SIG Proceedings Paper in LaTeX Format*

Extended Abstract

Ben Trovato
Institute for Clarity in Documentation
Dublin, Ohio
trovato@corporation.com

G.K.M. Tobin
Institute for Clarity in Documentation
Dublin, Ohio
webmaster@marysville-ohio.com

Lars Thørväld
The Thørväld Group
Hekla, Iceland
larst@affiliation.org

ABSTRACT

The Skip-Gram Model with negative sampling (SGNS) is a simple and effective algorithm to create word embeddings. SGNS uses Stochastic Gradient Descent (SGD) as it's learning algorithm. While a lot of effort has gone into increasing the throughput of words of the SGNS, not much work has gone into optimizing the convergence time. Therefore, our work focuses on the latter. To achieve a better convergence time we use advanced optimizers and input shuffling as optimization techniques. Here are the components used to evaluate our work: to train our model we used the text8 dataset - to measure the quality of our word embeddings we used the wordsim353 dataset, which assess that similar words have vector represenations that are close to each other. We trained our model with multiple advanced optimizers: Momentum, Nesterov Accelerated Gradient, Adagrad, Adadelta, and Adam. We also applied input shuffling to the above optimizers. Adam combined with input shuffling outperformed every optimizer. Adam with shuffling also outperformed the current state of the art implementation Gensim. Adam converged to a similarity value of 0.66 (state of the art) in 2 epochs, while Gensim took 4 epochs. Hence, this work shows that advanced optimizers combined with input shuffling do decrease the convergence time of the SGNS. These results must be confirmed with other datasets, but optimizing our model in terms of throughput has the potential of further reducing the overall run time of the SGNS.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

KEYWORDS

ACM proceedings, LaTeX, text tagging

ACM Reference Format:
Ben Trovato, G.K.M. Tobin, and Lars Thørväld. 2020. SIG Proceedings Paper in LaTeX Format: Extended Abstract. In *Proceedings of ACM SAC Conference (SAC'20).* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/xx.xxx/xxx_x

*Produces the permission block, and copyright information

SAC'20, March 30-April 3, 2020, Brno, Czech Republic

ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/xx.xxx/xxx_x

1 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 semantic meaning of the words, i.e. vector representations of words that are syntactically or semantically similar will be close to each other in the vector space. 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. [**?**] but more recently Mikolov et al. [**?**] introduced a software package called w2vec, which uses a simpler network that produces 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. Mikolov et al. [**?**] then introduced an alteration to the SGM called negative sampling. The goal of the Skip-Gram Model with negative sampling (SGNS) is to distinguish for a given word w, context words c_i (words that appear close to w in a sentence) from words drawn randomly (i.e negative samples) k_i. This is achieved by maximizing the dot product of w and c_i and minimizing the dot product of w and k_i.

The SGM, espacially the SGNS, 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! [**?**], proposed by Recht et al., where different threads can access a shared model and update it. As this is not an optimal solution Yi et al. [**?**] 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 [**?**]. 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 focused on improving the convergence time of the SGNS. To facilitate our calculations we propose a slightly altered version of the original SNGS, in which we compute the loss for a very large amount of training samples, i.e 2000. The loss of such a batch was computed by taking the sum over the loss all individual training samples in the specific batch We combined this batched approach with advanced optimizers and input shuffling to decrease he convergence time of the SGNS. In combination advanced optimizers and input shuffling allowed us to decrease the convergence time of the SGNS. If combined with an optimized throughput these results could lead to an overall decrease in runtime.
The rest of the paper is structured as follows: section 2 describes the SGM and the SGNS followed by its optimzations in Section 3. The reader is introduced to our Implementation in Section 4. Furthermore we will describe the used datasets, the measure applied to compare the quality of word embeddings and input shuffling in Section 5. Results are presented in Section 6. The last part will focus on the discussion of our results, and possible future work in Section 7 followed by a conclusion in Section 8.

2 BACKGROUND

This section gives in its first subsection a broad overview over the functioning of the Skip-Gram Model, and its second section describes the Technique called Negative Sampling introduced by Mikolov et al. [**?**] used to make the Skip-Gram model more efficient

2.1 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.

