



Lehrstuhl für Data Science

Optimizations of the Skip-Gram Model with negative Sampling

Bachelorarbeit von

Cédric Milinaire

PRÜFER

Prof. Dr. Michael Granitzer

April 4, 2019

Contents

1	Introduction	1
2	Background	2
2.1	The Skip-Gram Model	2
2.2	Negative Sampling	3
2.3	Optimization of the Skip Gram Model	5
2.3.1	Parallelization	5
2.3.2	Context sensitive word embedding	7
2.4	Optimizers in Machine learning	8
2.4.1	Stochastic Gradient Descent	8
2.4.2	Momentum	9
2.4.3	Nesterov	9
2.4.4	Adagrad	10
2.4.5	Adadelata	11
2.4.6	Adam	11
3	Methods	12
3.1	PyTorch	12
3.2	Implementation	13
3.2.1	class SkipGramModel	13
3.2.2	class Dataset	13
3.2.3	class W2Vec	15
4	Results	16
4.1	Dataset	16
4.1.1	Subsampling	16
4.1.2	Specification of datasets	17

Contents

4.2	Word similarity	17
4.2.1	wordsim353	18
4.3	Configuration of the network	18
4.4	Input Shuffling	19
4.5	Convergence time	19
4.6	Results by optimizer	20
4.6.1	SGD	20
4.6.2	Momentum and Nesterov	20
4.6.3	Adagrad	20
4.6.4	Adadelata	21
4.6.5	Adam	21
5	Discussion	22
5.1	Related Work	22
5.2	word2vec	22
5.3	Gensim	22
5.4	Challenges faced	25
5.5	Further Work	25
6	Conclusion	26
	Bibliography	27

Abstract

The Skip-gram Model with negative sampling (SGNS) is an effective algorithm to create word embeddings. SGNS uses Stochastic Gradient Descent (SGD) as its 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, Adadelata 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

2.1	Comparasion of the execution time in relation with the number of used cores [VEV16]	6
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]	8

List of Tables

4.1	Specifications of the dataset	17
-----	---	----

1 Introduction

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

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:

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

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:

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

where the parameters are the same as in 2.1. The basic Skip-Gram Model uses a classical Softmax to calculate the conditional probability $p(w_{t+j}|w_t)$:

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

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 feasible 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 :

2 Background

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

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). To put things in perspective let's 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:

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

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 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 been 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.

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

2 Background

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 execution time regressed after 8 cores, this seems to indicate that too much overwriting was happening, as the number of concurrent threads surpasses a certain threshold. This can be seen in 2.1, where c31 is the model proposed by Vuurens et al.[VEV16]. The model did not suffer any accuracy loss in comparison to the original Word2vec model.

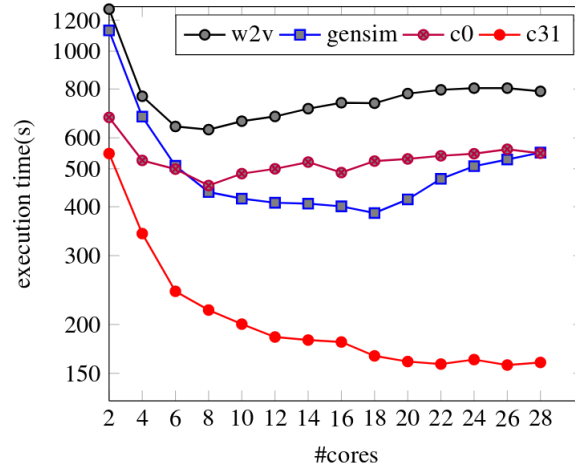


Figure 2.1: Comparison of the execution time in relation with the number of used cores [VEV16]

2.3.2 Context sensitive word embedding

A word does not always have the same meaning according to its 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 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 main idea is to change the way we compute the 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:

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

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 λ 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:

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

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 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 2.2.

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

This section will give a quick overview about a few optimizers existing in machine learning. First of all let's define a few notation: η will be the learning rate, θ all the parameters of our model, $J(\theta)$ our objective function. The goal of an optimizer is to find values for θ such that $J(\theta)$ can either be minimized or maximized depending on the task. Stochastic gradient descent is the most common optimization technique used in machine learning. We will address it in it's original form and the existing extensions.

