P(A\|B)$	Probability chain rule
$\prod$	Product
L_{CE}	Loss function
e	Euler's number
log	Logarithm
$\sum$	Summation
$\|v\|$	Vector v length
σ	Sigmoid function
w	Center word w embedding vector
C_{neg}, C_{pos}	Context word C embedding vector
$\hat{\mathbf{y}}$	predicted similarity between two word embedding vector
∂	Partial derivative
d	Derivative
$\mathbf{W}$	Embedding matrix storing word embeddings
η	Learning rate
s	Scoring function s

1. INTRODUCTION

Modern Generative Pre-Trained Transformers (GPTs) [1] based chatbots and other generative artificial intelligence applications are everyone's know today due to their recent research breakthroughs, which have led to massive performance improvements. The most successful generative models like ChatGPT [2] and DALL-E [3] can do things, that feel like magic but are statistically learned patterns from huge amounts of data in the end. That does not reduce fact, how impressive those generative models are, but instead surprises how complex things the computer can learn to do from picture and text-based training data, without the need for hard-coded instruction. To generate text or pictures from it, computers need at least some level of understanding of human language and its structure. To achieve that artificial intelligence (AI) comes into the picture, which has a long history of solving narrowed tasks in specific ways. This has created many subfields under artificial intelligence, which some of will be reviewed in the following paragraphs. Recent breakthroughs, which have helped to raise artificial intelligence into its current position, often combine the best methods of several subfields to create state of art applications.

Those state of art applications like ChatGPT have brought up discussions of the possible development of artificial general intelligence (AGI) [4], which can accomplish all the same tasks as humans. No one knows for sure when an AGI-level artificial intelligence system will be achievable, if never, but everyone agrees that it will need a deep level of understanding of human language. Understanding human language as general and profound is always been an extremely difficult task for computers, due to the various hidden meanings and patterns, which it contains. An additional challenge comes when artificial intelligence systems have to change between different languages, which have their own special features and structural differences.

Throughout history, there have been several ways and ideas on how to capture language information and convert it into computer-understandable form, which will be surveyed in the following chapters. There is not yet developed certain general way, that works best in all cases, but many of them perform well in several application areas. However, a common factor in almost every language-based application is that the words are represented in embedding vector format [5]. Those embedding vectors of words can capture widely different information about the words and their relations to other words, which is required in applications like translation [6 ch.13] and information search [7]. Those types of applications need more wider information about words, like context and semantic similarity with other words to function in a wanted way. Previously mentioned problems are fascinating and this thesis partly focuses on viewing proposed solutions in some of the application areas.

Evaluating how Word2vec and FastText algorithms apply to the Finnish language, is one of the research questions of my thesis. The second research question is looking answer to the question, does the synonym or semantic similarity detection with word embeddings help to find better alternative search results.

2. BACKGROUND

The Turing test is a generally accepted way to determine whether the machine exhibits intelligent behavior or not. Human language is complex, and it captures so much information, about intelligent behavior, which made language a natural choice for Alan Turing to determine the human type of intelligence in machines [8 p.20]. This choice created its own subfield for AI called natural language processing (NLP), which covers interactions between human language and computers [9].

NLP's purpose is utilizing machine learning (ML), statistical, and deep learning (DL) techniques for human language rule-based modeling [10]. Based on Russel and Norvig [8 p.20] there are three primary reasons for NLP, which are communication, learning, and scientific understanding. The first reason communication covers language-based interaction between humans and computers, for example, voice assistants and chatbots are direct examples of that interaction. To learn and know, the system needs to understand natural language, because hard coding every knowledge is not a scalable solution. The final reason is to expand the scientific understanding of language in general by using a combination of tools from AI with hand-to-hand information acquired from the fields of linguistics, cognitive psychology, and neuroscience.

2.1. History of NLP

NLP impacts our everyday life constantly, it is present when you search for information, translate text, use voice assistants, chatbots, and in countless other ways, which may not be so obvious. Development in the NLP area often comes by hand-to-hand with the progress of computers and the need of solving important tasks. It is not surprising, that after the Second World War, when two superpowers started competing with each other, and the field of NLP began to form, the first researches focused on machine translation (MT) between Russia and the English language [11]. MT focused NLP era continued from the 1940s to the late 1960s and suffered from a lack of computing power and small memory [11].

After that, the focus changed from MT to AI-flavoured and semantic-oriented topics, which enabled the simple question-answering type of communication between computers and humans, by using hard-coded rules and ready answer frames without the computers' actual understanding of language or conversation itself [11]. This phase lasted from the late 1960s to the late 1970s and contained successful systems like LUNAR, SHRDLU [11] and ELIZA [9].

From the late 1970s to 1980s dominant focus area was grammatico-logical, where the development of grammatical theory and AI field movement toward using logic for knowledge information influenced strongly [11]. This trend was reflected especially in the development of conceptual ontology programs, for example, MARGIE, QUALM and SAM, where real-world information is structured into computer readable data format [9].

In the 1980s many NLP works focused symbolic approach, which used complex hard-coded grammars and rules to language parse [12]. Other research lines stayed in the chatbot approach, which created programs like Racter and Jabberwacky [9]. After

the 1980s NLP could be roughly divided into two phases, where statistic machine learning focused phase lasted from the 1990s to the early 2000s, and deep learning models started to receive greater interest.

2.2. Statistical Machine Learning in NLP

Most of the traditional machine learning (ML) techniques in the NLP field use statistics and probabilities to perform specific wanted tasks. Those tasks can be for example natural language generation or machine translation, which uses probabilities obtained from the training phase to determine the most likely next word or translation of sentence [6 ch.3]. These types of models, which use assigned probabilities to the sequence of words are called language models (LMs), for example, ChatGPT which is a large language model (LLMs). But instead of using traditional ML techniques ChatGPT uses deep learning and massively large data sets to assign those probabilities. The simplest way to assign sentence based probabilities is to determine the probability of $P(w|h)$, where w is the certain word and h means sentence history before word w. To visualize that, we can consider example history h = *"ML is subfield of"* and we want to know the probability of how likely the next word w is *"AI"*. This probability is determined by:

$$P(w|h) = \frac{\text{C (ML is subfield of AI)}}{\text{C (ML is subfield of)}} \ , \tag{1}$$

where C represents count and we simply calculate how many times we have seen sentence *"ML is subfield of"* in training data and how many times word *"AI"* is followed that sentence history.

2.2.1. Markov Assumption

This type of probability calculating can work fine in many cases, but even the whole web isn't big enough to assign good probability estimations in most cases [6 ch.3]. We can use more clever ways to get estimates for sentence probabilities by using the Markov assumption, which assumes the probability of the next word depending only on the previous word [6 ch.3].

By using the Markov assumption and chain rule of probability we can determine the estimated probability of sentence, as follows

$$P(w_{1:n}) \approx \prod_{k=1}^{n} P(w_k|w_{k-1}) \quad . \tag{2}$$

If we want to, for example, calculate the probability of a sentence s = *"ML is subfield of AI."* we use Eq. (2) as

$$
\begin{aligned}
P(s) &\approx \prod_{k=1}^{6} P(w_k|w_{k-1}) \\
&\approx P(w_1|w_0) \cdot P(w_2|w_1) \cdot P(w_3|w_2) \cdot P(w_4|w_3) \cdot P(w_5|w_4) \cdot P(w_6|w_5) \quad ,
\end{aligned}
\tag{3}
$$

which corresponds to multiplying each words probability to respect to past word from it:

$P(s) = P(\text{ ML} \mid \text{<s>}) \cdot P(\text{is} \mid \text{ML}) \cdot P(\text{subfield} \mid \text{is}) \cdot P(\text{of} \mid \text{subfield}) \cdot P(\text{AI} \mid \text{of}) \cdot P(\text{</s>} \mid \text{AI})$.

Those individual probabilities we get by using maximum likelihood estimation (MLE), which again counts frequency of word pairs in training data. When looking at only one word form the past, like for example in $P(\text{AI} \mid \text{of})$ case, MLE is defined as

$$
P(w_n|w_{n-1}) \approx \frac{\mathbf{C}(w_{n-1}w_n)}{\mathbf{C}(w_{n-1})} \quad , \tag{4}
$$

where C in Eq. (4) is again the count and normalized estimated probability would be the count of how many times *"of AI"* appears in the data divided by the total count of word *"of"* appearance. $P(\text{ ML} \mid \text{<s>})$ and $P(\text{</s>} \mid \text{AI})$ contains special tags, which represents how certain is that a sentence starts or ends with that specific word, and the probability estimation is calculated the same way than others by using MLE. Sequencing words or characters is called N-grams, and the previous example, where we used word pairs and Markov assumption (N=2), is called bigram. If instead, we want to take notice of more than one word from the past, we just increase the size of N to the wanted size.

2.2.2. Word and Sentence Generating by Using N-Grams

Let us consider a case where we have a sentence generator, which has generated a partial sentence *"ML is subfield of"* and we want to use estimated 4-gram probability to see how likely next additional word is *"AI"*. We calculate the estimated 4-gram probability as follows

$$
\begin{aligned}
P(w_n|w_{n-N+1:n-1}) &= P(w_n|w_{n-3:n-1}) \\
&= P(\text{AI}|\text{is subfield of}) \\
&= \frac{C(\text{is subfield of AI})}{C(\text{is subfield of})} \quad .
\end{aligned}
\tag{5}
$$

This equals Eq. (1) calculation, but instead of using whole partial sentence history to estimate the next words probability, we only use a part of it. A good real-life example of using N-gram probabilities is search engines and their search suggestions. One way for search engines to give these suggestions as in Figure 1 is to calculate the probabilities to different sizes of N-grams from users' previous search query sentences and suggest the next word based on the highest probability. For example suggestion

icon could have been obtained from 3-gram probabilities, where the highest probability for w in $P(w|\text{machine learning})$ probabilities are looked at from words, which start with the character "i".

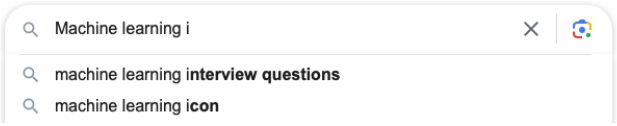

Figure 1. Example capture from Google search.

In practice, when we calculate n-gram or sentence probabilities in language models, we get really small numbers. To avoid underflow those probabilities are always reformed to log probabilities $= log(P_1 \times P_2 \times \cdots \times P_n)$, which modifies those probabilities a little bit higher. If we need reform those back to real probabilities, we just use exponent $P_1 \times P_2 \times \cdots \times P_n = e^{log(P_1 \times P_2 \times \cdots \times P_n)}$ to achieve that [6 ch.3].

2.3. Statistical ML in NLP

Statistical ML has helped to produce several real-world applications, which use natural language. Many of those modern ML algorithms utilize well known old probabilistic rules and assumptions, which are possible to be included into new generation algorithms after the increase of computational power. Naive Bayes classifier uses Bayes' rule to simplify classifying task to prior probability and likelihood multiplication, where prior probability corresponds to a certain share of class c, of the training examples and likelihood features f (contained by document d) individual probabilities to belong in a category c multiplied together. One of the first naive Bayes classifier adaptions was spam detection, where emails are classified as spam or not by looking at features (words), which it contains [6 ch.4].

Another popular ML algorithm for classification task is support vector machine (SVM), which modern version was first introduced in a paper [13]. SVM approach is to learn linear or non-linear hyperplane decision boundary to separate data points belonging to class A or B [14 p.20]. It is also applicable solution to classify more than two classes, for example by using one-versus-all (OVA) solution, where document d belongs to class A or class others, and finding the most suitable class by making A or others comparison, for document d respect to every possible class.

