Negative-Sampling Word-Embedding Skip-Gram Method

Preliminary remarks

After the paradigm shift from count-based language modelling towards predictions-based modelling techniques and the development of the Skip-Gram model architecture, the necessity for techniques that help decrease the number of needed calculations to train the network became evident. For the first exercise of the NLP-course, we focused on the technique "Negative-Sampling" in Skip-Gram that reduces the impact of each training sample, significantly reducing computation time, while increasing the quality of resulting word vectors.

Mathematical functionality of the Skip-Gram

The Skip-Gram method is based on the word2vec approach. This means the algorithm tries to predict a word based on the surrounding context after being given a corpus of words. The approach is therefore solving a probability maximisation problem that optimises the following cost function:

$$\underset{\theta}{\operatorname{argmax}}\ log\, p(w_1, w_2, \ldots, w_C | w_{center}; \theta)$$

Center word-based probability maximisation function

In the case of Skip-Gram, C represents the window size and w is a word vector that represents the context or center word. The parameter theta is the weight matrix that is being optimised. In order to facilitate the derivation, we take the logarithm of the cost function.
For Skip-Gram, the softmax function is used for context word classification. It maximises the probability of observing all C context words given the hidden projection layer h of a center word.

$$\underset{\theta}{\operatorname{argmax}}\ log \prod_{c=1}^{C} \frac{exp(W_{output_{(c)}} \cdot h)}{\sum_{i=1}^{V} exp(W_{output_{(i)}} \cdot h)}$$

Skip-Gram softmax probability maximisation function

In the equation, W_output represents a row vector for a specific context word of the output embedding matrix. The softmax is then put into the "Center word based probability maximisation function" explained before. This results in an objective function that maximises the probability of observing the context words based on a center word.

Functionality of Negative-Sampling

Although the mathematical approach seems streamlined in theory, it reaches some limitations in practice. The output embedding matrix W_output must be scanned through in order to receive the probabilities for all words. This results in a computational complexity O(x) that scales with the corpus of size x, making the solution not scalable.

Negative sampling tackles this issue by replacing softmax with a binary classifier. In practice, a certain set of n words are randomly selected. Instead of tweaking all weights in the neural network, negative sampling only updates the few chosen weights. For those weights, the prediction will be compared with 0 (not in context) and the prediction error gets back propagated. This saves a tremendous amount of computing time and reduces the complexity to O(n), n being the number of negative samples.

Implementation

In the following we will navigate through the code and comment on our implementation.

Data preparation

First, we will focus on Preprocessing. After the text corpus that will be trained on is loaded, it needs to be split into sentences and words. To do so, we made use of the already implemented *text2sentences* function where each row of the text is being saved as one list and every word within the sentence is saved as an element within the list. Using the regular expressions library we excluded all characters that are not alphabetical and converted all words to lower characters. We also chose to exclude white spaces.

Additionally to the preprocessing, we tried to make use of the SpaCy library. In particular, we thought excluding all stopwords in the English language as well as all words bigger than 2 would increase the accuracy of the model. Therefore we used the nlp.pipe pipeline with disabled Entity recognizer and parser for speed. After attempting to run it, we noticed however that spaCy took a considerable amount of time, which is why we deactivated it for the final version.

Skip-Gram Class

Next, we want to introduce our implementation of the Skip-Gram class. This consists of various functions as well as the actual training process that calls upon all said functions.

The class takes various hyperparameters that are predefined with a set of values. The class then takes care of the data handling and applies negative sampling by sampling the data with a unigram distribution.

Feed Forward propagation

For a specific sample, the input layer (word-embedding matrix containing the features for each word) is combined with the output layer (word-embedding matrix containing features of context words) at the respective index that is currently sampled. This results in a prediction matrix (y-pred) for each word in the corpus.

Backward propagation

The true results (y-true) come from the one-hot encoded matrix with 1 at the index of the context word. The error is then calculated by taking the difference between (y-pred,y-true). For each iteration, the input and output layer is then adapted via backpropagating the prediction error, therefore representing the basis for optimising the weight matrix theta.

The process is based on a stochastic gradient descent meaning that one update on the weight matrix theta is being made for each training sample center word.

Train Function

We partially adjusted the existing train function to include the option of iterating through a provided number of epochs. The standard number of epochs was set to 20 and we implemented a for loop going through the range of epochs. We also included a small notion that informs the user on the condition and progress during the training process by printing the number of epochs as well as the associated loss.

TrainWord Function

To train the model, we implemented a function that calls the feed_forward and backpropagation functions.

Save Function

For being able to save the weights, vocabulary and IDs mapped to words, we called the needed variables and stored them in the path provided by the function.

Similarity Function

Here, we defined the function needed to compute the similarity between two words within the vocabulary. To do so, we first transformed the provided words to numerical representations using the w2id and afterwards applied the already developed *pairwise.cosine_similarity* function of the *metrics* module from the *sklearn* library.

Load Function

Lastly, we implemented the loading function of the method. This includes calling the Skip-Gram class as our model and passing it the path containing the weights, text to pass through and the mapping of words from the vocabulary to IDs.

Fine Tuning of Hyper-Parameters

The optimal hyperparameter values are often known to be data and task dependent. We thus investigated the marginal importance of each hyperparameter through large hyperparameter grid searches. Results reveal that optimising many hyperparameters, namely negative sampling distribution, number of epochs, subsampling parameter and window-size, significantly improves performance on the task, and can increase it by an order of magnitude.

Initially, we started the investigation of optimal hyperparameters by running the model only for 1000 lines and were able to reach a 9.52% correlation score between the simlex and output similarity rates. This was achieved through embedding vectors of length 100, negative words rate of 10, window size of 5, a minimum count of 6 in order for a word to be included in the vocabulary and finally a random weight initialization from a uniform distribution. We also noticed, early on, that running the model for increasing the amount of lines substantially improves the model accuracy and were able to reach 12% accuracy by running it for 5000 lines and 10 epochs.

Next, we experimented with the learning rate in the range of 0.01 to 1. We found that smaller learning rates required more training epochs, as well as were not efficient in finding a good minimum value. Similarly, any value above 0.08 was too large and caused the model to converge to a suboptimal solution.

With regards to the number of epochs, while running the model for 1000 lines, 10 epochs were sufficient in order to reach the minimum loss that also did not overfit to the training dataset. For an increasing number of lines even less epochs were required.

In order to initialise the weights we have tried zero initialization, as well as random initialization from a uniform distribution and a normal distribution. As expected, random initialization was a better choice to break the symmetry and to avoid memorising the same function. Moreover, the random numbers from a uniform distribution with not too high or too low values resulted in the lowest loss.

Next to the already mentioned steps, we tried implementing the Automatic hyperparameter optimization function of fasttext. To do so, we installed the library and trained the input corpus using the *fasttext.train_unsupervised* function with a validation file included in the 1 Billion word modelling benchmark and an autotune duration of 300 seconds. We encountered, however, some issues during the training process, which is why we neglected the findings and stayed with our original parameters.

Final results

The Skip-Gram model seems to be very context dependent. The highest improvements seem to stem from including a bigger word corpus which highlights the fact that the benchmark is trained on a substantially larger dataset.

In theory, the model is very computation intensive and negative sampling provides a good way of reducing the amount of computation without substantially increasing error or training time.

Our model reached a maximum accuracy of 12%. However, for further development we would suggest to include a larger corpus and to tune the parameters using the fasttext automatic hyperparameter optimization. Furthermore, addressing potential bias in the input corpus could further increase the accuracy of the model.