Natural Language Processing

Skip Gram with Negative Sampling

Chloé Daems, Anne-Claire Laisney and Amir Mahmoudi

1. INTRODUCTION

Word2vec models have had a lot of impact on Natural Language Processing research and its applications. One of these models is the **Skip-gram model**, which uses a technique called **Negative Sampling** to train. Skip-gram is one of the unsupervised learning techniques used to find the most related words for a given word. It is used **to predict the context word for a given target word given a window of words (the input of the network)**. The input is the target word while the output are the context words.

In this paper, we will show our implementation of a Skip-gram with negative-sampling from scratch.

First, we will present the Skip-Gram model's optimization problem and how adding a negative sampling element simplifies it. Secondly, we will explain the different steps of our pre-processing. Then, we will detail our algorithm and training method using Stochastic Gradient Descent (SGD). Finally, we will show our best results after the fine-tuning of both the skip-gram and the training hyper-parameters.

2. SKIP-GRAM MODEL

Given an input, the Skip-gram model predicts the neighbors or context words. In the skip-gram model, let C be a corpus of words w and their contexts c. The objective of this model is to maximize the average log-probability of the context words occurring around the input word over the entire vocabulary. One approach for parameterizing the skip-gram model uses soft-max:

$$p(c|w;\theta) = \frac{exp(v_c.v_w)}{\sum_{(c' \in C)} e^{-v_c.v_w}}$$

where v_c and $v_w in R^d$ are vector representations for context word c and main word w respectively[1].

The objective is to solve the optimization problem:

$$(P) : argmax_\theta = \sum_{(w,c \in D)} log \frac{1}{1 + e^{-v_c.v_w}}$$

.

However, the main problem comes up when we want to calculate the denominator. It is a normalizing factor and has to be computed over all the vocabulary. Therefore, softmax is very costly when applied to a huge output layer. Indeed, it increases algorithm complexity exponentially.

To solve this problem, we will introduce a **Negative sampling** element to the approach (P) by adding randomly chosen negative pairs of word.

3. NEGATIVE SAMPLING

3.1. Explanation

Mikolov et al. present the negative-sampling approach as a more efficient way of deriving word embeddings. Negative sampling distribution is used to choose negative samples. It **allows us to only modify a small percentage of the weights**, rather than all of them for each training sample. Instead of trying to predict the probability of being a nearby word for all the words in the vocabulary, we try to predict the probability that our training sample words are neighbors or not. For example in Mikolov's papers the negative sampling expectation is formulated as:

$$log(\sigma(w,c)) + k.\mathbb{E}[log\sigma(-(w,c_n))]$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the *sigmoid* function.

3.2. Derivation of Cost Function

The derivations written here are based on the work of word2vec *Explained: Deriving Mikolov et al.'s Negative-Sampling Word-Embedding Method (Goldberg and Levy, 2014).*

Let's assume that (w, c) is a pair of words that appear near each other in the training data, with w a word and c its context. In other words, we can denote it as $p(D = 1|w, c)$ meaning that this pair came from the training data. Therefore, the probability that the pair did not come from the training data will be $p(D = 0|w, c) = 1 - p(D = 1|w, c)$. The trainable parameters of the probability distribution are denoted as θ and the notion of being out of training data as D'.

After some calculations, the optimization problem (P) with the addition of a negative sampling element is:

$$argmax_\theta = \sum_{(w,c \in D)} log(\sigma(v_c.v_w)) + \sum_{(w,c \in D')} log(\sigma(-v_c.v_w))$$

.

4. UNIGRAM DISTRIBUTION MODEL

The Unigram Model is a probability distribution for words that makes the assumption that the words in a sentence are completely independent from one another. Indeed, the **Unigram distribution is the probability of a single word occurring**.

We used this model to generate negative pair samples explained in [1]. The idea is to construct k samples $(w, c_1), ..., (w, c_k)$,where each c_j is drawn according to its unigram distribution raised to the $3/4$ power. This word

array is constructed such that the number of occurences of the word w is proportional to $p(w) = \frac{w^{3/4}}{\sum_{w'}(w'^{3/4})}$[4]. Therefore, we initialized the $unigramTable$ (i.e. the word array) with the size $unigramTableSize = 100 * 10^6$ [6] and the probabilities $p(w)$ for each word w.

5. PRE-PROCESSING

We pre-processed the training data set using:

- Lower-case
- Multi-word expressions
- Removing digits
- Removing punctuation
- Stemming

We called all our pre-processing functions in the function $text2sentences$.