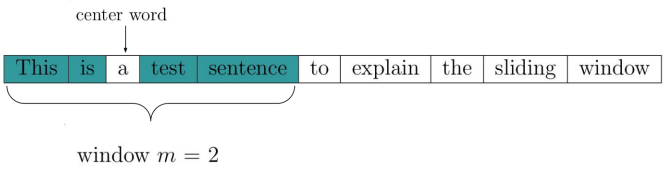

Figure 1: Example of a center word and it's context words

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 Figure 2 for an illustration. First the network has an input layer, a vector of size V (V being the size of the vocabulary), represent words as an one hot encoding. Secondly, the projection layer weight Matrix M_{in} stores V D-dimensional vectors for the corresponding word in the vocabulary in it's rows. The projection layer serves as a look up table 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 projection layer will represent our words as center word and the output layer as context words.

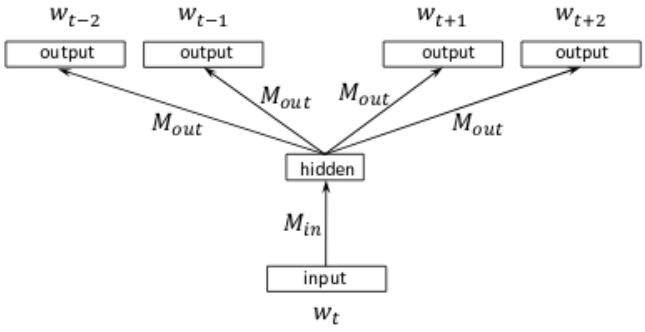

Figure 2: Structure of the network of the Skip Gram Model

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:

$$\prod_{i=1}^{V} \prod_{-m<j<m} p(w_{i+j}|w_i) \tag{1}$$

Where V is the number of words in the corpus data, w_i the i^{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:

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

where the parameters are the same as in Equation 1.

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

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

Here v_{w_i} and $\tilde{v}_{w_i}$ 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. 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 approach is unsuitable. Mikolov et al. [**?**] proposed different solutions. The first solution is to use a Hierarchical softmax introduced by Morin and Bengio [**?**]. 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.

2.2 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 [**?**], and first applied to NLP by Mnih and Teh [**?**]. This subsection will shortly describe NCE and then how Mikolov et al. [**?**] 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. [**?**] 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, 5-20 in practice. This will obviously save an enormous amount of computation time. The idea is that given a pair $(c, w) \in D$, wh$w$ ere c is a word in the context window of we select K random words k_i from the corpus data D. We assume those words do not appear in the context of w. By doing so we will only have to update $k + 1$ nodes. Mikolov et al. [**?**] introduced the following objective function J, that should be maximized:

$$J(c, w) = log(\sigma(v_c \tilde{v_w}) + \sum_{k \in K} \sigma(log(-v_c \tilde{v_k})) \quad (4)$$

Where v_c and $\tilde{v_w}$, can be interpreted as before in Equation 3 and $\sigma(x) = \frac{1}{1+e^{-x}}$. To understand why this loss function is working so well, one needs 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 therefore assume that their are not similar). 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 of dimension $d = 300$. 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 are the negative samples selected? Mikolov et al. [**?**] used the following unigram distribution, to define the probability $P(w)$ of a word w being drawn as a negative sample:

$$P(w) = \frac{f(w)^{\frac{3}{4}}}{\sum_{t=0}^{T} f(w_t)^{\frac{3}{4}}} \quad (5)$$

where $f(w)$ is the frequency of w_t. The value of $\frac{3}{4}$ is set empirically. Raising the unigram distribution to the 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. With all of the above knowledge it's understandable that the SGNS will will outperform the classical softmax in computation time. Now the question arises if the accuracy is maintained but according to Mikolov et al. [**?**] the negative sampling method "is an extremely simple training method that learns accurate representations". As a matter of fact, Mikolov et al. [**?**] 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 section introduces earlier approaches to optimize the SGM.

3 RELATED WORK

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.[**?**] used a Hogwild! approach proposed by Recht et al.[**?**]. The idea 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.[**?**] 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 optimize the throughput of the SGM.

