Lehrstuhl für Data Science

Optimizations of the Skip-Gram Model with negative Sampling

Bachelorarbeit von

Cédric Milinaire

Prüfer
Prof. Dr. Michael Granitzer

April 8, 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.

2 Background

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 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 quite easily 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 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. There lies a problem in this approach. 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. But 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 feasable to compute the softmax. Another possibility is the use of negative sampling which is used in the original word2vec implementation [Tom+13], which I 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.

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 4.1, 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.

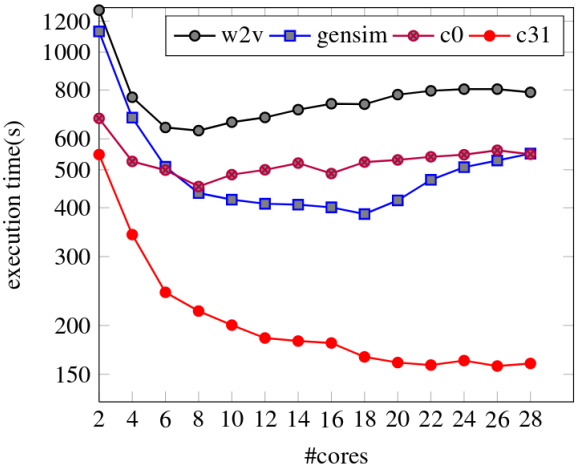

Figure 2.1: Comparasion of the execution time in relation with the number of used cores [VEV16]

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 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 substract a portion of this term from our parameter. The magnitude of the partion is often reffered 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 adress 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 osciliates 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 not ownly 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.

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:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\tilde{v_t} + \epsilon} \tilde{m_t} \tag{2.12}$$

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.

Forwarding

The forwarding method will take two vectors, and a matrix as an input. The first vector represents all the center words in a batch, the second one the context words. The two vectors need to have the same length. If we have two vectors v and w as an input, then our training batch X can be seen in the following way: $X = \{(v_i, w_i) \| \imath \in \{1, ..., n\}, n = |v|\}$. The matrix must be of dimension $|v|xn$, with k, being the number of negative samples per pair. First we will concatenate w and our Matrix A. To compute the dot product of each pair we only have to compute vA. When we sum the resulting matrix of and multiply it with -1 we get the wanted loss.

3.2.2 Creating the context pairs

This class should provide a way for the dataloader to access each word context pair. Here the challenge is that one cannot store all the pairs in a list as this would not be feasble for a large dataset. We therefore propose a way to compute th i-th word-context pair of the dataset. we therofore had to implement the len method. Here we knew that every sentence expcept the last would have a length of 20 words. therefore it was easy to compute the number of pairs in those sentences. We will distinguish two types of wordes in the sentences: first center words, those have the maxiumum amount of possible context words. The border pairs are the words that do not have this proprety because they are two close to the start or end of the sentence. The amount of context pairs per center pair is easy to compute, it's $2 * ctx$
$_w indow$. Therefore we get that for sentences with length LEN

$_{S}EN$, with $LEN\_SEN > ctx$
$_{w}indow * 2$:

```
number\_center\_pairs = ((LEN\_SEN - \\ ctx\_window*2)*self.ctx\_window*2)
```

We then need to compute the amount of borde words. For this we will only need to compute the following:

$$\sum\_i = 0^{ctx_{w}indow-1} ctx_{w}indow + i \tag{3.1}$$

Then we had to add the number of pairs from the last sentence and are done. First we have to check if the sentence is longer then $2ctx\_window$. If it's the case we can apply the samDe principle as shown above. If not we need to do the following:
Now we need to access each pair individually, so first we have to define an order. We will go over the Dataset sequentially adding first the pairs from the left then from the right. Now that we have defined an order, let's look at how we can acces theim. Given an item index x we need to find in which sentence the pairs is. Because we know how many pairs per sentence their is we can. We only have to devide x by the number of pairs that can be built within one sentence. Know we can access this sentence because we have the list that stores our dataset. then we have to iterate over all the possible pairs that we can builld within this sentence count theim, and stop when we arrived at the wished id. S

4 Results

Describe the experimental setup, the used datasets/parameters and the experimental results achieved

4.1 Dataset

In this implementation we used the following datasets: text8[1] and ewik9[2], these are both repsectively the first 30 and 100MB of clean text from Wikipedia. We chosed the text8 dataset becaus it was already used in related work and is very small, giving us a fast computation time. We needed a second more larger dataset to confrim our results, and a dataset with specific vocabulary for us to be in roder to compute the analogy task (more on this in the results). The first document consists of 1702 documents of 10k word. No punctuation. Therefore we needed to split the data set into sentences. We arbitrary chose a length of 20. Furthermore we applied subsampling (described in the next subsection) too both of these datasets. For more information look at table stats.

