Lehrstuhl für Data Science

Optimizations of the Skip-Gram Model with negative Sampling

Bachelorarbeit von

Cédric Milinaire

Prüfer
Prof. Dr. Michael Granitzer

April 14, 2019

Contents

Abstract

The Skip-gram Model with negative sampling (SGNS) is an 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 SGNS, not much work has gone into optimizing the convergence time. Therefore our work focused on the latter. We used two techniques to achieve a better convergence time, namely advanced optimizers and input shuffling. We compared our work to the state of the art implementation Gensim. We used the Text8 dataset to train our model, and measured the quality of our word embeddings with the wordsim353 dataset, which measures the quality of word embeddings by judging the similarity of different words. We trained our model with multiple advanced optimizers: momentum, nesterov accelerated gradient, Adagrad, Adadelta and Adam. We also applied input shuffling to all of 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 increase the convergence time of SGNS. This leads to a new question: can the throughput of words that was achieved with SGD, also be achieved with advanced optimizers? In future work these findings need to be confirmed with larger datasets.

List of Figures

List of Tables

gensim proper # rethink configuration of the network # add exmaples wordsim # datasets stats proper adadelta pseudo code algorithm implement better version of getting indices simulations with and wihtout calculations ?? verify gensim parameters describing sgd and optimizing

1 Introduction

Insert a short introduction to word2vec. and quickly talk about machine learning blablbablah.

Machine learning is the science of algorithms the execution of a certain task, for example the classification of images into categories. The goal is that after a training period the model will be able to execute the task. The training is done on a so called training dataset, after sufficient training samples the model will have learned the appropriate task and will be able to perform well on the wanted task. An application for machine learning are word embeddings. Word embeddings are vectors that represent words. This is usually done in a one hot manner. For a vocabulary of size n, we will create n vectors with dimension n. Each word will recieve an integer as an *id*. And the vector representation of the word will be as follows: every dimension of the vector will be 0, except for the one wich is equal to the *id* which will be 1. There lies a Problem in this approach, the dimensionality of the vector is very big, a typical size of a vocabulary is 3b. This representation does not capture any meaning /. Each vector has the same distance to each other. To tackle this issue, a lot of attempt were meade to create better word embeddings. And these have shown to facilitate a lot of tasks in natural language processing, such as machine translation and senteces classfication. Here there is a little trick at work to create word embeddings, as there is no real task to learn. Therefore the approaches to create word embeddings have used the following trick: Use a fake taks on a neural network, and then take the or some weights from the network and use these weights as embeddings. This was also the technique used by Mikolov et al., which ntroduced the Skip Gram Model, an algorithm to create word embedings. This work focused on improving the convergence time of the algorithm.
We will first give a short introduction to the Skip-Gram model, then discuss related work, and see that a lot of work has got into optimizeing the throughput of the skip gram model but not in getting a better convergence time. Then we will describe our implementation of the SGNS followed by the description of our results. There we will first describe our

dataset, the measure we used to compare the quality of word embedings. And finaly we will discuss our work by comparing it to the state of the art implementation gensim.

2 Background

Es muss ein Absatz unter jeden berpunkt, auch ywischen Background und 2.1. Ich kann dir wenn du magst die Folien von meinem Seminar nachher schicken, das hatte ich ja beim Granityer und da hat er ein bisschen drauf geschrieben, was er bei Scientific Writing wichtig findet, ich wei- aber nicht mehr, wie hilfreich die Folien sind. Aber das steht auf alle F'lle drin.

2.1 The Skip-Gram Model

The Skip Gram Model is a model used to embed words into vectors, by analyzing the context in which the words happens. It's trained to predict the words that are next to the input word w. It's achieving this by maximizing the following equation:

$$\prod_{t=1}^{T} \prod_{-m<j<m} p(w_{t+j}|w_t) \tag{2.1}$$

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. This equation can be transformed into sums by using log probabilities:

$$\sum_{t=1}^{T} \sum_{-m<j<m} log(p(w_{t+j}|w_t)) \tag{2.2}$$

where the parameters are the same as in Equation 2.1.

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

$$p(w_{t+j}|w_t) = \frac{exp(\tilde{v}_{w_{t+j}}^T v_{w_t})}{\sum_{w=1}^{v} exp(\tilde{v}_w^T v_{w_t})} \tag{2.3}$$

Here $\tilde{v}_{w_t}$ and v_{w_t} are the vector representations. The model stores vector representations of each word. One for the input words and another for the context word. Here $\tilde{v}_{w_t}$ is the input vector and v_{w_t} the output vector. There lies a problem in the approach with a classical softmax. As a matter of fact it is unsuitable to compute the softmax. For the computation of $\sum_{w=1}^{v} exp(v_w^T w_t)$ 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. Consequently different solutions were proposed by Mikolov et al. [Tom+13]. The first one is to use a Hierarchical soft max introduced by Morin and Bengio [MB05]. 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 this makes it suitable to compute the softmax. Another possibility is the use of negative sampling which is used in the original word2vec implementation [Tom+13], which we shall discuss in the next section.

2.2 Negative Sampling

An alternative to the Hierarchical Softmax is Noise Contrastive Estimation (NCE) which was introduced by Gutmann and Hyvärinen [GH10], and first applied to NLP by Mnih and Teh [MT12]. The idea behind NCE is to distinguish targets words from noise. It does so by reducing the problem to a logistic regression task, and does it by maximizing the log probability. The skip-gram Model 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. [Tom+13] simplified NCE and called it Negative Sampling. Let's dive into it.
The idea behind 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 will set $p(c|w) = 1$, here p is the score for our logistic regression. Then select K random words k_i from the corpus data and set $p(k_i|w) = 0$, more one the random distribution later. We will 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 will use logistic regression to update the weights of the k selected context words and c. By doing so we will only have to update $k + 1$ output nodes.

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

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

We see that to compute our objective function we will only have to compute the sum over K. Which in practice is very small (2-20). Too put things in perspective lets imagine our data set consists of 100000 words, we set $K = 2$ and let's say that each output neuron has weight vector v with $|v| = 300$. 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. [Tom+13] used the following unigram distribution:

$$P(w) = \frac{f(w)^{\frac{3}{4}}}{\sum_{w_k \in W} f(w_k)^{\frac{3}{4}}} \tag{2.4}$$

where $f(w)$ is the frequency of w in the Vocabulary W. The value of $\frac{3}{4}$ is set empirically.

It's quite easily observable that this approach will outperform the classical softmax in computation time. Now the question arises if the accuracy is good enough enough but according to Mikolov et al. [Tom+13] the negative sampling method "is an extremely simple training method that learns accurate representations". As a matter of fact Mikolov et al. [Tom+13] reported a 6% accuracy improvement in comparison to a Hierarchical Softmax model. We now have enough background knowledge about

word2vec and the skip gram model to look at how it can be optimized. In the next section we are going to cover what has already be done.

2.3 Optimization of the Skip Gram Model

Due to the popularity of the skip gram model, a lot of research went into optimizing it. This research can actually be divided into two categories, parallelization, and the optimization of the accuracy of the algorithm by allowing words to have multiple meanings. This section will first give an overview of parallelization and then present one paper that focused on context sensitive word embeddings.

2.3.1 Parallelization

In the original model the optimization is done with Stochastic Gradient Descent (SGD), which is a sequential algorithm. This process does not favor a parallelization. To deal with this specific problem Mikolov et al.[Tom+13] used a Hogwild tree proposed by Recht et al.[Rec+11]. The approach is to allow multiple threads to access a shared memory, in this case the single model. Therefore overwriting errors are bound to happen. But according to Recht et al.[Rec+11] the overwriting errors won't lead to a significant accuracy loss if the data access isn't too frequent. 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 subsections.

