Lehrstuhl für Data Science

Optimizations of the Skip-Gram Model with negative Sampling

Bachelorarbeit von

Cédric Milinaire

Prüfer

Prof. Dr. Michael Granitzer

April 23, 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 the SGNS, not much work has gone into optimizing the convergence time. Therefore our work focuses on the latter. In this work we used two techniques to achieve a better convergence time, namely advanced optimizers and input shuffling. 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 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 runtime.

List of Figures

List of Tables

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 semantics of the words, i.e. vector representations of words that are syntactically or semantically similar will be near to each other. By capturing semantic or syntactic meaning WE have shown to facilitate a lot of subsequent NLP tasks, such as entity recognition, machine translation or sentence classification. The first attempt to create WE with neural networks was made by Bengio et al. [Ben+03] but more recently Mikolov et al. [Mik+13] introduced a software package called w2vec that uses a simpler network and with this approach Mikolov et al. produced state of the art results. One of the proposed algorithms in this software package is the Skip-Gram Model (SGM). The SGM is an algorithm, that trains a neural network, on the task of predicting the neighboring words in a sentence. The weights of this network are then used as WE.

The SGM gained a lot of attention, as it achieved very good results for a very simplistic model. As a consequence, a lot of effort went into optimizing it. Most of this effort was trying to improve the throughput of the model, i.e the number of words that are processed per second. The SGM uses Stochastic Gradient Descent as an optimization algorithm and is therefore inherently sequential. To remedy this problem Mikolov et al. used Hogwild [Rec+11], where different threads can access a shared model and update it. As this is not an optimal solution Yi et al. [Ji+16] 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 [BY16]. Their goal was to parallelize the algorithm on GPU's. They, therefore, chose to parallelize the update

of the dimensions of each word representation. Both of these approaches and most of the literature are focused on improving the throughput of the model, but not the convergence time. Therefore one could ask if the convergence time of the SGM can be optimized while at the same time maintaining its accuracy. This work proposes an approach that uses advanced optimizers and input shuffling to optimize the convergence time of the SGM. In combination both of these techniques allowed us to decrease the convergence time of the SGM.

This work is structured as follows: Chapter 2 describes the SGM and its optimizations. Furthermore, we will give an extended explanation of the gradient descent optimizers used in this work. The reader is introduced to our Implementation in Chapter 3. Results are presented in Chapter 4 , where we will also describe used datasets, the measure applied to compare the quality of word embeddings, and finally the empirical results. The last part will focus on the discussion of our results, and possible future work in chapter 5 followed by a conclusion in chapter 6.

2 Background

This section gives an overview of the SGM and related work on its optimization. Furthermore the used optimizers in our experiments are explained.

2.1 The Skip-Gram Model

The SGM is a very simple model used to learn WE. The idea is to train the model, a neural network, on a fake task and then use some of the weights of the neural 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 2.1 as an example, where the context words are highlighted ones. An important thing 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.
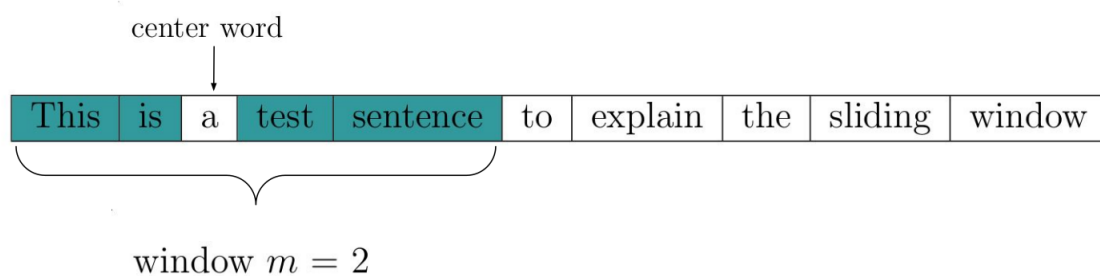
Figure 2.1: Example of a center word and it's context words

With these definition the fake task can be defined: given a pair of center and 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. One may ask how such

a network is built. The network consists of two matrices, a projection and an output layer. Both of these matrices store word embeddings. The first one, the projection layer, will store in each row the WE for one specific word. It will have dimension $T \times d$, with T being the size of our vocabulary and d the dimension of our WE. The second layer, the output layer, will store one WE in each column. The idea behind these 2 layers is that the input layer will represent our words as center word and the output layer as context words. The task of the network being to predict the probability of each word appearing in the context of a given center word, the following probability should be maximized:

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

Where T is the number of words in the corpus data, w_t the tth 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 2.1 can be transformed into sums by using log probabilities:

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

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, stored respectively in the projection Layer (first Matrix) and in the output layer (second Matrix). There is a problem with the classical softmax. As a matter of fact, it is unsuitable to compute the softmax. For the computation of $\sum_{w=1}^{v} exp(\tilde{v}_w^T w_t)$, the denominator in Equation 2.3 one has to go over the whole corpus data. As very big data sets are needed to train the model, this is not a solution. Mikolov et al. [Tom+13] porposed different solutions. The first solution is to use a Hierarchical softmax 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 makes it suitable to compute the softmax. Another possibility is the use of negative sampling which is discussed in the next section.

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 [GH10], and first applied to NLP by Mnih and Teh [MT12]. This section 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 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, which will be explained next.

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

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

$$p(c|w) = p(y=1|c,w) + \prod_{k \in K} p(y=0|k,c)$$

$$= p(y=1|c,w) + \prod_{k \in K} 1 - p(y=1|k,c)$$

$$= log((p(y=1|c,w)) + \sum_{k \in K} log(1 - p(y=1|k,c))$$

$$= log(\frac{1}{1+e^{-v_c \tilde{v_w}}}) + \sum_{k \in K} log(1 - \frac{1}{1+e^{-v_c \tilde{v_k}}})$$

$$= log(\frac{1}{1+e^{-v_c \tilde{v_w}}}) + \sum_{k \in K} log(\frac{1}{1+e^{v_c \tilde{v_k}}})$$

$$= log(\sigma(v_c \tilde{v_w}) + \sum_{k \in K} \sigma(log(-v_c \tilde{v_k})) \qquad \text{where, } \sigma(x) = \frac{1}{1+e^{-x}}$$