3.1 Parellelization by the use of caching

This idea was proposed by Vuurens et al. [**?**]. The architecture used here is the SGM with a hierarchical softmax. The general idea is to cache the most frequently used weight vectors, and update the model after a certain number of seen training samples, while taking in consideration the changes that happend during these number of seen training samples (the paper used the number 10). The unfrequent weight vectors are handled as in the original paper, i.e updated in shared memory after each training sample. This technique allows it to avoid collosision for the most frequent words, and the author therefore hoped to be able to increase the number of concurrent threads. 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.[**?**]. 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 SGM, as in particular, it allows using more cores during the computation. As this approached focused on the Hierarchical softmax, in

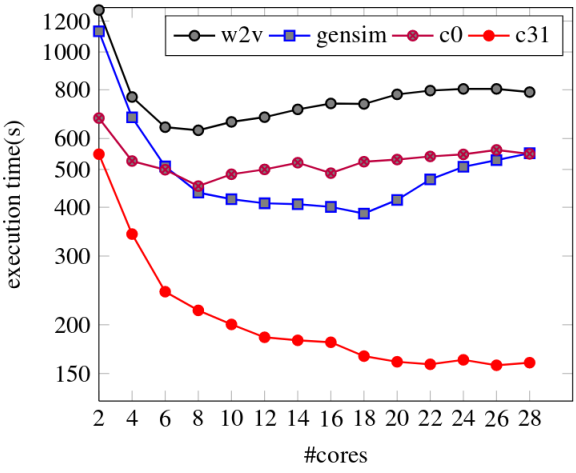

Figure 3: Comparasion of the execution time in relation to the number of used cores [?]

contrast to our work which uses negative sampling, the next subsection covers optimizations of the Skip-Gram Model with negative sampling(SGNS).

3.2 Parallization in shared and Distributed Memory

In contrast to the above approach, this one, introduced by Ji et al. [?] focused on the SGNS. The focus of this work was to speed up the computation of the loss function of the SGNS described in Equation 2.2. Where the dot product of the center word with negative samples as well as the context word is computed. The dot product is a level 1-BLAS operation and is not as efficiently implemented into hardware, as matrix multiplication which are a level 3-BLAS operation. To achieve such a transformation, Ji et al. decided to use the same negative samples for given amount of training samples. The training samples that will have the same negative samples are chosen as follows: given a word w_c one can construct all the training samples where w_c is a context word. The following training batch X will result: $X = \{({w_{in}}^i, w_c) | w_c \; appears \; in \; the \; context \; of \; {w_{in}}^i\}$. The matrix multiplication can be represented in the following way:

$$\begin{bmatrix} w_c \\ w_{n_1} \\ \vdots \\ w_{n_k} \end{bmatrix} * \begin{bmatrix} w_{in}^1 \\ \vdots \\ {w_{in}}^{2m} \end{bmatrix}$$

where, $w_{n_1} ... w_{n_k}$ are the shared negative samples. The above Matrix multiplication will compute the loss for each training sample and the weights of the used vectors will be updated after each batch. Ji et al.'s method achieves a 3.6 fold increase in throughput, by only losing 1

3.3 Accelleration of word2vec by using GPU's

Seulki and Youngmin [?] focused on getting a better throughput on the SGM with the use of GPU's. As the SGM is a sequential algorithm, it 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. Since the dimensions of the word embeddings are completly independent of each other Seulki and Youngmin [?] tried to parallelize their update. 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 the time used to read the sentences, it could be a smart choice to use multiple GPU's. According to Seulki and Youngmin [?], 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 acomplished 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. [?]. The speedup was achieved, while maintaining the accuracy of the model. The problem with this and all the above optimization is that the code is not easily available or of use. Therefore, we need an optimized implementation of the SGM that is easily available. This is provided by Gensim [?], which will be outlined in the next subsection.

3.4 Gensim

Gensim [?] 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 compared to the orignal C code from Mikolov et al. Gensim as a python library, and because it was already used in related work [?] made the comparison easy and of value.
This concludes our overview of related work. The next Section will cover our implementation of the Skip-Gram Model with negative sampling.

4 BATCHED SKIPGRAMMODEL

This section will give an overview of our implementation. The proposed implementation is a slighlty altered version of the origina Skip-Gram Model with negative sampling (SGNS). The idea behind this altering, is to compute the loss for multiple words and context pairs at the same time, this is inspired from mini-batch training, a model often used in other machine learning problem. Instead of computing the loss function of each pair, we computed the sum over a certain number of pairs (i.e 2000) as a loss. The exact process will be described in the following paragraph.