2.4.1 Stochastic Gradient Descent

In this work we will address stochastic gradient as mini-batch gradient descent. At each optimisation time step t , Stochastic gradient descent will compute the gradient of all our parameters. Denoted as $\nabla J(\theta_{t-1})$. We will then subtract this term, multiplied by our learning rate, from our weights θ . The gradients will give the direction of the optimization step, whereas the learning rate will give the amplitude of that step.

$$\theta_t = \theta_{t-1} - \eta \nabla J(\theta_{t-1}) \quad (2.7)$$

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

- The learning rate needs to be tuned, it's is often necessary to try out a set of learning rate to find the optimal one for the given network.
- Learning rate schedules, that are very often needed, as one gets closer to the optimal a smaller learning rate is often needed, must be set before the training starts and hence cannot adapt to the learning set

- SGD does not take the parameters into account, every parameter will be following the same update rule

Therefore there exists numerous advanced optimizers that take those issues into consideration. Let's look into them.

2.4.2 Momentum

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

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

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

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

2 Background

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:

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

SGD with 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 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, another word embedding system.

Each parameter θ_i , at time step t will have it's own learning rate $\eta_{t,i}$

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

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 achieves 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:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,i,i}} + \epsilon} g_{t,i}, \quad (2.11)$$

with $G_{t,i,i}$ being the diagonal Matrix of the sum of the squares of the gradients ($g_{t,i}$). 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 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.

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 consideration 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 v_t are initialized as vectors of zero. Therefore they are biased towards zero. therefore 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

We developed a full replication of the original w2vec implementation of the Skip Gram Model in Pytorch. And then tried to optimize it. This chapter will illustrate our proceeding. First it will give a short introduction to PyTorch, and then cover the most interesting parts of our implementation.

3.1 PyTorch

For our implementation we choosed the open source library PyTorch. It offers a few implementation features that enormlly facilitates the implemenatation process. First of all it computes the gradient of the loss function online, therofore there is no need for us to implement the calcluation of htose gradients. For the learning ew will only have to use the already implemented optimizers, that can be found in the module torch.optim. Another feature of Pytorch that we used are the classes dataset and dataloader. Both of these classes works closely together. The dataset 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 datsaet before each epochs.

3.2 Implementation

3.2.1 class SkipGramModel

This class is responsible for the forward pass of our data, and represents the neural network, that will result in the word embeddings. We used the class `torch.embeddings` to generate our word embeddings. This class allows us to give a single integer to the embedding and it will return our embedding.

Forwarding

. We will take as an input a number of word and their context and then use the same negative samples for all of those words. We will create a matrix made out of context words and samples. Create matrix containing all our words. Multiply the two matrices. This will result in a matrix having each score of equation ???. And we will then sum all our scores to return our loss. Let's examine an example: $(w_1, c_1)(w_2, c_2)(w_3, c_3)$ be our batch.

$$\text{Therefore } pos_u = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix}, pos_v = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} \text{ and } neg_v = \begin{bmatrix} k_{1,1} & k_{2,1} & k_{3,1} \\ k_{1,2} & k_{2,2} & k_{3,2} \\ k_{1,3} & k_{2,3} & k_{3,3} \end{bmatrix}$$

We then concatenate pos_v and neg_v , while negating neg_v resulting in:

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

We then multiply pos_u and $samples$ resulting in:

$$scores = \begin{bmatrix} w_1 \cdot c_1 & -w_1 \cdot k_{1,1} & -w_1 \cdot k_{2,1} & -w_1 \cdot k_{3,1} \\ w_2 \cdot c_2 & -w_2 \cdot k_{1,2} & -w_2 \cdot k_{2,2} & -w_2 \cdot k_{3,2} \\ w_3 \cdot c_3 & -w_3 \cdot k_{1,3} & -w_3 \cdot k_{2,3} & -w_3 \cdot k_{3,3} \end{bmatrix} \quad \text{Finally we sum up the score}$$

and multiply it with minus one to make it a minimizing problem:

$$-(\sum_{i=1}^3 w_i \cdot c_i - \sum_{j=0}^3 -w_i \cdot -k_{i,j})$$

3.2.2 class Dataset

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

Algorithm 1 My algorithm

```

1: procedure MYPROCEDURE
2:    $stringlen \leftarrow \text{length of } string$ 
3:    $i \leftarrow patlen$ 
4:   if  $i > stringlen$  then return false
5:    $j \leftarrow patlen$ 
6:   if  $string(i) = path(j)$  then
7:      $j \leftarrow j - 1.$ 
8:      $i \leftarrow i - 1.$ 
9:     goto loop.
10:  close;
11:   $i \leftarrow i + \max(delta_1(string(i)), delta_2(j)).$ 
12:  goto top.