Other important NLP areas like MT and part-of-speech (POS) tagging applications have utilized Hidden Markov Models (HMM), which is a statistical model, where the modeled system is assumed to be a Markov chain containing unobservable hidden states. HMM based solutions have been often used in application areas, which are believed to contain hidden causal factors and those are possible to observe from data only once it is generated [14 p.21]. Even though ML algorithms have shown good performance in many NLP related application areas, they contain limitations compared to deep learning algorithms. Traditional ML algorithms need structured training data and often lack possibility to scale, which poses no problem for deep learning algorithms. Deep learning algorithms can also find hidden and complex patterns from data, which is impossible to capture by traditional ML methods.

3. DEEP LEARNING

Deep learning (DL) is a subfield of machine learning, which uses neural networks as a general-purpose learning procedure [15]. The development of neural networks was inspired by the human brain and more specifically the neurons it contains. The neural network mathematical model was first introduced at paper [16] and a trainable version of that was demonstrated by Rosenblatt in 1957, which made it an active area of research until 1959 when Minsky and Papert showed its time-consuming weakness in a book titled "Perceptrons" [17]. Neural networks contained decades of very little research interest due to a lack of needed computational power like many other NLP-related models and algorithms, which later adapted to solve modern problems. Even though there was several pioneering neural network-related research, it was early 2010s when deep learning models started to outperform other ML methods.

3.1. Deep Learning in NLP

Deep learning started gaining attention in the NLP area, when Bengio et al. proposed their solution for feedforward neural language model (LM) at paper [18]. The structure of LM is based on feedforward neural network (FNN) architecture, where in simplified, data flows from the input layer to the network output layer without going backward. Bengio et al. proposed LM introduced building blocks to another type of network structures in the NLP field, which have then displaced classical FNN architectures [12]. Recurrent neural Networks (RNN) uses a different architectural approach compared to FNN, where instead of flowing data only forward, RNN flows data over history.

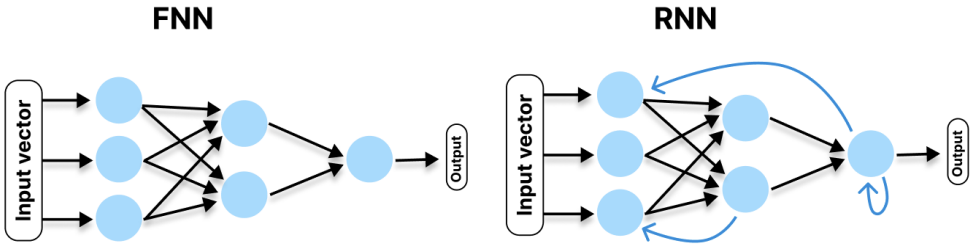

Figure 2. Simplified structural difference between FNN and RNN. Adapted From [19 p.6].

More precisely, RNNs contain hidden layer or layers, which can store and utilize internal state memory to process flowing data [6 ch.9]. As a weakness, RNNs face rapid information loss over time due to vanishing gradients, which led to the development of long short-term memory (LSTM) network solution [20]. Instead of being a completely different solution, LSTM is an extension of RNN model architecture, which utilizes RNN's internal state memory approach. As a difference compared to RNN, LSTM is able to remove information, that is not needed anymore, and add information, which is assumed to be needed later in decision-making. RNN and its extensions like LSTM started to gain attention in NLP due to [21] proposed idea to use RNN for NLP-related tasks, which has strongly impacted to especially

applications like MT and language modeling [8 p.826]. Currently, transformers and its variations are maybe the most important models in the NLP field and have achieved state-of-art solutions for almost every major NLP task [14 p.25]. Transformer model was introduced in the paper [22], which adapted a multi-head self-attention mechanism to model long-distance context without a sequential dependency. The original transformer model, which contains self-attention and a feedforward layer with residual connection, has inspired further solution extensions like masked language model (MLM) BERT [23], which have reached state-of-art performance in Q&A area and auto-regressive LLM ChatGPT [8 p.930].

3.2. Learning Types in DL

Typically ML algorithms are divided into three main learning types, which are supervised, unsupervised and reinforcement learning. Supervised learning means method, where the model tries to learn a way to generate wanted outputs for given inputs, which are detached from preprocessed data. Unsupervised learning uses the opposite approach, where the model is given only inputs and its task is to find structure from it. The last of the main types is reinforcement learning, which as simplified, learning by trying. Those reinforcement learning models have specific reward systems, which instruct it into desired functioning.

The previous learning types are the main categories, but there are also several other types of learning solutions, which are variations of some main category or combination of them. Deep learning uses a neural network as a structure, but it does not lock it into certain learning type, which varies according to network structure and specified task.

Modern language models like BERT and ChatGPT, which are based on transformer structure use transfer learning. The idea behind transfer learning is to use a pre-trained model as an initialization for the actual model, which is then further trained for its desired task. This can be compared to for example, if you have been trained to fly a commercial airplane, you most likely will learn to pilot smaller planes more quickly than someone without previous experience [8 p.832]. Simplistically, the purpose of pre-training in LLMs is to train the model, which generates and understands text using, during the pre-training phase learned embedding representation of subwords. That pre-trained model is then further trained usually by using reinforcement learning to wanted tasks like Q&A or chat.

3.3. Embeddings in NLP

The central idea of NLP is to analyze and understand human language to utilize that knowledge in a needed way. This leads to the question of how, we can get an NLP system to understand language without manual featuring engineering and still include information into word representation about it relations to other words [8 p.907].

Most simplest way to present a word is to use one-hot vector representation, where the length of the vector corresponds to the size of the vocabulary [8 p.907]. The size and language of vocabulary depend on the task we want our system to perform, if we

have for example system, of which vocabulary contains three words [A,B,C], we get the following one-hot vector representation of this vocabulary:

$$\begin{aligned} A &= [1, 0, 0] \\ B &= [0, 1, 0] \\ C &= [0, 0, 1] \ , \end{aligned} \tag{6}$$

where in Eq. (6) the corresponding index of the word in the vocabulary is 1 and all other positions are represented as 0. This is not a very informative way to represent words and does not give any information about their relation to each other.

There is a better and more efficient way called embedding, where instead of using long vectors, we reduce vector length to smaller and include relation information into it, instead of the index of word in vocabulary. This revolutionary idea was first introduced in article [24], where Osgood et al. used three manually featured values (Valence, Arousal, Dominance) to represent each word's meaning in 3-dimensional space [6 ch.6]. Words are not only information, that can be represented in N-dimensional space as an embedding format to capture meaning and relations, other options are for example documents or subwords.

3.4. Occurrence-Based Vectors

Embedding representation introduced by Osgood et al. needs feature engineering, which is not an effective way and also needs human determination of a words meaning. We have a simple way to capture some relation information by calculating word occurrence in a set of documents.

Let us consider the following example of four documents and the occurrence of some of the words in those documents. In the Figure 3, the count of word occurrence in each document is represented as rows and each document's word count is represented as column in the occurrence matrix. The occurrence matrix gives us the possibility to compare a word to another word and a document to another document.

	D1	**D2**	**D3**	**D4**
ai	36	58	2	6
nlp	114	80	62	89
computer	0	2	1	4
model	20	15	2	3

Figure 3. Occurrence matrix example.

As Figure 4 demonstrate, we can compare words to each other by using row vectors (Green) from the occurrence matrix and that way get information about word similarities. Using the same idea, if we want to find a similar document, we can compare document column vectors (Red) together.

	D1	D2	D3	D4
ai	36	58	2	6
nlp	114	80	62	89
computer	0	2	1	4
model	20	15	2	3

Figure 4. Word and Document comparison.

Figure 5 shows another way to compare words by using a document set is to calculate the co-occurrence matrix for words, where the value in ith column and jth row, means the sum of, how many documents contain jth and ith index word from the vocabulary.

	ai	···	nlp	cat	ml	···
language	36	···	487	0	297	···
model	1804	···	630	0	380	···
business	124	···	31	18	11	···
dog	0	···	0	233	0	···

Figure 5. Word co-occurrence example.

To compare two same-length vectors X and Y as in occurrence or co-occurrence matrix, we can use dot product, where vector X is multiplied by vector Y transpose. In case, we have long vectors, dot product may not be the best way to compare vectors and instead of use normalized cosine similarity:

$$cosine(v, w) = \frac{v \cdot w}{|v| \cdot |w|} = \frac{\sum_{i=1}^{N} v_i \cdot w_i}{\sqrt{\sum_{i=1}^{N} v_1^2} \cdot \sqrt{\sum_{i=1}^{N} w_1^2}} \quad , \tag{7}$$

where in Eq. (7) vector v and w dot product is divided by multiplication of their length.

3.5. TF-IDF

Comparing words based on their occurrence in documents, in other words by frequency, is not an effective method, because common words like "the" or "it" get too much impact and rare, more informative words, are ignored. One solution for that problem is to use **tf-idf** weighting instead of occurrence count, which is defined as

$$w_{t,d} = tf_{t,d} \cdot idf_f \ , \tag{8}$$

where in Eq. (8) t means word t in document d and $tf_{t,d}$ is simply word t occurrence count in document d. Multiplier idf_t is determined as $idf_t = log_{10}(\frac{N}{df_t})$, where N means total count of documents and df_t how many documents word t occurs. Another alternative solution is to use pointwise mutual information (PMI) proposed by Fano [25]. Although the above solutions can capture useful information about words, there is a more powerful way to capture relational hidden meaning of words like GloVe [26] or Word2vec [27], which are presented in the next chapter.

4. WORD2VEC

Depending on the source, Word2vec (W2v) is often described as an algorithm, this is partly true, but instead of considering W2v as a singular specific algorithm, it is a common designation for algorithms, that use neural networks to learn word embeddings in specific way [28]. Those word embeddings are N-dimensional vectors, which are trained to represent each word in training data vocabulary [6 ch.6]. The original version of the W2v algorithm was developed by four Google employees, who tried to find a better and more efficient way to detect semantically similar words by using a large text corpus as training data for a single-layer neural network [27]. Milkov et al. found that it is possible to train high-quality word embedding vectors by using simple model architectures, which helps to reduce computational complexity and enable a much larger data set to use in the training phase [27].

In paper [27], Milkov et al. use earlier proposed techniques [29] for measuring resulting embedding vector quality with the exception that similar words can have multiple degrees of similarity in addition that those words also tend to be close of each other in multidimensional vector space. Paper [29], shows that high-dimensional word vectors can disclose word similarity by using simple algebraic operations. For example, vector("King") - vector("Man") + vector("Woman") gives a result vector, which is closest to the vector("Queen") by comparing result vector cosine similarities with other word vectors in vocabulary.

Using embedding vectors as a representation of words, contained a long history before paper [27] and [18], where feedforward Neural Networks Language Models (NNLM) were proposed as the solution to learn those word vector embeddings [27]. At the time, when paper [27] was published, most of the proposed NNLM architecture solutions were computationally expensive to train, which inspired Milkov et al. to develop their problem solutions and further improve those proposed solutions later in the paper [30].

4.1. Structure

The architecture of the first version of W2v algorithm was a follow-up to Milkov et al. earlier proposed solutions in paper [31] and [32], which found out that it is possible to successfully train neural network language model in two steps: at the beginning use a simple model to learn continuous word vectors and then train N-gram NNLM on top of those distributed presentations of words [27]. Paper [27] proposed two new architecture models for algorithms to learn distributed word representation Continuous Back-of-Words (CBOW) and Skip-Gram, which are bases for most of the further developed improvement solutions.

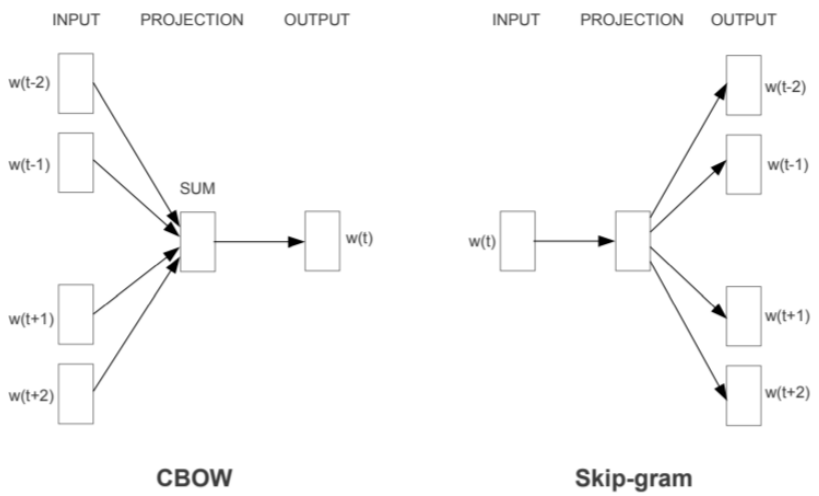

Figure 6. Structure of CBOW and Skip-gram models. Adapted From [27].