Parallization in shared and Distributed Memory

The first parallization solution which was proposed by Ji et al. [Ji+16], is to try to reduce the cost of our vector multiplication. The main idea in this paper is to convert the level 1-BLAS vector to vector operations to a level-3 BLAS matrix multiplication operation. This is achieved, buy 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, under the assumption that we will not loose accuracy by sharing the

same negative samples, into a matrix multiplication. The matrix multiplication can be represented 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.

Parallelization by the use of caching

This idea was proposed by Vuurens et al. [VEV16]. The architecture used here is the basic skip gram model with an hierarchical soft max. The general idea is to cache the most frequent used nodes of the binary tree used to memorize the probability distribution, and update them on the shared single model after a certain amounts of seen words (the paper used the number 10). The paper produced interesting results as they managed to increase execution time by increasing the number of cores used for the calculation. This is very powerful because in the original implementation the exectution time regressed after 8 cores, this seems to indicate that too much overwriting was happening, as the number of concurrent threads surpasses a certain threshold. This can be seen in 2.1, where c31 is the model proposed by Vuurens et al.[VEV16]. The model did not suffer any accuracy loss in comparison to the original Word2vec model.

2.3.2 Context sensitive word embedding

A word does not always have the same meaning according to it's context. This is a problem that is not addressed by word2vec and the general skipGram model. Some new models, that have taken this issue into consideration, were proposed. A lot of work has be done in this direction, Liu et al.[Liu+15], Bartunov et al.[Ser+15] for example, but the the one reporting the best results is Liu et al. [LQH15]. The main idea is to change

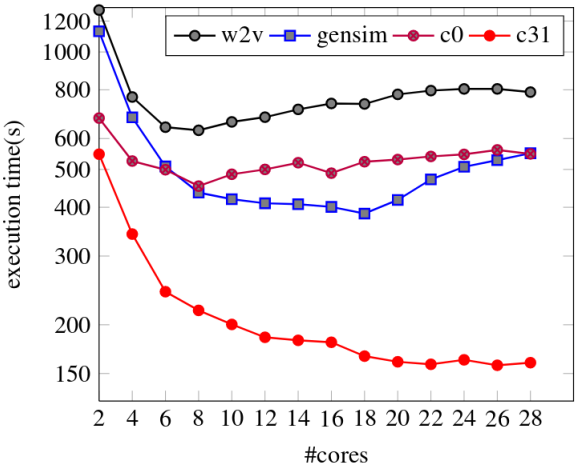

Figure 2.1: Comparasion of the execution time in relation with the number of used cores [VEV16]

the way we compute the and variables we use in our conditional probability. The idea is to look if a word given a certain context word matches to a topic. Bank would match too finance given the context word money. Bank would also match too nature if river was the given context word. But bank would not match too nature with the context word money. Now one could ask himself how to achieve such a context sensitive word embedding, first we have to introduce new variables, therefore let's look at the objective function used: First let's take a look at the objective function:

$$J(\Omega) = \sum_{(w,t,c)\in D} \sum_{(w,\tilde{t},\tilde{c}\in\tilde{D})} max(0, 1 - g(w,t,c) + g(w,\tilde{t},\tilde{c}))\lambda||\Omega||_2^2 \tag{2.5}$$

This approach uses the same negative sample technique as described in the previous sections, D is the corpus data and $\tilde{D}$ is the set of negative samples and $\lambda$is the hyperparameter used for the standard L_2 standarization. What is interesting here is the function $g(w, c, t)$, where w is a word, c the context word, and t the context in which the word appears, g is defined as follows:

$$g(w, c, t) = u^T \sigma(w^T M^{[1:k]} t + V_c^T (w \oplus t) + b_c) \tag{2.6}$$

where, u, V_c, b_c are standard parameters for a neural network, $\oplus$ is the vector concatenation, while the most important parameter is $M^{[1:k]}$, which a tensor layer, the tensor layer is used because of its ability to model multiple interactions in the data, as this will

be useful for multiple contexts. They used SGD for the optimization of this objective function. They achieved really interesting results as shown in 2.2.

Words	Similar Words
bank	depositor, fdicinsured, river, idbi
bank:1	river, flood, road, hilltop
bank:2	finance, investment, stock, share

Figure 2.2: "Nearest neighbor words by our model and Skip- Gram. The first line in each block is the results of Skip-Gram; and the rest lines are the results of our model" [LQH15]

2.4 Optimizers in Machine learning

The goeal of learning in machine learning is to minimze an objectiv function $J(\theta)$, where θ is the set of all parameters in our model. This happens by updating the parameters θ at every training time step t. We will denote θ_t as the parameters of our model at the t^{th} time step. In this work we only examined stochastic gradient descent algorithms.

2.4.1 Stochastic Gradient Descent

The idea in stochastic gradient descent optimization is to follow the path of steepest descent in the shape of our cost function. To get information about the shape of the objective function, one has to compute the gradients of all our parameters $\nabla J(\theta_{t-1})$, denoted asg_{t-1} . To follow the path of steepest descent we will have to subtract a portion of this term from our parameter. The magnitude of the portion is often referred to as the learning rate, denoted η. For illustration, this means that the gradients will give the direction of the optimization step, whereas the learning rate will give the amplitude of that step. An update at time step t will result in the following equation:

$$\theta_t = \theta_{t-1} - \eta \nabla g_{t-1} \tag{2.7}$$

SGD, through it's simplicity, is very limited, therefore some issues appear:

- The learning parameter is yet another hyper parameter to tune, as the optimum setting will largely vary depending on the training task and architecture of the network
- Learning rate schedules, that diminish the learning rate as the training progresses, are commonly accepted technique to improve accuracy. This schedule is most often set at the beginning of the training, and will be completely independent of the training set.
- Every parameter has the same learning rate

To tackle those issues numerous advanced optimizers were developed. They will be covered in the next sections.

2.4.2 Momentum

Momentum is a technique used to address one of SGD weak points. As a matter of fact because SGD can have trouble computing the optimum of objective function that are only steep in one directions. The problem here is that SGD often oscillates in the direction that is not very steep, and only takes small steps in the steep direction. This issue is addressed by SGD with momentum.

It does so by adding a percentage of the last gradient vector to the update vector. By doing so the gradient that go in the same direction will get bigger (building momentum) and gradients that go in different directions will anhill themselves. At the update t we will comppute our update vector v_t the following way:

$$v_t = \gamma v_{t-1} + \eta \nabla J(\theta) \tag{2.8}$$

The value 0.9 for γ has shown great results, but this, same as a learning rate, another hyper parameter that need to be tuned according to the specific task. And then we update our weights as usual: $\theta = \theta - v_t$

2.4.3 Nesterov

Momentum can be a powerful tool, but sometimes be it's own enemy. With momentum the learning algorithm often overshoots, and blows by the optima. Hence it will never converge. This problem was addressed by Nesterov. The idea behind his algorithm is to incorporate the momentum in the computation of our gradients. We will subtract the momentum vector, or just a fraction, from our parameters before computing the gradients. Therefore we will compute the gradients of the position where we would be with momentum, which will allow us to make a step in a better direction. The computation of the update vector will look the following way:

$$v_t = \gamma v_{t-1} + \eta \nabla J(\theta - \gamma v_{t-1}) \tag{2.9}$$

SGD wiht momentum and Nesterov accelerated gradient (NAG) has shown tremendous results in RNN's. But some of the earlier mentioned problems still remains. NAG, still treats every parameter the same way. Therefore we need a more complex optimization algorithm, that takes the frequency of a feature into account. Adagrad does just that.