```

feasible for a large dataset. We therefore propose a way to compute the i -th word-context pair of the dataset. we therefore had to implement the len method. Here we knew that every sentence except 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 words in the sentences: first center words, those have the maximum amount of possible context words. The border pairs are the words that do not have this property because they are too close to the start or end of the sentence. The amount of context pairs per center pair is easy to compute, it's $2 * ctx_window$. Therefore we get that for sentences with length LEN_SEN , with $LEN_SEN > ctx_window * 2$:

$$number_center_pairs = ((LEN_SEN - ctx_window * 2) * self.ctx_window * 2)$$

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

$$\sum_{i=0}^{ctx_window-1} ctx_window + i \quad (3.1)$$

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 than $2ctx_window$. If it's the case we can apply the same 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 access them. Given an item index x we need to find in which sentence the pairs is. Because we know how many pairs per sentence there is we can. We only have to divide x by the number of

pairs that can be built within one sentence. Now 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 build within this sentence count them, and stop when we arrived at the wished id. S

3.2.3 class W2Vec

tst

method train_with_loader

training

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: text8¹ and ewik9², these are both respectively the first 30 and 100MB of clean text from Wikipedia. We chose the first dataset because it is very small, hence giving us a fast computation time, but at the same time used in a lot of related work. [Ji+16] [gpu]. We needed the second dataset to compute the analogy task (more on this in results). Both of these datasets contain documents of 10k senteces. Therefore we needed to split the dataset into sentences. We arbitrarily chose a length of 20. Furthermore we applied subsampling (described in the next subsection) too both of these datasets. We also applied a Statistics regarding the dataset can be found in the following subsection. We also deleted every word that did not occur more than 5 times in the dataset.

4.1.1 Subsampling

Subsection 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 theim from the dataset. This will increase the computation time and should in theory increase the accuracy of the model. Another interesting fact of subsampling

¹matt mahoney

²matt

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

is that words that would not have appeared in the context of the word will now be it. Subsampling will be the first step of our preprocig. A word w will be eliminated with the following probability:

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

where $f(w)$ is the frequency of w , and t is a treshhold set empirically. (Mikolov et al.) recommends a value between 0 and $1e - 5$. We experimented with different values and $1e - 4$, seemed the most suted. We did this analysis by looking at and random set of sentences and analyzed the best context pairs.

4.1.2 Specification of datasets

todo add number of subsampled word number of words deleted by min count.

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 embeddings are of quality with other techniques. We therefore define the principle of word similarity. We need to define a measure for our word embeddings, if they are similar. 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 deviding it by the magnitude of v and w divided by each other. (equation x). We get

$$cos_{dis}(v, w) = 1 - \frac{v \cdot w}{|v||w|} \quad (4.2)$$

If we normalize the vectors then their 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 embeddings 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 embeddings on the pearson correlation coefficient between the cosine distance and the score attributed by humans.

4.3 Configuration of the network

The skip-gram model contains a lot of hyper parameters that can be tuned. Let's have a look at them and how they potentially could influence the network

- Negative Samples: the recommended range in [mikolov] is 10-20. The training accuracy should increase as the sample increases, therefore the computation time should go up. There is a tradeoff to find.
- 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. The recommended setting was 2-10.
- Dimension of the embedding: here the choice is less obvious, in their paper mikolov et al chose an embedding size of 300. their dataset was pretty huge, with 1 billion words. We therefore chose a smaller one.
- Batch size: more on the batch size in section intel description. Too big of a batch size will lead to too much overfitting and therefore not getting the best results. If we turn the batch size down too much we will get a higher computation time.
- Alpha: learning rate. We tried out a variety of lr in the range of 0.00001 to 2

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 $X = x_1 \dots x_n$ 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 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 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 update similar words at the same time. We then lose 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:

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

where ρ is the pearson coefficient with the wordsim353 task.