4.1.1. CBOW

CBOW has a similar type of architecture to NNLM, where all words share the same projection layer, and a non-linear hidden layer is removed [27]. This means that all vectors are averaged and words projected into the same position, which way word order history does not influence the projection [27]. CBOW takes in training phase t amount of context word vectors, where t means the size of $\pm$ context window [28]. In other words, this defines how many words before and after the center word w are taken notice of when the architecture tries to predict correct output word. In the CBOW model, the correct output is an actual center word and input words are all other words from inside the context window.

4.1.2. Skip-Gram

Skip-gram model has a similar type of architecture to CBOW, but it difference by the way it processes words in the training phase [27]. When the CBOW model architecture is trying to predict the center word by looking at words around it, Skip-gram does the opposite and tries to detect correct words around the given center word w. To compare these two models, while CBOW is faster to train it performs much weaker than the skip-gram model in total accuracy point of view [27]. Milkov et al. second paper [30] focuses on proposing improvements to the Skip-gram model architecture to lower training time which is why skip-gram is a more popular model solution than CBOW. The following structural explanations are based on skip-gram model architecture.

4.1.3. Embedding Matrix

The W2v model architecture core is the embedding matrix, which stores each word's embedding vectors. From a structural point of view, this matrix can be split into two individual matrices $\mathbf{W}1$ and $\mathbf{W}2$, where $\mathbf{W}1$ contains each word embedding vector representations as the center word and $\mathbf{W}2$ as the context word [28]. Both matrices $\mathbf{W}1$ and $\mathbf{W}2$ size is |V| x N, where |V| represents the amount of unique words in training data [28]. N is the length of each word embedding vector, in other words, dimension, which typically is between 50-1000 depending size of the training data set [6 ch.6]. Large training data sets can give high accuracy with smaller embedding vector dimensional, which again reduces computational complexity with each training epoch [6 ch.6].

4.2. Classifier

To predict the correct context word/words for a given input center word, the skip-gram model trains a probabilistic classifier, which assigns word probability to being context word by measuring its similarity with the center word [6 ch.6]. The similarity between center word w and context word c vectors can be calculated using dot product $(w \cdot c)$ by the assumption that two vectors are similar if they have high dot product. Dot product $(w \cdot c)$ is not yet probability, it is ranging number between $-\infty$ and ∞, which can be converted to probability by using the sigmoid [6 ch.6] or SoftMax [28] function. Eq. (9) shows the definition of the sigmoid function, as follows

$$Sigmoid : \sigma(x) = \frac{1}{1 + e^{-x}} \quad . \tag{9}$$

The sigmoid function modifies the negative dot product to positive and returns a number ranging between 0 to 1. By combining the sigmoid function and dot product, we can define that the classifier purpose is to maximize probability:

$$P(+|w, c) = \frac{1}{1 + e^{-w \cdot c}} \quad , \tag{10}$$

where w represents the current center word vector from embedding matrix $\mathbf{W}1$ and c is the context word vector from $\mathbf{W}2$ embedding matrix. We also want to minimize that non-real context words get high probability by minimizing the following eq.(11).

$$P(-|w, c) = 1 - P(+|w, c) = \frac{1}{1 + e^{\,w \cdot c}} \quad , \tag{11}$$

where w is again the center word embedding vector and this time c is a non-context word embedding vector. Equation (10) calculates probability only for one context word from context window $\pm$ t, which can contain multiple other context words. Skip-gram assumes context words independence, which allows just multiply word probabilities:

$$P(+|w, c_{1:t}) = \prod_{i=1}^{t} \sigma(w \cdot c_i) \quad , \tag{12}$$

and in log scale:

$$\log P(+|w, c_{1:t}) = \sum_{i=1}^{t} \log \sigma(w \cdot c_i) \quad \cdot \tag{13}$$

4.3. Skip-Gram with Negative Sampling

In Milkov et al. second paper [30] they introduced negative sampling as a solution for reducing computational complexity for learning word embedding vectors. The idea behind skip-gram with negative sampling (SGNS) is to calculate probability error in each training epoch only for real context words and k amount of randomly sampled non-context words, which are selected from training vocabulary [6 ch.6]. A suitable size of k depending amount of the training examples, with a smaller training data set the k is recommended to be a number between 5-20 [30]. In case there is a large training data set, it is enough that the value of k is a smaller number between 2-5 [30]. We can walk through how negative samples are generated by moving context-window, by exploring the following example, where we have the sentence " *I want to know more about NLP* " in training data. Let's determine that context window t is $\pm$ 2 and k = 2. We start moving the center word index at the sentence from the left and flow word by word to the right. In first training epoch center word w is "*I*", context words are w_{+1} = "*want*" and w_{+2} = "*to*". For each context word, we randomly select 2 non-context word, as follows

Positive examples:

t	c_{pos}	w
w_{+1}	want	I
w_{+2}	to	I

Negative examples:

c_{pos}	c_{neg1}	c_{neg2}
want	ai	more
to	more	like

Different context words can get the same randomly selected negative example word, but the negative example can not ever be the same word as some of the current context window [28]. In the fifth training epoch center word w is "**more**" and we again select context words around it and randomize negative examples, as follows

Positive examples:

t	c_{pos}	w
w_{-2}	to	more
w_{-1}	know	more
w_{+1}	about	more
w_{+2}	NLP	more

Negative examples:

c_{pos}	c_{neg1}	c_{neg2}
to	like	other
know	apple	car
about	ai	computer
NLP	banana	dog

4.4. Choosing Noise Words

Negative examples are randomly sampled with unigram probability, where more frequent words in vocabulary are selected more often to be c_{neg} than rare words [6

ch.6]. It is common to use weighted α when calculating unigram probability, which gives rare words a little bit more likely to get selected and leads to better performance [6 ch.6]. Weighted unigram probability is defined as follows

$$P_\alpha(w) = \frac{count(w)^\alpha}{\sum_{i=1}^{V} count(w_i)^\alpha} \quad , \tag{14}$$

where $count(w)$ is the number of how many times word w occurs in training data divided by total training data word count [6 ch.6]. Both the numerator and denominator are powered by weight α, which is usually 0.75 [6 ch.6].

4.5. Loss Function

The goal of the skip-gram algorithm is to adjust words embedding vectors by using loss function L, which tries to maximize dot product similarity of training pairs (w,c_{pos}) and minimize similarity of negative example pairs (w,c_{neg}) [6 ch.6]. In other words, we want that the real context words c_{pos} to be close to the center word w in embedding vector space and reduce noise words c_{neg} closeness by using loss function and stochastic gradient descent [6 ch.6].

According to [6 ch.6], in the SGNS case loss function can be determined as follows

$$\begin{aligned} L_{CE} &= -\log\left[P(+|w, c_{pos}) \prod_{i=1}^{k} P(+|w, c_{neg_i})\right] \\ &= -\left[\log P(+|w, c_{pos}) + \sum_{i=1}^{k} \log P(-|w, c_{neg_i})\right] \\ &= -\left[\log P(+|w, c_{pos}) + \sum_{i=1}^{k} \log(1 - P(+|w, c_{neg_i}))\right] \\ &= -\left[\log \sigma(w \cdot c_{pos}) + \sum_{i=1}^{k} \log \sigma(w \cdot -c_{neg_i})\right] \quad . \end{aligned} \tag{15}$$

Loss function calculates prediction error and it is good to notice how it behaves in extreme conditions. If our model give hypothetically 100% probability for $P(+|w, c_{pos})$ and 0% probability for every negative $P(+|w, c_{neg})$ sample, then $L_{CE} = -[\log(1) + k \cdot \log(1 - 0)]$, which equals prediction error $L_{CE} = 0$.

4.5.1. Gradient Descent Algorithm

To train machine learning models and neural networks, a commonly used way to achieve that is to use an optimization algorithm called gradient descent [33]. Gradient descent works hand-to-hand with the loss function, which calculates the error between the predicted output and the wanted output. By using loss function feedback about network error, it tries to modify parameters, so that the error is as small as possible [6 ch.6]. There are three types of gradient descent algorithm versions, which differ in

the way of updating weights [33]. A common solution in the Word2vec case is to use stochastic gradient descent (SGD), which uses each word in the corpus as a training epoch and updates center and context word embedding vectors after each epoch [6 ch.6]. With SGD training is less complex and more efficient from the computational point of view [33]. It can also help escape from the local minimum, where small changes to parameters do not reduce loss, which prevents finding a global minimum, where adjusted parameters give the smallest possible overall loss [33].

4.6. Stochastic Gradient Descent

The reason behind calculating the gradient for the network in each training epoch is to get information about how much each element affects prediction error by changing values at flow between input and output inside the neural network [33]. In the SGNS model, the only parameters that we need to adjust are context and center word embedding vectors, all other elements inside the network are just mathematical operations and their flowing output values [6 ch.6]. This network structure can be resented as a graph where different operations for variables are represented as boxes and connected to each other with lines.

Figure 7 shows an example of calculation operations in a situation where prediction error is calculated for real context word c_{pos}. In the case of calculating prediction error for non-real context words, i.e, noise word c_{neg}, the structure is otherwise the same but in noise word case $L_{CE} = -\log(1 - \sigma(y))$, instead of $L_{CE} = -\log(\sigma(y))$.

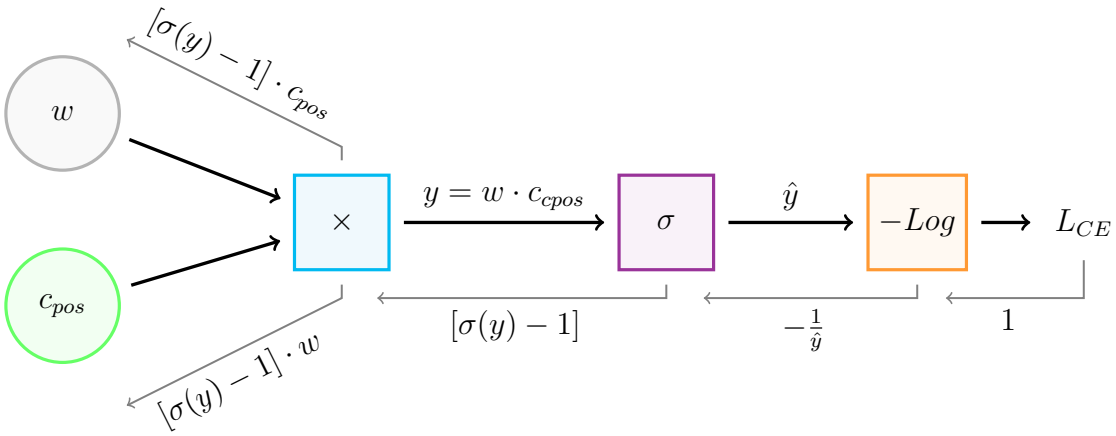

Figure 7. Graph example of the structure of calculating the loss function.

4.6.1. Partial Derivatives

Network gradient consists of partial derivatives, which are flowed back to the starting point in Figure 7 by grey arrows. This method is called backpropagation, which uses chain rule to backward pass network variables' partial derivatives with respect to loss function L_{CE} [33]. This way we can adjust effectively network parameters by mirroring their impact to computed prediction error i.e. loss [6 ch.6].