2.4.4 Adagrad

Adagrad first introduced by [**adagrad**] is an optimzer that tries to apply different learning rates to different parameters, according to their frequency. The idea is to give very frequent features a small learning rate, and very sparse features a high learning rate. This can be very important for our task of word embeddings, as rare words in the corpus are more important then very frequent ones. As a matter of fact Pennigton et al. used this algorithm for their training of Glove, another word embedding system.
Each parameter θ_i, at time step t will have it's own learning rate $\eta_{t,i}$

$$\eta_{t,i} = \frac{\eta_0}{\sqrt{\sum_{i=1}^{t} g_{t,i}^2 \epsilon}} \tag{2.10}$$

where $g_{t,i} = \nabla J(\theta_{t,i})$ is the partial derivative of the loss function with respect to the parameter θ_i at time step t.
We see that each parameter θ_i has it's one learning rate. For a very frequent feature the sum of the previous gradients will be very high, hence the learning rate low. This

is how Adagrad acheives a different learning rate for each feature. Therefore we have $\theta_{t+1,i} = \theta_{t_i} - \eta_{t,i} g_{t,i}$, and we can now construct our global parameter update as follows:

$$\theta_{t+1,i} = \theta_{t_i} - \frac{\eta}{\sqrt{G_{t_{i,i}}} + \epsilon} g_{t,i}, \tag{2.11}$$

with $G_{t_{i,i}}$ being the diagonal Matrix of the sum of the squares of the graditents ($g_{t,i}$). There lies one weakness in this approach: the sum of the squares of the previous gradients grows constantly. This means that after a certain number of epochs the learning rate will be insufficient, to update the model. This issue was addressed by the Adadelta algorithm, that will be covered in the next session.

2.4.5 Adadelta

Adadelta [**adadelta**] not only solves the constantly growing sum problem, but also the fact that one does not have to tune the learning rate by not having one. The gist of Adadelta is that instead of taking all the gradients to compute the sum we will only take a fixed number w of gradients. But instead of inefficiently storing w gradients we will take the exponantially decaying average of the squared gradient, denoted $E[g^2]$. The average at time step t will be computed in the following way:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma) g_t^2 \tag{2.12}$$

where γ is a hyper parameter similar to the one used in momentum. That decides how much the past is weighted in contrast to the current gradient. Since Adadelta is an extension of Adagrad the square root of $E[g^2]$ is need, which become the RMS:

$$RMS[g]_t = \sqrt{E[g^2]_t + \epsilon} \tag{2.13}$$

Which gives us the following update rule:

$$\theta_t = \theta_{t-1} - \frac{\eta}{RMS[g]_t} g_t \tag{2.14}$$

Here we have two problems, first the learning rate is still a hyper parameter. And the units do not match. This is a problem xyz wanted to address. That if the parameters

would have units the update parameters would have the same parameters. Therefore they define the exponentially decaying average of squared parameter updates:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2 \tag{2.15}$$

As before we can know us the root mean squared error:

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon} \tag{2.16}$$

As at time step t $RMS[\Delta\theta]_t$ is unknown, we approximat it with $RMS[\Delta\theta]_t - 1$. Know we replac η with $RMS[\Delta\theta]_t - 1$ and get the final update rule:

$$\theta_{t+1} = \theta_t - \frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t \tag{2.17}$$

2.4.6 Adam

Adaptive Moment Estimation (Adam), is a more recent optimization algorithm. It also computes adaptive learning rates. In comparison to Adagrad and Adadelta, it does not only take into considaration the decaying average of the previous squared gradients but also the decay of the past gradients. Let's introduce :
$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$ as the decaying average of the previous gradients
and $v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$, as the decaying average of the previous squared gradients .
One problem arises when using this formula, m_t and tv_t are initialized as vectors of zero. Therefore they are biased towards zero. therofore et al. advised to use a bias corrected version:
$\tilde{m_t} = \frac{m_t}{1-\beta_1^t}$
$\tilde{v_t} = \frac{v_t}{1-\beta_2^t}$
The general update is done exactly in the same way as in Adadelta:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\tilde{v_t} + \epsilon} \tilde{m_t} \tag{2.18}$$

3 Methods

Describe the method/software/tool/algorithm you have developed here

In this work we replicated, the skip gram models in a python implementation. We did modify it to achieve a faster computation. The following task, was then to try to optimize it. This chapter will illustrate our proceeding. First it will give a short introduction to PyTorch, and then cover our implementation. In the latter part we will talk about the modified version of the skip gram model that we used and the interesting parts of our implementation

3.1 PyTorch

For our implementation we chosed the open source library PyTorch.[1]. Thruogh it's simplicity of use it's one of the most used libraries for machine learning. One of the most important features is the calculation of gradients by Pytroch. All gradients are calculated online, therofore there is no need for us to implement the calcluation of those gradients. The second important feature that we used, is the large variety of optimizes already implemented in pyTorch. They are proposed in the package torch.optim. This means that we did not need to change anything in our implementation except for one line. The last important feature of Pytorch that we used are the classes `Dataset` and `Dataloader`. Both of these classes are meant to work closely together. The Dataset interfacea has to function to offer namely: "`__len__`" and " `__getitem__`". Those are then used by a data loader object. it will construct a batch based on those two function. The loader object has a very nice feature it can shuffle the dataset before each epoch.

[1] pytorch

3.2 Implementation

3.2.1 Batched SkipGramModel

We implemented the model in the `Model` interface from torch.model. We used two embeddings as they are proposed in torch.embedding. they do esactly what we need, given an integer they return a vector. Hence they can be seen as the look up matrix. The real interesting part of our implementation is the way we modified the loss function to get a faster computation. We wanted to achieve a faster computation therefore we had the idea to use a kind of a batch. The idea is to update the weights of different vectors.

Forwarding

The forwarding method will take 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 need to have the same length n. Our training batch X can be seen as in the following way: $X = \{(v_i, c_i) \| _1 \in \{1, ..., n\}, n = |v|\}$. Where (v_i, c_i) is a context pair. The matrix must be of dimension $n \times k$, with k, being the number of negative samples per pair. First we will concatenate w and our Matrix A. This will result in a Matrix $\tilde{A}$. Know we need to get our Embedings. Therefore we will create a Matrik E_v of dimensionality $n \times d$ where d is the dimension of our word embeding which will store the word embeding of the i^{th} word from our input vector v in it's i^{th} row. We will do the same for the our Matrix $\tilde{A}$ This will result in a $n \times k + 1 \times d$ Array E_v. To compute the dot product of each word vector with it's pair and the negative samples we will need some definitions: let A_j be the j^th row of the matrix A, know let $E_c(i, j)$ be the d dimensional embeding of the word stored in $\tilde{A}(i, j)$. To know compute the dot product we will do a so called batch multiplication[2] 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. Know we only have to multply the last k rows with minus one. We then sum the resulting matrix of and multiply it with -1 and we get the wanted loss. It is best understand with an example.

[2] torch.bmm

Let's examine an example: let $(v_1, c_1)(v_2, c_2)(v_3, c_3)$ be our batch.
Input:

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

We then concatenate c and A, resulting in:

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

Embeddings:

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

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

Batch multiplication and negation of samples:

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

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

As we know have a way to compute our loss, we need to access the context pairs, therefore we also had to create our own way of doing it. We will explaine the process in the next paragraph.

