Lehrstuhl für Data Science

Optimizations of the Skip-Gram Model with negative Sampling

Bachelorarbeit von

Cédric Milinaire

Prüfer
Prof. Dr. Michael Granitzer

March 28, 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 Adadelta 11

2.4.6 Adam 11

2.5 Input Shuffling 11

3 Methods **13**

3.1 PyTorch 13

3.2 Implementation 13

3.2.1 class SkipGramModel 13

3.2.2 class Dataset 14

3.2.3 class W2Vec 14

4 Results **15**

5 Discussion **16**

6 Conclusion **17**

Bibliography **18**

Abstract

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

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 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.

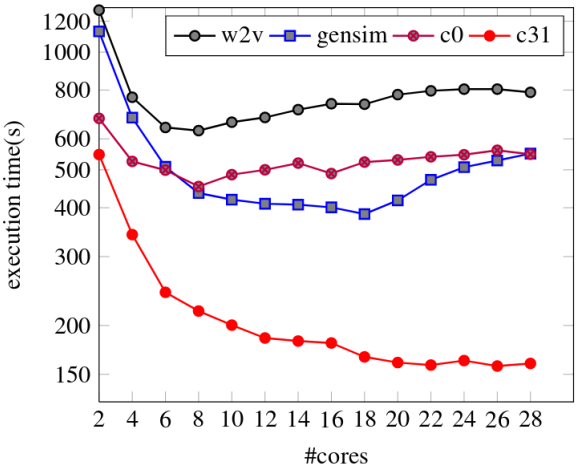

Figure 2.1: Comparasion of the execution time in relation with the number of used cores [VEV16]

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

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-batach gradient descent. At each optimiation time step t, Stochastic gradient descent will compute the gradient of all our parameters. Denoted as $\nabla J(\theta_{t-1})$. We will then substract 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.

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

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

- The learning rate needs to be tuned, it's is often necessacary 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

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

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:

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

2.5 Input Shuffling

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.

3 Methods

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

We developped 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

torch.optim dataloader etc.

3.2 Implementation

3.2.1 class SkipGramModel

This class is responsable 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 embedings. This class allows us to give a single integer to the embeding and it will return our embedding.

Method forward()

In the forward method we apply the idea of shared memory distribution [Ji+16]. 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 contex words and samples. Create matrix containg 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.

3.2.2 class Dataset

data loader, pairs neg samples etc.

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

5 Discussion

Discuss the results. What is the outcome of your experimetns?

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, 13).

[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).