Paper Title*

*Note: Sub-titles are not captured in Xplore and should not be used

1st Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

3rd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

4th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

5th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

6th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

Representing words as vectors, i.e word embeddings (WE) is a fundamental aspect of Natural Language Processing (NLP). There are two ways to create such WE, either arbitrarily or with the purpose of capturing the semantics of the words, i.e. vector representations of words that are syntactically or semantically similar will be near to each other. By capturing semantic or syntactic meaning WE have shown to facilitate a lot of subsequent NLP tasks, such as entity recognition, machine translation or sentence classification. The first attempt to create WE with neural networks was made by Bengio et al. [1] but more recently Mikolov et al. [2] introduced a software package called w2vec that uses a simpler network and with this approach Mikolov et al. produced state of the art results. One of the proposed algorithms in this software package is the Skip-Gram Model (SGM). The SGM is an algorithm, that trains a neural network, on the task of predicting the neighboring words in a sentence. The weights of this network are then used as WE.

The SGM gained a lot of attention, as it achieved very good results for a very simplistic model. As a consequence, a lot of effort went into optimizing it. Most of this effort was trying to improve the throughput of the model, i.e the number of words that are processed per second. The SGM uses Stochastic Gradient Descent as an optimization algorithm and is therefore inherently sequential. To remedy this problem Mikolov et al. used Hogwild [3], where different threads can access a shared model and update it. As this is not an optimal solution Yi et al. [4] tried to optimize it, by using a mini-batch like approach and converting vector to vector multiplications into matrix to matrix multiplications. This yielded two consequences: First the model is updated less frequently leading to less overwriting and offering the possibility to parallelize more. Secondly, it transformed level-1 BLAS operation into level-3 BLAS operations, and the algorithm could therefore effectively use computational resources. Another attempt at optimizing the throughput was made by Seulki and Youngmin [5]. Their goal was to parallelize the algorithm on GPU's. They, therefore, chose to parallelize the update of the dimensions of each word representation. Both of these approaches and most of the literature are focused on improving the throughput of the model, but not the convergence time. Therefore one could ask if the convergence time of the SGM can be optimized while at the same time maintaining its accuracy. This work proposes an approach that uses advanced optimizers and input shuffling to optimize the convergence time of the SGM. In combination both of these techniques allowed us to decrease the convergence time of the SGM. If combined with an optimized throughput these result could lead to an overall decrease in runtime.

This work is structured as follows: subsection II describes the SGM and its optimizations. Furthermore, we will give an extended explanation of the gradient descent optimizers used in this work. The reader is introduced to our Implementation in subsection **??**. Results are presented in subsection V , where we will also describe used datasets, the measure applied to compare the quality of word embeddings, and finally the empirical results. The last part will focus on the discussion of our results, and possible future work in subsection VII followed by a conclusion in subsection VIII.

II. Background

This subsection gives an overview of the SGM and related work on its optimization. Furthermore, the used optimizers in

Identify applicable funding agency here. If none, delete this.

our experiments are explained.

A. The Skip-Gram Model

The SGM is a very simple model used to learn WE. The idea is to train a neural network, on a fake task and then use some of the weights of the network as embeddings. To understand this fake task center and context words must be defined. The center word is any given word in a sentence, from which we want to learn the WE. The context words of this specific center word are the words left and right in a given window m in the sentence. See Figure 1 as an example, where the context words are highlighted. An important aspect to notice, is that instead of having a fixed window size m, each word will randomly have a window size between 1 and m. The idea behind having different window sizes is that words that are further away in the sentence, have less semantic correlation to the center word.

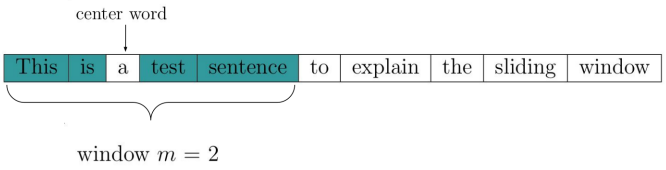

Fig. 1. Example of a center word and it's context words

With these definitions the fake task can be defined: given a pair consisting of a center and a context words, which will be our training samples, the goal of the network is to predict the probability of each word to appear in the context of the center word. To achieve this goal, Mikolov et al. introduced the following model. The network consists of three layers. See fig x for an illustration. First the ntw has an input layer, a vector of size V (V being the size of the vocabulary), represint words as an one hot encoding. Secondly, the projection layer wight Matrix M_{in} soters V D-dimensional vectors for the corresponding word in the vocabulary in it's rows. The projection layer serves as a look up tavle for our WE (if a one hot vector is multiplied with M_{in} the corresponding WE will result). Finally, the output layer wight matrix M_{out} also stores V D-dimensional vector representation of words.The idea behind these 2 Matrices is that the input layer will represent our words as center word and the output layer as context words. Since the task of the network being to predict the probability of each word appearing in the context of a given center word, the following probability should be maximized:

$$\prod_{t=1}^{T} \prod_{-m<j<m} p(w_{t+j}|w_t) \tag{1}$$

Where T is the number of words in the corpus data, w_t the t^{th} word in the corpus data and m is the context window. This means that the m nearest words to w are considered as context words. Equation 1 can be transformed into sums by using log probabilities:

$$\sum_{t=1}^{T} \sum_{-m<j<m} log(p(w_{t+j}|w_t)) \tag{2}$$

where the parameters are the same as in Equation 1.

The basic Skip-Gram Model uses a classical Softmax to calculate the conditional probability $p(w_{t+j}|w_t)$:

$$p(w_{t+j}|w_t) = \frac{exp(\tilde{v}_{w_{t+j}}^T v_{w_t})}{\sum_{w=1}^{v} exp(\tilde{v}_w^T v_{w_t})} \tag{3}$$

Here $\tilde{v}_{w_t}$ and v_{w_t} are the vector representations, stored respectively in the projection Layer (M_{in}) and in the output layer (M_{out}). There is a problem with the classical softmax. As a matter of fact, it is unsuitable to compute the softmax. For the computation of $\sum_{w=1}^{v} exp(\tilde{v_w}^T w_t)$, the denominator in Equation 3, one has to go over the whole corpus data. As very big data sets are needed to train the model, this is not a solution. Mikolov et al. [6] proposed different solutions. The first solution is to use a Hierarchical softmax introduced by Morin and Bengio [7]. In this model, the probability distribution of the output nodes is saved in a binary tree which gives one a logarithmic computation time for each of these probabilities and makes it suitable to compute the softmax. Another possibility is the use of negative sampling which is discussed in the next subsection.