Where v_c and $\tilde{v_w}$, can be interpreted as before in Equation 2.3. The goal is know to maximize this objective function. Another way too look at this function, apart from logistic regression, and understand why it's working so well, is to assume that two vectors are similar if their dot product is high. Therefore we maximize the dot product of similar words (w,c) (c appears in the context of w) and minimize it for dissimilar words (w,k_i) (those were selected randomly). We see that to compute our objective function we will only have to compute the sum over the number of negative samples K. Which, is verys small in practice (2-20). To put things in perspective lets imagine our data set consists of 100000 words, we set $K = 2$. Assume each output neuron has a weight vector v with $|v| = 300$. 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:

$$P(w) = \frac{f(w)^{\frac{3}{4}}}{\sum_{t=0}^{T} f(w_t)^{\frac{3}{4}}} \qquad (2.4)$$

where $f(w)$ is the frequency of w_t. The value of $\frac{3}{4}$ is set empirically. Raising the unigram distribution to power of $\frac{3}{4}$ makes it less likelier for a word to be drawn if it appears often in the dataset in comparison to the basic unigram distribution. See figure

2.2 for an example. It's quite easily observable that this approach will outperform the
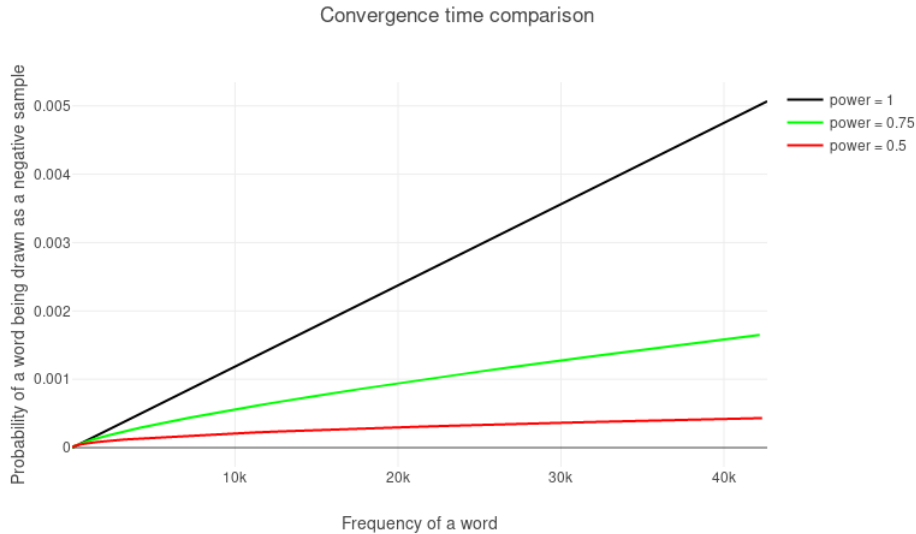
Figure 2.2: Probabililty of a word, of the text8 dataset (sampled), to be chosen at random according to its frequency and the power to which the unigram distribution is raised

classical softmax in computation time, as it only compute the sum over K output nodes. Now the question arises if the accuracy is good enough but according to Mikolov et al. [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. 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.

2.3 Optimization of the Skip Gram Model

Due to the popularity of the SGM, a lot of research went into optimizing it. This research can be divided into two categories, optimization of the throughput and the optimization of the algorithms accuracy. The latter was achieved by allowing words to have multiple meanings, also called context-sensitive word embedings. For our work, the optimization of the throughput is of big interest while the semantic optimization is aimed at giving

the reader a more holistic comprehension of the possible research directions. This section will first give an overview of the optimization of the throughput and then present one paper that focused on context-sensitive word embeddings.

2.3.1 Optimization of the throughput

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.[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. 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 parallelly and have access to the shared memory. Therefore overwriting errors are bound to happen. According to Recht et al.[Rec+11] the overwriting errors won't lead to a significant accuracy loss if the data is sparse enough. But in the case of NLP, the problem seems to be a bit more significant, and especially for word embedding, as many words share the same context words. There were several attempts at solving this issue, and we are going to cover a few of them in the following subsections.

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 a hierarchical softmax. The general idea is to cache the most frequently used nodes of the binary tree used to memorize the probability distribution and update them on the shared single model after a certain number of seen words (the paper used the number 10). The paper produced interesting results as they managed to decrease execution time by increasing the number of cores used for the calculation. This is very powerful because in the original implementation the execution time regressed after using more than 8 cores. It seems to indicate that too much overwriting was happening, as the number of concurrent threads surpasses a certain threshold. This can be seen in Figure 2.3, where c31 is the model proposed by Vuurens et al.[VEV16]. 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 to

8

Figure 2.3: Comparasion of the execution time in relation to the number of used cores [VEV16]

use more cores during the computation. As this approached focused on the Hierarchical softmax, in contrast to our work which used negative sampling, the next section covers optimizations of the Skip-Gram Model with negative sampling(SGNS).

Parallization in shared and Distributed Memory

The first parallelized 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 (Basic linear subprogram) vector to vector operations to a level 3-BLAS matrix multiplication operation, which are efficiently implemented into hardware and consequently faster. This is achieved, by using the same negative samples for each context word of a given word w. Instead of using for each context word a vector to vector multiplication we can transform this into a matrix multiplication, under the assumption that we will not lose accuracy by sharing the same negative samples. The matrix multiplication can be represented in the following way.

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

where w is our given word, $w_{n_1}...w_{n_k}$ are the shared negative samples, with $k \in [5, 20]$, and $w_{c_1}...w_{c_{2m}}$ are the words inside of the context window m of w, with $m \in [10, 20]$, also called a batch of input context words. After each batch the model updates the weights of the used vectors. This model achieves a 3.6 fold increase in throughput, by only losing 1% of accuracy. An aspect that is not as useful to us is that the experiments were done on CPU, as modern GPU's are often used in many machine learning libraries, as with CUDA for example, there still need work to be done to optimize it with GPU's. This was done by Seulki and Youngmin [BY16], which is described in the next section.

Accelleration of word2vec by Using GPU's

Seulki and Youngmin [BY16] focused on getting a better throughput on the SGM when using GPU's. As the SGM is a sequential algorithm, is is not easy to parallelize it, especially if one wants to parallelize the training of individual training samples. As the algorithm goes sequentially over a sentence, the samples next to each other, in order of execution, will almost every time have the same input word. Therefore it's very hard to parallelize at this level. To solve this problem, Seulki and Youngmin [BY16] was to parallelize the update of each dimension of the word embedding, since those are completely independent of each other. They achieved this by mapping each dimension to a CUDA thread while mapping each sentence to a CUDA block. As each CUDA block runs independently, the training of the sentences is parallelized, and the fact that sentences have different length is of no problem. If the execution time of the GPU kernel is greater than time used to read the sentences, it could be a smart choice to use multiple GPU's. According to Seulki and Youngmin [BY16], if multiple GPU's are used, there is a need for synchronizing the model, which will hinder run time performance. They achieved their best results with 2 concurrent GPU'S. The accomplished results were very good as they managed a 20x speedup compared to a single threaded CPU execution, which is a 3x increase in comparison to the original C code, with no loss in accuracy. The problem with this and all the above optimization is that the code is not easily available. Therefore we need an optimized implementation of the SGM that is easily available. This is provided by Gensim [ŘS10], which will be outlined in the next section.