3.2.2 Creating the context pairs

We needed to provide a way for the dataloader to access each word context pair. The straight forward way would be to go once over the whole dataset and create a list that stores all those pairs. The Problem with this approach is that it would not be possible to do so for a very large dataset. As one would get a list of over 10 billion entries, therefore

we propose a way to compute th i^{th} word-context pair of the dataset. Here we had two tasks, compute the number of possible pairs in our dataset, and given an index i return the wanted pair. Both of these task were respectively done in the methods `__len__` and `__getitem__`. For the first task we knew that every Senetece except the last one had a length of 20 words (more on this in Chapter 4).

Number of pairs in dataset

As the number was fixed we only needed to compute once the number of pairs in a sentece of length 20. This was done in the following way. We will distinguish two types of words in a sentence: first center words, those have the maxiumum amount of possible context words, and the border words that do not have this property because they are two close to the start or end of the sentence. We know that every center word has exactly 2 times the amount of the window size as context words. To compute the the number of border pairs in a sentence one only has to substract the length of the sentence with the double of the window size. The border pairs can be computed by the following equation.

$$\sum_{i=0}^{ctx\_window-1} ctx\_window + i \tag{3.1}$$

Finally we have to compute the number of pairs in the last sentence, here the challenge is to compute it if the sentence is shorter then twice the length of our window, because then the equation described previously does not work.A pseudo code description can be found in Algorithm 1. We will iterate over the length and add each time step we will compute the number of context pairs the given word has. First we check if the idex is smaller then our window. If this is the case we have to distinguish one special case, namely if our sentence is smaller then where we would be if we added the maximum number of window. If this is the case we add the length of our sentence -1 to the number of pairs (line 4). A short explanation: Instinctively one would first add the j pairs that are left of our word, and then compute the number of pairs right of the word. For this one would do $len\_last\_sen - 1 - j$, because len_last_sen -1 is the index of the last word in our sentence and to get the amount of words between the last and the current j^{th} word one needs to substract j. Therefore the number of pairs is $len\_last\_sen - 1 - j + j = len\_last\_sen - 1$. Then we have to check if our word is to close to the end of the last sentence (line 7) to add the context pairs. Here we apply the same procedure as above, add the window

amount context pairs from the left, we can do this because we know that $j \geq window$ from line2, we also need to add the words right to the current word therefore we take the same difference as before: $len\_last\_sen - 1 - j$. The final case is simply calculating the amount of pairs per center word, if there is any.

Algorithm 1 Computing the number of pairs in the last sen

```
1: for j = 0 to len_last_sen -1 do
2:   if j < window then
3:     if j + window ≥ len_last_sen then
4:       pairs_last_sen += len_last_sen-1
5:     else
6:       pairs_last_sen += j+window
7:   else if j ≥ len_last_sen − window then
8:     pairs_last_sen += len_last_sen -1 -j + window
9:   else
10:    pairs_last_sen += 2*window
11: return pairs_last_sen
```

Accessing each pair individually

Now we need to access each pair individually,a pseudo code description can be found in 2. Given an item index idx we need to find in which sentence the pair is. Because we know that how many pairs their is in each sentence, excpet the last one this is not a Problem. We only have to devide idx by the number of pairs that can be built within one sentence (line2). Once this is done we have access to the sentence where our pair is located. (line3). We have to find the id of our pair within our sentence. We know the number of pairs in all sentence before our sentence(this also holds true if our sentence is the last one), we can substract the number of pairs that are in all the snetnces before our sentence. (line4). Once this is done we only have to iterate over all the possible pairs in our sentence and keep count and return when we find the good pair (line 6-15).

Algorithm 2 Getting the context pair from the id

```
1: n_pairs_in_sen = border_pairs + center_pairs
2: id_sen = ⌊idx / n_pairs_in_sen⌋
3: sen = dataset[id_sen]
4: pair_id_in_sen = idx − id_sen ∗ (n_pairs_in_sen)
5: counter = 0
6: for i = 0 to len_sen do
7:     for j = 0 to window do
8:         if i + j < len_sen then
9:             if counter == pair_id_sen then
10:                return (word2idx[word], word2idx[sen[i+j]])
11:            counter += 1
12:        if i − j ≥ 0 then
13:            if counter == pair_id_sen then
14:                return (word2idx[word], word2idx[sen[i-j]])
15:            counter += 1
```

4 Results

This section will give an overview over the used datasets, the used metric to evaluate our models, the configuration of our model and finally the experimental results achieved.

4.1 Dataset

In this implementation we mainly used the text8 [1] dataset. We chose this dataset for two reasons. First of all it's a very small dataset, more on exact specifics later, that enabled us to do a lot of computations. Secondly this data set was used in a lot of related work (cite gpu, cpu,) hence giving us a very good benchmark. The text8 dataset consists of 1702 lines of 1000 words, there is no punctuation. Know 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 loss in accuracy empirically. We chose a sentences length of 20. Furtheremore we applied a technique called subsampling that reduces the data set size. We needed a second more larger dataset to confrim our results. We therefore chose the enwik9 dataset[2]. This datast neede some more preprocessing as it's plain html. We therefor used the script that can be found at the bottom of this page[3]. We applied exactly the same preprocessing technique afterwards. We will know explain the technique called subsampling.

4.1.1 Subsampling

Subsampling is a technique introduced by Mikolov et al. in [**mikolov**] to reduce the dataset size while at the same time increasing the quality of the dataset, i.e getting

[1] matt mahoney.zip
[2] matt enwik9
[3] script

```
['anarchism', 'originated', 'as', 'a', 'term', 'of', 'abuse', 'first', 'used', 'against', 'early', 'working', 'class', '
radicals', 'including', 'the', 'diggers', 'of', 'the', 'english']
```

(a) Treshold = 10^{-1}

```
['anarchism', 'originated', 'as', 'a', 'term', 'of', 'abuse', 'first', 'used', 'against', 'early', 'working', 'class', '
radicals', 'including', 'the', 'diggers', 'of', 'the', 'english']
```

(b) Treshold = 10^{-2}

```
['anarchism', 'originated', 'as', 'a', 'term', 'of', 'abuse', 'first', 'used', 'against', 'early', 'working', 'class', '
radicals', 'including', 'the', 'diggers', 'of', 'the', 'english']
```

(c) Treshold = 10^{-3}

```
['anarchism', 'originated', 'as', 'a', 'term', 'of', 'abuse', 'first', 'used', 'against', 'early', 'working', 'class', '
radicals', 'including', 'the', 'diggers', 'of', 'the', 'english']
```

(d) Treshold = 10^{-4}

```
['anarchism', 'originated', 'as', 'a', 'term', 'of', 'abuse', 'first', 'used', 'against', 'early', 'working', 'class', '
radicals', 'including', 'the', 'diggers', 'of', 'the', 'english']
```

(e) Treshold = 10^{-5}

Figure 4.1: Example of a sentence with different sampling tresholds

better word embeddings. Certain words that appear very frequently in the dataset such as: "the, as, it, of" do not give an intrinsic value to the words that appear in it's context. Therefore the idea of subsampling is to delete such words from the dataset. This will decrease the computation time 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 know, because words between have been deleted, do. Know the question arises, how one chooses to delete a word. Mikolov et al. chose the following equation to compute the deletion of a word w in the data set:

$$P(w) = 1 - \sqrt{\frac{t}{f(w)}} \tag{4.1}$$

where $f(w)$ is the frequency of w, and t is a threshold set empirically. (Mikolov et al.) recommends 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 simply looking at a random set of sentences and humanly judging the results. Stats about subsampling can be found in table 4.1. Examples in figure 4.1

Sampling Treshhold	0	1^{-1}	1^{-2}	1^{-3}	1^{-4}
Size of Dataset	16 mio	15mio	11 mio	8mio	4 mio

Table 4.1: Size of preproccessed text8 dataset according to sampling treshold

Min count

We also applied a threshold in the lower bound, we deleted every word that did not appear more than 5 times in our dataset. We got this tecnique from gensim [ŘS10], that introduced this parameter into their training. This is a good technique because of two reasons: first certain words of our data sets do not appear in a common lexicon (twigleg, melqu), or words from a foreign language (gastarbeiter), or names and acronyms. Secondly a word that only appears one time in our dataset will be very dependent on it's original initialization as it will only be updated with it's context pairs. Therefore we applied this technique. It should in theory, such as subsampling, improve the accuracy and decrease computation time.