B. Negative Sampling

A second alternative solution to the use of a classical softmax is negative sampling. Negative Sampling is a simplification of Noise Contrastive Estimation (NCE) which was introduced by Gutmann and Hyvärinen [8], and first applied to NLP by Mnih and Teh [9]. This subsection will shortly describe NCE and then how Mikolov et al. [6] simplified it to create the technique called negative sampling.

The idea behind NCE is to distinguish data from noise. It does so by reducing the problem to a logistic regression task and does it by maximizing the log probability. The SGM is only interested in good word representation, hence the probability of the word is not meaningful as long as the quality of the word representations remains high. Mikolov et al. [6] simplified NCE and called it Negative Sampling, which will be explained next.

The main goal of negative sampling is to only update the output nodes of certain words. This will obviously save an enormous amount of computation time. The idea is that given a pair $(c, w) \in D$, where c is a word in the context window of w we select K random words k_i from the corpus data D. We assume those words do not appear in the context of w. We denote the score that the (c, w) wasn't drawn at random the following way: $p(y = 1|c, w)$, and if (k, w) is chosen at random this way: $p(y = 0|k, w)$. Now we use logistic regression to update the weights of the k selected context words and c. By doing so we will only have to update $k + 1$ nodes.

Let's look at how we construct our objective function for a given word w and one of its context words c:

$$
\begin{aligned}
p(c|w) &= p(y=1|c,w) + \prod_{k \in K} p(y=0|k,c) \\
&= p(y=1|c,w) + \prod_{k \in K} 1 - p(y=1|k,c) \\
&= log((p(y=1|c,w)) + \sum_{k \in K} log(1 - p(y=1|k,c)) \\
&= log(\frac{1}{1+e^{-v_c \tilde{v_w}}}) + \sum_{k \in K} log(1 - \frac{1}{1+e^{-v_c \tilde{v_k}}}) \\
&= log(\frac{1}{1+e^{-v_c \tilde{v_w}}}) + \sum_{k \in K} log(\frac{1}{1+e^{v_c \tilde{v_k}}}) \\
&= log(\sigma(v_c \tilde{v_w}) + \sum_{k \in K} \sigma(log(-v_c \tilde{v_k}))
\end{aligned}
$$

Where v_c and $\tilde{v_w}$, can be interpreted as before in Equation 3 and $\sigma(x) = \frac{1}{1+e^{-x}}$. The goal is to maximize this objective function. Another way to look at this function, apart from logistic regression, and understand why it's working so well, is to assume that two vectors are similar if their dot product is high. Therefore, we maximize the dot product of similar words (w, c) (c appears in the context of w) and minimize it for dissimilar words (w, k_i) (those were selected randomly). We see that to compute our objective function we will only have to compute the sum over the number of negative samples K, which is very small in practice (2-20). To put things in perspective lets imagine our data set consists of 100000 words, we set $K = 2$. Assume each output neuron has a weight vector v with $|v| = 300$. In consequence, when updating our weights we would only update $0.2 * 10^{-2}$ of the 300 million weights in the output layer.

One question remains: how do we choose our random words? Mikolov et al. [6] used the following unigram distribution:

$$P(w) = \frac{f(w)^{\frac{3}{4}}}{\sum_{t=0}^{T} f(w_t)^{\frac{3}{4}}} \tag{4}$$

where $f(w)$ is the frequency of w_t. The value of $\frac{3}{4}$ is set empirically. Raising the unigram distribution to power of $\frac{3}{4}$ makes it less likely for a word to be drawn if it often appears in the dataset in comparison to the basic unigram distribution. See figure 2 for an example. It's quite easily observable that this approach will outperform the classical softmax in computation time, as it only compute the sum over K output nodes. Now the question arises if the accuracy is good enough but according to Mikolov et al. [6] the negative sampling method "is an extremely simple training method that learns accurate representations". As a matter of fact, Mikolov et al. [6] reported a 6% accuracy improvement in comparison to a Hierarchical Softmax model. Therefore, it is a good solution to the problem raised by the classical softmax. We now have enough background knowledge about the SGM to look at how it can be optimized. The next subsection introduces earlier approaches to optimize the SGM.

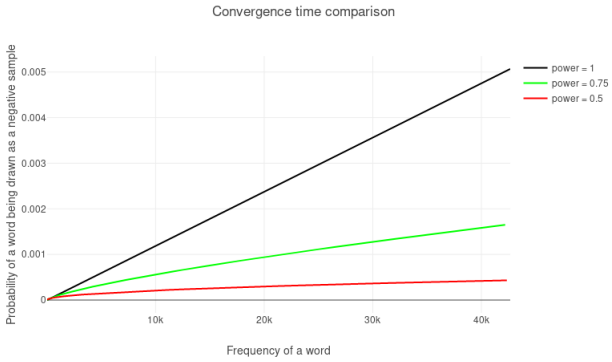

Fig. 2. Probabililty of a word, of the text8 dataset (sampled), to be chosen at random according to its frequency and the power to which the unigram distribution is raised

III. Related Work

Due to the popularity of the SGM, a lot of research went into optimizing it. This research can be divided into two categories, optimization of the throughput and the optimization of the algorithm's accuracy. The latter was achieved by allowing words to have multiple meanings, also called context-sensitive word embedings. For our work, the optimization of the throughput is of big interest while the semantic optimization is aimed at giving the reader a more holistic comprehension of the possible research directions. This subsection will first give an overview of the optimization of the throughput and then present one paper that focused on context-sensitive word embeddings.