Gensim

Gensim [ŘS10] 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 us 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 runtime. Gensim is an important tool as it allows us, as a python library, to compare our data rather easily. It was also used in related work [Ji+16] and is therefore of value, as it allows to us compare our work in a simplier way. This concludes our overview of the optimizations of the throughput of the SGM. In the next section, we give a quick outlook of what has been done in the field of context-sensitive word embeddings.

2.3.2 Context sensitive word embedding

A word has different meanings dependant on its context. This is a problem that is not addressed by the SGM. Some new models, that have taken this issue into consideration, were proposed. A lot of work has been done in this direction, Liu et al.[Liu+15], Bartunov et al.[Ser+15] for example, but the one reporting the best results is Liu et al. [LQH15]. The concept of this approach is to change the way we compute the objective function 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 to *finance* given the context word *money*. *Bank* would also match to *nature* if *river* was the given context word. But *Bank* would not match to *nature* with the context word *money* Now one could ask 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:

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

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 standardization. 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. The most important parameter is $M^{[1:k]}$, which is 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 2.4.

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

Figure 2.4: "Nearest neighbor words 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]

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

2.4 Gradient Descent Optimizers

The goal of learning in machine learning is to minimize an objective 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 tth time step. In this work, we only examine gradient descent algorithms.

2.4.1 Gradient Descent

The idea in gradient descent optimization is to follow the path of steepest descent in the shape of the objective function. To get information about the shape of the objective function, one has to compute the gradients of all our parameters $\nabla J(\theta_t)$, where J is our

objective function, and θ all of our parameters, t denotes the time step at which the parameters are taken. We will define g_t as the gradients of our parameters θ_t according to our objective function J. 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 as 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:

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

Where all the variables can be interpreted as above. This will also be the case for further Equations. There are three main variations of Gradient Descent, they differ at the moment they chose to update the parameters.

Stochastic Gradient Descent (SGD)

In this variant, the model will update the parameters after each training sample. The problem with this approach is that the variance of the direction of the training step will be very high, as each sample will influence the step individually.

Batch Gradient Descent

Here the update of the model happens after having gone through the entire dataset. The problem with this approach is that one updates the model way to infrequently which will lead to a high convergence time.

Mini-Batch Gradient Descent

Update the model after having gone through a specific number of training samples. The idea is that the batch will be representative of the entire dataset. Which will allow the model to learn quicker, as the updates are more frequent (in comparison to batch gradient descent) and less variant (in comparison to SGD).

Problems with Gradient Descent Algorithms

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

- The learning parameter is yet another hyperparameter 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 is only steep in one direction. 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 update vector to the current update vector. By doing so the gradient that goes in the same direction will get bigger (building momentum) and gradients that go in different directions will annul themselves. At the update t we will comppute our update vector v_t the following way:

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

Where v_t and v_{t-1}, respectively are the current and the last update vector, and γ a hyper parameter, usually set to 0.9, that defines the importance of the momentum term. We then update our weights as usual: $\theta_t = \theta_{t-1} - v_t$

2.4.3 Nesterov

Momentum can be a powerful tool, but sometimes be its own enemy. With Momentum, the learning algorithm often overshoots and blows by the Optima. Hence it will never converge. This problem was addressed by Yurii Nesterov [**nesterov**]. The idea behind his algorithm is to incorporate the momentum in the computation of our gradients. We will subtract the previous update 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:

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

Where the parameters as the same used with momentum in equation 2.4.2. SGD with momentum and Nesterov accelerated gradient (NAG) has shown tremendous results in RNN's. But some of the earlier mentioned problems still remain. 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 [DHS11] is an optimizer 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 than very frequent ones. As a matter of fact, Pennigton et al. used this algorithm for their training of Glove [PSM14], 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, and ϵ is a smoothing factor, so that one does not divide by 0, at the beginning of the computation and η_0 is the global learning rate.

We see that each parameter θ_i has it's one learning rate. For a very frequent feature the sum of the previous square gradients will be very high, hence the learning rate low. This is how Adagrad achieves a different learning rate for each feature. Therefore we have $\theta_{t,i} = \theta_{t-1,i} - \eta_{t-1,i}g_{t-1,i}$, and we can now construct our global parameter update as follows:

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

with $G_{t_{i,i}}$ being the diagonal Matrix of the sum of the squares of the graditents $(g_{t,i}^2)$. 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 [Zei12] 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 exponentially 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 hyperparameter 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 needed which become the Root Mean Squared (RMS) Error:

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

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 hyperparameter. And the units do not match. This is a problem Mathew Zeiler wanted to address. That if the parameters would have units the update parameters units would match. Therefore they define the exponentially decaying average of squared parameter updates:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2 \qquad \text{,where } \Delta\theta = -\frac{\eta}{RMS[g]_t}g_t \qquad (2.15)$$

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

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

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

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

2.4.6 Adam

Adaptive Moment Estimation (Adam) [**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 decaying average of the past gradients. Let's introduce :

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t \qquad (2.18)$$

as the decaying average of the previous gradients, with β_1 being a hyperparameter similar to γ in the previous optimizers, and

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2 \qquad (2.19)$$

as the decaying average of the previous squared gradients, with β_2 having the same role as β_1
One problem arises when using this formula, m_t and tv_t are initialized as vectors of zero. Therefore they are biased towards zero. Therefore a bias corrected version was introduced:

$$\tilde{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{2.20}$$

$$\tilde{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{2.21}$$

The general update is done exactly in the same way as in Adadelta:

$$\theta_t = \theta_{t-1} - \frac{\eta}{\tilde{v}_{-1} + \epsilon} \tilde{m_{t-1}} \tag{2.22}$$

with ϵ as a smoothing factor and η as the global learning rate. This concludes our overview of gradient descent optimizers, and more generally our background chapter. In the next chapter the reader will be given an outline of our implementation of the Skip-Gram Model.

3 Implementation

In this work our goal is to optimize the skip-gram model with negative sampling (SGNS), we did implemented a very slightly different version compared to the original w2vec implementation [Mik+13]. This chapter will illustrate our proceeding. First, it will give a short introduction to PyTorch, and then cover our implementation. The latter part explained the altered version of the SGM, and its challenges.