4.1 Forwarding

As the computation of the new loss function 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 one of its context words.

4.1.1 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:

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

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

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

4.1.3 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.

$$E_v = \begin{bmatrix} \tilde{v_1}_1 & \dots & \tilde{v_1}_d \\ \tilde{v_2}_1 & \dots & \tilde{v_2}_d \\ \tilde{v_3}_1 & \dots & \tilde{v_3}_d \end{bmatrix}, \text{where } \tilde{v_i} = \begin{bmatrix} \tilde{v}_{i1} & \dots & \tilde{v}_{id} \end{bmatrix} \text{ is the embedding of } v_i.$$

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

4.1.4 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 2.2. 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 $S(i, j) = E_c(i, j) \cdot E_{vj}$. This will result in a $n \times k + 1$ Matrix S. Now we only have to negate the last k rows. The sum of each row represents the loss computed in Equation 2.2, for each word context pair. Since computation time is too long with this approach the loss function is altered

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

Table 1: 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

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

4.1.5 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. [**?**] the exact update per pair.
$L = -\sum_{(i,j) \in k \times n} S(i, j)$

This finishes the description of our batched approach. The next section will describe our experimental setting and the possibilities that exists to evaluate the quality of word embeddings.

5 EVALUATION

This section gives an overview of the used datasets, the used metric to evaluate our models, the configuration of our model, and finally an explanation about input shuffling

5.1 Dataset

In this implementation we only used the text8 [2] dataset. We chose this dataset for two reasons. First of all, it's a small dataset, that allowed us to do a lot of computations. Secondly, this dataset was used in related work [**?**] hence giving us a very good benchmark. The text8 dataset consists of 1702 lines of 1000 words, with a vocabulary of roughly 63000 words. 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 1. 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.

5.1.1 Subsampling. Subsampling is a technique introduced by Mikolov et al. [**?**] to reduce the size of the dataset while at the same time increasing the quality of the dataset, i.e increasing the quality of the word embeddings (WE). 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. Subsampling 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

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

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. [**?**] chose the following equation to compute the probability of a word w to be deleted from the data set:

$$P(w) = 1 - \sqrt{\frac{t}{f(w)}} \tag{6}$$

where $f(w)$ is the frequency of w, and t is a threshold set empirically. 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 on the text8 dataset. We did this by looking at a random set of sentences and judging the results manually. As Equation 6 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. An example of the first sentence with different sampling thresholds can be found in Table **??**. 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 **??**.

5.1.2 Min count. We also deleted every word that did not appear more than 5 times in our dataset. This idea was also introduced by Mikolove et al. [**?**]. 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. The weights of the network that represents the word, will only be updated when the word itself is used as input. As the word appears very infrequently the word representation will not differ much from the initialization, hence being very dependent from it. For all the above reasons, we applied this technique. Similar to subsampling, it will decrease computation time and should in theory improve the quality of the word embeddings.

5.2 Evaluating word embedings

Evaluating word embedding is not an easy task, such as evaluating the accuracy of a common classifier for example. 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, to ensure the quality of word embeddings other techniques have to be used.To define quality we first need to define a measure of similarity between two vectors. For this task we will use the cosine distance. Two vectors will be judged similiar if they point in the same direction.

5.2.1 Word similarity and wordsim353. Too measure the quality of our word embedding, in question of word similarity we must use a dataset to compare our self 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.

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

Table 2: Example of pairs and their rating in wordsim353

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

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 Table 2. We will rank our embeddings on the Pearson correlation coefficient between the cosine distance and the scores attributed by humans, as this is the common procedure. This concludes our description of the evaluation methods of word embeddings. The next section will present the configuration of our network used to optimize the SGNS.