4.1.1 Subsampling

Subsampling is a technique introduced by Mikolov et al. in [**mikolov**]. 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 them from the dataset. This will decrease the computation time and should in theory increase the accuracy of the model. Another reason how subsampling can increase the accuracy, is that words that would not have appeared in the context of each other,

[1] matt mahoney
[2] matt

Dataset	Voc size	Numb of sentences (20)	Number of Sentences(length 20) with sampling
text8	250k	4000	2000
enwik9	750k	100000	5000

Table 4.1: Specifications of the dataset

may know do, because words between have been deleted. Subsampling will be the first step of our preprocessing. A word w will be eliminated with the following probability:

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

where $f(w)$ is the frecquency of w, and t is a treshhold set empirecally. (Mikolov et al.) recommends a value between 0 and $1e-5$. We experimented with different values and $1e-4$, seemed the most suted. See figure 3 for examples and table stats for inforamtion about sampling. We did this analysis by looking at and random set of sentences and analyzed the best context pairs.

Min count

We also applied a treshhold in the lower bound, we deleted every word that did not appear more than 5 times in our dataset. We got this idea from gensim [ŘS10]. This is a good idea becaus of two reasons: first a few words of our data sets had no meaning (aldk, ahdfo) VERIFY. Secondly a word that only appears one time in our dataset will be very dependent on it's original initialization.Therefore we applied this technique. It should in theory, such as subsampling, improve both the accuracy and computation.

4.1.2 Specification of datasets

- Text 8 original:
  17 mio words
  1702 docs of 10k words/ no sentences
- Applied subsampling: 8mio words. 1702 w of 5k words
- Transformed to 250k sen of 20 words

4.2 Word similarity

Evaluating word embedding is not an easy task. We cannot split our data set into train and test set. Therefore we need to verify that our embedings are of quality with other techniques. We therefore need to define a measure that will give us the similarity of 2 word 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 deg aparts from each other and 2 if they are pointing exactly in the opposited direction. It's 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

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

From a geographic percpective, this measeures the angle between two vectors. Hence the distance will be 0 if v and w if they are pointing in exactly the same direcction. Magnitude does not play any role. Hence for our tasks, two vectors will be considered equal if they are of different amgnitude but point in the same direction. If we normalize the vectors then their the cosine angle becomes the dot product which allows us to compute the distance much faster using gpu's.

4.2.1 wordsim353

To rate our word embedings we will need a dataset that gives us the similarity of words. This is exactly what wordsim353 is. It's 353 pairs of words rated by humans on their similarity. The score goes from 10 and 1, 10 being the most similar. We will rank our embedings on the pearson corelation coefficient between the cosine distance and the score attributed by humans.

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

Figure 4.1: 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*
- `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

4.6.1 SGD

The first think we had two manage was finding a correct learning rate for our model. We tried the learning rate from gensim first then expirmanted with different learning rates. As expected a bell curve shape resulted. the optimal setting that we found is 0.0075. We converged in 11 epochs. We then added input shuffling to the equation. We see that for all the learning rates this helped to increase the convergenge time. Our models

know converges in 7 epochs. An interesting point to notice is that we achieve this with a higher learning rate.