3.1 PyTorch

The implementation was done with the open source library PyTorch.[1]. Through it's the 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, therefore there is no need for us to implement the calculation of those gradients. The second important feature that was used, is the large variety of optimizes already implemented in pyTorch. They are proposed in the package `torch.optim`[2]. Two important features of PyTorch are the classes `Dataset`[3] and `Dataloader`[3]. Both of these classes are meant to work closely together. The Dataset interface has two functions to offer namely: "`__len__`" and "`__getitem__`". Those are then used by a data loader object, that will construct a batch based on those two functions. The loader object will facilitate the training process, as it facilitates the iteration over the dataset and the creation of batches. Furthermore, the data loader can shuffle the dataset before each epoch.

---

[1] https://pytorch.org/
[2] https://pytorch.org/docs/stable/optim.html
[3] https://pytorch.org/docs/stable/data.html

3.2 Batched SkipGramModel

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

3.2.1 Forwarding

As this represents the challenging part of our Implementation the the forwarding method is explained step by step. Each time step is illustrated to make the explanation clearer. Let $X = (v_1, c_1), (v_2, c_2), (v_3, c_3)$, be the training batch, where (v_i, c_i) is a training sample constituted by a word and on of its context words. *Input:*
The forwarding method will accepts 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. 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. This means the ith row will store the negative samples for the ith word context pair.
The input can be illustrated as follows:

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

Concatenation of samples:
The concatenation of the vector c and the Matrix A will result in a Matrix $\tilde{A}$, with:
$$\tilde{A} = \begin{bmatrix} c_1 & k_{1,1} & k_{2,1} & k_{3,1} \\ c_2 & k_{1,2} & k_{2,2} & k_{3,2} \\ c_3 & k_{1,3} & k_{2,3} & k_{3,3} \end{bmatrix}$$

Embeddings:

Know 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 ith word from our input vector v in it's i^{th} row. The same is done for the our Matrix $\tilde{A}$. This will result in a $n \times k + 1 \times d$ Array E_c.

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

$$E_c = \begin{bmatrix} \tilde{c}_1 & \tilde{k_{1,1}} & \tilde{k_{2,1}} \\ \tilde{c}_2 & \tilde{k_{1,2}} & \tilde{k_{2,2}} \\ \tilde{c}_3 & \tilde{k_{1,3}} & \tilde{k_{2,3}} \end{bmatrix}, \text{ where each entry of the matrix is a vector of dimension } d$$

Batch multiplication and negation of samples:

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 $j^t h$ row of the matrix A, let $E_c(i,j)$ be the d dimensional embedding of the word stored in $\tilde{A}(i,j)$. To the dot product is computed with a do a so-called batch multiplication[4] which will result in a matrix S where $S(i,j) = E_c(i,j) \cdot A_j$. This will result in a $n \times k + 1$ Matrix S. Now we only have to negate the last k rows with minus one. The sum of each row represents the loss computed in Equation 2.2, for each word context pair. Since computation time is too long with this approach the loss function is altered

$$S = \begin{bmatrix} \tilde{v}_1 \cdot \tilde{c}_1 & -\tilde{v}_1 \cdot \tilde{k_{1,1}} & -\tilde{v}_1 \cdot \tilde{k_{2,1}} & -\tilde{v}_1 \cdot \tilde{k_{3,1}} \\ \tilde{v}_2 \cdot \tilde{c}_2 & -\tilde{v}_2 \cdot \tilde{k_{1,2}} & -\tilde{v}_2 \cdot \tilde{k_{2,2}} & -\tilde{v}_2 \cdot \tilde{k_{3,2}} \\ \tilde{v}_3 \cdot \tilde{c}_3 & -\tilde{v}_3 \cdot c_3\tilde{k_{1,3}} & -\tilde{v}_3 \cdot c_3\tilde{k_{2,3}} & -\tilde{v}_3 \cdot \tilde{k_{3,3}} \end{bmatrix}$$

Loss function:

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

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

---

[4]Documentation of the batch multiplication can be found at https://pytorch.org/docs/stable/torch.html#torch.bmm

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

3.2.2 Creating the context pairs

We need to provide a way for the `dataloader` to access each word context pair. The straight forward way would be to go over the whole dataset once and create a list that stores all those pairs. This approach is not suitable for very large datasets. The amount of RAM needed to store a list of all the possible pairs for a small dataset (250k words, more on the dataset in Section 4.1) is roughly 4GB. As some used datasets are 100x bigger in practice, this is not a suitable solution. Therefore we propose a way to compute the ith word-context pair of the dataset, by only storing the dataset in the RAM. Here we had two tasks, compute the number of possible pairs in our dataset, and given an index i return the according pair. These task were done in the methods `__len__` and `__getitem__`.

Number of pairs in the dataset

For the first task, calculating the number of pairs in the dataset, we knew that every sentence except the last one had a length of 20 words (more on this in Section 4.1). As the number is fixed we only need to compute the number of pairs once in a sentence of length 20. This was done in the following way. We will distinguish two types of words in a sentence: first words that have the maximum amount of possible context words, and the border words that do not have this property because they are too 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. The border pairs can be computed by the following equation.

$$2 * \sum_{i=0}^{window-1} window + i \tag{3.1}$$

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 of the sentence and at each

time step compute the number of context pairs the given word has. First, we check if the index is smaller than our window. If this is the case, this word does not have the maximum amount of context words, as it is to close to the begining of the sentence. Therefore we have to distinguish distinguish one special case, namely if our sentence is smaller than the position of the given word plus the 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. Therefore, one would do $len\_last\_sen - 1 - j$ because len_last_sen -1 is the index of the last word in our sentence. To get the number of words between the last and the current j^{th} word one needs to subtract j. Therefore the number of pairs is $len\_last\_sen - 1 - j + j = len\_last\_sen - 1$.

Next we have to check if the 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 line 2. 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 number of pairs per center word if there are any.

Know we can simply add the number of pairs in our last sentence to the previously computed number, and return the number of pairs in our dataset.

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** $j + window \geq len\_last\_sen$ **then**
4:       pairs_last_sen += len_last_sen-1
5:     **else**
6:       pairs_last_sen += j+window
7:   **else if** $j \geq len\_last\_sen - window$ **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

The dataloader needs a way to access each pair individually, a pseudo code description can be found in 2. Given an item index idx we need to find the corresponding sentence.