5.3 Configuration of the network

The SGNS 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:` For this parameter we have to find a trade-off between, a parameter that is too high (better accuracy slower computation time) and too low (worse accuracy faster computation time). For smaller data sets a higher number of negative samples is often needed. In their original paper Mikolov et al. [**?**] 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 size the more training examples the network will have, but if the window size is too big the semantic meaning of the window size will be erased. Mikolov et al. [**?**] proposed a value between 2-10 - *5*
- `Dimension of the embedding`: The choice of the dimension 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 [**?**]. We, therefore, used Gensim to find the best embedding possible. - *100*
- `Batch size`: As described in Section 4, there is a trade off to find between quality and training time. We first used a batch size of 5000, but then decide after non conclusive results 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)*

This conlcudes our overview of the possible parameters of our SGNS, in the next section the reader will be given a description about the process of input shuffling.

5.4 Input Shuffling

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

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 an input at each epoch. The idea behind this technique is to present our optimizer with different surfaces of our loss function, in order to find the best optimum. Furthermore, it's easier for the neural network to escape a local minimum. As for example if a network, without using shuffling, had converged to a local minimum after one epoch it probably cannot escape it. 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 SGNS. Firstly, we read our words sequentially, therefore, words that only appear very early in the dataset 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 4. When we do not use input shuffling the same word will appear with all of its context words as training samples in a batch, therefore the updae will be an average of all of these training samples. But if we would use input shuffling instead, then the probability of a word appearing multiple time in one batch would be smaller. In consequence the accuracy of the model should be optimized hence, leading to a better convergence time. A definition of convergence time is given in the next section.

To summarize the evaluation, we deleted words that appear very often (subsampling) and very infrequently (min count) from our dataset, i.e text8 dataset. To evaluate the resulting word embeddings we defined the cosine distance, which is 1 minuse the cosine of the angle of two vectors. We will measure the quality of the embeddings by analyzing the correlation of the cosine distance of pairs of vectors to humanly judged similarity score of the corresponding words, and used the wordsim353 to collect these scores. The reader was also given an explanation about the configuration of our network and introduced to the concept of input shuffling. With all this knowledge we can know present the achieved experimental results in the next section.

6 RESULTS

As stated in the previous section we ran multiple experiments for each optimizer with the above configuration and datasets. . This Section will give an overview of the experimental results. Each subsection will give an explanation over the achieved result with a specific optimizer.

6.0.1 SGD. The first challenge for each optimizer was to find a correct learning rate. As SGD is the optimizer used in Gensim [**?**] we first tried the same learning rate as the default value in Gensim [**?**], i.e 0.01, 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, know 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.

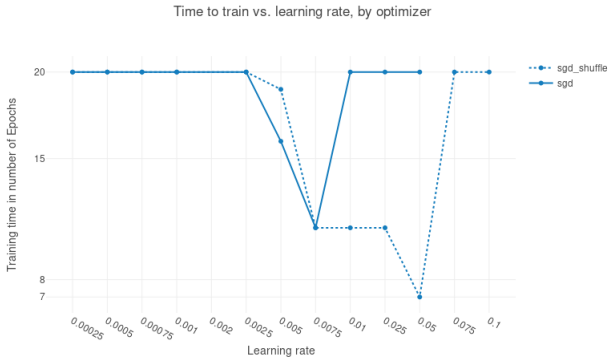

Figure 4: Training time Stochastic Gradient Descent with input Shuffling

6.0.2 Momentum and Nesterov. Momentum and Nesterov Accelerated Gradient (NAG) [**?**] both have an additional hyperparameter γ, that defines the percentage of the previous gradient that will be added to the current gradients. We set $\gamma = 0.9$ as this is a typical value and did not alter it during our experiments. Momentum and NAG alone respectively only slightly decrease or increase the convergence time. Momentum optimally converges in 9 epochs and NAG 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.

6.0.3 Adagrad. Adagrad [**?**] is a very interesting optimizing for creating word embeddings as it decreases the learning rate for very frequent occurring features, and vice versa. Words (features in our case) that appear very frequently (low learning rate) often do not have a semantic gain that is as important as words that appear less frequently (high learning rate) to their context words. So, in theory, Adagrad is particularly well suited for our task, as for example Pennington et al. used Adagrad in the training of Glove [**?**], another system which word embeddings. Our results confirmed empirically that Adagrad can work better for creating word embeddings. The model converged in 4 epochs. When combined with shuffling Adagrad only took 3 epochs to converge. These results confirms two hypothesis: first the tendency of the Skip-Gram Model to converge faster with input shuffling and second the big impact of having different learning rate for each feature. Another interesting fact to notice is that a higher learning rate combined with input shuffling

