NLP Project Report

Nastaran Okati

1 Task 1: Word Embeddings

In this section, I describe my approach for extracting the word embeddings in detail.

1.1 Introduction

Despite image classification or speech recognition tasks in which the data can be directly fed into the Neural Network, in Natural Language Processing (NLP), one needs to first get the vector representation of the words and then perform the task on these word vectors. A word vector is a projection of a word from a $\mathcal{V}$ dimensional vector where $\mathcal{V}$ is the size of the unique words existing in the dataset into a lower dimensional space (typically a few hundred). The projection is done by adding a hidden layer which can be thought of as feature extractor. A CNN can then be trained on top of these word vectors.

1.2 Methodology

In order to extract the word embeddings, I will use Word2Vec [5]. I will go through the code step by step and describe what and why each step was done.

1.2.1 Data Preparation

In the data preparation step, I remove all the words which are not valid from the dataset. A word is valid if it satisfies some of the criteria defind in the "valid" function. For example if the word is solely a puctuation, or it starts with "@" or it is a url, or it is a stopword, it is not considered as a valid word. Later on, I add all the valid words to an array named "new tweets" (here I also save the label of each tweet to later on use for the classification task). Note that I do not remove the words starting with hashtags since usually they are good key words which can tell us important information about the context of the tweet.

1.2.2 Build the Vocabulary

In this task, I need to save all the distinct words into an array named V. To do so, I define an empty map and add each word that I see to this map and only add a word to V if it is not added to the map so far.

1.2.3 One hot encoding

to represent each word by a vector, we use one-hot encoding of the word. In one-hot encoding, the word is represented by a vector of size $|V|$ (size of the dictionary) and the i'th element of this vector is 1 if the word is the i'th word of the dictionary.

1.2.4 Subsampling

We need to calculate the probability of keeping each word in the context which is related to the frequency of the word in the dataset.

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

1.2.5 Skip-Grams

In this part, we define a context for each sentence. The context is defined based on the words before and after each word in a window (we use window size of 5 as the original paper). We iterate through all words in a sentence and check the neighbors of that word which fall into the window and then based on the sampling probability decide whether to keep that word in the context or not. Basically the sampling probability of the word is the parameter p of b binomial distribution which outputs 1 with probability p and 0 with probability $(1 - p)$ and we sample from this distribution and if the output is 1, we keep the word.

1.2.6 Setting Hyper parameters

We use window size of 5 and embedding dimension of 300 as the original paper. By looking at the training loss, I stop training whenever the training loss stops decreasing. More formally, I define a patience threshold t and stop training if the traininig loss does not get better after t number of steps (described more fomally in the third section). I tried several learning rates and picked the one giving me the best result on the validation set.

1.2.7 The Word2Vec Module

The implementation is exactly based on the paper. The hidden layer gets an input of size $|V|$ which is the size of the one-hot representation of each word and outputs a vector of size embedding size (300 in our case). The last (second) layer gets input of embedding size and outputs a vector of size $|V|$. There is a log softmax layer which calculates the log softmax of the second layer. Note that the second layer as well as the log softmax layer are only needed for training the Word2Vec module (since we need to optimize the weights based on some loss function and we can use negative log likelihood loss if the output of the network can be interpreted as log likelihood which is done by the log softmax layer.

1.2.8 Loss function and Optimizer

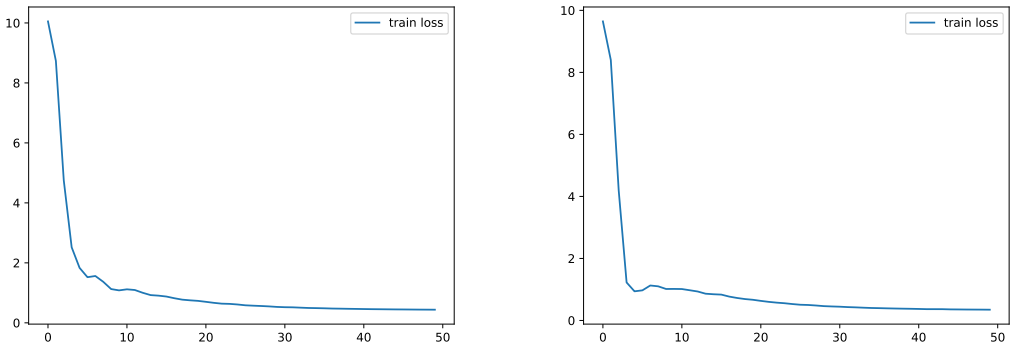
I use Negative Log Likelihood loss. The last layer of our Word2Vec module is a logsoftmax layer which can be interpreted as the log-likelihood of each class (in our case the interpretation is the likelihood of each each word of the dictionary being in the context of the current word). Hence, knowing the context of each word and getting the log-likelihood out of the Word2Vec model, we can calculate the negative log-likelihood (negative of the output of the model corresponding to the ground truth label (note that the ground truth label is the context of the word which we calculated in the last part)). As the optimizer I use Adam optimizer since it is the state of the art and has shown better convergence than other methods. I tried with other optimizers such as SGD, AdaGrad, AdaDelta, LBFGS and got similar or worse results and hence continued with Adam.