4.2 Word similarity

Evaluating word embedding is not an easy task. 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 are of quality with other techniques. To define quality we first need to define a measure of similarity between two vectors. We will use the cosine distance for this task. The cosine distance of vectors v and w is 1 minus the cosine of the angle between the two vectors. The cosine distance is 0 if two vectors are pointing in the same direction. It's 1 if they are 90 degrees from each other and 2 if they are pointing exactly in the opposite direction. The cosine of the angle between two vectors is 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 then for the cosine distance:

$$cos\_dis(v, w) = 1 - \frac{v \cdot w}{|v||w|} \tag{4.2}$$

By subtracting 1 from the angle we create a distance 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. This technique a part from being shown empirically to work very well to measure the quality of word embedding 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.

Know that we have a measure to compute the similairty of two vectors let us introduce a way to rate the quality of our embedings.

wordsim353

To measure the quality of our word embedding we will need a dataset to compare our results too. We chose wordsim353[4] for this task, as it's the most used in similar literature. The data set consits of 353 pairs of words rated by humans on their similarity. The similarity score is in the range of 1 and 10. See 4.2 for two examples of such pairs. We will rank our embedings on the pearson corelation coefficient between the cosine distance and the scores attributed by humans.

```
['FBI', 'investigation', '8.31', 'Mars', 'scientist', '5.63']
```

Figure 4.2: Example of pairs and their rating in wordsim353

4.3 Configuration of the network

The skip gram model, has a lot of possible options, that can be tuned. We configured one model and then only changed the learning rate. The explanation of the parameters will be structerd as follow: `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 dataset a higher negative samples is often needed. We experimented early with 5, 10,15 - *10*
- `Context Window:` The bigger the window the more training examples the network will have, but if the window is to big the semantic meaning of the window will be erased. Mikolov et al. proposed a setting between 2-10. - *5*
- `Dimension of the embedding`: Here the choice is less obvious, the higher the dimension the better the embedding should be.(cite paper dimension vectors talk about gensim) but when tested on gensim 100 yielded better results than 300 - *100*

[4] wordsim353

- `Batch size`: As described in section FORWARD, there is a trade off to find between accuracy 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`: learnng rate, this hyper parameter was tuned in every optimizer therefore only the range will be indicated - *(1e-5,1)*

4.4 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 an input at each epoch. The idea behind this technique, is the same as the use of mini batches. We want to present our optimizer with different loss surfaces, so that it's able to find the best optimum. But both combined can be a very powerful, there always lies a risk that a mini-batch isn't a good representative of the true gradient. This way, by shuffling the input, one would avoid this bias. There are two reasons why we thnik that input shuffling is particulary 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 thing is that we used the special batch technique described in section x. 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 updat similar words at the same time. We then loose some accuracy. But if instead we would use input shuffling then in one batch the words would likely not be similar and therefore overwriting will be less likely. One thing to consider is that when using input shuffling we cannot use a sliding window size.

4.5 Convergence time

To optimize convergence time we have to define convergence time first. Therefore we used the already available implementation gensim. we know from [Ji+16]. that a score

of 0.66 in the task of word similarity is the state of the art. We also know after testing Gensim (more on this process in section discussion), that it takes 4 epochs to converge. Therefore we defined the following criteria for convergence:
$\rho - \rho_{prev} < 0.009 \vee or \rho = 0.66$
where ρ is the pearson coefficient with the wordsim353 task.

4.6 Results by optimizer

We ran multiple experiments for each optimizer. This section will only give an overview over the quantified results.

4.6.1 SGD

The first challenge for each optimizer was to find a correct learning rate. We first tried the learning rate suggested by gensim, and then performed a random search to find the best one.As expected a bell curve shape resulted, with a learning rate that is to high our model diverged and with a learning rate that is two low or model would not learn quick enough. The best setting that we found is 0.0075. We converged in 11 epochs. The second experiment was to add input shuffling. As seen is figure 4.3, for every learning rate the convergence time decreased. Our model know converges in only 7 epochs. Another intersting point to point out of figure 4.3 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.

4.6.2 Momentum and Nesterov

Momentum and Nesterov accelerated gradient both have an additional hyper parameter γ, that, as described in Section 2.4, 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. As the first one optimally converges in 9 epochs and the second one in 13. If we add input shuffling to

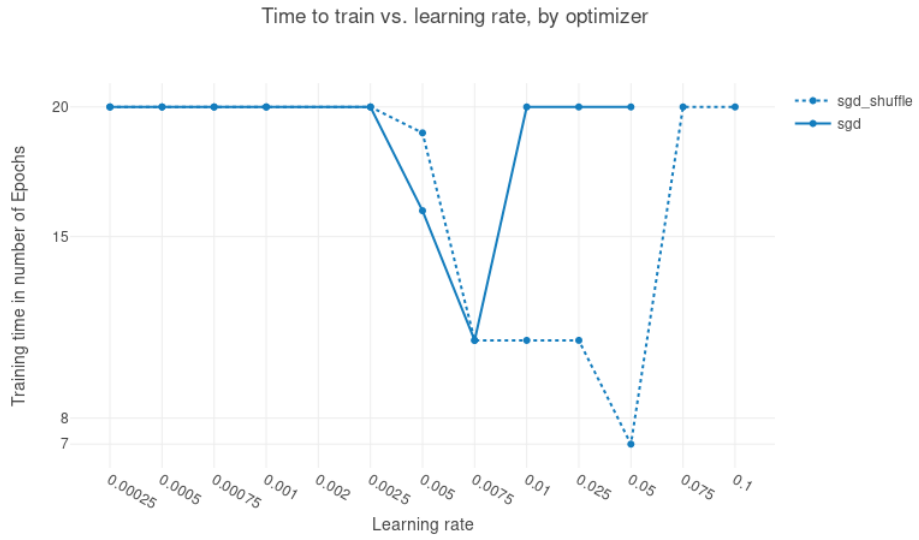

Figure 4.3: Training time Stochastic Gradient Descent with input Shuffling

the equation, interestingly the same things appear as with plain SGD. The convergence time gets better, 8 and 3 epochs until convergence respectively, and higher learning rate do also yield better results.