Because we know how many pairs are in each sentence, except the last sentence, we only have to divide idx by the number of pairs that can be built within one sentence (line 2). This division also holds for the last sentence, as it's the only one with a different number of words is positioned at the end of the dataset. Once this is done we have access to the sentence where our pair is located. (line3). We have to find the index of the pair within the sentence. We now the number of pairs in all sentence before our sentence (this also holds if our sentence is the last one), therefore we can subtract the number of pairs that are in all the sentences before the sentence from idx, and will get the index of our pair within it's sentence (line4). Once this is done we only have to iterate over all the possible pairs in our sentence, keep count and return when we find the correct 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 = $\lfloor \frac{idx}{n\_pairs\_in\_sen} \rfloor$
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[sen[i]], word2idx[sen[i+j]])
11:          counter += 1
12:       **if** $i - j \geq 0$ **then**
13:          **if** counter == pair_id_sen **then**
14:            **return** (word2idx[sen[i]], word2idx[sen[i-j]])
15:          counter += 1

Adaptability to other datasets

The above-described implementation is limited to datasets formatted as ours, namely having all sentences except the last of the same length. Our model can be quickly modified to achieve the same results on any given dataset. To compute the length of the dataset one would have to use Algorithm 2 to compute the length of each sentence, and take the sum over the length of each sentence. To access the context-pairs the challenge lies in getting the id of the sentences. Therefore one would have to go over the dataset

and compute the number of pairs for each sentence, sum them up and wait until the sum is greater than the searched pair. Once this is done the algorithm stays the same.

4 Results

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

4.1 Dataset

In this implementation we only used the text8 [1] dataset. We chose this dataset for two reasons. First of all, its a very small dataset, that allowed us to do a lot of computations. Secondly, this data set was used in a lot of related work [BY16], [**cpu**] hence giving us a very good benchmark. The text8 dataset consists of 1702 lines of 1000 words, with a vocabulary of roughly 63000 words. Conventiently, 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 4.1. 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 section.

---

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

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

Table 4.1: Training and Convergence time according to choice of the length of sentences in text8 dataset

4.1.1 Subsampling

Subsampling is a technique introduced by Mikolov et al. [Mik+13] to reduce the dataset size while at the same time increasing the quality of the dataset, i.e getting better word embeddings with it. The idea behind subsampling is the removal of very frequent words such as: "the, as, it, of". These words do not give an intrinsic value to the words that appear in its context. Therefore the goal of subsampling is to delete such words from the dataset. This will decrease the computation time, as it will reduce the number of training samples, and should, in theory, increase the accuracy of the model. The increase in accuracy can also be explained by the fact that words that would not have appeared in the context of each other, may know do because words between have been deleted. To choose which word to delete, Mikolov et al. [Tom+13] 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. As Equation 4.1 is a probability, subsampling is not a deterministic procedure, words that may have been deleted with a threshold of 10^{-2} may stay in the dataset with a lower threshold, as shown in Table 4.3. Mikolov et al. recommend a value between 0 and 10^{-5}, depending on the size of the dataset. We experimented with different values and 10^{-4} seemed the most suited. We did this by looking at a random set of sentences and judging the results manually. An example of the first sentence with different sampling thresholds can be found in Table 4.3. 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 4.2.

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

Table 4.2: Size of the preprocessed text8 dataset according to sampling treshold

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

Table 4.3: Example of a sentence with different sampling tresholds

Min count

We also deleted every word that did not appear more than 5 times in our dataset. We got this technique from Gensim [ŘS10], that introduced this parameter into their training. This is a good technique because of three reasons: First certain words of our data sets do not appear in a common lexicon (twigleg, melqu), or come from a foreign language (Volksvereinigung), or are names and acronyms. Secondly, each document often has spelling mistakes, those (as long as the same spelling mistake does not appear too often, what should be avoided in practice) would be deleted by sampling too, as the words do not have any meaning. Lastly, a word that only appears one time in our dataset will be very dependent on its original initialization. This is the case because it will only be updated with its context pairs once, which is only a dozen of times in practice and then won't be updated any more. For all of the above reasons, we applied this technique. Similar to subsampling, it should in theory improve the quality of the word embeddings and will decrease the computation time.

4.2 Evaluating word embedings

Evaluating word embedding is not an easy task, such as evaluating the accuracy of a common classifier. We cannot split our data set into train and test set. As the task that the network is learning is of no interest to us. Therefore we need to verify that our embedding is of quality with other techniques. To define quality we first need to define

a measure of similarity between two vectors. This requires knowledge of the Cosine distance, which is introduced in the following section.

4.2.1 Cosine distance

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

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

The cosine of 0° is 1, it's 0 for two vectors that are orthogonal to each other and vectors that point in the opposite direction will have a cosine of their angle of -1. This is not a good distance measure as -1 is smaller than 0, and therefore two vectors pointing away from each other would be closer than two orthogonal vectors, but by subtracting 1 from the cosine of the angle we can create a good distance measure between the two vectors. This distance does not take into account any order of magnitude. Hence for our tasks, two vectors will be considered equal if they are of different magnitude but point in the same direction. This technique apart 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 similarity of two vectors let us introduce a way to rate the quality of our embeddings.

4.2.2 Word similarity and wordsim353

To measure the quality of our word embedding we will need a dataset to compare our results too. We chose wordsim353[2] for this task, as it's the most used in the related literature. The data set consists of 353 pairs of words rated by humans on their similarity. The similarity score is in the range of 1 and 10, an example for two of such pairs can be found in Figure 4.4. We will rank our embeddings on the Pearson correlation coefficient

---

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

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

Table 4.4: Example of pairs and their rating in wordsim353

between the cosine distance and the scores attributed by humans, as again this is what is done in the literature.

4.3 Configuration of the network

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

- `Negative Samples` Here we have to find a trade-off between, setting the parameter too high which will result in increased accuracy but a longer computation time. For smaller data sets a higher number of negative samples is often needed. In their original paper Mikolov et al. recommend a value of 5-20. We tested a few values in the range of 5 to 15, as 10 yielded state of the art results we chose this value. - *10*

- `Context Window:` The bigger the window the more training examples the network will have, but if the window is too big the semantic meaning of the window will be erased. Mikolov et al. proposed a setting between 2-10, as all our sentences are of size 20 we chose 5. - *5*

- `Dimension of the embedding`: Here the choice is less obvious, as the dimension needs to be high enough to capture the meaning, but cannot be too high as this leads to a decrease in performance as shown by Yin and Shen [YS18]. We, therefore, used Gensim to find the best embedding possible. - *100*

- `Batch size`: As described in section 3.2, there is a tradeoff to find between quality and training time. We first used a batch size of 5000, but then decide after non conclusive results (see 5.3.2) 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)*

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

There are two reasons why we think that input shuffling is particularly well suited for the skip gram model. The first one has to do with the fact that when we read our words sequentially that words that only appear very early will not benefit from the context words being already updated from others. The second idea is that we used the special batch technique described in Section 3.2. When using this technique and not using shuffling we will always have words that appear next to each other in a batch and will, therefore, update the same words at the same time. We, therefore, lose some quality. But if instead, we would use input shuffling then in one batch the words would likely not be similar and therefore only taking the average of a small part of pairs with the same words will be less likely.