Because SGNS calculates prediction error in each training epoch only for real context word c_{pos} and noise words c_{neg} instead of all words in the vocabulary, we can define partial derivatives as follows:

$$\frac{\partial_{L_{CE}}}{\partial_{c_{pos}}} = [\sigma(w \cdot c_{pos}) - 1] \cdot w \quad , \tag{16}$$

$$\frac{\partial_{L_{CE}}}{\partial_{c_{neg}}} = [\sigma(w \cdot c_{neg})] \cdot w \quad , \tag{17}$$

where equations (16) and (17) show, partial derivatives for noise words c_{pos} and real context word c_{neg}.

The total partial derivative of w is the sum of its individual partial derivatives representing to prediction error of c_{pos} and noise c_{neg} noise words, which leads definition as follow:

$$\frac{\partial_{L_{CE}}}{\partial_{w}} = [\sigma(w \cdot c_{pos}) - 1] \cdot c_{pos} + \sum_{i=1}^{k} [\sigma(w \cdot c_{neg_k})] \cdot c_{neg_k} \quad . \tag{18}$$

4.7. Parameter Updating

After each training epoch, network parameters are adjusted to minimize future prediction error based it is effect of prediction error L_{CE}, which equals parameter partial derivative with respect to L_{CE} [6 ch.6]. SGNS model parameter updating is defined as follows:

$$c_{pos}^{t+1} = c_{pos}^{t} - \eta[\sigma(w^t \cdot c_{pos}^{t}) - 1] \cdot w^t \tag{19}$$

$$c_{neg}^{t+1} = c_{neg}^{t} - \eta[\sigma(w^t \cdot c_{neg}^{t})] \cdot w^t \tag{20}$$

$$w^{t+1} = w^t - \eta \left[[\sigma(w^t \cdot c_{pos}) - 1] \cdot c_{pos} + \sum_{i=1}^{k} [\sigma(w^t \cdot c_{neg_i})] \cdot c_{neg_i} \right] \quad , \tag{21}$$

where t represent time step and η learning rate [6 ch.6]. Intermediate stages of derivatives and the example of full training epoc with parameter updating can be found in the first appendix.

5. FASTTEXT

Even though previous Word2vec algorithm solution example and other similar versions of that can effectively capture the semantic similarity of words, they have certain deficiencies. Morphological-rich languages like Finnish, contain internal level word structure information, which is not utilized by those models. Another deficiency is their inability to handle out-of-vocabulary (OOV) words, which means that the network can not represent embedding vectors for those words, which are not in the training set. As a solution to rectify those deficiencies, four Facebook employees proposed improvements to the original Word2vec algorithm in three parts [34], [35], and [36]. Paper [35] focuses mainly on model compressing and ways to reduce memory demand, while papers [34] and [36] proposed architectural improvements, which we focus on in this chapter.

Joulin et al. found out that their proposed techniques beat almost all state-of-art solutions in sentiment accuracy and all of them clearly in training speed [34]. The only algorithm in the comparison group, which gave slightly better sentiment accuracy with certain data sets, was the very deep convolutional neural network (DCNN) proposed by Conneau et al. [37], but in single epoch training time comparison, FastText was at best even 1100 times faster than DCNN. They also found out that the FastText algorithm is also applicable to modified tag prediction task, where it reached reasonable accuracy for predicting correct tags into input text sentences [34].

5.1. Structure

In paper [36] Joulin et al. introduced the FastText algorithm structure, which is broadly similar to the previous chapter SGNS Word2vec version. The difference between SGNS and FastText comes in the way of handling input words and how to calculate the probability of word c to be context word for word w [36]. In the original Word2vec paper [30], the proposed classifier defined probability for words to be context word to specific word w by using the SoftMax function, defined as follows:

$$P(w|c) = \frac{exp(c \cdot w^T)}{\sum_{i=0}^{V} exp(c_i \cdot w^T)} \quad , \tag{22}$$

where w is the center word and c is the context word, which probability we want to know. In the denominator, the dot product between each word in the vocabulary and the center word w is summed up to normalize calculated probability. This solution does not consider probabilities as independent for each word, because those probabilities add up to 1, which can limit the learning of word embeddings.

Previous chapter SGNS example used the sigmoid function to calculate probabilities instead of SoftMax, which way probabilities were considered as independent for each word. Paper [36] proposed the following solution to calculate probabilities for real context words C_{pos} :

$$P(+|w, c) = 1 + e^{-s(c_{pos}, w)} \tag{23}$$

and negative samples c_{neg} :

$$P(-|w,c) = 1 + e^{s(c_{neg},w)} \quad . \tag{24}$$

Those modifications give us a slightly simpler loss function compared to SGNS, which can be defined as follows:

$$L_{CE} = \Big[\log(1 + e^{-s(c_{pos},w)}) + \sum_{i=0}^{k} \log(1 + e^{s(c_{neg},w)}) \Big] \quad , \tag{25}$$

where again k is amount of negatives samples for each positive c_{pos} example and s represent scoring function, which will be reviewed later.

5.1.1. N-Gram

Another structural change compared to SGNS related into the input layer and first embedding matrix $\mathbf{W}1$. Previous Word2vec algorithm solutions lack to utilization of words' internal structure, which Joulin et al. wanted to reform by using words n-grams to capture words' internal structure. N-gram representation of word splits word to a bag of characters the size of N, which we further represent as lower case n. In the FastText algorithm, the size of n is typically between 3-6, and the lower or upper size of n does not capture words' internal information as precisely [36]. A common way is also to add special boundary symbols < and > for at the beginning and end of each word to distinguish words start and ending from other character sequences.

Let's take an example where n = 4 and the word is the Finnish word "paloauto", which corresponds to a fire engine in English. We add special boundary symbols beginning and end of the word "paloauto" and split it into a bag of characters size of 4:

$$\Big[< pal,\ \ palo,\ \ aloa,\ \ loau,\ \ oaut,\ \ auto,\ \ uto >,\ \ < paloauto > \Big] \quad .$$

Notice that the 4-gram representation of a word in the FastText case also contains a special sequence $< paloauto >$, which way the model learns to represent each word by the sum of its n-grams and the word itself. The reason to add n-grams to architecture can be visualized by looking at the words and their n-grams, for example, the word "paloauto" is a combination of two Finnish words "palo" and "auto", which equals in English "fire" and "car". This way model does not represent the word "paloauto" as a completely different word than "auto" or "palo", because it also contains those words as n-gram information. However, it does not mean that 4-gram $auto$ is equal to special word sequence $< auto >$, which represents word $auto$ itself.

5.1.2. Scoring Function

Compared to SGNS, where scoring function s is word context word c_{neg} or c_{pos} dot product with center word w, we need a little modification to include n-grams for that.

According to Joulin et al. in a paper [36], the FastText scoring function s is defined as follows:

$$s(w, c_{pos/neg}) = \sum_{g \in G_w} z_g^T v_c \quad , \tag{26}$$

where g is an index of G_w, which represents a bag of center word w n-grams embedding vectors and v_c is an embedding vector of context word c. This can be simplified for future examples by using the vector calculus rule: $A \cdot C + B \cdot C = (A + B) \cdot C$, which leads scoring function:

$$s(w, c_{pos/neg}) = \Big[\sum_{g \in G_w} z_g^T \Big] \cdot v_c \quad , \tag{27}$$

where all n-gram embedding vectors of center word w are added together and then multiplied with context word c embedding vector.

5.1.3. Embedding Matrix

Adding words n-gram representations to the model demands also modification to the embedding matrix. In previous Word2vec algorithm examples, we had two embedding matrices $\mathbf{W}1$ and $\mathbf{W}2$, which both were the size of $|V| \times N$, where $|V|$ represents the number of unique words in training data and N length of each word w embedding vector. FastText algorithm case $\mathbf{W}2$ matrix is again the size of $|V| \cdot N$ and it contains each word embedding vector as a context word. Embedding matrix $\mathbf{W}1$ contains each word special sequences $< word >$, which equals to word w embedding vector as center word and also all possible n size n-grams embedding vectors. This leads to embedding matrix $\mathbf{W}1$ size of $(|V| + l^n) \times N$, where n is the size of n-gram (n is not equal to embedding vector size N) and l means how many unique letter vocabulary contains. Including n-grams into embedding matrix $\mathbf{W}1$ increases its size, but it also creates other benefits than just capturing words' internal structure. We can represent OOV words by looking at which n-grams that word contains and adding its n-gram embedding vectors together, which was not possible when $\mathbf{W}1$ contained only words embedding vectors.

5.2. Partial Derivatives

Similar to most other Word2vec algorithm versions, FastText uses stochastic gradient descent (SGD) to train embedding vectors, where corresponding embedding vectors are adjusted after each training epoch.

In Figure 8, we again can represent algorithm operations by using a graph, where the size of n = 3 and the center word w is "nlp". At first, we sum all center word-related embedding vectors from matrix $\mathbf{W}1$ together to get a vector size of $1 \times N$ and represent that by symbol w. We get the aforementioned vector by multiplying embedding matrix $\mathbf{W}1$ with one-hot encoded input vector $[0_0, 1_1, 1_2, 0_3 \cdots 1_{900} \cdots 0_{|V|+l^n}]$, where corresponding indexes of word w n-grams and <word> itself is 1, all others index

locations are 0. Symbol c represent context word c_{pos} or c_{neg} from embedding matrix $\mathbf{W2}$, in case of c_{pos} we will use $1 + e^{-y}$ and in noise word case c_{neg} we use $1 + e^{y}$.

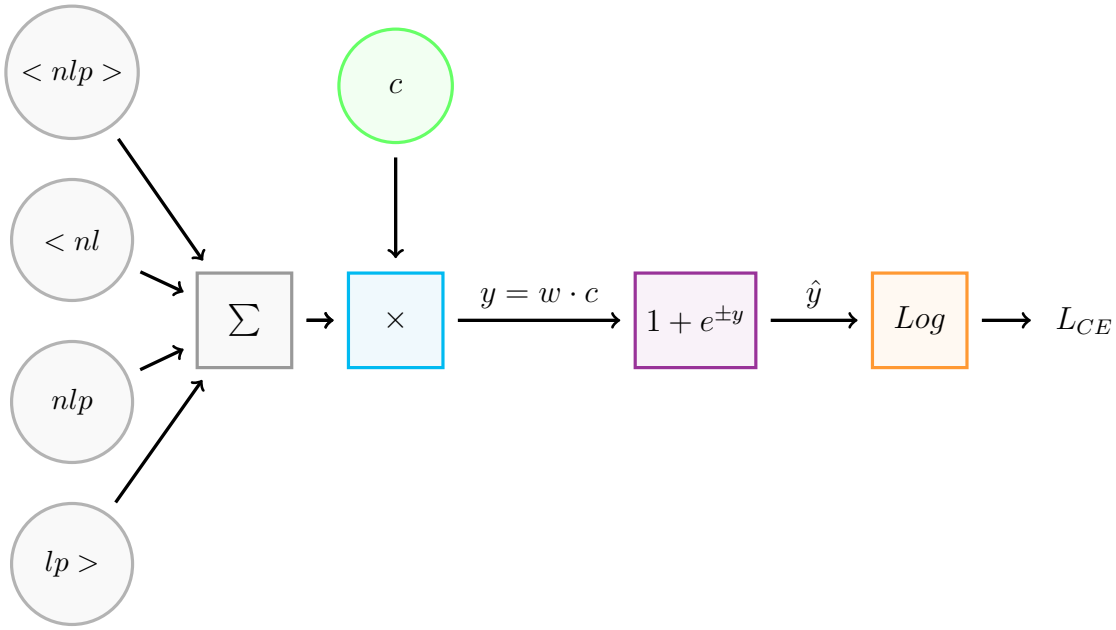

Figure 8. Graph example of calculating the loss function of FastText structure.

