Word Vectorization using SkipGram model with Negative Sampling

Asrorbek Orzikulov, Sauraj Verma, Shivaditya Meduri, and Raghuwansh Raj

CentraleSupélec, M.Sc in Data Science

February 19, 2022

Introduction

In this project, we worked on creating a Word2Vec model utilizing the SkipGram method with negative sampling for faster training. The word2vec algorithm uses a neural network model to learn word associations from a large corpus of text. Once trained, such a model can detect synonymous words or suggest additional words for a partial sentence. As the name implies, word2vec represents each distinct word with a particular list of numbers called a vector. The vectors are chosen carefully such that a simple mathematical function (the cosine similarity between the vectors) indicates the level of semantic similarity between the words represented by those vectors. In this paper, we discuss the mathematical derivation of the training loop, hyper-parameter tuning and additional experiments that we did.

Architecture, training steps derivation, and execution

Architecture of the model

Word2Vec model has 1 hidden layer with number of nodes equal to the embedding dimension size. The model uses sigmoid function to learn to differentiate the actual context words (positive) from randomly drawn words (negative) from the noise distribution. The idea is that if the model can distinguish between the likely (positive) pairs vs unlikely (negative) pairs, good word vectors will be learned. Negative sampling converts multi-classification task into binary-classification task. The new objective is to predict, for any given word-context pair (w,c), whether the word (c) is in the context window of the the center word (w) or not. Since the goal is to identify a given word as True (positive, D=1) or False (negative, D=0), we use sigmoid function instead of softmax function. The model's set of parameters include the weights of the input weight matrix and the output weight matrix. We optimize the gradients using gradient descent approach to maximize the likelihood of positive (context) words and input word occurring together and negative (noise) words and input word not occurring together.

Note: For mathematical derivations, we referred to this article.

$$p(D = 1|w, c; \theta) = \frac{1}{1 + exp(-c \cdot w)} \tag{1}$$

The probability of a word (c) appearing within the context of the center word (w) can be defined as above. The likelihood of set of words, positive words and context words should be maximizied, the objective

function to be maximized can be found below.

$$\underset{\theta}{\text{argmax}} \; p(D=1|w, c_{pos}; \theta) \prod_{c_{neg} \in W_{neg}} (1 - p(D=1|w, c_{neg}; \theta)) \tag{2}$$

for which we can apply logarithm to simplify the process of optimizing. We also convert the maximizing optimization problem to minimization optimization problem by multiplying it with -1. After simplifying the objective function, the final objective function to be optimized can be as seen below

$$\underset{\theta}{\text{argmin}} \; -\log \sigma(c_{pos} \cdot w) - \sum_{c_{neg} \in W_{neg}} \log \sigma(-c_{neg} \cdot w) \tag{3}$$

Derivation of the training update steps

We have to find the training steps to update the input weight matrix as well as the output weight matrix. We will find the update steps by reducing the parameters in the negative direction of the gradient at a rate provided by the user in the form of a hyper-parameter called the learning rate. We will initially find the gradients with respect to output and input weight matrices which are the prerequisities to finding the training steps. With negative sampling, we do not update the entire output weight matrix Woutput, but only a fraction of it. We update K+1 word vectors in the output weight matrix — c_{pos} c, c_{neg1}, ... c_{negK}. We take partial derivatives to the cost function defined in with respect to positive words and negative words.

$$\frac{\partial J}{\partial c_{pos}} = (\sigma(c_{pos} \cdot w) - 1) \cdot w \tag{4}$$

$$\frac{\partial J}{\partial c_{neg}} = \sigma(c_{neg} \cdot w) \cdot w \tag{5}$$

We take partial derivatives of the cost function with respect to the input word

$$\frac{\partial J}{\partial w} = (\sigma(c_{pos} \cdot w) - 1) \cdot c_{pos} + \sum_{c_{neg} \in W_{neg}} \sigma(c_{neg} \cdot w) \cdot c_{neg} \tag{6}$$

$$= \sum_{c_j \in \{c_{pos}\} \cup W_{neg}} (\sigma(c_j \cdot w) - t_j) \cdot c_j \tag{7}$$

Now that we have the gradients, we can write the update steps for the input and the output weight matrix which is given below.

Weight update steps for the output weight matrix:

$$c_{pos}^{(new)} = c_{pos}^{(old)} - \eta \cdot (\sigma(c_{pos} \cdot w) - 1) \cdot w \tag{8}$$

$$c_{neg}^{(new)} = c_{neg}^{(old)} - \eta \cdot \sigma(c_{neg} \cdot w) \cdot w \tag{9}$$

Weight Update steps for the input weight matrix:

$$w^{(new)} = w^{(old)} - \eta \cdot \sum_{c_j \in \{c_{pos}\} \cup W_{neg}} (\sigma(c_j \cdot w) - t_j) \cdot c_j \tag{10}$$

Now that we have the weight update steps, we can start implementing the train function to update the

randomly generated input and output weight matrices with the update steps we derived.