4.5 Convergence time

To optimize convergence time we have to define it first. Therefore we used the already available implementation Gensim [ŘS10]. Gensim is an open source software that proposes an implementation of the SGNS in python. It is also written in cython, therefore

it has a fast computation time, but can be used inside a python implementation. Together with the knowledge from Ji et al.[Ji+16] that a score of 0.66 in the task of word similarity, with the text8 dataset, is the state of the art, we tested Gensim (more on this process in Chapter 5 and found out that it took 4 epochs to converge. Therefore we defined the following criteria for convergence:

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

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

4.6 Results by optimizer

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

4.6.1 SGD

The first challenge for each optimizer was to find a correct learning rate. As SGD is the optimizer used in Gensim [ŘS10] we first tried the same learning rate as Gensim [ŘS10] 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 4.1, 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 4.1 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.
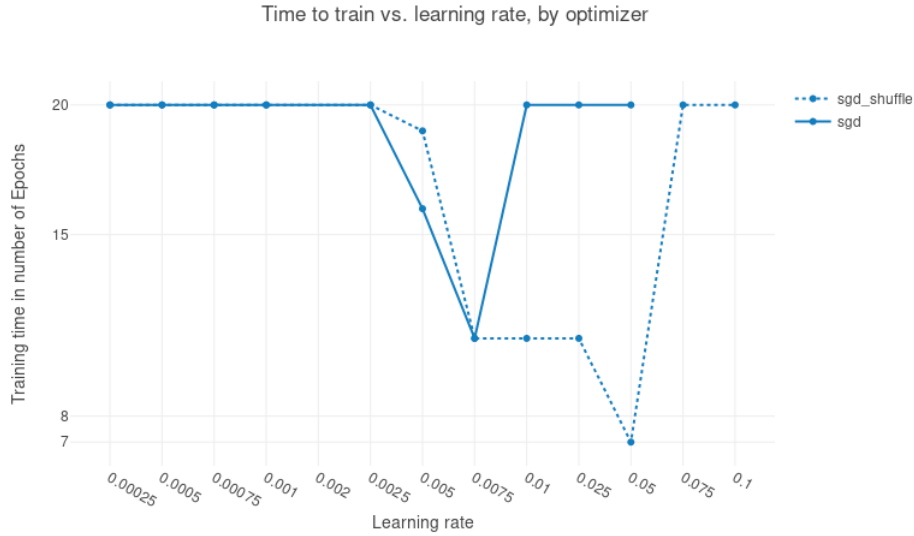
Figure 4.1: Training time Stochastic Gradient Descent with input Shuffling

4.6.2 Momentum and Nesterov

Momentum and NAG [Nes83] both have an additional hyperparameter 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 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.

4.6.3 Adagrad

Adagrad [DHS11] is a very interesting tool for learning word embeddings as it decreases the learning rate for very frequent occurring features, and vice versa (this is explained in detail in Section 2.4.4). Because words that appear very frequently often do not have a semantic gain that is as important as words that appear less frequently to their context

Figure 4.2: Training time Momentum with input Shuffling
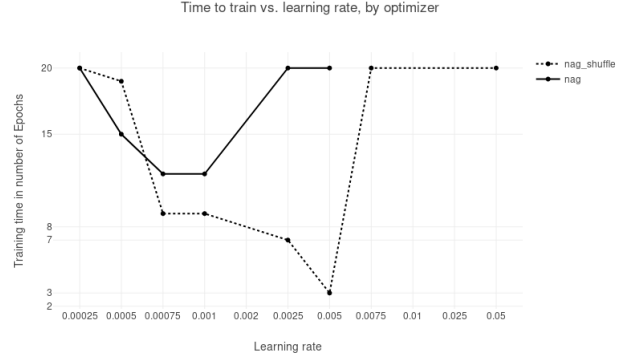
Figure 4.3: Training time Nesterov with input Shuffling

words, it's good to have a lower learning rate. So, in theory, Adagrad is particularly well suited for our task. This was confirmed empirically as our model converged in 4 epochs. When combined with shuffling adagrad only took 3 epochs to converge. This shows the tendency of the skip gram model to converge faster with input shuffling and the big impact of having different learning rate for each feature. Here it's interesting to notice that a higher learning rate combined with input shuffling did not yield better results than without shuffling. Both of our best results happened with a learning rate of 0.1, as can be seen in Figure 4.4.

Figure 4.4: Training time Adagrad with input Shuffling

4.6.4 Adadelta

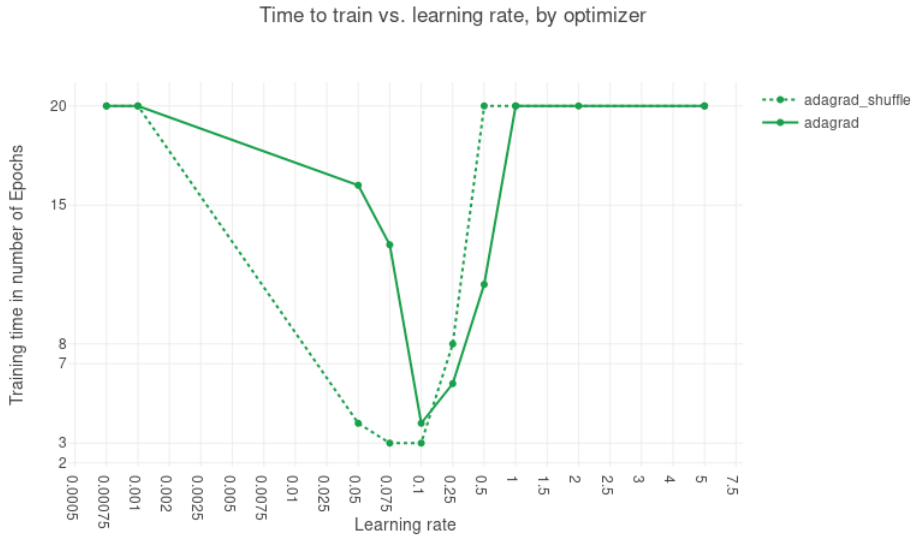In theory Adadelta [Zei12] should outperform Adagrad as it's an extension of the former. Because it didn't have any learning rate to tune, we only did 2 experiments, with and without input shuffling. We are aware of the fact that there are additional hyperparameters by did, for simplicity reason, and because their effect is not as high as the learning rate in other optimizers, decide no to tune it. We left it to it's default value $\rho = 0.9$. This parameter defines the percentage taken when calculating the exponentially decaying average of past gradients, as explained in 2.4.5Adadelta did not manage to achieve a word similarity of 0.66. It only converged to a similarity of 0.59. It did this in 20 epochs without input shuffling and in 3 with input shuffling, as can be seen in Table 4.5