In the original model, the optimization is done with Stochastic Gradient Descent (SGD), which is a sequential algorithm. This process does not favor parallelization. To deal with this specific problem Mikolov et al. [6] used a Hogwild tree proposed by Recht et al. [3]. The approach is to allow multiple threads to access a shared memory, in this case, the single model. In the original SGM, the threads are constructed as follows: at the beginning of training, the dataset is split into N evenly sized chunks and each of these chunks will be processed by a one thread. The threads run parallel and have access to the shared memory. Therefore, overwriting errors are bound to happen. According to Recht et al. [3] the overwriting errors won't lead to a significant accuracy loss if the data is sparse enough. But in the case of NLP, the problem seems to be a bit more significant, and especially for word embedding, as many words share the same context words. There were several attempts at solving this issue, and we are going to cover a few of them in the following subsubsections.

a) Parallelization by the use of caching: This idea was proposed by Vuurens et al. [10]. The architecture used here is the basic skip gram model with a hierarchical softmax. The general idea is to cache the most frequently used nodes of

the binary tree used to memorize the probability distribution and update them on the shared single model after a certain number of seen words (the paper used the number 10). The paper produced interesting results as they managed to decrease execution time by increasing the number of cores used for the calculation. This is very powerful because in the original implementation the execution time regressed after using more than 8 cores. It seems to indicate that too much overwriting was happening, as the number of concurrent threads surpasses a certain threshold. This can be seen in Figure 3, where c31 is the model proposed by Vuurens et al. [10]. The model did not suffer any accuracy loss in comparison to the original SGM model. This work proposes a very good way to parallelize the

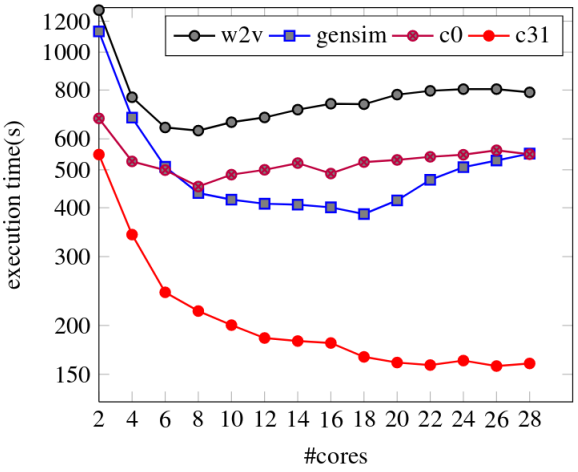

Fig. 3. Comparasion of the execution time in relation to the number of used cores [10]

SGM, as in particular, it allows using more cores during the computation. As this approached focused on the Hierarchical softmax, in contrast to our work which used negative sampling, the next subsection covers optimizations of the Skip-Gram Model with negative sampling(SGNS).

A. Parallization in shared and Distributed Memory

The first parallelized solution which was proposed by Ji et al. [4], is to try to reduce the cost of our vector multiplication. The main idea in this paper is to convert the level 1-BLAS (Basic linear subprogram) vector to vector operations to a level 3-BLAS matrix multiplication operation, which are efficiently implemented into hardware and consequently faster. This is achieved, by using the same negative samples for each context word of a given word w. Instead of using for each context word a vector to vector multiplication we can transform this into a matrix multiplication, under the assumption that we will not lose accuracy by sharing the same negative samples. The matrix multiplication can be represented in the following way.

$$\begin{bmatrix} w \\ w_{n_1} \\ \vdots \\ w_{n_k} \end{bmatrix} * \begin{bmatrix} w_{c_1} \\ \vdots \\ w_{c_{2m}} \end{bmatrix}$$

where w is our given word, $w_{n_1}...w_{n_k}$ are the shared negative samples, with $k \in [5, 20]$, and $w_{c_1}...w_{c_2m}$ are the words inside of the context window m of w, with $m \in [10, 20]$, also called a batch of input context words. After each batch the model updates the weights of the used vectors. This model achieves a 3.6 fold increase in throughput, by only losing 1% of accuracy. An aspect that is not as useful to us is that the experiments were done on CPU, as modern GPU's are often used in many machine learning libraries, as with CUDA for example, there still need work to be done to optimize it with GPU's. This was done by Seulki and Youngmin [5], which is described in the next subsection.

B. Accelleration of word2vec by Using GPU's

Seulki and Youngmin [5] focused on getting a better throughput on the SGM when using GPU's. As the SGM is a sequential algorithm, is is not easy to parallelize it, especially if one wants to parallelize the training of individual training samples. As the algorithm goes sequentially over a sentence, the samples next to each other, in order of execution, will almost every time have the same input word. Consequently, it's very hard to parallelize at this level. To solve this problem, Seulki and Youngmin [5] proposed the idea to parallelize the update of each dimension of the word embedding, since those are completely independent of each other. They achieved this by mapping each dimension to a CUDA thread while mapping each sentence to a CUDA block. As each CUDA block runs independently, the training of the sentences is parallelized, and the fact that sentences have different length is of no problem. If the execution time of the GPU kernel is greater than time used to read the sentences, it could be a smart choice to use multiple GPU's. According to Seulki and Youngmin [5], if multiple GPU's are used, there is a need for synchronizing the model, which will hinder run time performance. They achieved their best results with 2 concurrent GPU'S. The accomplished results were very good as they managed a 20x speedup compared to a single threaded CPU execution, which is a 3x increase in comparison to the original C code from Mikolov et al. [6], with no loss in accuracy. The problem with this and all the above optimization is that the code is not easily available. Therefore, we need an optimized implementation of the SGM that is easily available. This is provided by Gensim [11], which will be outlined in the next subsection.

C. Gensim

Gensim [11] is a pure Python library that holds a state of the art implementation of the SGM. Gensim is written in Cython, which first allowed Gensim to have the same runtime as the original C code. Furthermore, it made use of BLAS's and precomputed sigmoid tables, while also trying to parallelize

the training of different sentences. This finally yielded in a 4x speedup in run time. Gensim is an important tool as it allows us, as a python library, to compare our data rather easily. It was also used in related work [4] and is therefore of value, as it allows to us compare our work in a simpler way. This concludes our overview of the optimizations of the throughput of the SGM. In the next subsection, we give a quick outlook of what has been done in the field of context-sensitive word embeddings.

This will conclude our overview of the related work. We will now give the reader an outline of the different Gradient Descent Optimizer used in our experiments.

IV. Batched SkipGramModel

This section will give an overview of our implementation. First, it will give a broad overview of the implementation idea and process. Then it will go into detail, explaining the forward process of our model and the construction of our dataset, by the use of the `Dataset` class. The proposed implementation is a slighlty altered version of the original SGNS. The idea behind this altering, is to compute the loss for multiple words and context pairs at the same time. The exact process will be described in the following paragraph.

A. Forwarding

As this represents the challenging part of our Implementation the forwarding method is explained step by step. Each time step is illustrated to make the explanation clearer.

Let $X = (v_1, c_1), (v_2, c_2), (v_3, c_3)$, be the training batch, where (v_i, c_i) is a training sample constituted by a word and on of its context words.

Input:

The forwarding method will accept two vectors v and c, and a matrix A as an input. The first vector represents all the center words in a batch, the second one the context words. The Matrix represents the negative samples. The two vectors are of the same length, defined as n. The matrix must be of dimension $n \times k$, with k being the number of negative samples per pair. This means the i^{th} row will store the negative samples for the i^{th} word context pair.