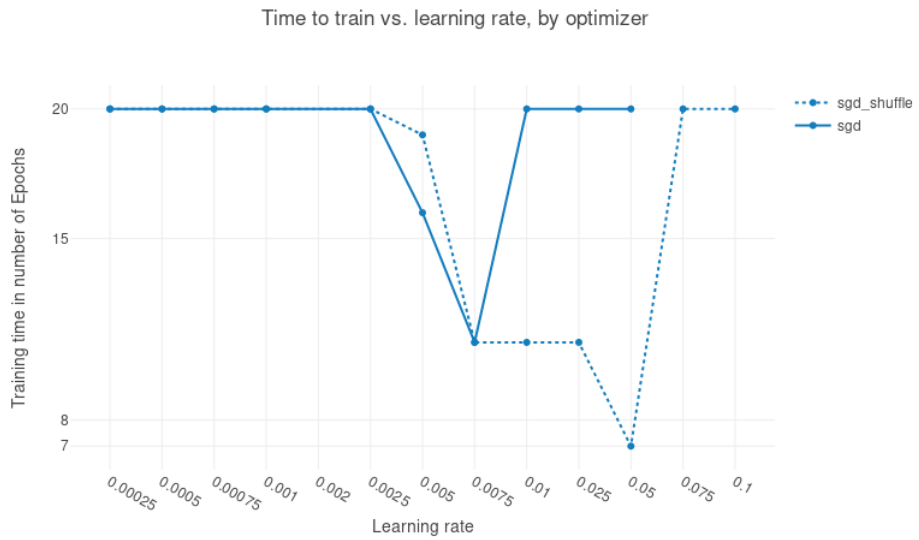

Figure 4.2: Training time Stochastic Gradient Descent with input Shuffling

4.6.2 Momentum and Nesterov

Momentum and nesterov accelerated gradient both have an additional hyper parameter γ, that, as described in secction background, defines the importance of the previous gradient taking in the equation. We set $\gamma = 0.9|$ this is the typical value. Let's talk results: Momentum and NAG only slighltly respectively decrease or increasethe convergence time. NAG convergences in 12 epochs while momentum took 8 epochs.

4.6.3 Adagrad

Adagrad is a very interesting tool for learning word embedding as they decrease the loearning 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. this is also confirmed by the results adagrad converges in 4 epochs. When combined with shuffling adagrad only

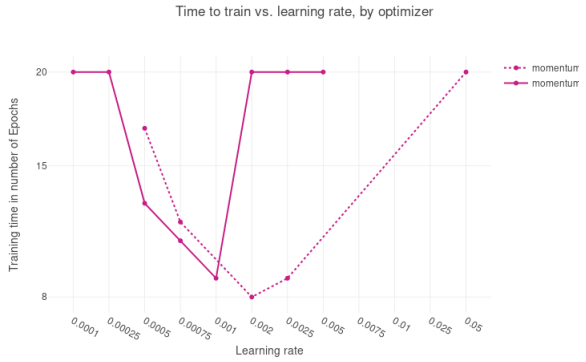

Figure 4.3: Training time Momentum with input Shuffling

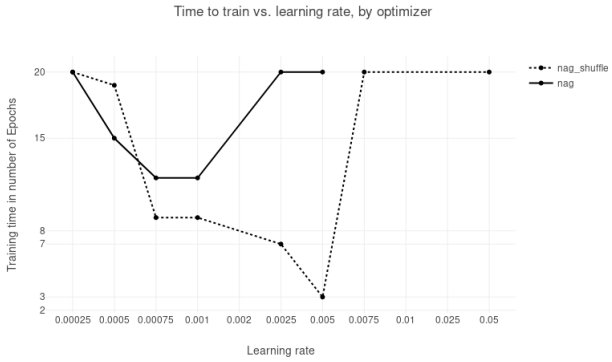

Figure 4.4: Training time Nesterov with input Shuffling

takes 3 epochs to converge. This shows the tendency of the skip gram model to converge faster with input shuffling.

