NNTI Winter 2020/21 NLP Final Project

Alice Chase
7007281
Saarland University
alch00001@stud.uni-saarland.de

Kushagra Sharma
7010529
Saarland University
kush00001@stud.uni-saarland.de

1 Introduction

We took two approaches to the code. Our initial attempt of Task 1 and 2 consisted of simple but effective pre-processing and networks using PyTorch. We implemented a basic Skipgram model and for the sentiment analysis used a simple Long Short-Term Memory (LSTM) network. After running these two models over both Hindi and Bengali data sets we went back for Task 3 in which we modified both the word vectors in Task 1 by implementing negative sampling and replacing the LSTM model with a Convolutional Neural Network. These improvements, along with hyper-parameter tuning resulted in an increase in test accuracy and the need for less epochs. Besides using Hindi word embeddings for Hindi sentiment and Bengali embeddings for Bengali sentiment we also used our final sentiment analysis model trained on Hindi to classify Bengali. While findings are summarized here, exact output of the code and word embeddings can be found in the Data folder.

2 Methodology

2.1 Task 1

2.1.1 Text Processing

We loaded the datasets into pandas dataframes to clean and process it. We decided to remove both emojis and hashtags even though there is evidence, especially for platforms such as Twitter, that keeping these objects improves sentiment classification, particularly for positive sentiments[1]. However, for emojis only a limited few are usually useful in improving classification and that their usage can vary widely depending on the demographic[2]. Given that only a minority of emojis can be reliable sentiment indicators and their context of use may be different depending on the language and culture, we decided to exclude them. For similar reasons, while hashtags can be useful their sparsity and reliability is variable[1] so we removed them for simplicity.

For the Hindi dataset we did try removing the English words, as mixing the languages common in that dataset. We did this using a few NLTK capabilities but the difference in loss values when running the Skipgram model was negligible so we decided to simplify the pre-processing further by retaining the English words.

Finally, we also defined some variables that weren't explicitly asked for but made other functions easier; corpus which is all the words in the sentences as they occur rather than just the unique words, and two dictionaries – word2idx and idx2word. The former was useful for counting word frequencies and the dictionaries were for creating a context tensor in the form that the loss function could handle. They were also useful in debugging to ensure that our word vectors were mapping to the correct vector.

2.1.2 Model Training

The initial model was very simple and used Pytorch. It had two fully connected linear layers, the hidden layer for our word embeddings that we would keep to use for Task 2 and output layer for the

word probabilities that was discarded. We decided to use the Cross Entropy loss function and Adam optimizer.

The target/center word was passed into the model as a one-hot encoding while the context was used in the loss function where it's one-hot encoding was generated implicitly The loss values were high. For the initial development set, using a batch size of 10 and learning rate of 0.001 the loss value started at around 7 and decreased to 2.8 after 15 epochs. For the full sets, both Hindi and Bengali had higher loss values at around 9 and decreased to around 5. Even after hyper-parameter tuning this was the best we could get, so we came back to this as part of Task 3 to try and improve the quality of our word vectors.

As we used Torchtext for the sentiment analysis models we stored our word vectors with their corresponding vocabulary file in a text file in the format of: word value 1 value 2...value n. You may view these files in the Data folder.

2.2 Task 2

2.2.1 LSTM

The data from the Hindi Dataset was pre-processed using the same pipeline created in task one. The hashtags, usernames, emojis and punctuations were removed from the data and the sentences were tokenized to be processed further using torchtext library.

A train set was made using 75% of the data and 25% of the data was used for test set. Using the build vocab attribute of the Field module, the text and labels were created to be used in the model. The build vocab attribute automatically applies padding to the sentences to make it of equal size, this makes loading the data in minibatches very easy as all the sentences in a batch need to be of equal size. Using BucketIterator we make batches of similar length of sentences so least amount of padding has to be used.