The input can be illustrated as follows:

$$v = \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix}, c = \begin{bmatrix} c1 \\ c2 \\ c3 \end{bmatrix} \text{ and } A = \begin{bmatrix} k_{1,1} & k_{2,1} & k_{3,1} \\ k_{1,2} & k_{2,2} & k_{3,2} \\ k_{1,3} & k_{2,3} & k_{3,3} \end{bmatrix}$$

Concatenation of samples:

The concatenation of the vector c and the Matrix A will result in a Matrix $\tilde{A}$, with:

$$\tilde{A} = \begin{bmatrix} c_1 & k_{1,1} & k_{2,1} & k_{3,1} \\ c_2 & k_{1,2} & k_{2,2} & k_{3,2} \\ c_3 & k_{1,3} & k_{2,3} & k_{3,3} \end{bmatrix}$$

Embeddings:

Now it's necessary to access the word embeddings. For this purpose a Matrix E_v of dimensionality $n \times d$ where d is the dimension of our word embedding is created. E_v stores the word embedding of the i^{th} word from our input vector v in it's i^{th} row. The same is done for the other Matrix $\tilde{A}$. This will result in a $n \times k+1 \times d$ Array E_c.

$E_v = \begin{bmatrix} \tilde{v_{11}} & \dots & \tilde{v_{1d}} \\ \tilde{v_{21}} & \dots & \tilde{v_{2d}} \\ \tilde{v_{31}} & \dots & \tilde{v_{3d}} \end{bmatrix}$, where $\tilde{v_i} = \begin{bmatrix} \tilde{v_{i1}} & \dots & \tilde{v_{id}} \end{bmatrix}$ is the embedding of v_i.

$E_c = \begin{bmatrix} \tilde{c_1} & \tilde{k_{1,1}} & \tilde{k_{2,1}} \\ \tilde{c_2} & \tilde{k_{1,2}} & \tilde{k_{2,2}} \\ \tilde{c_3} & \tilde{k_{1,3}} & \tilde{k_{2,3}} \end{bmatrix}$, where each entry of the matrix is a vector of dimension d

Batch multiplication and negation of samples:

Now we need to compute the dot product of each word vector with its pair and the negative samples, exactly as done as in the original loss function of Mikolov et al. shown in Equation II-B. For this task some definitions are necessary: let A_j be the j^th row of the matrix A, let $E_c(i,j)$ be the d dimensional embedding of the word stored in $\tilde{A}(i,j)$. The dot product is computed with a so-called batch multiplication[1] which will result in a matrix S where $S(i,j) = E_c(i,j) \cdot A_j$. This will result in a $n \times k+1$ Matrix S. Now we only have to negate the last k rows with minus one. The sum of each row represents the loss computed in Equation II-B, for each word context pair. Since computation time is too long with this approach the loss function is altered

$$S = \begin{bmatrix} \tilde{v_1} \cdot \tilde{c_1} & -\tilde{v_1} \cdot \tilde{k_{1,1}} & -\tilde{v_1} \cdot \tilde{k_{2,1}} & -\tilde{v_1} \cdot \tilde{k_{3,1}} \\ \tilde{v_2} \cdot \tilde{c_2} & -\tilde{v_2} \cdot \tilde{k_{1,2}} & -\tilde{v_2} \cdot \tilde{k_{2,2}} & -\tilde{v_2} \cdot \tilde{k_{3,2}} \\ \tilde{v_3} \cdot \tilde{c_3} & -\tilde{v_3} \cdot c_3 \tilde{k_{1,3}} & -\tilde{v_3} \cdot c_3 \tilde{k_{2,3}} & -\tilde{v_3} \cdot \tilde{k_{3,3}} \end{bmatrix}$$

Loss function:

Summing the matrix S and multiplying it with -1 (to make the problem a minimizing problem) results in the loss for our entire batch. As some words may appear more than once in the batch this will more be an average of it than as with Mikolov et al. [6] the exact update per pair.

L = $-\sum_{(i,j) \in k \times n} S(i,j)$

This algorithm provides a way to compute the loss, to make the implementation successful, one still needs to access the context-pairs. The process of doing so is described in the next paragraph.

V. Evalutation

This subsection gives an overview of the used datasets, the used metric to evaluate our models, the configuration of our model and finally, the experimental results achieved.

A. Dataset

In this implementation we only used the text8 [2] dataset. We chose this dataset for two reasons. First of all, it's a very small dataset, that allowed us to do a lot of computations. Secondly, this dataset was used in related work [4] hence giving us a very good benchmark. The text8 dataset consists of 1702 lines of 1000 words, with a vocabulary of roughly 63000 words.

[1] Documentation: https://pytorch.org/docs/stable/torch.html#torch.bmm

[2] http://mattmahoney.net/dc/enwik8.zip

TABLE I
TRAINING AND CONVERGENCE TIME ACCORDING TO CHOICE OF THE LENGTH OF SENTENCES IN TEXT8 DATASETCAPTION

Length of Sentences	10	20	40	1 Document
Training Time for one batch	8min	10min	11min	18min
Convergence Time in epochs	4	3	3	3
Word Similarity	0.65	0.66	0.66	0.66

TABLE II
SIZE OF THE PREPROCESSED TEXT8 DATASET ACCORDING TO SAMPLING TRESHOLD

Sampling Treshhold	0	10^{-1}	10^{-2}	10^{-3}	10^{-4}
Words in Dataset	16mio	15mio	11mio	8mio	4mio

Conveniently, there is no punctuation in the dataset. Therefore, we had to choose between building arbitrary sentences and keeping the dataset as it is. We chose the first option because it gives us a faster computation time, and did not show any significant loss in quality empirically, as shown in Table II. We chose a length of sentences of 20. Furthermore, we applied a technique called subsampling to reduce the data set size, which is explained in the following subsection.