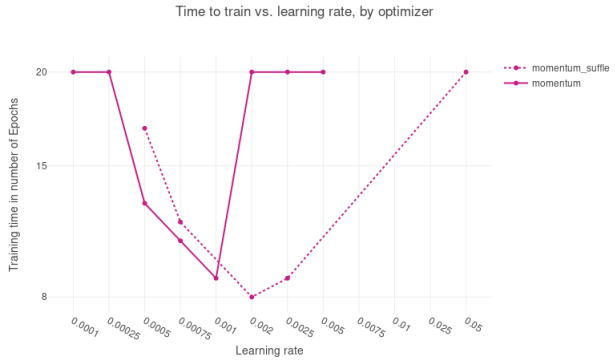

Figure 5: Training time Momentum with input Shuffling

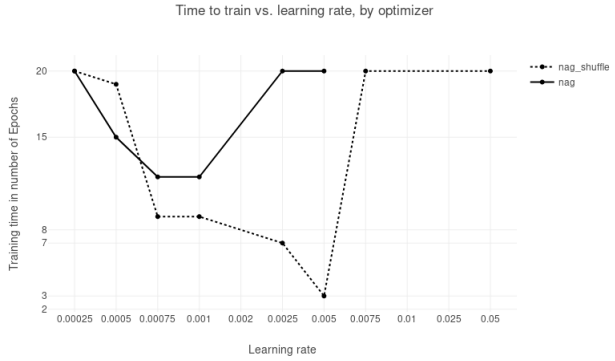

Figure 6: Training time Nesterov with input Shuffling

did not yield better results than without shuffling, in comparison to the earlier described optimzers. Adagrad performed best with a learning rate of 0.1, as shown in Figure 7.

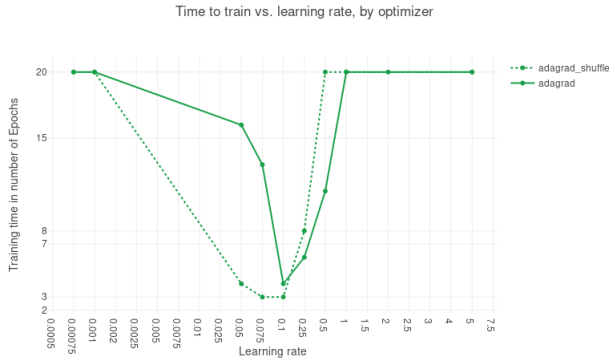

Figure 7: Training time Adagrad with input Shuffling

6.0.4 Adadelta. In theory Adadelta [**?**] should outperform Adagrad as it's an extension of the former. Adadelta doesn't have any learning rate to tune, therefore we only did 2 experiments, with and without input shuffling. We are aware of the fact that there are additional hyper parameters to Adadelta. We decide not to tune them for two reasons: first for simplicity reasons and second because their effect is not as high as the learning rate. The hyper parameter, ρ that defines the percentage taken when calculating the exponentially decaying average of past gradients was set to $\rho = 0.9$. Adadelta did not manage to achieve a word similarity of 0.66, in comparison to all the others optimzers. 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 3

Table 3: Convergence Time and Quality with Adadelta

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

6.0.5 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 a learning rate of 0.05 and 2 with shuffling and a learning rate of 0.001, as shown in Figure 8. Those are the best results achieved in any of our experiment. Exacctly as Adagrad, Adamd did not react to input shuffling the same way as SGD did.

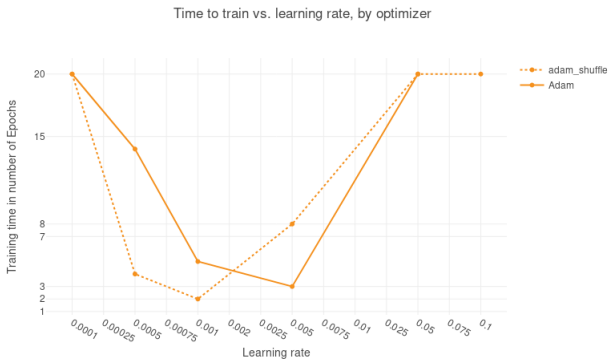

Figure 8: Training time Adam with input Shuffling

7 DISCUSSION

This section shortly discusses the empirical results and then extensively compares the findings of this work to the existing literature while trying to explain some of the differences. It is concluded by a subsection describing the limitations and possible extensions of this work.