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

Table 4.5: Convergence Time and Quality with Adadelta

4.6.5 Adam

Adam is the most advanced of all the optimizers used in our experiments and did yield the best results as seen in Figure 4.5. Adam converged in 3 epochs without shuffling and 2 with. This is the best result that we got with any optimizer. It's also interesting to note that as same as with Adagrad it did not react to input shuffling the same way as SGD did. In fact, it worked in the opposite direction, as we achieved our best result with input shuffling while having a lower learning rate 0.001 then we used to achieve the best result without input shuffling 0.05.

Time to train vs. learning rate, by optimizer

Figure 4.5: Training time Adam with input Shuffling

5 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 differences. It is followed by a section describing the limitations and possible extensions of this work. This chapter is concluded by the problems we encountered during the implementation so others can avoid them in the future.

5.1 Our work

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

5.1.1 Shuffling and learning rate with SGD

When the model used SGD as an optimizer it was able to use higher learning rates when input shuffling was used, as shown in figures 4.1, 4.2 and 4.3 Therefore arises the questions why these phenomena happen, especially as it did not happen with advanced optimizers. One possibility is the use of the batched version of the SGNS. In consequence, when the input is not shuffled the same word will often appear in one batch, hence the optimizer is presented with an average value of the gradient, which can be imprecise. This is counter-attacked by advanced optimizers as they have adaptive learning rates, i.e a frequent appearing feature will have a low learning rate.

5.1.2 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 4.1 and 4.3, plain SGD. We do not only attribute those resultst to input shuffling but partially also to a good random initialization guess. Due to a lack of time these results where not replicate more than once.

5.2 Comparison to Gensim

This section will compare our finding extensively to Gensim [ŘS10]. As explained in Section **??**, 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.

5.2.1 Configuration of Gensim

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

- `sentences` (iterable of iterables) – *Dataset* – text8 document splitted into sentences of 20 words

- `size`(int) – *Dimensionality of the word vectors* – 100

- `window` (int) –*Maximum distance between the current and predicted word within a sentence* – 5

- `min_count` (int) – *Ignores all words with total frequency lower than this* – 5

- `workers` (int) – *Number of threads used to train the model* – 4

- `sg` (0, 1) – *Training algorithm: 1 for skip–gram*–1

- `negative` (int) –*Number of negative samples*– 10

- **ns_exponent** (float) – *Exponent in the unigram distribution, when choosing random samples, as shown in Equation 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 Equation 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

- **callbacks** (iterable of CallbackAny2Vec) – *Set of functions executed in order to follow the loss and the progress of the model in word similarity – see Appendix*

5.2.2 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[1], which is 23x faster than plain Numpy. Since we used PyTorch the difference is not that big, but still remains. As shown in Figure 5.1 the convergence time was not the same between our implementation and Gensim. There are different possible reasons why this could be the case:

First, our batched approach could hinder performance in terms of convergence since our loss function is not exactly the same. When a word appears more than once in our batch, the gradient will be an average over the gradients of each pair alone, as it is done by Gensim.

Secondly, a difference to our implementation is the fact that Gensim checks whether negative samples are not equal to the context word. If that is the case Gensim selects a new random sample. Therefore, the learning of the input and output context is optimized.

Finally, another possibility is the decay of the learning rate used by Gensim. In fact, decaying the learning rate has been proven in a lot of work to decrease the convergence time. Gensim linearly decreases the learning rate, as we did not use this technique, the decay of the learning rate could help explain the noted differences.

---

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

The first hypothesis, the fact that we used a batched approach, may be confirmed by the fact that when combined with input shuffling SGD does perform closer to Gensim, going from 11 to 7 epochs to converge, as input shuffling reduces the number of co-occurrence of the same word in a batch. Now the question arises if the 3 epochs, that Gensim is better, can be explained by the selection of better negative samples and the learning rate decay?

5.2.3 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 5.1.
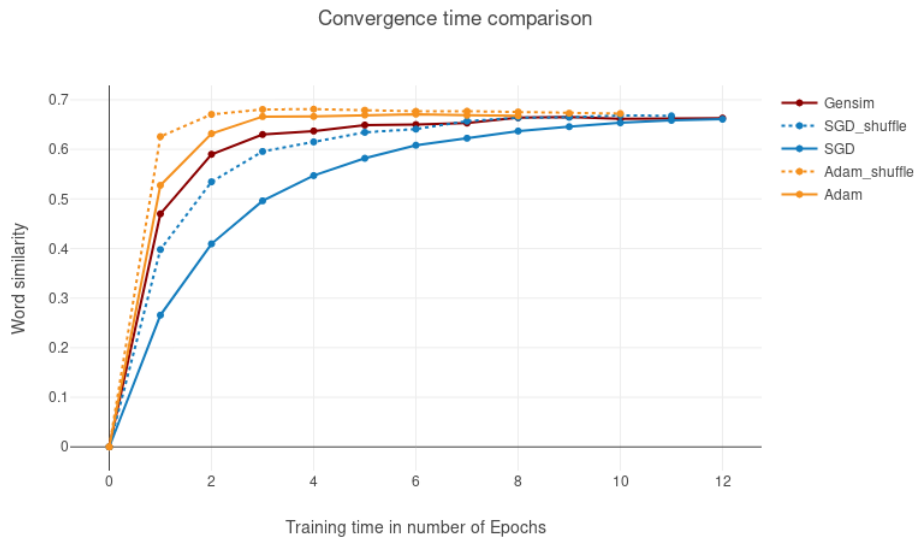

Figure 5.1: Convergence time of SGD and Adam compared to Gensim

5.3 Challenges faced

During our implementation we did encounter some problems, this section has the purpose of preventing the scientific community from making the same mistakes.