4.6 Results by optimizer

4.6.1 SGD

The first thing we had to manage was finding a correct learning rate for our model. We tried the learning rate from gensim first then experimented 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 convergence time. Our models now converge in 7 epochs. An interesting point to notice is that we achieve this with a higher learning rate.

4.6.2 Momentum and Nesterov

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

4.6.3 Adagrad

Adagrad is a very interesting tool for learning word embeddings as they decrease the learning rate for very frequent occurring 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 takes 3 epochs to converge. This shows the tendency of the skip gram model to converge faster with input shuffling.

4.6.4 Adadelta

Adadelta was the Advanced optimizer that showed the most disappointing results. Because it didn't have any learning rate to tune, we only did 2 experiments, with and without input shuffling. The results were very much disappointing. 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 than the latter.

4.6.5 Adam

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

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

5.3 Gensim

Gensim does propose the class `w2vec` with the following possible parameters, we will describe them, show our setting, and why we chose the following:

- `sentences` (iterable of iterables, optional) – The sentences iterable can be simply a list of lists of tokens, but for larger corpora, consider an iterable that streams the sentences directly from disk/network. See `BrownCorpus`, `Text8Corpus` or `LineSentence` in `word2vec` module for such examples. See also the tutorial on data streaming in Python. If you don't supply sentences, the model is left uninitialized – use if you plan to initialize it in some other way.

- `corpus_file` (str, optional) – Path to a corpus file in LineSentence format. You may use this argument instead of `sentences` to get performance boost. Only one of `sentences` or `corpus_file` arguments need to be passed (or none of them, in that case, the model is left uninitialized). `size` (int, optional) – Dimensionality of the word vectors. `window` (int, optional) – Maximum distance between the current and predicted word within a sentence.
- `min_count` (int, optional) – Ignores all words with total frequency lower than this.
- `workers` (int, optional) – Use these many worker threads to train the model (=faster training with multicore machines).
- `sg` (0, 1, optional) – Training algorithm: 1 for skip-gram; otherwise CBOW.
- `hs` (0, 1, optional) – If 1, hierarchical softmax will be used for model training. If 0, and `negative` is non-zero, negative sampling will be used. `negative` (int, optional) – If $\neq 0$, 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.
- `ns_exponent` (float, optional) – The exponent used to shape the negative sampling distribution. A value of 1.0 samples exactly in proportion to the frequencies, 0.0 samples all words equally, while a negative value samples low-frequency words more than high-frequency words. The popular default value of 0.75 was chosen by the original Word2Vec paper. More recently, in <https://arxiv.org/abs/1804.04212>, Caselles-Dupré, Lesaint, & Royo-Letelier suggest that other values may perform better for recommendation applications.
- `cbow_mean` (0, 1, optional) – If 0, use the sum of the context word vectors. If 1, use the mean, only applies when `cbow` is used.
- `alpha` (float, optional) – The initial learning rate.
- `min_alpha` (float, optional) – Learning rate will linearly drop to `min_alpha` as training progresses. `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).

- `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.
- `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.
- `sample` (float, optional) – The threshold for configuring which higher-frequency words are randomly downsampled, useful range is (0, 1e-5).
- `hashfxn` (function, optional) – Hash function to use to randomly initialize weights, for increased training reproducibility.
- `iter` (int, optional) – Number of iterations (epochs) over the corpus.
- `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 \leq `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.

- `sorted_vocab` (0, 1, optional) – If 1, sort the vocabulary by descending frequency before assigning word indexes. See `sort_vocab()`.
- `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.)

- `compute_loss` (bool, optional) – If True, computes and stores loss value which can be retrieved using `get_latest_training_loss()`.
- `callbacks` (iterable of `CallbackAny2Vec`, optional) – Sequence of callbacks to be executed at specific stages during training.

5.4 Challenges faced

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. 3).
- [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. 5, 16, 19).
- [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. 7, 8).
- [Liu+15] Yang Liu, Zhiyuan Liu, Tat-Seng Chua, and Maosong Sun. “Topical Word Embeddings.” In: *AAAI*. 2015, pp. 2418–2424 (cit. on p. 7).
- [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. 3).
- [MB05] Frederic Morin and Yoshua Bengio. “Hierarchical probabilistic neural network language model.” In: *Aistats*. Vol. 5. Citeseer. 2005, pp. 246–252 (cit. on p. 2).
- [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. 5).

- [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.
- [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. 7).
- [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. 2–5).
- [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. 6).