Contents

1 Background

1.1 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:

$$\prod_{t=1}^{T} \prod_{-m<j<m} p(w_{t+j}|w_t) \tag{1.1}$$

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:

$$\sum_{t=1}^{T} \sum_{-m<j<m} log(p(w_{t+j}|w_t)) \tag{1.2}$$

where the parameters are the same as in 1.1. The basic Skip-Gram Model uses a classical softmax to calculate the conditional probability $p(w_{t+j}|w_t)$:

$$p(w_{t+j}|w_t) = \frac{exp(\tilde{v}_{w_{t+j}}^T v_{w_t})}{\sum_{w=1}^{v} exp(\tilde{v}_w^T v_{w_t})} \tag{1.3}$$

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 [**mikolov2**]. The first one is to use a Hierarchical soft max introduced by [**hsoftmax**]. 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 [**mikolov2**], which I shall discuss in the next section.

1.2 Negative Sampling

An alternative to the Hierarchical Softmax is Noise Contrastive Estimation (NCE) which was introduced by Gutmann and Hyvrinen, and first applied to NLP by Mnih and Teh. The idea behind 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. [**mikolov2**] 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 probability 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. [**mikolov2**] used the following unigram distribution:

$$P(w) = \frac{f(w)^{\frac{3}{4}}}{\sum_{w_k \in W} f(w_k)^{\frac{3}{4}}} \tag{1.4}$$

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. [**mikolov2**] the negative sampling method "is an extremely simple training method that learns accurate representations". As a matter of fact Mikolov et al. [**mikolov2**] 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.

1.3 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.

1.3.1 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 [**mikolov2**] used a Hogwild tree proposed by [**hogwild**]. 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 [**hogwild**] 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 [**intel**], 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 [**efficient**]. 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 is happening, as the number of concurrent threads surpasses a certain threshold. This can be seen in 1.1, where c31 is the model proposed by [**efficient**]. The model did not suffer any accuracy loss in comparison to the original Word2vec model.

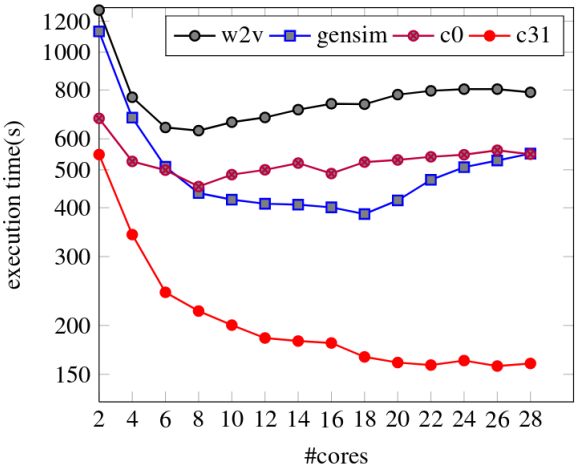

Figure 1.1: Comparasion of the execution time in relation with the number of used cores [**efficient**]

1.3.2 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, [**topicalWE**], [**breaking**] for example, but the the one reporting the best results is [**contextWithTensor**]. 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:

$$J(\Omega) = \sum_{(w,t,c)\in D} \sum_{(w,\tilde{t},\tilde{c}\in\tilde{D})} max(0, 1 - g(w,t,c) + g(w,\tilde{t},\tilde{c}))\lambda||\Omega||_2^2 \tag{1.5}$$

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:

$$g(w, c, t) = u^T \sigma(w^T M^{[1:k]} t + V_c^T (w \oplus t) + b_c) \tag{1.6}$$

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 1.2.

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

Figure 1.2: "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" [**contextWithTensor**]

2 Research Questions

This paper will address the following questions:

1. Can the convergence time of the skip Gram Model be optimized by the use of advanced optimizers, while at the same time maintaining it's accuracy?
2. Can the convergence time of the skip Gram Model be optimized by the use of input shuffling, while at the same time maintaining it's accuracy?

3 Project Plan

This section will cover the main project plan, we will discuss exactly what we wish to implement and how we are going to test our implementation. The whole process will be done the following way:

- `Phase 1: Research`
  In this phase we will become a broad overlook on the subject, research possible libraries that we can use and start planing the following phases.

- `Phase 2: Implementation`
  This phase will be our main work, as we will implement our own word2vec version.

- `Phase 3: Testing`
  Here we will first test if our optimization ideas were succesfull, and if they were we will test the accuracy of our Model.

- `Phase 4: Writing`
  In this phase we will summarize Phase 1-3 in our thesis.

More details on phase 2 and 3:
`Phase2` First we will implement our own version of the skip gram model. We will implement the optimization techniques stated in 2, this means we will use input shuffling and advanced optimizers. There exists a python implementation of the original word2vec mode, that is called Gensim [**gensim**]maybe it's possible to tweak it to fit our needs, if not we are going to implement our own version.
`Phase3` First we compare our model against the original gensim word2vec implementation. For this we wil use the dataset *text8*[1], that was created by Matt Mahoney [2]. We will first compare the convergence time. If we see promising results we will then test the

[1] http://mattmahoney.net/dc/text8.zip
[2] mattmahoney.net

accuracy of our model. This is quite difficult as the quality of word embeddings are often task dependent, but Mikolov et al. [**mikolov2**] presented a word analogy task[3] and [**wSimilarity**] presented a word similarity evaluation [4]. The analogy tasks evaluates semantic and syntactic analogies. The idea is to guess where a specific vector x should be located, and if the closest (using the cosine distance) vector to the guess is x then the model passed the test. For example let $x = vec(\text{"}Berlin\text{"}) - vec(\text{"}Germany\text{"}) + vec(\text{"}France\text{"})$ then the closest vector to x should be $vec(\text{"}France\text{"})$. The above described test would be a semantic test, for a syntactic test one could use the following words: quick, quickly, slow, slowly. The similarity task dataset consists of tuples of words assigned with a similarity value. This value is assigned by human annotator. For example (king, cabbage) have a low similarity but (king, queen) have a high similarity. Therefore the cosine distance of (king,queen) should be small and the one of (king, cabbage) should be high. We will test our model on both of these tasks if we achieve a significant optimization.

[3] http://download.tensorflow.org/data/questions-words.txt
[4] http://www.leviants.com/ira.leviant/WS353_ALL_Langs_SIM_TXT_Format.zip