1.2.9 Training Procedure

For each of the tweets, we first need to get the context of all of the words inside it and then turn the context (which is some word) into its one-hot encoding and use it as the label for training this network. Then we start the actual training procedure: Using batch size of 64, in each epoch, for each batch of data, we send it to the Word2Vec module and get its output and calculate the Negative log-likelihood loss of the ground truth label and the model output. The gradient is then back propagated through the network to update the learnable paramers through loss.backward(). The optimizer then updates the parameters based on the gradients and the learning rate using optimizer.step(). At the end of each epoch the average training loss over batches is displayed to make sure the values are reasonable and decreasing which is the case.

1.2.10 Saving the model

Now that the training is finished, we have our feature extractor! The weights of the first layer should be saved so that if we input a word, we can get the 300-dim vector which can be thought of as its feature vector.



(a) Hindi (b) Bengali

Figure 1: The training loss of the Word2Vec model trained on Hindi and Bangali datasets averaged over all training samples during 50 epochs. The loss is the negative log likelihood loss. We can see that the loss is decreasing during all training epochs. The rate of decrease is faster in the beginning of training (exponential) and gets slower as we proceed which is due to the use of adaptive learning rate which decreases as we manage to settle in more reasonable parts of the optimization space.

1.3 Discussion and Results

When I monitored the loss on the validation set, I noticed that it is not very nicely decreasing. As this is a sign of overfitting, I tried to solve this issue by the normal tricks that one can use to overcome overfitting (discussed more in the next section) but non of them worked. I am not sure if this is due to distribution change between the train and test set which is inherent in the dataset or it is a problem with my code. Figure 1 shows the training curve for both of the dataset which illustrates that the hyperparameters are well tuned and the loss is monotonically decreasing.

2 Task 2: Sentiment Classifier and Word Embeddings

In this section, I describe my approach for text classification, as well as the CNN I use and its training procedure in detail.

2.1 Methodology

In order to classify the text, I will use a CNN designed for text classification [3]. The architecture of this network is described later in detail.

2.1.1 Preprocessing

In our dataset, different tweets can have different lengths but we want all of them to be of same length since a CNN cannot accept inputs of different lengths. To do so, I pick the average lengths of tweets across all tweets as the final length of tweets and for those which are longer than this size I cut the first words and for those which are shorter I pad them with random numbers between 0 and 1 (the same as [3]). Also note that I already cleaned the tweets and removed the stop words etc in the embeddings extraction part and saved the cleaned tweets so that I only load them in this step. I then split the data into three sets: Train, Validation and Test. I use 80% of the data for training, 10% for validation and 10% for testing. It is crucial to leave a set as validation set in order to monitor the training process for overfitting/under fitting and tune the hyperparameters. The embeddings as well as the ground truth labels for all three subsets are saved and are ready to be fed into the CNN.

Note that for the Bangali data we had to pick a portion of the data which has the same statistics as the Hindi dataset since we want to fine-tune the classifier which was trained on the Hindi dataset. To do so I make sure that I pick roughly equal number of samples from each class since this is the case for the Hindi dataset.

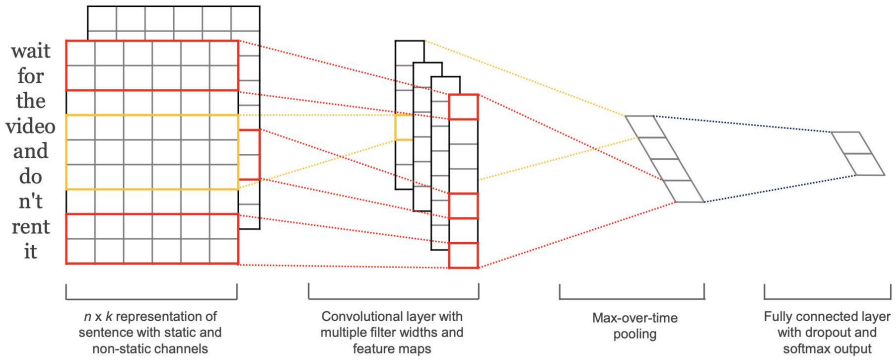


Figure 2: The CNN architecure used for text classification, the figure is taken from [3]. As can seen from the figure, each word in the sentence is first represented by its embeddings which are taken from the trained Word2Vec module in our case. The embeddings are concatenated to form the final representation of the sentence (with croppings and paddings to make the length of all sentences equal). Then 100 convolutional filters of size 3,4,5 are applied over the data to further do the feature extraction. Max pooling is then applied after these layers to reduce the dimensionality. All the outputs of the convolutional layers are concatenated and passed to a fully connected layer with 50% dropout rate. A softmax is then applied to get the class probailities. (In our case, I used log-softmax to later on use the Negative Log Likelihood loss to train the model).