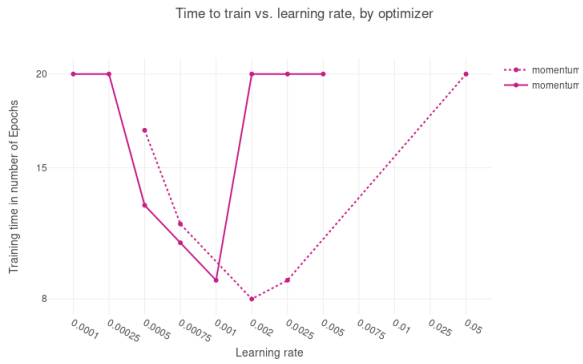

Figure 4.4: Training time Momentum with input Shuffling

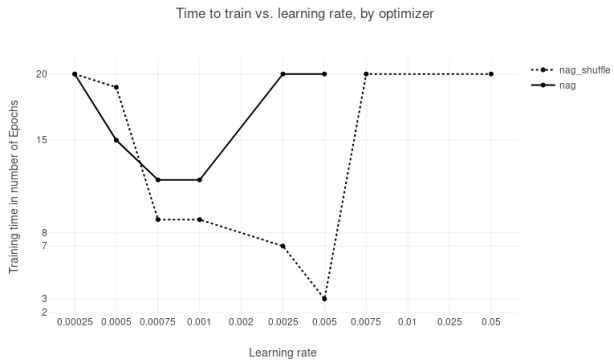

Figure 4.5: Training time Nesterov with input Shuffling

4.6.3 Adagrad

Adagrad is a very interesting tool for learning word embedding as they decrease the learning rate for very frequent occuring features, and vice versa for low frequent words.

Because words that appear very frequently often do not have a real semantic gain to their context words, it's good to have a low learning rate. So in theory Adagrad is particullary 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 then without shuffling. Both of our best results happened with a learning rate of 0.1.

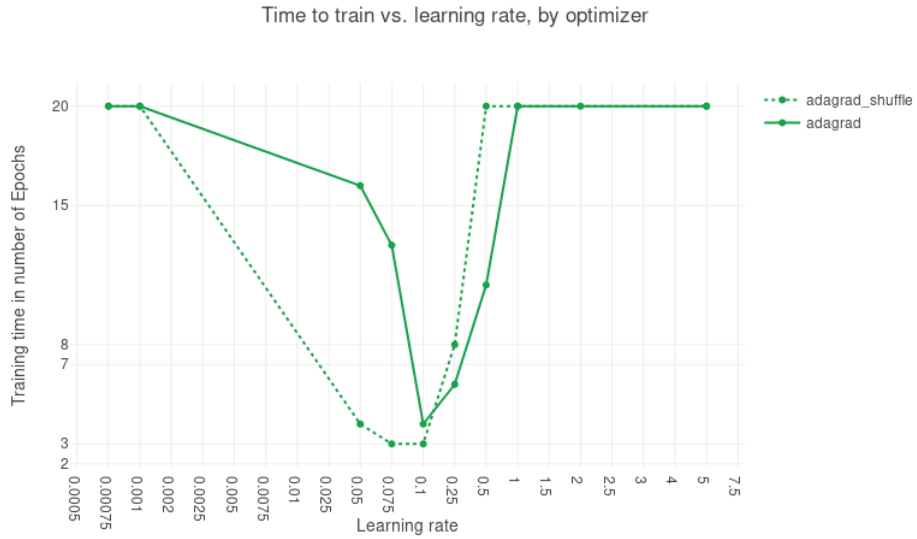

Figure 4.6: Training time Adagrad with input Shuffling

4.6.4 Adadelta

In theory Adadelta should outperfrom Adagrad as it's its upgrade. But it did not hold up to our excpectations. Because it didn't has any learning rate to tune, we only did 2 experiments, with and without input shuffling. 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 seen in table 4.2

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

Table 4.2: Convergence Time and Quality with Adadelta

4.6.5 Adam

Adam is the most advanced of all the optimizers used in our experiments. Did it therefore yield the best results? Indeed this was the case, as seen in figure 4.7, Adam converged in 3 epochs without shuffling and 2 with. This are the best result that we got with any optimizer. It's also intersting 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 opposit direction. We achieved our best result with input shuffling with a lower learning rate 0.001 then we used to achieve the best result without input shuffling 0.05

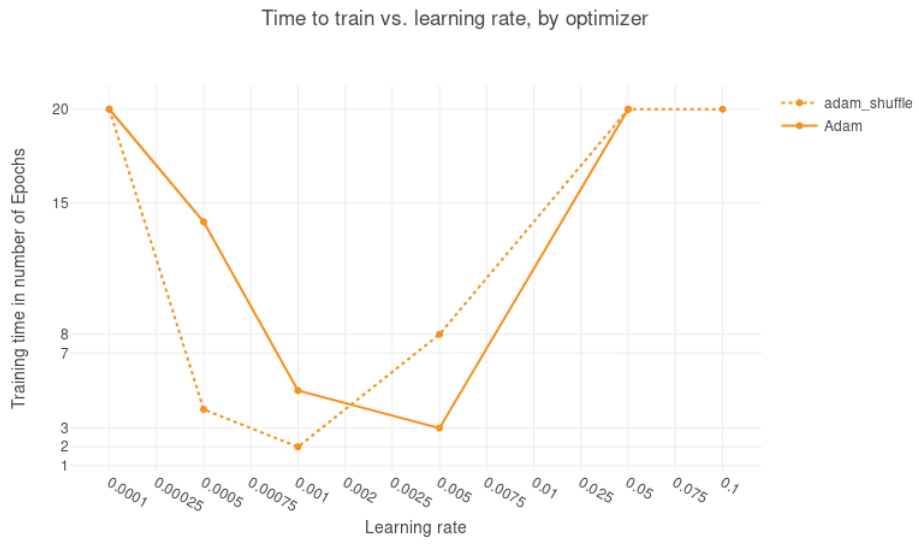

Figure 4.7: Training time Adam with input Shuffling

5 Discussion

In this secftion we will shortly discuss our results then extensively compare our work to related work, try to make sense out of our results, discuss some challenges faced and finally possible future work.

5.1 Our work

In this section we will quickly discuss our findings, and try to give some explanation to it.

5.1.1 Shuffling and learning rate with SGD

As shown in figures 4.3, 4.4 and 4.5, the model, when using SGD as an optimizer, was able to use a higher learning rate when the input was shuffled as when not. Therefore arises the questions why those this phenomen happen. One possibility is that the model is presented with a slightly different loss function every time, which is closer to the optimal loss function, therefore the steps taken by the optimizer are closer to the optimum and can therefore be bigger.

Large differences with nag and sgd when using shuffling

As swhon in figures 4.3 and 4.5, plain SGD and Nesterov Accelerated gradient, greatly differ in their convergence time when using shufflin in to comparison when not. We attribute these results partially to a good random initialization guess and not only input shuffling. Due to a lack of time these results where not replicate more then once.

5.2 Related Work

In this section we will compare our work to related work. We will first compare us to the baseline model, the original c implementation from Mikolov et al [**Mikolov**]. As Mikolov et al. did not use the same datasets as we, this comparison will loose in quality. We therefore compare ourself extensively with Gensim [ŘS10], a open source python implementation of the SGNS. Gensim is optimized to have a very high throuhgput which allowed us to, achieve a lot of computaions. Furthermore Gensim gave us access to the loss and the resulting word embedings, which made it easy to compare

5.2.1 word2vec

As mikolov et al. published the original paper which introduced the SGNS, it is of course relevant to compare ourselves to theim. The first important thing to take under account is that Mikolov et al. only trained their model on a very large google news dataset incorporating more than 3 billion words. This makes the comparison of our work more difficult. But we will make some assumptions, as they can be interesting. In their original paper, Mikolov et al. reported results from computations that took 1 and 3 epochs. We accord these good results to the very large dataset and furthermore as a matter of fact their result are better with 3 than with 1 epochs. We do not have any informtaion about the convergence. Hence it would be intersting to use their dataset for comparison.
One thing we can compare is the quality of our word embeddings. Mikolov et al. did not report any results on their model with the text8 dataset, but they therefore published their code, and Ji et al [Ji+16] tested the model on the text8 dataset. They reported a similarity of 0.63 on the wordsim task. This is obviously outperformed by all our models. We did not find any explanation on why those results differ as much.
The final assumption is that an advanced optimizer could maybe outperform SGD in terms of quality on a large data set. This will be discussed in further work.