7.1 Our work

This subsection discusses our findings. First it will analyze the possible reasons behind the influence of input shuffling to the learning rate. Second it will conclude with the discussion of unexpected results.

7.1.1 Shuffling and learning rate with SGD. In this subsubsection we will use the term SGD for SGD, Momentum and NAG, and advanced optimzers for Adagrad and Adam. The models was able to use a higher learning rates when SGD optimizers where combined with input shuffling in comparison to as when not, 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.
First, we suspect our batched version of the Skip-Gram model with negative sampling (SGNS), to be at the origin of this phenomena. In consequence to this batched version, when input shuffling is not used a lot of words will appear multiple times in the same batch, as explained in Section 5.4. Therefore, the gradients will be an average of all the training samples. When the input is shuffled, less words will appear multiple time which will make the gradients more precise.
Second, advanced optimizers have a way to counter-attack this issue, namely adaptive learning rates, explained in Section 6.0.3. Adaptive learning rates counter attack the issue in the following way: high frequency words will appear more often multiple times in a batch, they will also have lower learning rates. Therefore the impact of appearing in the same batch will be reduced.
The two above arguments could explain why a higher learning rate combined with SGD achieved better results with input shuffling but not when advanced optimizers are used.

7.1.2 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. 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 experiments were not replicate more than once.

7.2 Comparison to Gensim

This subsection will compare our finding extensively to Gensim [**?**]. As explained in Section 3.4, 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. This subsection is structured as follows: first it will describe the used configuration of Gensim, second it will compare Gensim to our implementation of the SGNS with the use of SGD and finally compare our model with Adam to Gensim. A graphic showing the comparison of our models to Gensim can be found in Figure 9.

7.2.1 Configuration of Gensim. To compare our-self in a correct manner we used the same dataset, with the same preprocessing parameters, i.e subsampling and min count. The hyper parameters of the network (window size, embedding size, number of negative sample, exponent to which the unigram distribution, which decides how a negative sample is used, is raised) are equivalent to our parameters described in Section 5.3. The only difference with our model is the learning rate. Gensim has a starting learning rate of 0.025 and linearly decrease it at every epoch to 0.0001.

7.2.2 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 time since our loss function is not exactly the same. We compute the loss for multiple training samples by taking the sum over each score which is individually used as a loss 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. Know 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.

7.2.3 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.

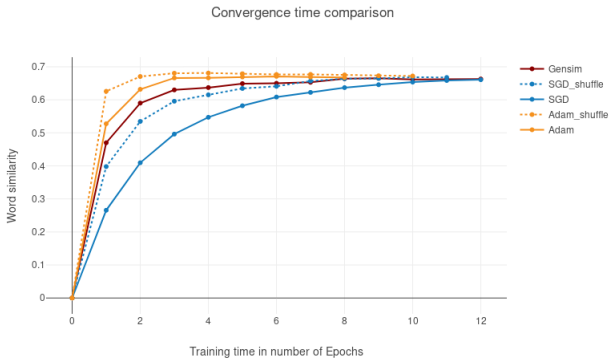

Figure 9: Convergence time of SGD and Adam compared to Gensim

7.3 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

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

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.

Problem with a small dataset:

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.

Problem with using only one dataset:

It has been shown that some optimizers perform better with specific shapes of loss functions. To make a compelling argument it's necessary to show that our model with the use of input shuffling and Adam as its optimizer also outperforms Gensim with other data sets.

Finally, as it wasn't the goal of our implementation to outperform Gensim in run time, one could improve an already existing, optimized version, with input shuffling and advanced optimizers and should achieve a better run time than Gensim.

8 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 task. We decided to use advanced optimizers and input shuffling to achieve a better convergence time. 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. The batched version allowed us to compute more models and analyze the convergence time faster. We used the text8 dataset to train our model and word similarity as a quality measure for the word embeddings (WE). To compare our work, Gensim was used, which holds a state of the art implementation of the SGNS. 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.