1) Subsampling: Subsampling is a technique introduced by Mikolov et al. [2] to reduce the dataset size while at the same time increasing the quality of the dataset, i.e getting better word embeddings with it. The idea behind subsampling is the removal of very frequent words such as: "the, as, it, of". These words do not give an intrinsic value to the words that appear in its context. Therefore, the goal of subsampling is to delete such words from the dataset. This will decrease the computation time, as it will reduce the number of training samples, and should, in theory, increase the accuracy of the model. The increase in accuracy can also be explained by the fact that words that would not have appeared in the context of each other, may now do because words between have been deleted. To choose which word to delete, Mikolov et al. [6] chose the following equation to compute the deletion of a word w in the data set:

$$P(w) = 1 - \sqrt{\frac{t}{f(w)}} \tag{5}$$

where $f(w)$ is the frequency of w, and t is a threshold set empirically. As Equation 5 is a probability, subsampling is not a deterministic procedure, words that may have been deleted with a threshold of 10^-2 may stay in the dataset with a lower threshold, as shown in Table III. Mikolov et al. recommend a value between 0 and 10^{-5}, depending on the size of the dataset. We experimented with different values and 10^{-4} seemed the most suited. We did this by looking at a random set of sentences and judging the results manually. An example of the first sentence with different sampling thresholds can be found in Table III. The table shows the first 20 words of our dataset, without the words that were subsampled according to a threshold sample. Stats about subsampling can be found in Table V-A1.

2) Min count: We also deleted every word that did not appear more than 5 times in our dataset. We got this technique from Gensim [11], that introduced this parameter into their training. This is a good technique because of three reasons: First certain words of our data sets do not appear in a common lexicon (twigleg, melqu), or come from a foreign language (Volksvereinigung), or are names and acronyms. Secondly, each document often has spelling mistakes, those (as long as the same spelling mistake does not appear too often, what should be avoided in practice) would be deleted by sampling too, as the words do not have any meaning. Lastly, a word that only appears one time in our dataset will be very dependent on its original initialization. This is the case because it will only be updated with its context pairs once, which is only a dozen of times in practice and then won't be updated anymore. For all the above reasons, we applied this technique. Similar to subsampling, it should in theory improve the quality of the word embeddings and will decrease the computation time.

B. Evaluating word embedings

Evaluating word embedding is not an easy task, such as evaluating the accuracy of a common classifier. We cannot split our data set into train and test set. As the task that the network is learning is of no interest to us. Therefore, we need to verify that our embedding is of quality with other techniques. To define quality we first need to define a measure of similarity between two vectors. This requires knowledge of the Cosine distance, which is introduced in the following subsection.

1) Cosine distance: The cosine similarity, this is not the cosine distance, of vectors v and w is the cosine of the angle between the two vectors It can be calculated by taking the dot product of v and w and dividing it by the magnitude of v and w multiplied with each other. We get:

$$cos\_sim(v, w) = \frac{v \cdot w}{|v||w|} \tag{6}$$

The cosine of 0° is 1, it's 0 for two vectors that are orthogonal to each other and vectors that point in the opposite direction will have a cosine of their angle of -1. This is not a good distance measure as -1 is smaller than 0, and therefore two vectors pointing away from each other would be closer than two orthogonal vectors, but by subtracting 1 from the cosine of the angle we can create a good distance measure between the two vectors. This distance does not take into account any order of magnitude. Hence, for our tasks, two vectors will be considered equal if they are of different magnitude but point in the same direction. Apart from measuring the quality of word embedding well, this technique has another advantage. By normalizing the vectors the calculation of the cosine angle becomes the dot product of the two vectors. Which can be computed very fast on modern GPU's. Knowing that we have a measure to compute the similarity of two vectors let us introduce a way to rate the quality of our embeddings.

TABLE III
EXAMPLE OF A SENTENCE WITH DIFFERENT SAMPLING TRESHOLDS

Sampling Treshold	First sentence of Dataset
10^{-1}	Anarchism originated as a term of abuse first used against early working class radicals including the diggers of the english
10^{-2}	Anarchism originated as a term of abuse first used against early working class radicals including diggers of english
10^{-3}	Anarchism originated a term abuse first used against early working class radicals including diggers the english
10^{-4}	Anarchism originated abuse used against working class radicals diggers english
10^{-5}	against radicals diggers

TABLE IV
EXAMPLE OF PAIRS AND THEIR RATING IN WORDSIM353

Word1	Word2	Score
"FBI"	"Investigation"	8.31
"Mars"	"scientist"	5.63

2) Word similarity and wordsim353: To measure the quality of our word embedding we will need a dataset to compare our results too. We chose wordsim353[3] for this task, as it's the most used in the related literature. The data set consists of 353 pairs of words rated by humans on their similarity. The similarity score is in the range of 1 and 10, with 10 being the best score. An example for two of such pairs can be found in Figure IV. We will rank our embeddings on the Pearson correlation coefficient between the cosine distance and the scores attributed by humans, as this is a common procedure.

C. Configuration of the network

The skip gram model has a lot of possible parameters, that can be tuned. We experimented with different models and finally decided for one that we tried to optimize. This subsection will give a short overview of each parameter, where we will explain the process in which we chose the value of the given parameter. The explanation of the parameters will be structured as follows: `Parameter` - Description and tuning - *Value*

- `Negative Samples`: Here we have to find a trade-off between, setting the parameter too high which will result in increased accuracy but a longer computation time. For smaller data sets a higher number of negative samples is often needed. In their original paper Mikolov et al. [6] recommend a value of 5-20. We tested a few values in the range of 5 to 15, as 10 yielded state of the art results we chose this value. - *10*
- `Context Window`: The bigger the window the more training examples the network will have, but if the window is too big the semantic meaning of the window will be erased. Mikolov et al. [2] proposed a setting between 2-10, as all our sentences are of size 20 we chose 5. - *5*
- `Dimension of the embedding`: Here the choice is less obvious, as the dimension needs to be high enough to capture the meaning, but cannot be too high as this leads to a decrease in performance as shown by Yin and Shen [15]. We, therefore, used Gensim to find the best embedding possible. - *100*
- `Batch size`: As described in subsection IV, there is a tradeoff to find between quality and training time. We first used a batch size of 5000, but then decide after non conclusive results (see **??**) that 2000 would be better - *2000*
- `Alpha`: learning rate, this hyperparameter was tuned in every optimizer therefore only the range will be indicated - *(1e-5,1)*

D. Input Shuffling

We used input shuffling as a technique to optimize the skip gram model. We will first describe input shuffling in a general way and then explain why we suppose that input shuffling could work well on the skip gram model.

Let $X = x_1...x_n$ be our input data set. Input Shuffling describes the process of taking a random permutation of the dataset as input at each epoch. The idea behind this technique is to present our optimizer with different loss surfaces so that it's able to find the best optimum. Therefore, it's easier for the neural network to escape a local minimum. As for example if a network had converged to a local minimum after one epoch it could not escape it as all the parameters are the same. But if we change the shape of the loss function, by input shuffling, then there would be a greater probability for the network to escape the local minimum.