5.2.2 Gensim

Gensim proposes a class `W2vec`[1]. The constructor has a lot of possible parameter, an extended list can be found in the appendix. In this section we will only describe the parameters we changed from the default setting. The description will be presented in the following way:
`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* – 5
- `window` (int) –*Maximum distance between the current and predicted word within a sentence* – 100
- `min_count` (int) – *Ignores all words with total frequency lower than this* – 5
- `workers` (int) – *Use these many worker threads to train the model (=faster training with multicore machines)* – 4
- `sg` (0, 1) – *Training algorithm: 1 for skip–gram; otherwise CBOW.* –1
- `negative` (int) –*Number of negative samples*– 10
- `ns_exponent` (float) – *Exponent in the unigram distribution, when choosing random samples, as shown in Equation 2.4* – 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 4.1.* – 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

[1] param

- `callbacks` (iterable of CallbackAny2Vec) – *Set of functions that will be executed at given training times, in order to follow the loss and the progress of the model in word similarity* – see Appendix

Gensim vs. SGD

First, as said earlier, we are not going to compare ourselves to gensim in runtime. This does not make any sense as our code is written in slow python and gensim is implemented in cython. [2]. There are a few interesting contradictions to note between gensim and our own implementation of SGNS. First of all the convergence time was not the same. There are different possibilities why this could be the case. First our batched approach could hinder performance in term of convergence as some overwriting may happen. Another difference between our implementation is the fact that gensim checks whether negative samples are real negative samples. This means for every negative sample, gensim checks whether or not the sample does appear in the context of the word somewhere in the data set, and if it's the case select a new random sample. Therefore the learning of the input and output context is optimized. The first hypothesis 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. Input shuffling does reduce the amount of overwriting that happens, as words next to each other will not be in the same batch as often anymore.

Gensim vs. Adam

The Adam optimizer did outperform the Gensim application in performance (only slighltlly: 0.01 correlation coefficient better) and convergence time. Adam converged in 2 epochs while Gensim in 4. To be sure of our results we ran each computation 40 time. The results can be seen in Figure 5.1.

[2] link to tutorial thesis

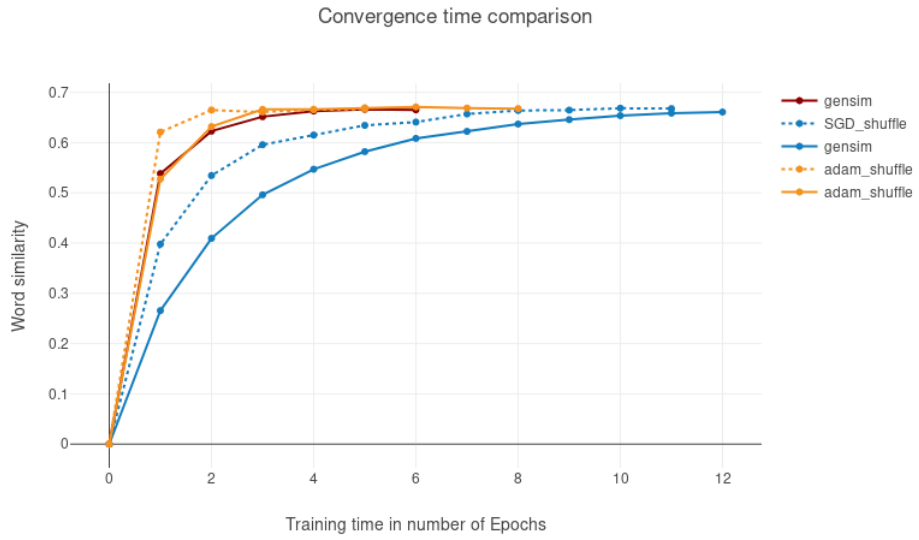

Figure 5.1: Training time Stochastic Gradient Descent with input Shuffling

5.3 Further Work

This work only focused on testing the approach of using advanced optimizers and input shuffling to improve the convergence time of the SGNS. While we did show that in theory it could work there still needs a bit of work to be done to show that this claim holds consistently.First and foremost an aspect of our implementation that can be prejudicial is that we only extensively tested our model with one small dataset. Both of these aspects are problematic. By using a very small dataset we do not use the model in the condition it is mostly needed for as the dataset used in practice a usually 3b words. There is a small argument that can be made for machine translation as the use of small parllel corpus is not unusual . But the main issue with using only one data set it that it has been shown that some optimizers perfrom better with specif shapes of loss functions. Hence it's necessary to show that Adam also outperfrom gensim with other data sets, to make a compelling argument. Furthermore having a better convergence time is not very useful if the runtime is hindered by a too complex computation, as for example second order optimization techniques. Therefore there is still need to show that Adam does not hinder computation time. One would have to improve one of the already efficient implementation in order to show that we can imporve the convergence time while at the same time maintaining the same convergence time.

5.4 Challenges faced

During our implentation we did make some mistakes, this section has the purpose of informing the scientific community to not make the same mistakes.

5.4.1 Using the wrong embeddings

To start with, we did not chose the same initialization value as gensim in our word embeddings. We initialized the input layer with a normal distribution between (-1,1). We did this because as we first started, we did indeed do the same thing as gensim, initializing the input layer with 0. But then our model did not seem to train well. Retrospectively we accord this to a learning rate too low. But back then we changed the input layer intitialization. After a few trainings we saw that we did not perform as good as gensim. And as we changed theim back to 0 we performed well again. Therefore this is a recommendation to future work to not set the initialization to (-1,1).

5.4.2 Batch size and loss function adjustements

During our experiments we faced a moment where we needed to us a very high learning rate (50) to achieve good results. As this is the complete opposite of what is standard we got suspiscous. To remedy the problem we made to changes. Firs we needed to find a good batch size. During the rise of the above training we used a batch size of 5000. We then decided to take a batch size of 2000.
Secondly, at the same time we did experiment with different loss functions. As our batched model does not have exactly the same loss function as introduced by Mikolov et al. [**mikolov**]. At the beginning of our experiment we took the average of all the scores stored in our final matrix. But then chose to take the sum as the training did not seem optimal.
With these two changes we did increase convergence time and word similarity, and while at the same time having a usual learning rate.
Know the question arises why this happened? Is this because the shape of the loss function better suits our advanced optimizers. Or because the loss function is closer to it's original form?

6 Conclusion

Summarize the thesis and provide a outlook on future work.