Execution in Python code

```
def trainWord(self, wordId, posId, negIds):
    centerWord = self.inputWeights[wordId]
    posWordVec = self.contextWeights[posId]
    gradCenterWord = (sigmoid(np.dot(posWordVec, centerWord)) - 1) * posWordVec
    gradPosWord = (sigmoid(np.dot(posWordVec, centerWord)) - 1) * centerWord
    self.contextWeights[posId] -= self.lr * gradPosWord
    for negId in negIds:
        negWordVec = self.contextWeights[negId]
        gradNegWord = sigmoid(np.dot(negWordVec, centerWord)) * centerWord
        self.contextWeights[negId] -= self.lr * gradNegWord
        gradCenterWord += sigmoid(np.dot(negWordVec, centerWord)) * negWordVec
    self.inputWeights[wordId] -= self.lr * gradCenterWord
```

We take a center word, positive context word and k negative context words where k is a hyperparameter called the negativeRate, we initially convert the wordIds to the embedded representations using the weight matrices which are later fitted in the update equations to update the weights in the direction of the increasing the maximum likelihood function.

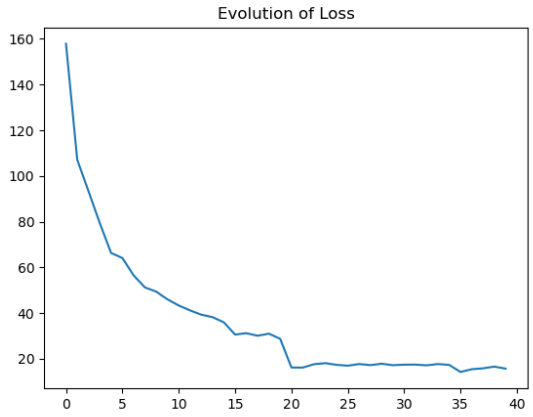

Figure 1: Average Loss over 100 lines

The evolution of the log likelihood loss can be seen in the figure above for a lr=0.01, nembed=300, winSize=5 and negRate=5. The loss is the average loss taken for every 100 sentences trained, we trained for 2000 sentences for 2 epochs; therefore 4000 sentences mod 100, will give us 40 losses, which are plotted in the above figure

Hyper-parameter Tuning

We trained the model with different set of hyperparameters(Learning rate, number of epochs, Embedded Vector Size and Window Size to find the best set of hyperparameters for training. To evaluate the model

performance, we used SimLex word pairs to find the cosine similarities of the word vectors outputted by our models and find the correlation with the actual similarities of the word pairs. Our training data consists of the first 2000 sentences of news.en-00024-of-00100 corpus of sentences.

Table 1: Hyperparameters Used for Training.

Learning Rate	Epochs	Embedded Vector Size	Window Size	Correlation
0.01	2	300	5	0.1126
0.001	2	300	5	0.1022
0.01	2	100	7	0.1014
0.001	2	300	5	0.0895
0.1	2	300	5	0.0795

After testing different set of parameters, we concluded that a learning rate of 0.01, for 2 epochs, an embedding dimension size of 300, context window size of 5 appear to be the best set of hyper-parameters with a correlation score of 11.26%. Therefore we are using those set of hyper-parameters for the final submission code

Additional Experiments

Tokenization

To break sentences into tokens, we tried to use Treebank Word Tokenizer from the NLTK package. However, only a negligible improvement in performance (0.002) was observed over the base case, where we just removed punctuation marks and numbers. When we analyzed the reason why tokenization did not help, we noticed that given sentences had already been tokenized: Punctuation marks were separated by space from last words, and contractions like "hasn't" were broken down into "hasn" and "'t". As a result, further tokenization did not offer a significant increase in the correlation score.

Combination of embeddings

We tried three different embeddings to compute similarity scores, given the optimal hyperparameters obtained in the previous part. First, we simply used the vectors of the input matrix. Second, we used those obtained from the context matrix. Finally, a simple average of the two matrices was computed and the resulting average vectors were used for the similarity scores.

The rationale for trying different embeddings was that for every context-center word pair, only 1 row of the input matrix is updated (by the sum of 6 gradient vectors). Meanwhile, the algorithm updates 6 rows of the context matrix, but only one gradient vector is used for each row. Thus, we were interested in seeing how these different updates affect the quality of embeddings.

Removing stopwords

In many applications, removing stopwords boosts the performance of NLP algorithms. We tried to see if removing the 179 stopwords given in the NLTK package can result in better word embeddings, keeping all the hyper-parameters.