5.3.1 Using the wrong embeddings

To start with, we did not choose the same initialization value as Gensim in our word embeddings. We initialized the projection layer with a normal distribution between (-1,1), as opposed to Gensim which initializes all the weights to 0. The reason for this, is that at the begining of our experiments we did not achieve good results with the same initialization as Gensim. Retrospectively we accord this to a learning rate that was too low. After a few simulations, we saw that we did not perform as good as Gensim. And as we changed the initialization of the projection layer back to 0, and already had adjusted the learning rate, we achieved good results. Therefore, this is a recommendation to future work to not set the initialization to (-1,1).

5.3.2 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 made a few changes to remedy the problem.

First, we needed to find a good batch size. During the above-explained raise of the learning rate, 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. [Mik+13], we needed to find one that suited our goals the best. At the beginning of our experiment, we took the average of all the scores stored in our final matrix, as explained in 3.2. 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, while at the same time having a usual learning rate.

Know the question arises why this happened? Is this because of the shape of the loss function better suits our advanced optimizers? Or because the loss function is closer to its original form? Our hypothesis looks the following way: When taking the average, our loss function is represented as follows:

$$\frac{-1}{b} * \sum_{x \in X} loss(x) \tag{5.1}$$

where X is our batch and $b = |X|$. When taking the partial derivative of each parameter, the constant $\frac{-1}{b}$ will remain the same. Therefore it's the same as taking the sum as the loss function and multiplying each gradient with $\frac{-1}{b}$. As our batch size was really high, i.e 2000 and 5000, this could highly influence the learning rate. This is also the conclusion of Goyal et al. [Goy+17], that worked with very high batch sizes, i.e 8000: "When the minibatch size is multiplied by k, multiply the learning rate by k". This makes a little argument for our explanation.

5.4 Future Work

This work shows that the convergence time of the SGNS could be improved by using input shuffling and advanced optimizers. As with every work, there still exists possible extensions. First and foremost an aspect of our implementation that can be prejudicial is that we only extensively tested our model with one small dataset. The fact that we only used one dataset as well that it's a small one is problematic. By using a very small dataset we do not use the model in the condition it is most needed for, as the dataset used in practice usually consists of more than 1 billion words. There is a small argument that can be made for machine translation as the use of small parallel corpus is not unusual in this field. But the main issue with using only one data set it that it has been shown that some optimizers perform better with specific shapes of loss functions. To make a compelling argument it's, therefore, necessary to show that our model with the use of Input shuffling and Adam as its optimizer also outperform Gensim with other data sets, so that the claim can hold consistently, it needs to be confirmed with other datasets as well. Furthermore, our implementation did not outperform Gensim in run time, as this was not the goal of our work. Therefore, one could improve an already existing, optimized version, with input shuffling and advanced optimizers and should achieve a better run time than Gensim.

6 Conclusion

This work provides an overview of the Skip Gram Model with negative Sampling (SGNS) and the numerous successful attemps of optimizing the throughput of the model. As this is the case, no effort went into optimizng the convergence time of the SGNS, therefore this work focused on this point. We decided to use advanced optimizers and input shuffling as optimizing techniques. After giving a short overview over Gradient Descent algorithms this work proposes a slighlty altered version of the SGNS. Where the idea is to compute the loss over the sum of a lot of training samples instead of computing it for each invidiually. This allowed us a faster runtime. We did this as it allowed us to compute more models and analyze the convergence time faster. We used the text8 dataset for most of our models. And used the word similarity as a quality measure for the word embeddings (WE). We used the State of the art implementation Gensim to compare ourself. We did achieve a better convergence time then gensim with Adam as an optimizer and the use of input shuffling. Gensim convereged in 4 epochs to a word similairty of 0.66 and our model only took 2 epochs to achieve the same quality. We did confirm these findings with one test run on the enwik9 dataset. As gensim took 5 epochs to converge and our model only 3. Those results still need to be confirmed with more datasets. Finally if this work would combined with an optimized throughput could improve the state of the art runtime of the SGNS.

A Code

A.1 Gensim

The code used to retrieve information from Gensim computation.

```python
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, self.cum_loss, score))
```

```python
        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 .
            BaseWordEmbeddingsModel '
             Current model.
        """
        self . cum_loss += model . get_latest_training_loss ()
```

B Math

C Parameters

D Dataset

These are the parameter that we did not alter during the training of Genism:

- `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

Bibliography

[BY16]   Seulki Bae and Youngmin Yi. "Acceleration of Word2vec Using GPUs". In: *Neural Information Processing*. Ed. by Akira Hirose, Seiichi Ozawa, Kenji Doya, Kazushi Ikeda, Minho Lee, and Derong Liu. Cham: Springer International Publishing, 2016, pp. 269–279. ISBN: 978-3-319-46672-9 (cit. on pp. 1, 10).

[Ben+03]   Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. "A neural probabilistic language model". In: *Journal of machine learning research* 3.Feb (2003), pp. 1137–1155 (cit. on p. 1).

[DHS11]   John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of Machine Learning Research* 12.Jul (2011), pp. 2121–2159 (cit. on pp. 15, 34).

[Goy+17]   Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour". In: *CoRR* abs/1706.02677 (2017). arXiv: 1706.02677. URL: http://arxiv.org/abs/1706.02677 (cit. on p. 42).

[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. 1, 9, 11, 31).

Bibliography

[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. 11, 12).

[Liu+15]   Yang Liu, Zhiyuan Liu, Tat-Seng Chua, and Maosong Sun. "Topical Word Embeddings." In: *AAAI*. 2015, pp. 2418–2424 (cit. on p. 11).

[Mik+13]   Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013) (cit. on pp. 1, 19, 27, 41).

[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. 4).

[Nes83]   Yurii E Nesterov. "A method for solving the convex programming problem with convergence rate O (1/kˆ 2)". In: *Dokl. akad. nauk Sssr*. Vol. 269. 1983, pp. 543–547 (cit. on p. 32).

[PSM14]   Jeffrey Pennington, Richard Socher, and Christopher Manning. "Glove: Global vectors for word representation". In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543 (cit. on p. 15).

[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 pp. 1, 8).

[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. 10, 11, 28, 31, 32, 38).

Bibliography

[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. 11).

[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. 4–8).

[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).

[YS18]   Zi Yin and Yuanyuan Shen. "On the Dimensionality of Word Embedding". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Curran Associates, Inc., 2018, pp. 887–898. URL: http://papers.nips.cc/paper/7368-on-the-dimensionality-of-word-embedding.pdf (cit. on p. 30).

[Zei12]   Matthew D. Zeiler. "ADADELTA: An Adaptive Learning Rate Method". In: *CoRR* abs/1212.5701 (2012). arXiv: 1212.5701. URL: http://arxiv.org/abs/1212.5701 (cit. on pp. 16, 34).