There are two reasons why we think that input shuffling is particularly well suited for the skip gram model. The first one has to do with the fact that when we read our words sequentially that words that only appear very early will not benefit from the context words being already updated from others. The second idea is that we used the special batch technique described in Section IV. When using this technique and not using shuffling we will always have words that appear next to each other in a batch and will, therefore, update the same words at the same time. We, therefore, lose some quality. But if we would use input shuffling instead, then the words in one batch would likely not be similar and therefore only

[3]http://www.cs.technion.ac.il/ gabr/resources/data/wordsim353/wordsim353.zip

taking the average of a small part of pairs with the same words will be less likely.

E. Convergence time

To optimize convergence time we have to define it first. Therefore, we used the already available implementation of Gensim [11]. Gensim is an open source software that proposes an implementation of the SGNS in Python. It is also written in Cython, therefore it has a fast computation time, but can be used inside a python implementation. Together with the knowledge from Ji et al. [4] that a score of 0.66 in the task of word similarity, with the text8 dataset, is the state of the art, we tested Gensim (more on this process in subsection VII and found out that it took 4 epochs to converge. Therefore, we defined the following criteria for convergence:

$\rho - \rho_{prev} < 0.009 \vee \rho = 0.66$

where ρ is the Pearson coefficient on the wordsim353 task. We also stopped computation, if it took more then 20 epochs to converge.

VI. Results

We ran multiple experiments for each optimizer. This subsection will only give an overview of the achieved results. Each subsection will give an explanation over the achieved result with a specific optimizer.

1) SGD: The first challenge for each optimizer was to find a correct learning rate. As SGD is the optimizer used in Gensim [11] we first tried the same learning rate as Gensim [11] and then performed a random search to find a better one. As expected a bell curve shape resulted, a learning rate that is too high leads to diversion and a learning rate that is too low leads to a training time that is too slow. The best value that we found for the learning rate is 0.0075. With this setting SGD converged in 11 epochs. The second experiment was to add input shuffling. As seen in Figure 4, for almost every learning rate the convergence time decreased. Our model, with the best setting, now converges in only 7 epochs. Another interesting fact to point out from Figure 4 is that with input shuffling we achieved better results with higher learning rates. As for learning rates of 0.01 and 0.025 we did converge in 11 epochs with input shuffling but did not converge in 20 epochs without it.

2) Momentum and Nesterov: Momentum and NAG [16] both have an additional hyperparameter γ, that, as described in Section **??**, defines the percentage of the previous gradient that will be added to the current gradients. We set $\gamma = 0.9$ as this is the typical value and did not alter it during our experiments. Momentum and Nesterov alone respectively only slightly decrease or increase the convergence time. Momentum optimally converges in 9 epochs and Nesterov in 13. If we combine these optimizers with input shuffling, interestingly the same phenomena as with plain SGD appear. The convergence time gets better, 8 epochs for Momentum and 3 epochs for NAG. The phenomena that a higher learning rat yields better results also happens with both of the optimizers. As Momentum does not converge in 20 epochs with a learning rate of 0.002 but does in 8 with input shuffling.

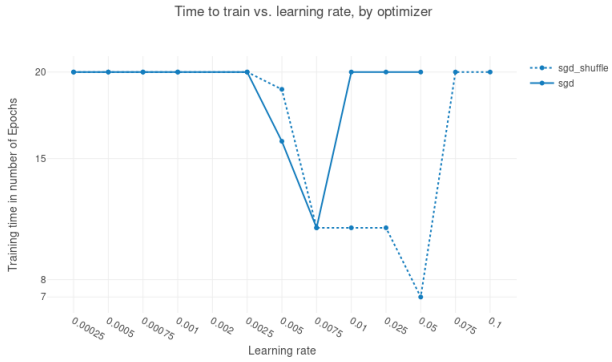

Fig. 4. Training time Stochastic Gradient Descent with input Shuffling

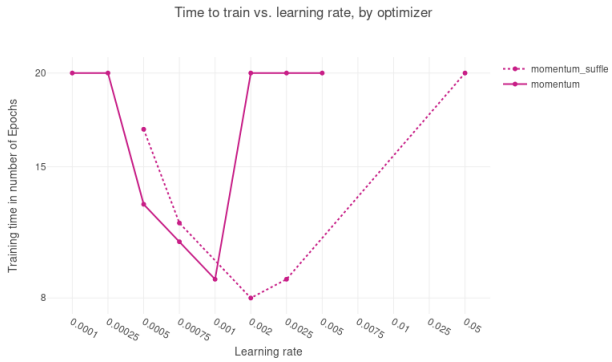

Fig. 5. Training time Momentum with input Shuffling

3) Adagrad: Adagrad [17] is a very interesting tool for learning word embeddings as it decreases the learning rate for very frequent occurring features, and vice versa (this is explained in detail in Section **??**). Because words that appear very frequently often do not have a semantic gain that is as important as words that appear less frequently to their context words, it's good to have a lower learning rate. So, in theory, Adagrad is particularly well suited for our task. This was confirmed empirically as our model converged in 4 epochs. When combined with shuffling Adagrad only took 3 epochs to converge. This shows the tendency of the skip gram model to converge faster with input shuffling and the big impact of having different learning rate for each feature. Here it's interesting to notice that a higher learning rate combined with input shuffling did not yield better results than without shuffling. Both of our best results happened with a learning rate of 0.1, as shown in Figure 7.

4) Adadelta: In theory Adadelta [18] should outperform Adagrad as it's an extension of the former. Because it didn't have any learning rate to tune, we only did 2 experiments, with and without input shuffling. We are aware of the fact that there are additional hyperparameters but did, for simplicity reason, and because their effect is not as high as the learning rate in other optimizers, decide not to tune it. We left it to its