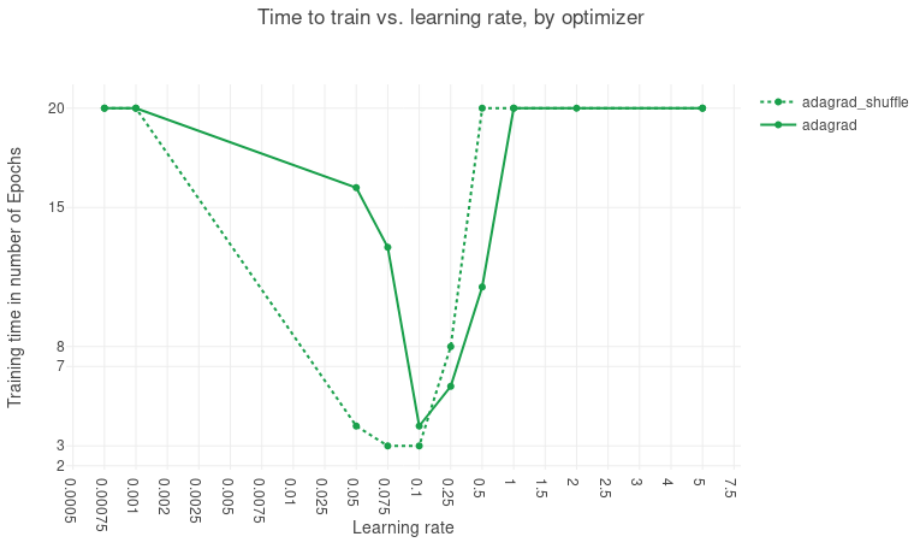

Figure 4.5: Training time Adagrad with input Shuffling

4.6.4 Adadelta

Adadelta was the Advanced optimizer that showed the most disapointing results. Because it didn't has any learning rate to tune, we only did 2 experiments, with and

without input shuffling. the results where very much disapointing. Adadelta converged in 10 epochs with shuffling and 7 epochs without. Therefore barely beating sgd. This is interesting as Adadelta is an expansion of Adagrad, and is supposed, in theory to work better then the latter.

4.6.5 Adam

Adam is the most advanced of all the optimizers used in our experiemnts. Did it therefore yield the best results? Indeed this was the case, as seen in figure 2, Adam converged in 3 epochs without shufffling and 2 with. This are the best result that we got.

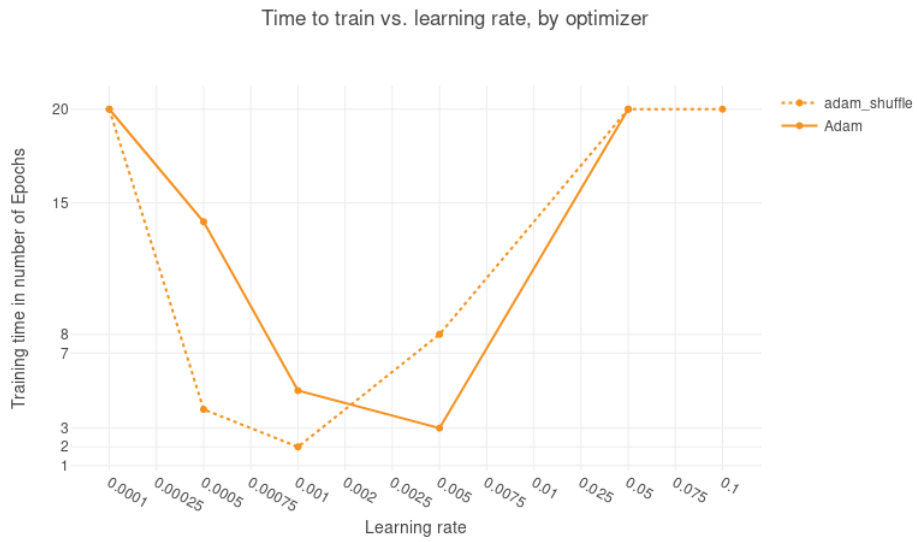

Figure 4.6: Training time Adam with input Shuffling

5 Discussion

In this section we will 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 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**]. We will then compare our self to Gensim, we do this because Gensim allowed us to easily and fastly compute the embeddings, while having access to the loss, and being a python implementaion made everything easier.

5.2 word2vec

word2vec only trained their model on the very large google news datset incorporating more than 3 billio words. Therefore we have a problem to compare our work. But we will make some assumptions. the original paper paer of word2vec they show results from one or 3 epochs. We accord these good results to the very large dataset. And as a matter of fact their result are better with 3 than with 1 epochs. We do not have any informtaion about the convergence. But we can talk about the quality of the word embeddings as Ji et al. reported wordsim ressults on text8 with the original c code, namely 0 .63. This is worse than our results with sgd and adam. we did not find anything that would explain those results as shuffling and wihtout shuffling VERIFY beat the original c code. It is also possible that if we would apply an advanced optimizer to the large dataset we would get a better word embedding as we achieved a wordsim score of 0.67 with adam and gensim only 0.66.