As in the SGNS case, we calculate partial derivatives to adjust parameters and use the chain rule to get know how much each parameter impacts to model loss L_{CE}. Partial derivatives are:

$$\frac{\partial_{L_{CE}}}{\partial_{L_{CE}}} = 1 \tag{28}$$

$$\frac{\partial_{L_{CE}}}{\partial_{\hat{\mathbf{y}}}} = \frac{d_{L_{CE}}}{d_{\hat{\mathbf{y}}}}\Big[\log(\hat{\mathbf{y}})\Big] \cdot \frac{\partial_{L_{CE}}}{\partial_{L_{CE}}} = \frac{1}{\hat{\mathbf{y}}} \cdot 1 = \frac{1}{\hat{\mathbf{y}}} \quad , \tag{29}$$

where $\hat{\mathbf{y}} = 1 + e^{\pm y}$. We again get slightly different partial derivatives for noise and real content word c. For real context word c_{pos} we get:

$$\frac{\partial_{L_{CE}}}{\partial_{y}} = \frac{d_{\hat{\mathbf{y}}}}{d_{y}}\Big[1 + e^{-y}\Big] \cdot \frac{\partial_{L_{CE}}}{\partial_{\hat{\mathbf{y}}}} = -e^{-y} \cdot \frac{1}{1 + e^{-y}} = \frac{-1}{1 + e^{y}} \tag{30}$$

$$\frac{\partial_{L_{CE}}}{\partial_{c_{pos}}} = \frac{d_{y}}{d_{c_{pos}}}\Big[c \cdot w\Big] \cdot \frac{\partial_{L_{CE}}}{\partial_{y}} = w \cdot \frac{-1}{1 + e^{y}} = \frac{-w}{1 + e^{c_{pos} \cdot w}} \tag{31}$$

$$\frac{\partial_{L_{CE}}}{\partial_{w}} = \frac{d_{y}}{d_{w}}\Big[c \cdot w\Big] \cdot \frac{\partial_{L_{CE}}}{\partial_{y}} = c_{pos} \cdot \frac{-1}{1 + e^{y}} = \frac{-c_{pos}}{1 + e^{c_{pos} \cdot w}} \quad , \tag{32}$$

and for noise word c_{neg} we get:

$$\frac{\partial_{L_{CE}}}{\partial_{y}} = \frac{d_{\hat{\mathbf{y}}}}{d_{y}}\Big[1 + e^{y}\Big] \cdot \frac{\partial_{L_{CE}}}{\partial_{\hat{\mathbf{y}}}} = e^{y} \cdot \frac{1}{1 + e^{y}} = \frac{1}{1 + e^{-y}} \tag{33}$$

$$\frac{\partial_{L_{CE}}}{\partial_{c_{neg}}} = \frac{d_y}{d_{c_{neg}}}\Big[c \cdot w\Big] \cdot \frac{\partial_{L_{CE}}}{\partial_y} = w \cdot \frac{1}{1+e^{-y}} = \frac{w}{1+e^{-c_{neg} \cdot w}} \tag{34}$$

$$\frac{\partial_{L_{CE}}}{\partial_w} = \frac{d_y}{d_w}\Big[c \cdot w\Big] \cdot \frac{\partial_{L_{CE}}}{\partial_y} = c_{neg} \cdot \frac{1}{1+e^{-y}} = \frac{c_{neg}}{1+e^{-c_{neg} \cdot w}} \quad . \tag{35}$$

We used the symbol w to represent the sum of n-grams and center word embedding vectors. Partial derivatives of those corresponding n-grams and center word are equal to the partial derivative of w with respect to a calculated loss L_{CE}, because:

$$\frac{\partial_{L_{CE}}}{\partial_{n_1}} = \frac{d_w}{d_{n_1}}\Big[n_1 + \cdots + n_G + v_{word}\Big] \cdot \frac{\partial_{L_{CE}}}{\partial_w} = 1 \cdot \frac{\pm c_{neg/pos}}{1+e^{\pm y}} \quad , \tag{36}$$

where n represent n-gram embedding vector in set $[1 \cdots G]$, which contains all n-grams of center word w and v_{word} is embedding vector of center word itself.

5.3. Parameter Updating

In FastText network parameters can be adjusted the same way as in SGNS after each training epoch to minimize future prediction error. This time, the parameters that we need to adjust are context word c_{pos}, c_{neg}, and all n-grams, which the center word contains in addition to the center word itself. This leads following embedding vector adjustment:

$$c_{pos}^{t+1} = c_{pos}^{t} - \eta\left[\frac{-w^t}{1+e^{c_{pos}^t \cdot w^t}}\right] \tag{37}$$

$$c_{neg}^{t+1} = c_{neg}^{t} - \eta\left[\frac{w^t}{1+e^{-c_{pos}^t \cdot w^t}}\right] \tag{38}$$

$$w^{t+1} = w^t - \eta\left[\frac{-c_{pos}^t}{1+e^{c_{pos}^t \cdot w^t}} + \sum_{i=1}^{k} \frac{c_{neg_i}^t}{1+e^{-c_{neg_i}^t \cdot w^t}}\right] \quad , \tag{39}$$

where t represents time step and η learning rate. Adjustment w^{t+1} will be done center word embedding vector and all its contained n-gram embedding vectors in $\mathbf{W}1$ matrix.

6. IMPLEMENTATION AND ANALYSIS

To analyze and learn possible usage possibilities of word embeddings, specific Word2vec and FastText versions were trained and in this chapter, the intermediate stages and outcomes of the experimentation are presented.

6.1. Training Data and Preprocessing

Regardless of what type of NLP model is wanted to train, one of the most important individual element is training data. That's why trained example models used the Finnish Wikipedia as a training data set because it contains text from widely different categories, which minimizes OOV words. Different language Wikipedia versions are easily available and downloadable as dumps, generated by the Wikimedia Foundation (The latest Finnish version [38]). Those dumps contain every Wikipedia page in a raw text format for a specified language. As seen in Figure 9, dumps contain a lot of additional data like URLs and tags, which need to be removed before training.

```
Azerbaidžan osallistui ensimmäisen kerran [[Eurovision laulukilpailu]]ihin vuonna [[Eurovision laulukilpailu 2008|2008
]]&lt;ref name=&quot;eurovision&quot;&gt;http://www.esctoday.com/news/read/9443&lt;/ref&gt; ja voitti ne ensimmäisen k
erran vuonna [[Eurovision laulukilpailu 2011|2011]]&lt;ref&gt;[http://satumaa.yle.fi/euroviisut Azerbaidžanin presiden
tti nimitti Euroviisujen järjestelykomitean] {{Wayback|1=http://satumaa.yle.fi/euroviisut |päiväys=20110428191816 }} Y
LE Satumaa&lt;/ref&gt;.

=== Juhlapäivät ===
{| border=&quot;1&quot; cellpadding=&quot;2&quot; cellspacing=&quot;0&quot; style=&quot;margin: 0 auto; border-collaps
e: collapse; border: 1xp solid #aaa;&quot;
|-
! style=&quot;background-color: #efefef;&quot; | Pvm&lt;ref name=&quot;AZ-holiday&quot;/&gt;!! style=&quot;background-
color: #efefef;&quot; | Suomalainen nimi !! style=&quot;background-color: #efefef;&quot; | Azeriksi&lt;ref name=&quot;
AZ-holi-presi&quot;&gt;{{Verkkoviite | Osoite=http://en.president.az/azerbaijan/holidays| Nimeke= The Labor Code of th
e Republic of Azerbaijan Article 105. Holidays| Julkaisija = president.az | Viitattu=30.12.2014}}&lt;/ref&gt; !! style
=&quot;background-color: #efefef;&quot; | Huomautuksia
|-
| 1.&amp;ndash;2. tammikuuta || uusivuosi ||  || &amp;nbsp;
|-
| 8. maaliskuuta || [[kansainvälinen naistenpäivä]] ||  || &amp;nbsp;
|-
| noin 20.&amp;ndash;24. maaliskuuta || [[Novruz]] (”Uuden päivän” juhla) || Novruz Bayramı || kevätpäiväntasauksesta
alkava viisipäiväinen juhla
```

Figure 9. Example clip of Wikipedia dump file.

To get rid of tags, non-alphabet characters etc., specific Python code was written to generate a processed training data set of 89 million total words, which was small compared to selected comparison versions, of which for example Turku NLP Group Word2vec version was trained with 4.5 billion words [39].

6.2. Training Models

To train the traditional or FastText version of the Word2vec model, there are many downloadable code package solutions, in which training time is fast and optimized. Both traditional skip-gram and FastText versions of the Word2vec model with two different embedding vector sizes to compare performance and functional differences were trained. The FastText models were trained by using the FastText API package [40] and the Skip-gram models by using Gensim [41].

6.2.1. Training Parameters

Gensim and FastText API contains several parameter options to train the model, which can be modified according to need. Those parameters are for example: learning rate, structure: CBOW vs Skip-gram, context window size, amount of negative samples, length of embedding vectors etc. and in FastText case you can also modify min and max n-gram size.

The trained models used embedding vector sizes 300 and 75, context window size 6, Skip-gram structure, 6 negative samples for each C_{pos} and minimum word count 2, other parameter options were kept as default. Of the above, especially the minimum word count is convenient, because the data parsing phase often leaves some garbage inside data, which can be removed by setting a threshold value to 2 for the minimum appearance of a word in the vocabulary.

6.3. Evaluating Models

A common evaluation method for word embedding vectors is to use test sets like SimVerb or SimLex, which calculates word pair similarities and compares the result to the correct value. Some of those test sets have translated Finnish language versions, but a different approach in evaluation was wanted. Instead of calculating correct word pair similarity, a test scenario, where models have to contain correct synonyms or semantically similar words in top N most similar words result for the input word, was made. This way, it does not matter, if the input word dot product is enough high with its synonym or semantically similar word if it is not one of the top N most similar words. The total amount of word pairs in the test scenario was 200, where 75 were testing synonyms and 125 semantically similar word detection. In addition to four trained models, FastText and Word2vec skip-gram with 300 and 75-dimensional word embedding vectors, there were also two comparison models. Both comparison models Facebook FastText [40] and Word2vec [39] used 300-dimensional word embedding vectors.

6.3.1. Evaluation Set

The generated evaluation test set to measure model performance contained some easier and some trickier word pairs. There were, for example, several word pairs in both synonym and semantic evaluation test sets, which none of the models could identify. However, there were also many evaluation test pairs, which all the models could identify correctly without a problem.

Some of the evaluation test pairs, which none of the models could identify were surprising because those are commonly used in language. Those word pairs, were for example, synonym pairs: ("**loma**","**vapaa**") = "**holiday**" , ("**matematiikka**", "**matiikka**") = "**math**" and ("**bugi**", "**vika**") = "**bug**" or semantically similar pairs like ("**kahvi = coffee**", "**tee = tea**") and ("**kello = watch**", "**koru = jewerly**").

The main reason for that, in the case of trained models, may be the lack of different types of training examples, but in the case of comparison models, accurate speculation

is not possible, because specific information about training examples is not available. For example, many words can have two or more different meanings in language and be synonyms with each other only by one meaning. Those connections are not usually visible in a standard language, in which way, for example, Wikipedia is mainly written.

The evaluation test set included also trickier evaluation test pairs, which were mainly semantically similar. Those pairs had clear semantic connections to each other, but the purpose was to test, which type of semantic similarity the models could identify and could those models identify multiple types of semantic similarity. For example, Finnish word "**lehti**" can mean in English word "**leaf**" or "**magazine**", due to which evaluation test set contains pairs ("**lehti = leaf**", "**puu = tree**") and ("**lehti = magazine**", "**sanomalehti = news paper**") . Some of the models identify semantic connection between "**lehti**" and "**sanomalehti**", but none of them between "**lehti**" and "**puu**". Also connections like ("**paloauto = fire truck**", "**palomies = fireman**") were for models too difficult to identify and most similar words for input "**paloauto**" were another type of vehicles like "**ambulanssi = ambulance**". The full list of evaluation set pairs is available in [42].

6.3.2. Evaluating Result

Table 1 shows that the Word2vec comparison model outperform in both evaluation sets clearly and it gave almost every time really good or reasonable results into input words in a synonym or semantically similarity point of view. In synonym testing Facebook FastText model gave the second-best performance, but in semantic similarity testing Word2vec skip-gram model trained in this work with 75-dimensional word embedding vectors outperformed the Facebook model and got the second-best performance.