5.1. Lower-Case

We put every sentence in lower case so that a word and the same word starting with an upper case letter will be considered the same. Indeed, we will need to predict the words given a certain context. If we don't lower case our data, our model might treat a word which is in the beginning of a sentence with a capital letter different from the same word which appears later in the sentence but without any capital latter, which might impact the results.

5.2. Multi-word expressions

We counted the repetition of the words, and therefore incremented an attribute $counter$. We want to keep track of the number of occurences and the unique indexes of our set of words.

5.3. Removing digits and punctuation

We removed digits and punctuation. We also removed superfluous spaces, dashes and underscores.

5.4. Stemming

Finally, we removed the suffix from a word to reduce it to its root word.

6. ALGORITHM

6.1. Initialization

To initialize the Skip-Gram class, we implemented in the $init$ method all the class attributes, the loss and support variables, the training parameters, an index for the words not in the vocabulary, the context size, and finally the two embeddings for each word (one as context, one as word).

A word embedding is the mathematical representation of word in the form of vectors to which some linear algebra concepts can be applied to extract important features of text[5]. Each word can be represented in the form of vector with value ranging between 0–1 and number of values in each vector can go up to any length. Before training, we start by randomly initializing the embedding matrices $wEmbed$ and $cEmbed$. Therefore, we initialized both $cEmbed$ and $wEmbed$ (respectively word and context embedding) with a matrix of random floats of $np.random.random$ with the size of the vocabulary and the size of $nEmbed$. We then multiplied both matrices with a proportion $\alpha = 0.1$. We tried different values including $np.zeros$ for $cEmbed$, but this version gave us the best results.

6.2. Training

The aim of the $train$ method is updating both embeddings in each iteration **to find parameters to maximize the probabilities that all of the observations indeed came from the data**.
To perform the SGD, we used the two methods $sigma$ and $gradient$. The first one computes the sigmoid function for the input. The second one computes the gradient for the sigmoid function, which is equal to $\nabla x = x.(x-1)$. Then, the SGD algorithm is coded in the method $trainWord$.
For each epoch, we split it into different batches and we updated both word and context embeddings ($wEmbed$ and $cEmbed$ respectively) using a Stochastic Gradient Descent (SGD) with a constant learning rate. At each iteration, we modified slightly the values to improve over the loss function. With $\alpha = learningRate$:

$$\theta = \theta - \alpha \frac{\partial L(x_i, \theta)}{\partial \theta}$$

.

6.3. Cosine similarity

The main technique for calculating the similarity and relevance of words meaning using the Word2Vec model from word embeddings. We generated the similarity value using the Cosine Similarity formula of the word vector values. Cosine similarity calculates similarity by measuring the cosine of angle between two vectors.

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}, \quad (1)$$

It is a better metric than euclidian distance because if the two text document far apart by euclidian distance, there are

still chances that they are close to each other in terms of their context. Therefore, it is better adapted to our case.

6.4. Pearson Correlation

In our version, the Pearson Correlation Coefficient assigns a value between 0 and 1, where 0 is no correlation, 1 is total positive correlation. The formula of the Pearson correlation[7]:

$$corr = \frac{n\sum xy - \sum x \sum y}{\sqrt{[n\sum x^2 - (\sum x)^2][n\sum y^2 - (\sum y)^2]}}$$

.

7. EXPERIMENTS

We trained our Skip-Gram model on the $test - format.sh$ file. We used to train our model on a billion words training set.
We took into account the peculiarities of the contexts used in the *word2vec* software mentioned in [1]. That includes dynamic window size, the effect of subsampling and rare-word pruning. This means that we made sure that *"words appearing less than $minCount$ times are not considered as either words or contexts, and frequent words are down-sampled"*.

7.1. Training Hyper-parameters

7.1.1 Epochs: $epoch$

Increasing the number of epochs should improve the learning process and therefore make our embeddings better. We changed a lot the number of epochs over our experiments, from 2 epochs to 300. We observed that reducing the loss on the train too much was not the right thing to do. Even if we had a 0.88 loss the pearson's correlation was 0.02. We then observed that having a relatively small number of epochs allow us to train the model but not overfit it. Over 50 epochs the pearson's correlation was not increasing or at worst decreasing.

7.1.2 Batch Size : $batchSize$

A batch size of 256 means that we will create samples of size 256 from the training data set will be used to estimate the error gradient before the model weights are updated. This allows us to work on a different batch samples at each epoch. Having a batch size too small will reduce the robustness of our model and it will not be trained enough. 256 came up as the best value tried on the grid search as it is not too small and big enough to take a relatively big part of the train set. However, the training time was really long. So we decided to split the training set to a batch size of 256. We have better performance in less time.