A Code

A.1 Gensim

```
from gensim.models.callbacks import CallbackAny2Vec
from gensim.models import Word2Vec
vocab = set(text8_ds1)
gensim_emb = dict()


class EpochLogger(CallbackAny2Vec):
    def __init__(self):
        self.epoch = 0
        self.cum_loss = 0
        self.loss_list = []
        self.ws_list = []
        self.prev_score = -1
        self.no_improvement =0


    def on_epoch_end(self, model):
        for word in vocab:
            gensim_emb[word] = model.wv[word]

        score = -1*wordsim_task(gensim_emb)[0][1]
        self.ws_list.append(score)

        if(score - self.prev_score < 0.0009):
```

```
            self.no_improvement +=1

        print("Epoch #{} end: cum_loss={}, ws_score={}".format(self.epoch,


        if(self.no_improvement == 2):
            print("No improvement in word similarity early stoppage")



        self.epoch += 1
        self.prev_score = score

    def on_batch_end(self, model):
        """Method called at the end of each batch.
        Parameters
        ----------
        model : :class:`~gensim.models.base_any2vec.BaseWordEmbeddingsMode
            Current model.
        """
        self.cum_loss += model.get_latest_training_loss()
```

B Math

C Parameters

D Dataset

- `hashfxn` (function) – *Hash function to use to randomly initialize weights, for increased training reproducibility.* – VERIFY
- `hs` (0, 1) – *If 1, hierarchical softmax will be used for model training. If 0, and negative is non–zero, negative sampling will be used.* – 0
- `corpus_file` (str, optional) – None
- `cbow_mean` (0, 1, optional) - Unnecessary since cbow is not used
- seed (int, optional) – Seed for the random number generator. Initial vectors for each word are seeded with a hash of the concatenation of word + str(seed). Note that for a fully deterministically-reproducible run, you must also limit the model to a single worker thread (workers=1), to eliminate ordering jitter from OS thread scheduling. (In Python 3, reproducibility between interpreter launches also requires use of the PYTHONHASHSEED environment variable to control hash randomization). - None
- `max_vocab_size` (int, optional) – Limits the RAM during vocabulary building; if there are more unique words than this, then prune the infrequent ones. Every 10 million word types need about 1GB of RAM. Set to None for no limit. - None
- `max_final_vocab` (int, optional) – Limits the vocab to a target vocab size by automatically picking a matching min_count. If the specified min_count is more than the calculated min_count, the specified min_count will be used. Set to None if not required. - None item `trim_rule` (function, optional) –Vocabulary trimming rule, specifies whether certain words should remain in the vocabulary, be trimmed away, or handled using the default (discard if word count ¡ min_count). Can be None (min_count will be used, look to keep_vocab_item()), or a callable that accepts parameters (word, count, min_count) and returns either gensim.utils.RULE_DISCARD,

gensim.utils.RULE_KEEP or gensim.utils.RULE_DEFAULT. The rule, if given, is only used to prune vocabulary during build_vocab() and is not stored as part of the model.

The input parameters are of the following types: word (str) - the word we are examining count (int) - the word's frequency count in the corpus min_count (int) - the minimum count threshold. - None

- `sorted_vocab` (0, 1, optional) – If 1, sort the vocabulary by descending frequency before assigning word indexes. See sort_vocab(). - None
- `batch_words` (int, optional) – Target size (in words) for batches of examples passed to worker threads (and thus cython routines).(Larger batches will be passed if individual texts are longer than 10000 words, but the standard cython code truncates to that maximum.) - None

Let's examine an example: $(w_1, c_1)(w_2, c_2)(w_3, c_3)$ be our batch. Therefore $pos_u = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix}, pos_v = \begin{bmatrix} c1 \\ c2 \\ c3 \end{bmatrix}$ and $neg_v = \begin{bmatrix} k_{1,1} & k_{2,1} & k_{3,1} \\ k_{1,2} & k_{2,2} & k_{3,2} \\ k_{1,3} & k_{2,3} & k_{3,3} \end{bmatrix}$
We then concatenate pos_v and neg_v, while negating neg_v resulting in:
$samples = \begin{bmatrix} c_1 & -k_{1,1} & -k_{2,1} & -k_{3,1} \\ c_2 & -k_{1,2} & -k_{2,2} & -k_{3,2} \\ c_3 & -k_{1,3} & -k_{2,3} & -k_{3,3} \end{bmatrix}$

We then multiply pos_u and $samples$ resulting in:
$scores = \begin{bmatrix} w_1 \cdot c_1 & -w_1 \cdot k_{1,1} & -w_1 \cdot k_{2,1} & -w_1 \cdot k_{3,1} \\ w_2 \cdot c_2 & -w_2 \cdot k_{1,2} & -w_2 \cdot k_{2,2} & k_{3,2} \\ w_3 \cdot c_3 & -w_3 \cdot c_3 k_{1,3} & -w_3 \cdot c_3 k_{2,3} & k_{3,3} \end{bmatrix}$ Finnally we sum up the score and multiply it with minus one to make it a minimizing problem:
$-(\sum_{i=1}^{3} w_i \cdot c_i - \sum_{j=0}^{3} -w_i \cdot -k_i, j)$

Bibliography

[02] "Placing Search in Context: The Concept Revisited". In: *ACM Trans. Inf. Syst.* 20.1 (Jan. 2002), pp. 116–131. ISSN: 1046-8188. DOI: 10.1145/503104.503110. URL: http://doi.acm.org/10.1145/503104.503110.

[GH10] Michael Gutmann and Aapo 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 (cit. on p. 5).

[Ji+16] Shihao Ji, Nadathur Satish, Sheng Li, and Pradeep Dubey. "Parallelizing Word2Vec in Shared and Distributed Memory". In: *CoRR* abs/1604.04661 (2016). arXiv: 1604.04661. URL: http://arxiv.org/abs/1604.04661 (cit. on pp. 7, 25, 31).

[LQH15] Pengfei Liu, Xipeng Qiu, and Xuanjing Huang. "Learning Context-Sensitive Word Embeddings with Neural Tensor Skip-Gram Model." In: *IJCAI*. 2015, pp. 1284–1290 (cit. on pp. 8, 10).

[Liu+15] Yang Liu, Zhiyuan Liu, Tat-Seng Chua, and Maosong Sun. "Topical Word Embeddings." In: *AAAI*. 2015, pp. 2418–2424 (cit. on p. 8).

[MT12] Andriy Mnih and Yee Whye Teh. "A fast and simple algorithm for training neural probabilistic language models". In: *arXiv preprint arXiv:1206.6426* (2012) (cit. on p. 5).

[MB05] Frederic Morin and Yoshua Bengio. "Hierarchical probabilistic neural network language model." In: *Aistats*. Vol. 5. Citeseer. 2005, pp. 246–252 (cit. on p. 5).

[Rec+11] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. "Hogwild: A lock-free approach to parallelizing stochastic gradient descent". In: *Advances in neural information processing systems*. 2011, pp. 693–701 (cit. on p. 7).

[ŘS10] Radim Řehůřek and Petr Sojka. "Software Framework for Topic Modelling with Large Corpora". English. In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. http://is.muni.cz/publication/884893/en. Valletta, Malta: ELRA, May 2010, pp. 45–50 (cit. on pp. 23, 31).

[Ser+15] Bartunov Sergey, Kondrashkin Dmitry, Osokin Anton, and P. Vetrov Dmitry. "Breaking Sticks and Ambiguities with Adaptive Skip-gram". In: *CoRR* abs/1502.07257 (2015). arXiv: 1502.07257. URL: http://arxiv.org/abs/1502.07257 (cit. on p. 8).

[Tom+13] Mikolov Tomas, Ilya Sutskever Qiu, Chen Kai, Corado Greg, and Dean Jeffrey. "Distributed Representations of Words and Phrases and their Compositionality". In: *CoRR* abs/1310.4546 (2013). arXiv: 1310.4546. URL: http://arxiv.org/abs/1310.4546 (cit. on pp. 5–7).

[VEV16] Jeroen BP Vuurens, Carsten Eickhoff, and Arjen P de Vries. "Efficient Parallel Learning of Word2Vec". In: *arXiv preprint arXiv:1606.07822* (2016) (cit. on pp. 8, 9).