Table 1. Evaluation result table of accuracy in percent

Test set and version	Synonym N=5	Semantic N=10
W2v Turku NLP	**57.33**	**56.00**
W2v D75	21.33	48.00
W2v D300	22.67	40.00
FastText Facebook	33.33	44.00
FastText D75	10.67	21.60
FastText D300	10.67	13.60

Both FastText models with 300 and 75-dimensional embedding vectors, which were trained by using the same training data set as Word2vec skip-gram models, performed quite badly. In general, training embedding vectors with smaller training data sets like in this thesis, smaller dimensional word embedding vectors usually perform better than higher dimensional like 300.

6.3.3. Note on Evaluation

To evaluate the models, there are general things related to training data, which need to be taken into account. For example in the Finnish language **anna** means **give** in English, but **Anna** means name in both languages. If you change all characters in training data to lowercase, the model considers "anna" and "Anna" as the same word. But in case you do not lowercase them, the model considers some of the same words as two different words, because every sentence starts with a word with an uppercase character. Another case is punctuation marks because in Wikipedia text file contains many punctuation marks between words, which are not compound words. Punctuation marks were chosen to be replaced with spaces in training data, but this decision affected the models, in a way that those could not recognize compound words.

6.3.4. Summary of Evaluation

Even though the trained FastText models, performed badly compared to others, there were common synonyms or semantically similar words, which all models captured as the most similar word. For example, all models detect semantically similar color words like red - blue or numbers five - six easily. Those semantic connections are easily recognizable for humans, but clearly also strongly visible in training data.

In evaluation, there were also some cases in which some models detected the correct target synonym as the top 6 results, but threshold values were decided to be kept in 5 as synonym evaluation and 10 in semantic case. Because synonym or semantic similarity word pairs, which were selected for the evaluation set, contained mainly clear connections, which are easily identifiable for humans.

6.4. Visualization of Models

From trained models, the Word2vec model with a 75-dimensional embedding vector, was selected to be used in visual analysis, because it performed best in comparison to other models in both evaluation tests. To visualize word embeddings, used method was principal component analysis (PCA), which is directly available as a function in for example in scikit-learn software library [43]. With PCA, selected 75-dimensional word embedding vectors were reduced into 2-dimensional to see how certain words distribute in embedding space as shown in Figure 10.

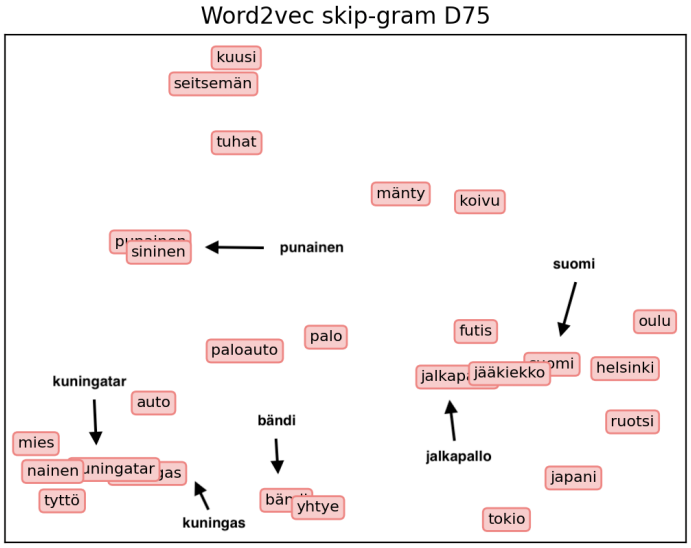

Figure 10. Visualization of Word2vec skip-gram model.

There can be seen some overlapping between words, which are synonyms together or semantically similar. For the visualization, there were also purposely chosen a couple of tricky words to see how they are located compared to each other in vector space. For example, the Finnish word **"kuusi"** means the number 6 and also specific wood. In Figure 10, there is two other wood **"mänty"** and **"koivu"**, which are correctly close to each other, but **"kuusi"** is further away, because the model keeps it more similar with words **"seitsemän = 7"** and **"tuhat = 1000"**. Also in the wood case "kuusi" is closer to the word "jääkiekko = hockey" than "mänty", which can be due to the successful finnish hockey player brothers Mikko and Saku Koivu. Overlapped words in the Figure 10 were expected, because they have a strong connection to each other. For example **"bändi"** and **"yhtye"** are synonyms together for English word band. Also words like **"kuningas = king"** and **"kuningatar = queen"** are semantically so similar, that they were expected to be really close to each other.

Figure 11 shows another interesting observation which was made by visualizing the same model. The evaluation test set contained a word pair ("**lasku**", "**yhtälö = equation**"), where the word "**lasku**" has three different semantic connections. It has strong semantic connections in Finnish to words like "**matematiikka = math**", "**yhtälö = equation**", "**mäki = hill**", "**rinne = slope**", "**osto = purchase**" and "**kauppa = deal**", but none of those were among top ten most semantically similar with the word "**lasku**". The observations led to consideration, of whether could it be possible that, because of its semantic connection to three different topic groups, those different groups are pulling the word "**lasku**" near them alternately. This may be true on some level because the word "**lasku**" is located almost in the middle of those three different groups in Figure 11.

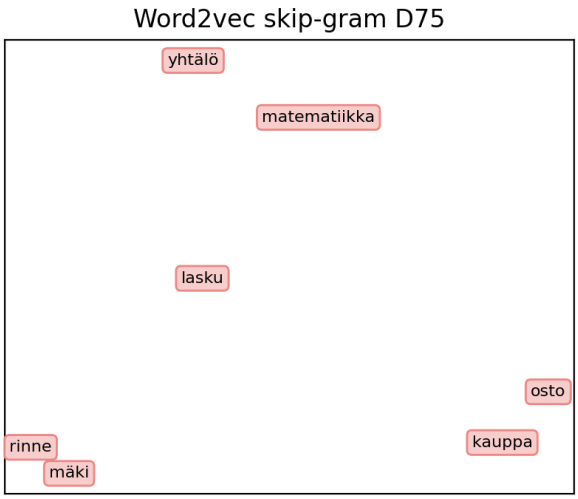

Figure 11. Visualization of Word2vec skip-gram model.

6.5. Analogy Operations

As mentioned on page 17, word embedding vectors are capable of analogy operations, where the models try to find the most similar word embedding vector by using simple +- operations and cosine similarity. The aforementioned type of testing was done with the Word2vec skip-gram D75 model, which shows that in some cases it works fine, but the same structure usually does not work in multiple cases.

Table 2 shows some examples, that the model succeeded in identifying correctly or poorly.

Table 2. Analogy testing examples

Correctly	**Poorly**
mies + tyttö - poika = nainen	sisko + poika - tyttö = pojanpoika
suomi + obama - niinistö = usa	italia + audi - ferrari = unkari
helsinki + japani - suomi = tokio	kuningas + nainen - mies = kuninkaan
jalkapallo + selänne - litmanen = jääkiekko	isä + nainen - mies = isoisä

Table 3 on the other hand shows, some examples, of where the structure works in certain cases, but not after changing a word or words.

Table 3. Analogy testing examples

Correct	**Not correct**
laulaja + dicaprio - shakira = näyttelijä	laulaja + dicaprio - adele = kehonrakentaja
saksa + monarkia - demokratia = alankomaat	suomi + monarkia - demokratia = tanskalle
usa + nokia - apple = suomi	suomi + apple - nokia = esityskielen

To test the analogy some cases, word pairs do not even need to be related to each other. For example, model gave the correct answer to this analogy:

neliö + mies - kolmio = nainen

where the word "**mies** = **man**" as well "**nainen** = **women**" are sexs and word "**neliö** = **square**" as well "**kolmio** = **triangle**" are shapes.

6.6. Observations

Major observation in the evaluations was the importance of training data amount and its quality. It is possible to train the model with a comparatively small training data set, which works well in clear and narrow topic areas, but if you want generally good performance, you need a big training data set. Also, experience due to testing four trained and two pre-trained models is that architectural difference (FastText vs. Traditional Word2vec) is emphasized when the size of training data is smaller. The performance difference between Facebook FastText and Turku NLP Word2vec model was a lot smaller and those work really similarly compared to the performance between two architecturally different trained models.

Trained Word2vec skip-gram models, worked similarly to comparison models, but models, based on FastText architecture, worked completely differently in some cases. FastText models, which were trained perform quite badly in synonym and semantic similarity recognition, but those models recognize quite well for example inflectional forms for input verbs. As example, input word **"juosta = run"** with other models give verbs like **"kävellä = walk"** or **"hypätä = jump"** as result, which are semantically similar words, while results of trained FastText models, were inflections like **"juosten"**, **"juostiin"**, and **"juoksemalla"**. However, that type of behavior looks to changes when the training data set is enough big or it contains in addition other types of text than Wikipedia articles. This can be seen by comparing trained model outputs for inputs like **"juosta"** with comparison models, which were trained by using data from several sources.

7. WORD EMBEDDINGS IN SEARCH APPLICATIONS

Using word embeddings in a search engine environment, where similar types of information are sought, has many possibilities to get alternative results, if the user query does not match any good indexed information. Next, we will get through a couple of scenarios, how to possibly use those word embeddings to do that by using visual examples of Google search results and word embedding operations of model **Word2vec skip-gram D75**. To remark, Google uses LM called BERT [23] in their search engine to give better search results. BERT differs from the Word2vec model from a word embedding point of view, in that instead of having a single embedding vector for each word, it has several to capture word meaning in different contexts. As mentioned on page 34 the problem of that word "lasku" has an individual meaning in three different contexts and the purpose of BERT is to capture all of them.

An easy way to visualize the possible use of word embeddings to get alternative search results is to consider the case that you are buying a new wallet. In Finnish, there are a couple of commonly used words for a wallet, which are "**lompakko**" and "**rahapussi**". As you can see in Figure 12, the user query is "**ruskea rahapussi**", but the search result uses the word "**rahapussi**" synonym "**lompakko**" instead of it. Models, which are trained with large data sets, have an easy task to recognize the word "**lompakko**" as one of the most similar words for input "**rahapussi**" and search engines can use those most similar words as alternative search words.

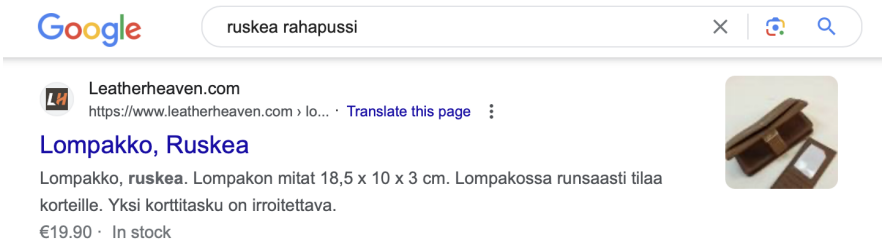

Figure 12. Example search from a synonym point of view.

Figure 13 shows an example of a semantic point of view.

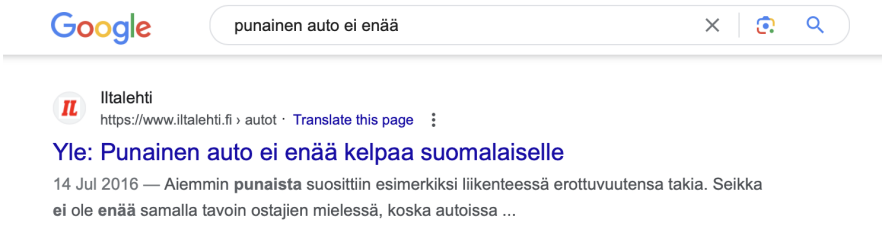

Figure 13. Example search from a semantic point of view.