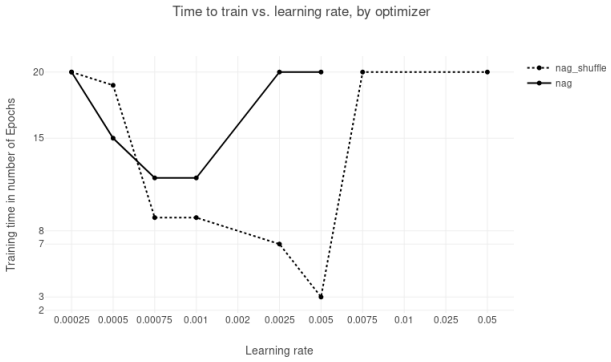

Fig. 6. Training time Nesterov with input Shuffling

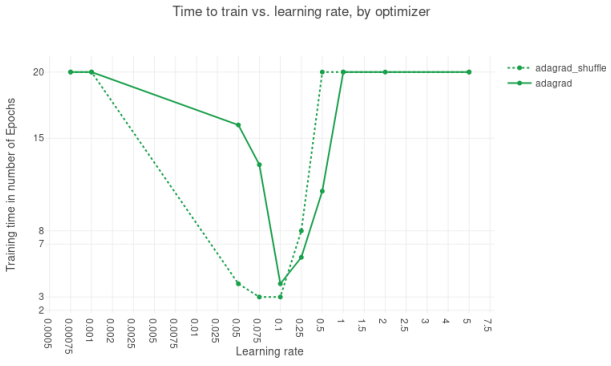

Fig. 7. Training time Adagrad with input Shuffling

default value $\rho = 0.9$. This parameter defines the percentage taken when calculating the exponentially decaying average of past gradients, as explained in **??** Adadelta did not manage to achieve a word similarity of 0.66. It only converged to a similarity of 0.59. It did this in 20 epochs without input shuffling and in 3 with input shuffling, as can be seen in Table VI

TABLE V

5) Adam: Adam is the most advanced of all the optimizers used in our experiments and did yield the best results as seen in Figure 8. Adam converged in 3 epochs without shuffling and 2 with. This is the best result that we got with any optimizer. It's also interesting to note that as same as with Adagrad it did not react to input shuffling the same way as SGD did. In fact, it worked in the opposite direction, as we achieved our best result with input shuffling while having a lower learning rate 0.001 then we used to achieve the best result without input shuffling 0.05.

TABLE VI
CONVERGENCE TIME AND QUALITY WITH ADADELTA

Adadelta Model	Convergence Time	Word similarity
Without Shuffling	20	0.59
With Shuffling	3	0.59

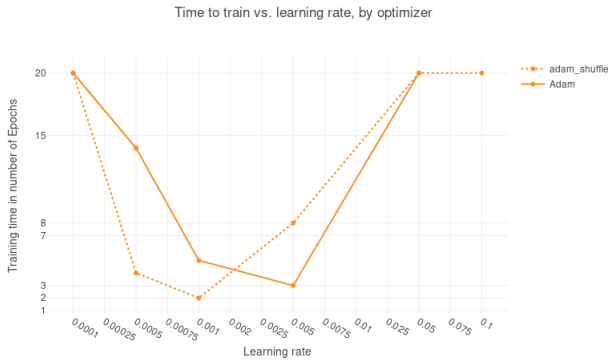

Fig. 8. Training time Adam with input Shuffling

VII. DISCUSSION

This subsection shortly discusses the empirical results and then extensively compares the findings of this work to the existing literature while trying to explain some differences. It is followed by a subsection describing the limitations and possible extensions of this work. This subsection is concluded by the problems we encountered during the implementation so others can avoid them in the future.

A. Our work

This subsection discusses our findings, first we will analyze how input shuffling could influence the learning rate and conclude with the discussion of unexpected results.

1) Shuffling and learning rate with SGD: When the model used SGD as an optimizer it was able to use higher learning rates when input shuffling was used, as shown in figures 4, 5 and 6 Therefore arises the questions why these phenomena happen, especially as it did not happen with advanced optimizers. One possibility is the use of the batched version of the SGNS. In consequence, when the input is not shuffled the same word will often appear in one batch, hence the optimizer is presented with an average value of the gradient, which can be imprecise. This is counter-attacked by advanced optimizers as they have adaptive learning rates, i.e a frequent appearing feature will have a low learning rate.

2) Large differences with NAG and SGD when using shuffling: SGD and NAG both have very different values when using shuffling in comparison to unshuffled input, as shown in figures 4 and 6, plain SGD. We do not only attribute those results to input shuffling but partially also to a good random initialization guess. Due to a lack of time these results were not replicate more than once.

B. Comparison to Gensim

This subsection will compare our finding extensively to Gensim [11]. As explained in Section III-C, Gensim is optimized to have a very high throughput, this made it possible to achieve a lot of computations. Furthermore, Gensim provides access to the loss and the resulting word embeddings, which facilitated the comparison process.

1) Configuration of Gensim: The training with Gensim has a lot of possible parameters an extended list can be found in the appendix. This subsection will only describe the parameters we changed from the default setting. The description of each parameter has the following form:
`name` (type) – *Description* – Value
Parameters:

- `sentences` (iterable of iterables) – *Dataset* – text8 document splitted into sentences of 20 words
- `size`(int) – *Dimensionality of the word vectors* – 100
- `window` (int) –*Maximum distance between the current and predicted word within a sentence* – 5
- `min_count` (int) – *Ignores all words with total frequency lower than this* – 5
- `workers` (int) – *Number of threads used to train the model* – 4
- `sg` (0, 1) – *Training algorithm: 1 for skip–gram*–1
- `negative` (int) –*Number of negative samples*– 10
- `ns_exponent` (float) – *Exponent in the unigram distribution, when choosing random samples, as shown in Equation 4* – 0.75
- `alpha` (float) – *The initial learning rate.* – 0.025
- `min_alpha` (float) – *Learning rate will linearly drop to min_alpha as training progresses* –0.0001
- `sample` (float) – *Treshold for subsampling as described in Equation 5* – 1e–4
- `iter` (int) – *Number of iterations (epochs) over the corpus.* – 10
- `compute_loss` (bool) –*If True, loss is stored at the end of each batch*– True
- `callbacks` (iterable of CallbackAny2Vec) – *Set of functions executed in order to follow the loss and the progress of the model in word similarity* – see Appendix

2) Gensim vs. SGD: As stated earlier, we are not going to compare this work to Gensim in run time. Gensim is heavily optimized and written in cython[4], which is 23x faster than plain Numpy. Since we used PyTorch the difference is not that big, but still remains. As shown in Figure 9 the convergence time was not the same between our implementation and Gensim. There are different possible reasons why this could be the case:
First, our batched approach could hinder performance in terms of convergence since our loss function is not exactly the same. When a word appears more than once in our batch, the gradient will be an average over the gradients of each pair alone, as it is done by Gensim.
Secondly, a difference to our implementation is the fact that Gensim checks whether negative samples are not equal to the context word. If that is the case Gensim selects a new random sample. Therefore, the learning of the input and output context is optimized.
Finally, another possibility is the decay of the learning rate used by Gensim. In fact, decaying the learning rate has been proven in a lot of work to decrease the convergence time. Gensim linearly decreases the learning rate, as we did not use this technique, the decay of the learning rate could help explain the noted differences.
The first hypothesis, the fact that we used a batched approach, may be confirmed by the fact that when combined with input shuffling SGD does perform closer to Gensim, going from 11 to 7 epochs to converge, as input shuffling reduces the number of co-occurrence of the same word in a batch. Now the question arises if the 3 epochs, that Gensim is better, can be explained by the selection of better negative samples and the learning rate decay?