5.3 Gensim

Gensim does propose the class w2vec with the following possible parameters[1], we will describe theim, show our setting, and why we chose the following, we will only describe the prametrs we changed from the default value will be explained, the rest can be found in the appendix. They will be presented in the following way:
`name` (type) - *Description* - Value

- `sentences` (iterable of iterables) – Dataset - we used the Original text8 document splitted into sentences of length 20
- `size` (int) – Dimensionality of the word vectors - 100
- `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
- `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
- `negative` (int) – If ¿ 0, negative sampling will be used, the int for negative specifies how many "noise words" should be drawn (usually between 5-20). If set to 0, no negative sampling is used - 10
- `ns_exponent` (float) – Exponent in the unigram distribution, when choosing random samples REFERENCE EQUATION. - 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) – The threshold for configuring which higher-frequency words are randomly downsampled, useful range is (0, 1e-5). - 1e-4

[1] link to gensim params

- `hashfxn` (function) – Hash function to use to randomly initialize weights, for increased training reproducibility. - VERIFY
- `iter` (int) – Number of iterations (epochs) over the corpus. - 10
- `compute_loss` (bool) – If True, computes and stores loss value which can be retrieved using get_latest_training_loss(). - True
- `callbacks` (iterable of CallbackAny2Vec) – Sequence of callbacks to be executed at specific stages during training. - see Appendix

5.3.1 Gensim vs. SGD

First we have to note that we cannot compare ourselves to gensim in computation time, this woudl not be very smart as our code is written in python and gensim uses advanced cython routines[2]. The first interesting thing to note is that we did not have the same values in convergence time as gensim when using sgd. There are differnet possiblilities why this could be the case. First our batched approach could hinder performance in term of convergence as some overwriting may happen. Another differnce between our implementation is the fact that gensim checks whether negative samples are real negative samples. Thererfore the learning of the input and output context is optimized. The first hypothesis may be confirmed by the fact that our input shuffling reduces the number of overwritings and therefore we achieved conv time of 7 epcohs which is closer to 4 epochs. and the last difference maybe explained by the eg samples thing.

5.3.2 Gensim vs. Adam

The Adam optimizer did outperform the Gensim application in performance (only slighltlly: 0.01 corr coeff advantage) and convergence time. Adam converged in 2 epochs vs. Gensim in 4. The case to be made here is that one would have to look at the computational advantage to calculate the simple sgd vs. the adam update rule. Another aspect to look at is that we mainly tested our implementation on the text8 dataset. We only experienced once one the enwik9 dataset. Here the argument that can be made is that, some optimizers work better on specific loss functions in comparison to others,

[2]link to tutorial thesis

hence we need to show that our optimizer words on amultiued on theim and that it is not just a statistical anomaly.

5.4 Challenges faced

5.4.1 Using the wrong embeddings

To start our comutation we did choose some wrong, at leat not in coordinance with gensim, initialization embeddings. The context vectors were set between (-1,1) distributed normally. Gensim does it buy putting theim all to 0. As we did the latter the first time our model did not train, but this is possably due to a learning rate that was too low. So we chosed theim to (-1,1). With theim we did not acheive the same similarity as gensim. But when we changed theim back we performed the same results as gensim. Therefore this is a recommendation to future work to not set the initialization to (-1,1)

5.4.2 Too big of a batch size

Another challenged we faced was finding a good batch size. We trained our batch size for a long time with a bs of 5000. But this seemed to worked fine in the beginning but a high lr seemed to work too which was very suspect. We then decided to take a batch size of 2000. Which slightly decreased the learning time, and increased the convergence time and adjusted the learning rate.

5.4.3 Correct loss function

We expirimented with different loss function as we did not have exactly the same as introduced by mikolov et al. We therefore first ook the average of each score in our tensor. But we then took the sum which seemed to work better. Is this because the shape of the loss function suits bettter advanced optimizers. Or because it approaches the original loss function more.

5.5 Further Work

6 Conclusion

Summarize the thesis and provide a outlook on future work.

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. 4).

[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. 6, 16, 20).

[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, 9).

[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. 4).

[MB05] Frederic Morin and Yoshua Bengio. "Hierarchical probabilistic neural network language model." In: *Aistats*. Vol. 5. Citeseer. 2005, pp. 246–252 (cit. on p. 3).

[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. 6).

[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 p. 17).

[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. 3–6).

[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 p. 7).