Let's consider the case that the user remembers the same news title partly, which occurs in Figure 13, but the user remembers the color wrong. In that case, user search by query "**keltainen auto ei enää**", but Google does not find the same news anymore. There are not even matching results for the part sentence "**ei enää**" in the top result

for query "**keltainen auto ei enää**" as Figure 14 shows. In this type of case, where a perfect result fitting into the query is not found, could be the potential use case for finding alternative search results by using word embedding vector similarity. For a model, detecting the correct color in this case "**punainen**" is an easy task, when the input is "**keltainen**", because those colors are semantically so similar.

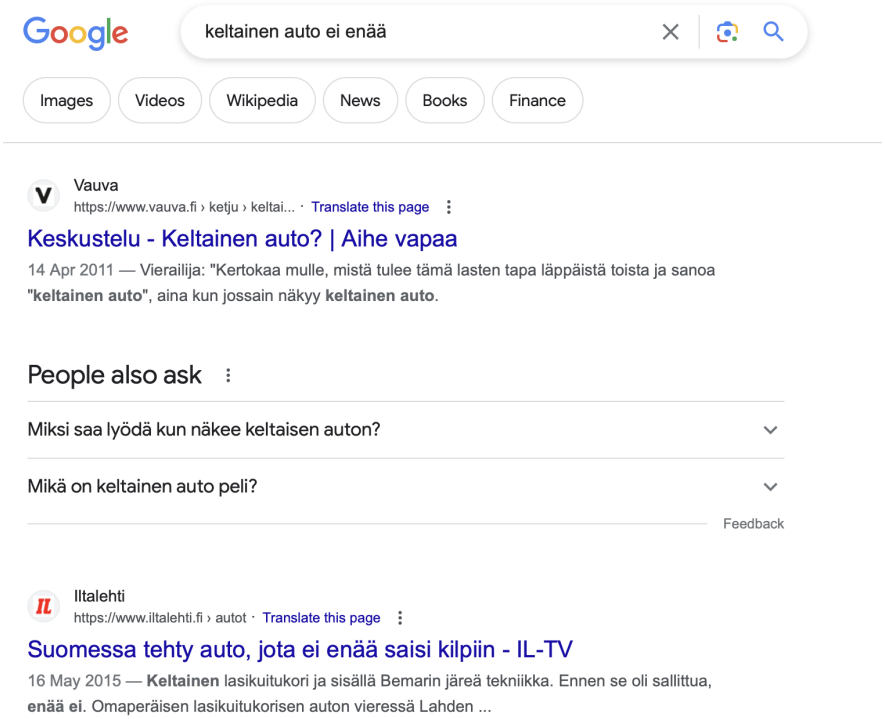

Figure 14. Comparison search result for semantically similar query.

Another example is to use sentence comparison by using word embeddings, which for example Gensim API [41] enables. For this, we can use an example, where someone is seeking information about the leader of Canada. There is a possibility that some people mistakenly thinks that Canada's leader is the President instead of the Prime minister. Naturally, there is not any indexed information about Canada's President, which matches the sentence "**kanadan presidentti = The President of Canada**". We again can use word embeddings to detect the most similar words to input "**presidentti = president**", where the top 3 results in a model **Word2vec skip-gram D75** are: "**presidentin = presidential**", "**varapresidentti = vice president**" and "**pääministeri = prime minister**". So in this case correct target prime minister is the third most similar word, but if we calculate sentence similarities, where "**presidentti**" is changed for alternatively most similar words, the sentence "**kanadan pääministeri = Canada's prime minister**" gets the highest similarity score. **Similairty scores**: "kanadan presidentin = 0.2199", "kanadan varapresidentti = 0.2367" , "kanadan pääministeri = **0.2553**" for comparison sentence "kanadan presidentti".

8. DISCUSSION

As mentioned in the previous section, the amount of training data has a big impact on that how well the models work. However, each of the models showed comparatively good performance to a limited extent and were able to recognize synonyms and semantically similar words. The amount of training data used in the models, which were trained, was small compared to the comparison models, although it contains almost a hundred million words in total. The functional differences due to a smaller amount of training data were especially visible for synonyms and semantically similar words, which are not as clear as, for example, the semantically strongly connected words like **"men"** and **"women"**. However, the clearest difference to the comparison models was highlighted when looking at the ten most similar words given by the models to the inputs, which often contained words that have no clear semantic or synonymous connection with the input word. The comparison models, especially the Word2vec version [39], succeeded very well and the ten most similar words of the input words almost always contained a clear semantic connection with the input word, if it was not a synonym of the word.

Also, character-specific choices made in the preprocessing and parsing of the training data, for example removing punctuation marks, had clear effects on the functioning of the models. As a result, data parsing and preprocessing can be considered to be the most challenging step because it is difficult to estimate in advance how certain choices in the preprocessing step affect the final model. This is not as critical if you are using "clean" training data that does not contain non-normal notation methods for words and other extras, such as Wikipedia's text data, which needs those notation methods for the correct creation of articles on the web page. The sources of the teaching data also matter, because naturally, for example, Wikipedia articles and the news are written in different text styles, which affects the final functionality of the different models. This was noticeable for the comparison models that also contained another type of training data than the Finnish language Wikipedia articles. In further studies, the intention is to investigate more precisely the effect of the preprocessing phase decisions and other parameters such as context window size on the functioning of the models. In addition to this, another tendency of further research is also to investigate the effect of different types of training data on the model's functionality and more precisely possible use cases for those models.

For studying the effect of training data style on the functioning of models, a potential and interesting way could be pre-editing the training data to mimic different text styles. This means that different versions of the training data are created, where the training data is rewritten to mimic for example news article style by using ChatGPT or another generative model, in which case the order of words or some of the words will change in sentences. The potential uses for the Word2vec and FastText models are in practically almost any application, where similar information is looked for based on text or it is wanted to be classified. Depending on the use case, there are often more powerful solutions, that are specified for that particular use case, but the advantage of Word2vec and FastText models is their fast and easy training. One possible application area, that could be interesting to study more in the future is the possibility of using FastText architecture for word error correction, where character level n-grams are used to detect the most similar word from vocabulary if the input word is not found in it.

9. SUMMARY

The main focus area of this thesis was word embeddings and their usage possibilities for synonym and semantically similar word detection in the Finnish language. As the results show, it is possible to train a model, that performs well in this type of task, but it also contains limitations. Some of those limitations could possibly be remedied by adding specific types of training examples into training data, but others are complex to fix. For example in cases, where the same word has multiple meanings in the language. In some situations, there could be possible benefits of using word embeddings to detect word synonyms or semantically similar words in search engine cases, if the perfectly fitting match is not found by using a search query. Those possible benefits can be seen by changing words in search queries for synonyms or other semantically similar words and comparing those search results. Another point of view could be to train models to find the most similar historical search queries, which to use as alternative queries and compare those search results.

The evaluation also showed the importance of the amount and quality of training data, which, however, do not guarantee generally great performance in all topic areas at least in the case of the Finnish language. The thesis introduces some of the application areas, where word and sentence information can be used and recognizing words synonyms or semantically similar words, have various application possibilities of areas like search engines, which seek similar types of information from data sets. There are also other interesting discovered use cases for word embedding, which have been trained by using the Word2vec algorithm or its variations. This type of possible use purposes are for example word and phrase translation [44] and analyzing change of words wider public meaning over different points of time [45].

One of this thesis work's additional purposes was to give a comprehensively better understanding of natural language processing strengths and weaknesses in the word embedding area for readers regardless of background knowledge of the topic and also motivate them to do their own research about the topic.

10. REFERENCES

[1] What is GPT? URL:https://aws.amazon.com/what-is/gpt/. Accessed 24.04.2024.

[2] ChatGPT. URL:https://chat.openai.com/auth/login. Accessed 24.04.2024.

[3] Dall.E 3. URL:https://openai.com/dall-e-3. Accessed 24.04.2024.

[4] Getting ready for artificial general intelligence with examples. URL:https://www.ibm.com/blog/artificial-general-intelligence-examples/. Accessed 24.04.2024.

[5] What are Embeddings? URL:https://learn.microsoft.com/en-us/semantic-kernel/memories/embeddings. Accessed 24.04.2024.

[6] Jurafsky D. & Martin J. (2023) Speech and Language Processing (3rd ed. draft). Book (Online).

[7] Vectors in Azure AI Search. URL:https://learn.microsoft.com/en-us/azure/search/vector-search-overview. Accessed 24.04.2024.

[8] Russel S. & Norvig P. (2021) Artificial Intelligence, A Modern Approach, Fourth Edition. Pearson Education Limited, 1161 p.

[9] Natural Language Processing (Wikipedia). URL:https://en.wikipedia.org/wiki/Natural_language_processing. Accessed 20.11.2023.

[10] What is natural language processing? (IBM). URL:https://www.ibm.com/topics/natural-language-processing. Accessed 1.11.2023.

[11] Jones Spark K. (1994), Natural language processing: A historical review. URL:https://dmice.ohsu.edu/bedricks/courses/cs662/pdf/sparck_jones_1994.pdf. Accessed 5.11.2023.

[12] A Brief History of Natural Language Processing — Part 1 (MEDIUM)). URL:https://www.ibm.com/topics/natural-language-processing. Accessed 1.11.2023.

[13] Cortes C. & Vapnik V. (1995) Support-vector networks. Springer 20, pp. 273–297. DOI: https://doi.org/10.1007/BF00994018.

[14] Vajjala S., Majumder B., Gupta A. & Surana H. (2020) Practical Natural Language Processing. O'Reilly Media, 454 p.

[15] LeCun Y., Bengio Y. & Hinton G. (2015) Deep learning. Nature 521, pp. 436–444. DOI: https://doi.org/10.1038/nature14539.

[16] McCulloch W. & Pitts W. (1943), A logical calculus of the ideas immanent in nervous activity. DOI: https://doi.org/10.1007/BF02478259.

[17] Hardesty L., Explained: Neural networks (MIT News). URL:https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414. Accessed 3.12.2023.

[18] Bengio Y., Ducharme R., Vincent P. & Janvin C. (2003) A neural probabilistic language model. J. Mach. Learn. Res. 3, pp. 1137–1155. URL: https://api.semanticscholar.org/CorpusID:221275765.

[19] Krenker A., Bester J. & Kos A. (2011) Introduction to the Artificial Neural Networks. Accessed 10.1.2024.

[20] Hochreiter S. & Schmidhuber J. (1997) Long short-term memory. Neural computation 9, pp. 1735–80.

[21] Mikolov T., Karafiát M., Burget L., Cernocký J. & Khudanpur S. (2010) Recurrent neural network based language model. vol. 2, pp. 1045–1048.

[22] Vaswani A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez A., Kiser L. & Polosukhin I. (2017), Attention is all you need. DOI: https://doi.org/10.48550/arXiv.1706.03762.

[23] Devlin J., Chang M.W., Lee K. & Toutanova K. (2018), Bert: Pre-training of deep bidirectional transformers for language understanding. DOI: https://doi.org/10.48550/arXiv.1810.04805.

[24] Osgood C. E. S.G.J..T.P.H. (1957) The measurement of meaning. univer. Illinois Press .

[25] Fano R. (1961) Transmission of information. A Statistical Theory of Communication URL: https://mitpress.mit.edu/9780262561693/transmission-of-information/.

[26] Pennington J., Socher R. & Manning C. (2014) GloVe: Global vectors for word representation. In: A. Moschitti, B. Pang & W. Daelemans (eds.) Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Association for Computational Linguistics, Doha, Qatar, pp. 1532–1543. URL: https://aclanthology.org/D14-1162.

[27] Milkov T., Chen K., Corrado G. & Dean J. (2013), Efficient estimation of word representations in vector space. DOI: https://doi.org/10.48550/arXiv.1301.3781.

[28] word2vec (TersonFlow). URL:https://www.tensorflow.org/text/tutorials/word2vec. Accessed 8.11.2023.

[29] Mikolov T., Yih W.t. & Zweig G. (2013) Linguistic regularities in continuous space word representations. In: L. Vanderwende, H. Daumé III & K. Kirchhoff (eds.) Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Association for Computational Linguistics, Atlanta, Georgia, pp. 746–751. URL: https://aclanthology.org/N13-1090.