3) Gensim vs. Adam: The Adam optimizer did outperform the Gensim application in quality of word embeddings (only slightly: 0.01 correlation coefficient better) and convergence time. Adam converged in 2 epochs while Gensim in 4. To confirm the achieved results we ran each computation 40 times. The results can be seen in Figure 9.

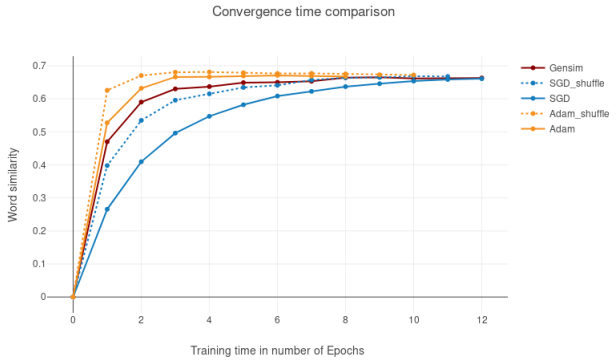

Fig. 9. Convergence time of SGD and Adam compared to Gensim

C. Future Work

This work shows that the convergence time of the SGNS could be improved by using input shuffling and advanced optimizers. As with every work, there still exists possible extensions. First and foremost an aspect of our implementation that can be prejudicial is that we only extensively tested our model with one small dataset. The fact that we only used one dataset as well that it's a small one is problematic. By using a very small dataset we do not use the model in the condition it is most needed for, as the dataset used in practice usually consists of more than 1 billion words. There is a small argument that can be made for machine translation as the use of small parallel corpus is not unusual in this field. But the main issue with using only one data set it that it has been shown that some optimizers perform better

[4]https://rare-technologies.com/word2vec-in-python-part-two-optimizing/

with specific shapes of loss functions. To make a compelling argument it's, therefore, necessary to show that our model with the use of Input shuffling and Adam as its optimizer also outperform Gensim with other data sets, so that the claim can hold consistently, it needs to be confirmed with other datasets as well. Furthermore, our implementation did not outperform Gensim in run time, as this was not the goal of our work. Therefore, one could improve an already existing, optimized version, with input shuffling and advanced optimizers and should achieve a better run time than Gensim.

VIII. Conclusion

This work provides an overview of the Skip Gram Model with negative Sampling (SGNS) and the numerous successful attempts of optimizing the throughput of the model. As this is the case, no effort went into optimizng the convergence time of the SGNS, therefore this work focused on this point. We decided to use advanced optimizers and input shuffling as optimizing techniques. After giving a short overview over Gradient Descent algorithms this work proposes a slighlty altered version of the SGNS, where the idea is to compute the loss over the sum of a high number of training samples, i.e 2000, instead of computing it for each individually. We did this as it allowed us to compute more models and analyze the convergence time faster. We used the text8 dataset and used word similarity as a quality measure for the word embeddings (WE). We used the State of the art implementation Gensim to compare our self. We did achieve a better convergence time than gensim with Adam as an optimizer and the use of input shuffling. Gensim convereged in 4 epochs to a word similarity of 0.66 and our model only took 2 epochs to achieve the same quality. Those results still need to be confirmed with more datasets. Finally, if this work would be combined with an optimized throughput it could improve the state of the art run time of the SGNS.

References

[1] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[2] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[3] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in neural information processing systems*, 2011, pp. 693–701.

[4] S. Ji, N. Satish, S. Li, and P. Dubey, "Parallelizing word2vec in shared and distributed memory," *CoRR*, vol. abs/1604.04661, 2016. [Online]. Available: http://arxiv.org/abs/1604.04661

[5] S. Bae and Y. Yi, "Acceleration of word2vec using gpus," in *Neural Information Processing*, A. Hirose, S. Ozawa, K. Doya, K. Ikeda, M. Lee, and D. Liu, Eds. Cham: Springer International Publishing, 2016, pp. 269–279.

[6] M. Tomas, I. S. Qiu, C. Kai, C. Greg, and D. Jeffrey, "Distributed representations of words and phrases and their compositionality," *CoRR*, vol. abs/1310.4546, 2013. [Online]. Available: http://arxiv.org/abs/1310.4546

[7] F. Morin and Y. Bengio, "Hierarchical probabilistic neural network language model." in *Aistats*, vol. 5. Citeseer, 2005, pp. 246–252.

[8] M. Gutmann and A. Hyvärinen, "Noise-contrastive estimation: A new estimation principle for unnormalized statistical models," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 2010, pp. 297–304.

[9] A. Mnih and Y. W. Teh, "A fast and simple algorithm for training neural probabilistic language models," *arXiv preprint arXiv:1206.6426*, 2012.

[10] J. B. Vuurens, C. Eickhoff, and A. P. de Vries, "Efficient parallel learning of word2vec," *arXiv preprint arXiv:1606.07822*, 2016.

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

[12] Y. Liu, Z. Liu, T.-S. Chua, and M. Sun, "Topical word embeddings." in *AAAI*, 2015, pp. 2418–2424.

[13] B. Sergey, K. Dmitry, O. Anton, and P. V. Dmitry, "Breaking sticks and ambiguities with adaptive skip-gram," *CoRR*, vol. abs/1502.07257, 2015. [Online]. Available: http://arxiv.org/abs/1502.07257

[14] P. Liu, X. Qiu, and X. Huang, "Learning context-sensitive word embeddings with neural tensor skip-gram model." in *IJCAI*, 2015, pp. 1284–1290.

[15] Z. Yin and Y. Shen, "On the dimensionality of word embedding," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 887–898. [Online]. Available: http://papers.nips.cc/paper/7368-on-the-dimensionality-of-word-embedding.pdf

[16] Y. E. Nesterov, "A method for solving the convex programming problem with convergence rate o (1/kˆ 2)," in *Dokl. akad. nauk Sssr*, vol. 269, 1983, pp. 543–547.

[17] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[18] M. D. Zeiler, "ADADELTA: an adaptive learning rate method," *CoRR*, vol. abs/1212.5701, 2012. [Online]. Available: http://arxiv.org/abs/1212.5701