2.1.2 CNN architecture

The CNN uses 300 convolutional filters of size 3,4,5 (100 filters for each filter size). All the filters are convolved over the image (the shapes of all of the vectors are mentioned in the code through comments after the operation) and then a ReLU non-linearity is applied over them. The resulting tensor is then passed through a max pooling layer which selects the maximum between the activations resulting from the convolutions (hence for each of the filters we will have one output and in our case since we have 300 filters we will have 300 outputs from each of the filter sizes). These tensors are all cocatenated and passed through a dropout layer which randomly ignores 50% of the neurons. In the end, we have a fully connected layer (which is in fact a linear layer) which gets the resulting tensor of previous steps and outputs a vector of size number of classes (in our case which is a binary classification, 2). I later on use a logsoftmax layer on top of it to be able to use the Negative Log Likelihood loss for training the mode. (Please note that I chose 100 filters in the original paper which proposes the CNN architecture, they used 100).

2.1.3 Training Procedure

I use negative log-likelihood as the loss function and Adam as the optimizer and batch size of 32. The prepared data is first loaded and then in each epoch we go through the batches of data and feed them to our model and calculate the loss between the output of the model and the ground truth label and back propagate the loss through the network to update the parameters. After each epoch, we also pass through the validation set and calculate the loss there. We save the model which performed best on the validation set as the final model.

Note that for the Hindi dataset the classifier is trained from scratch while for the Bangali dataset it is fine tuned from the classifier trained on Bangali dataset.

2.1.4 Results and Discussion

I could get an accuracy of 72% and 69% on Hindi and Bangali datasets respectively. One visible problem with my approach is that the validation loss starts increasing after only a few epochs while the training loss keeps decreasing. This is a clear sign of overfitting which I tried to solve by reducing the number of parameters (using less number of filters, using smaller filter sizes, increasing drop out

etc) but non of them helped. I am not sure if this is a problem with my code or that in general the data distribution varies between different samples causing the train and validation sets being from diverse distributions and hence a model fitted on the train set is not able to generalize on the validation or test set.

3 Task3: Challenge

Here are the approaches I tried for improving my model:

3.0.1 CNN Architecture

First of all, there are many text classifier CNNs which were introduced since text classification got popularity such as [1, 6, 4]. The reason why I picked [3] is that it is state-of-the-art CNN classifier for text classification which is a simple CNN that achieves excellent results on multiple benchmarks according to its authors (I had also previously used this CNN for classifying the tweets in Hatespeech [2] dataset and could get 89% accuracy). Some of the charactristics of the model which may be a reason of its good performance are as follows:

3.0.2 Cleaning the data

Since the data also contains English phrases, I also tried removing English stopwords to make the Word2Vec model more accurate (It did not help much with the loss of the Word2Vec model though. I also remove the Urls from the data and all the punctuations.

3.0.3 Monitoring the training procedure

This was done by setting aside a subset of the data as the Validation set and checking the train and validation curves during the training procedure. In a good model both train and validation loss must be decreasing as we go through epochs and if one sees the training decreasing while the validation increasing (overfitting) they should stop training or change parameters (if the training loss is not low enough).

3.0.4 Model Architecture

First of all, there are many text classifier CNNs which were introduced since text classification got popularity such as [1, 6, 4]. The reason why I picked [3] is that it is state-of-the-art CNN classifier for text classification which is a simple CNN that achieves excellent results on multiple benchmarks according to its authors (I had also previously used this CNN for classifying the tweets in Hatespeech [2] dataset and could get 89% accuracy). Some of the charactristics of the model which may be a reason of its good performance are as follows:

- The dropout layers: The 50% dropout layers help in generalization of the model by adding some stochasticity to the output of the network.
- The max pooling layers: The CNN contains three max pooling layers which picks the highest Neuron value among the ones output by each filter (So for each filter one value is returned and since we have 300 filters of each size in the end we have 300×3 values which are then concatenated together and fed to the fully connected layer).
- Batch Learning: It is often observed that the Machine Learning models which are trained on batches have better performances than the ones trained instantly on all the data points or the ones trained using stochastic gradient descent (batch size of 1). I tried different values of batch size and in the end picked the one with best performance on the validation set.
- Shuffling the data: At the beginning of each epoch, one can shuffle the data in order to add more randomness and stochasticity to the model. I used it both for training the Word2Vec model and the CNN.

References

[1] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3(null):1137–1155, Mar. 2003.

[2] T. Davidson, D. Warmsley, M. Macy, and I. Weber. Automated hate speech detection and the problem of offensive language. ICWSM, pages 512–515.

[3] Y. Kim. Convolutional neural networks for sentence classification, 2014.

[4] Q. V. Le and T. Mikolov. Distributed representations of sentences and documents, 2014.

[5] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space, 2013.

[6] W.-t. Yih, K. Toutanova, J. C. Platt, and C. Meek. Learning discriminative projections for text similarity measures. In *Proceedings of the Fifteenth Conference on Computational Natural Language Learning*, pages 247–256, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.