[30] Milkov T., Sutskever I., Chen K., Corrado G. & Dean J. (2013), Distributed representations of words and phrases and their compositionality. DOI: https://doi.org/10.48550/arXiv.1310.4546.

[31] Mikolov T. (2008) Language models for automatic speech recognition of czech lectures. URL: https://api.semanticscholar.org/CorpusID:13934103.

[32] Mikolov T., Kopecký J., Burget L., Glembek O. & ernocký J.H. (2009) Neural network based language models for highly inflective languages. 2009 IEEE International Conference on Acoustics, Speech and Signal Processing , pp. 4725–4728URL: https://api.semanticscholar.org/CorpusID:14518311.

[33] What is gradient descent? (IBM). URL:https://www.ibm.com/topics/gradient-descent. Accessed 1.11.2023.

[34] Joulin A., Grave E., Bojanowski P. & Mikolov T. (2017) Bag of tricks for efficient text classification. In: M. Lapata, P. Blunsom & A. Koller (eds.) Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers, Association for Computational Linguistics, Valencia, Spain, pp. 427–431. URL: https://aclanthology.org/E17-2068.

[35] Joulin A., Grave E., Bojanowski P., Douze M., Jégou H. & Milkov T. (2016), Fasttext.zip: Compressing text classification models. DOI: https://doi.org/10.48550/arXiv.1612.03651.

[36] Bojanowski P., Grave E., Joulin A. & Mikolov T. (2016) Enriching word vectors with subword information. Transactions of the Association for Computational Linguistics 5, pp. 135–146. URL: https://api.semanticscholar.org/CorpusID:207556454.

[37] Conneau A., Schwenk H., Barrault L. & Lecun Y. (2017) Very deep convolutional networks for text classification. In: M. Lapata, P. Blunsom & A. Koller (eds.) Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers, Association for Computational Linguistics, Valencia, Spain, pp. 1107–1116. URL: https://aclanthology.org/E17-1104.

[38] Wikipedia dumps (Finnish). URL:https://dumps.wikimedia.org/fiwiki/latest/. Accessed 15.10.2023.

[39] Turku NLP Word2vec demo. URL:http://epsilon-it.utu.fi/wv_demo/. Accessed 18.10.2023.

[40] FastText. URL:https://fasttext.cc. Accessed 15.10.2023.

[41] Gensim. URL:https://radimrehurek.com/gensim/. Accessed 15.10.2023.

[42] Evaluation word pairs. URL:https://github.com/Jtapsa/WordEmbedding/blob/main/evaluation.txt. Created 13.1.2024.

[43] PCA (scikit-learn). URL:https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html. Accessed 26.12.2023.

[44] Jansen T. (2017), Word and phrase translation with word2vec. DOI: https://doi.org/10.48550/arXiv.1705.03127.

[45] Hamilton W.L., Leskovec J. & Jurafsky D. (2016) Diachronic word embeddings reveal statistical laws of semantic change. In: K. Erk & N.A. Smith (eds.) Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Association for Computational Linguistics, Berlin, Germany, pp. 1489–1501. URL: https://aclanthology.org/P16-1141.

11. APPENDICES

Appendix 1 Partial derivatives of Word2vec

$$\frac{\partial_{L_{CE}}}{\partial_{L_{CE}}} = 1 \tag{40}$$

$$\frac{\partial_{L_{CE}}}{\partial_{\hat{\mathbf{y}}}} = \frac{d_{L_{CE}}}{d_{\hat{\mathbf{y}}}}\Big[-\log(\hat{\mathbf{y}})\Big] \cdot \frac{\partial_{L_{CE}}}{\partial_{L_{CE}}} = -\frac{1}{\hat{\mathbf{y}}} \cdot 1 = -\frac{1}{\hat{\mathbf{y}}} \quad , \tag{41}$$

where $\hat{\mathbf{y}} = \sigma(y)$.

$$\begin{aligned} \frac{\partial_{L_{CE}}}{\partial_y} &= \frac{d_{\hat{\mathbf{y}}}}{d_y}\Big[\sigma(y)\Big] \cdot \frac{\partial_{L_{CE}}}{\partial_{\hat{\mathbf{y}}}} = \Big[\sigma(y) \cdot (1-\sigma(y)\Big] \cdot \frac{-1}{\hat{\mathbf{y}}} \\ &= \frac{-[\sigma(y) - \sigma(y)^2]}{\sigma(y)} = [\sigma(y) - 1] \end{aligned} \tag{42}$$

$$\begin{aligned} \frac{\partial_{L_{CE}}}{\partial_{c_{pos}}} &= \frac{d_y}{d_{c_{pos}}}\Big[w \cdot c_{pos}\Big] \cdot \frac{\partial_{L_{CE}}}{\partial_y} = w \cdot \frac{\partial_{L_{CE}}}{\partial_y} \\ &= [\sigma(y) - 1] \cdot w = [\sigma(w \cdot c_{pos}) - 1] \cdot w \end{aligned} \tag{43}$$

Partial derivative for noise words c_{neg} differences from real context word c_{pos} only for slightly. Prediction error for noise words is determined as follow $\log(P(-|w, c_{neg_i}))$, which equals $\log(1 - P(+|w, c_{neg_i}))$ and leads following (44) modification for partial derivative in respect to (43).

$$\begin{aligned} \frac{\partial_{L_{CE}}}{\partial_{c_{neg}}} &= \frac{\partial_{L_{CE}}}{\partial_{L_{CE}}} \cdot \frac{\partial_{L_{CE}}}{\partial_{\hat{\mathbf{y}}}} \cdot \frac{\partial_{\hat{\mathbf{y}}}}{\partial_y} \cdot \frac{\partial_y}{\partial_{c_{neg}}} \\ &= 1 \cdot \frac{d_{L_{CE}}}{d_{\hat{\mathbf{y}}}}\Big[-\log(1-\sigma(\hat{\mathbf{y}}))\Big] \cdot \frac{\partial_{\hat{\mathbf{y}}}}{\partial_y} \cdot \frac{\partial_y}{\partial_{c_{neg}}} \\ &= \frac{1}{1-\hat{\mathbf{y}}} \cdot \frac{d_{\hat{\mathbf{y}}}}{d_y}\Big[\sigma(y)\Big] \cdot \frac{\partial_{\hat{\mathbf{y}}}}{\partial_y} \\ &= \frac{\sigma(y) - \sigma(y)^2}{1-\sigma(y)} \cdot \frac{\partial_{\hat{\mathbf{y}}}}{\partial_y} \\ &= \sigma(y) \cdot \frac{d_y}{d_{c_{neg}}}\Big[w \cdot c_{neg}\Big] \\ &= [\sigma(w \cdot c_{neg})] \cdot w \end{aligned} \tag{44}$$

Individual partial derivative of w: $\frac{\partial_{L_{CE}}}{\partial_w}$ differences from $\frac{\partial_{L_{CE}}}{\partial_{C_{pos}}}$ and $\frac{\partial_{L_{CE}}}{\partial_{C_{neg}}}$ only by outer multiplier. Instead of:

$$\frac{\partial_{L_{CE}}}{\partial_{c_{pos}}} = \frac{\partial_{L_{CE}}}{\partial_y} \cdot \frac{d_y}{d_{c_{pos}}}[w \cdot c_{pos}] = [\sigma(w \cdot c_{pos}) - 1] \cdot w \quad , \tag{45}$$

we get:

$$\frac{\partial_{L_{CE}}}{\partial_w} = \frac{\partial_{L_{CE}}}{\partial_y} \cdot \frac{d_y}{d_w}[w \cdot c_{pos}] = [\sigma(w \cdot c_{pos}) - 1] \cdot c_{pos} \quad , \tag{46}$$

and for c_{neg} case we get:

$$\frac{\partial_{L_{CE}}}{\partial_w} = [\sigma(w \cdot c_{neg})] \cdot c_{neg} \quad \cdot \tag{47}$$

The total partial derivative of w is the sum of its individual partial derivatives representing to prediction error of c_{pos} and noise c_{neg} noise words, as follows

$$\frac{\partial_{L_{CE}}}{\partial_w} = [\sigma(w \cdot c_{pos}) - 1] \cdot c_{pos} + \sum_{i=1}^{k} [\sigma(w \cdot c_{neg_k})] \cdot c_{neg_k} \quad \cdot \tag{48}$$

To visualize part of the SGNS training epoch we can use three elements: stored vocabulary index table, $\mathbf{W}1$ and $\mathbf{W}2$ embedding matrix, which stores center and context word embedding vectors:

Vocabulary

1	w_1
$\vdots$	$\vdots$
500	ai
$\vdots$	$\vdots$
$V-1$	nlp
V	zebra

Index word

$\mathbf{W}1$

w_1^1	w_1^2	$\cdots$	w_1^N
$\vdots$	$\vdots$	$\cdots$	$\vdots$
w_{500}^1	w_{500}^2	$\cdots$	w_{500}^N
$\vdots$	$\vdots$	$\cdots$	$\vdots$
w_{V-1}^1	w_{V-1}^2	$\cdots$	w_{V-1}^N
w_V^1	w_V^2	$\cdots$	w_V^N

V × N

$\mathbf{W}2$

c_1^1	$\cdots$	c_{500}^1	$\cdots$	c_{V-1}^1	c_V^1
c_1^2	$\cdots$	c_{500}^2	$\cdots$	c_{V-1}^2	c_V^2
$\vdots$	$\cdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$
c_1^N	$\cdots$	c_{500}^N	$\cdots$	c_{V-1}^N	c_V^N

N × V

Let us consider a case, where we have the word **ai** as center word and one of the context words c_{pos} is **nlp**. There is also marked noise word c_{neg}, which is **zebra**.

To get the center word w embedding vector, we just multiply the embedding matrix $\mathbf{W}1$ by One-hot encoded input vector, where the corresponding centre word index is 1 and all others is 0: $[0_1, 0_2 \cdots 1_{500} \cdots 0_{V-1}, 0_V] \times \mathbf{W}1 = [w_{500}^1, w_{500}^2 \cdots w_{500}^N] = w$

Next step is to multiply word **ai** embedding vector (w), with context word **nlp** embedding vector, which located in embedding matrix $\mathbf{W}2$: $[w_{500}^1, w_{500}^2 \cdots w_{500}^N] \times [c_{V-1}^1, c_{V-1}^2 \cdots c_{V-1}^N]^T = w \cdot c_{pos}$

To get loss L_{CE} use Sigmoid (σ) and calculate: $-\log[\hat{\mathbf{y}}]$, where $\hat{\mathbf{y}}$ correspond $\frac{1}{1+e^{-w \cdot c_{pos}}}$, after we know L_{CE}, we can adjust corresponding embedding vectors using partial derivatives.

Updating center word **ai** embedding vector:

$$w_{500}^{t+1} = w_{500}^t - \eta \Big[[\sigma(w_{500}^t \cdot c_{V-1}^t) - 1] \cdot c_{V-1}^t \Big]$$

Updating context word **nlp** embedding vector:

$$c_{V-1}^{t+1} = c_{V-1}^t - \eta \Big[\sigma(w_{500}^t \cdot c_{V-1}^t) - 1 \Big] \cdot w_{500}^t$$

Which equals following embedding vector adjustment:

$$w_{ai}^{t+1} = \boxed{w_{500}^1 - \eta \times [(\hat{\mathbf{y}} - 1) \times c_{V-1}^1]} \; \boxed{\cdots} \; \boxed{w_{500}^N - \eta \times [(\hat{\mathbf{y}} - 1) \times c_{V-1}^N]}$$

$$c_{nlp}^{t+1} = \boxed{c_{V-1}^1 - \eta \times [(\hat{\mathbf{y}} - 1) \times w_{500}^1]} \; \boxed{\cdots} \; \boxed{c_{V-1}^N - \eta \times [(\hat{\mathbf{y}} - 1) \times w_{500}^N]}$$

where t represents the time step and η learning rate.