7.1.3 Learning Rate: $learningRate$

The learning rate controls how quickly the model is adapted to the problem. It represents the proportion of modification in our gradient descent. A large learning rate risks that the algorithm jumps over the minima. On the other hand, a small learning rate slows down the convergence and may converge to a local minima which isn't the global minimum. We did a trade-off while trying large and small learning rates. Our experiments led us to a learning rate of 0.01.

7.2. Skip Gram Hyper-parameters

7.2.1 Negative Rate: $negativeRate$

The negative rate is the number of negative words in a training vocabulary. During the experiments, we saw that having a high negative rate slows down the computation time and have a negative effect on the pearson correlation. But one that's too small was not good either. After multiple grid search, 6 came up with the best results.

7.2.2 Window Size: $winSize$

The window size shows how many words before and after a given word are included as its context words. It can affect the training accuracy and computational cost (complexity of the model). Larger windows tend to capture more topic information while smaller windows tend to capture more about word itself[2].
However, we observed that a smaller window size has a negative impact on our results. We chose to train our model on a larger window size, as it is independent from the training data size. We observed, when the window size is bigger (taking more than 2 to 10 words (20 to 25)), the algorithm have a better understanding of the context and performs better, going higher than those values.

8. RESULTS

The results are given in the table below.

Parameters	Value
nEmbed	70
winSize	20
minCounts	3
negativeRate	6
epochs	50
batchSize	256
learningRate	0.01

Table 1: The best hyper-parameters as results of our fine-tuning

We put these best parameters as default values in our $init$ function, that is called when a $SkipGram$ is created.

8.1. Observations

We made our model the most robust possible. **In 3 distinct runs, we have values between 0.109 and 0.12 approximately for the Pearson Correlation**.

```
[13]  1 from scipy import stats
      2 stats.pearsonr(sim_true, sim_exp)

      (0.11987429884634077, 0.00014597363294488886)
```

Figure 1: Pearson's correlation for our best hyperparameters

9. CONCLUSION

In conclusion, we can conclude that the Skip-Gram Model with Negative Sampling is efficient and solves the computational time issues that the word2vec can face. This exercise was challenging and complex in terms of process implementation and fine-tuning over the best model. It was long to train which made the process complex. We learned a lot about the word2vec model and we found it really interesting. Most importantly, we learned about the impact of each training and skip-gram hyper-parameters, as we played with them to make the best model possible.

In order to make our model even more robust and efficient, we could add more epochs and initialize differently the embeddings matrices using the uniform distribution or $np.zeros$ for $cEmbed$ for example.

NB: We have a mistake in the test-format.sh saying the output of the results is not a number. This is the docker pull command: $dockerpullamirmahmoudi/nlp$. The docker image is really heavy as all the environment is downloaded on the $dockerpullcommand$, including all the packages, libraries and most importantly the billion words $.tar.gz$ file.
So, we recommend you to download our $skipGram.py$ **and use your own shell script.** Here, you will find our $skipGram.py$: $https$: $//github.com/anneclairelaisney/NLPExercise1$.

References

[1] Yoav Goldberg, and Omer Levy. "Word2vec Explained: Deriving Mikolov Et Al.'s Negative-Sampling Word-Embedding Method." ArXiv.org, 14 Feb. 2014, https://arxiv.org/abs/1402.3722.

[2] Yoav Goldberg, and Omer Levy. "Dependency-Based Word Embeddings" ArXiv.org, 14 Feb. 2014, https://arxiv.org/abs/1402.3722.

[3] Kulshrestha, Ria. "NLP 102: Negative Sampling and Glove." Medium, Towards Data Science, 12 July 2021.

[4] Kaji Nobuhiro, Kobayashi Hayato. "Incremental Skip-gram Model with Negative Sampling". "Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing", 2017.

[5] Brownlee, Jason. "What Are Word Embeddings for Text?" Machine Learning Mastery, 7 Aug. 2019, https://machinelearningmastery.com/what-are-word-embeddings/.

[6] "Word2Vec Tutorial Part 2 - Negative Sampling." Word2Vec Tutorial Part 2 - Negative Sampling · Chris McCormick, 11 Jan. 2017, http://mccormickml.com/2017/01/11/word2vec-tutorial-part-2-negative-sampling/.

[7] Derry Jatnikaa, Moch Arif Bijaksanaa, Arie Ardiyanti Suryania, "Word2Vec Model Analysis for Semantic Similarities in English Words Semantic Similarities in English Words", 12-13 september 2013