In addition to our pretrained weights we had two other rows in the embedding matrix, padding and unknown. The padding is for what is explained in the above paragraph, and is a 0 vector for adjusting our sentence length. The unknown was also intialized to 0 but unlike the padding, it will train word embeddings for when the model encounters an unknown word, for example if a word was dropped during subsampling.

The model was created using an embedding layer, a LSTM (long short term memory) layer and a linear layer. The embedding layer is a fully connected layer which takes the sparse one hot encodings as input and converts it to dense vectors for the LSTM layer. Unlike RNN (recurrent neural network), LSTM can hold information over longer intervals[4]. Along with the hidden state, LSTM also maintains a cell state which helps in storing information over long intervals. So, at any timestamp the LSTM has 3 inputs the hidden state, cell state and the observation at time t. Two such layers were used for the model before the final hidden state from LSTM was fed to a fully connected linear layer which transforms it to the required output dimension. The forward method was called to feed data to the model through batches and sequentially execute all the three layers and finally the forward method returns the output of the linear layer as predictions. We referenced code from the github repository[6] as a basis for our model which we expanded and modified for our own use.

We define a training function to calculate the loss and do backpropagation on the training split. To calculate loss the criterion used is `BCEWithLogitsLoss()`, since we only have two classes and the output is a one-dimensional tensor with each value between 0 and 1. We perform the gradient descent using the Adam optimizer. The train method returns the epoch loss and accuracy averaged over all the values in the batch. We use the function `binary_accuracy()` to caluclate the accuracy. Before calling it, the sigmoid layer is applied on the predictions to scale the values between 0 and 1. The output from the sigmoid layer is rounded off to get the classification and the accuracy is then calculated on the proportion of correctly classified samples. The loss and accuracy for the test set is calculated with the help of an evaluate function.

2.3 Task 3

2.3.1 Negative Subsampling

Going back to the first Skipgram model we implemented negative sampling (see the SGNS.ipynb file) which has been shown to improve the quality of word embeddings [3].

To generate the 'negative' samples we followed closely what the original code did specified in the paper[3], although it was written in C. The first part involved generating a table from which to pull the words. In the function `gen_negative_sample_table()` in the SGNS.pynb file we created a numpy array of 1e5. The original authors used 1e8 but it was likely scaled to fit the size of their dataset which was much larger. To fill it we chose words from the corpus and followed the formula in the paper:

$f(w_i)^{\frac{3}{4}} \sum_{j=0}^{n} = (f(w_j)^{\frac{3}{4}}) P(w_i) = f(w_i)^{\frac{3}{4}} \sum_{j=0}^{n} = (f(w_j)^{\frac{3}{4}})$

This is the same as what was used in the sampling function for drop probability, with the $exponent = .75$ because the authors of the paper stated that's what generated the best results[3]. Unlike the sampling function whose purpose is to drop common words, in the `gen_negative_sample_table()` function it is used to generate words to fill the table so more common words will appear more times.

A separate function, `get_sample_table(table, count)` is used to extract words from the table, sampling by generating a random number between 0 and the 1e5-1. The function also takes 'count' as input to specify how many words you want to return.

The final function we modified before the model was `get_target_context(sentence)`. In addition to generating our positive samples we also generate the negative examples as it is easier to do it in this function so we can ensure that the negative examples don't also appear in the positive examples for our target words. In case there were repeats, we set the number for the parameter of how many words to return from `gen_negative_sample_table()` function to 10. However, for every target words only 5 negative samples were generated as the paper recommended windows of 5-20 for smaller data sets[3]. For the target, context lists a third element was added to label whether the example is a positive or negative pair – 0 and 1, respectively. All of the examples are shuffled together and the label is used in the training loop by comparing the model output to these labels using our loss function.

The model itself was rewritten from our original word2vec code because negative sampling uses Sigmoid instead of Softmax. Both the target and context words are inputted to the model as one-hot tensors. There were two fully connected linear layers like our original code but the dimensions are the same (instead of reversed) because the first one is for the weights of the target and the second for the context words. They are dotted together to produce an output between 0 and 1 which we can then compare with our labels element.

2.3.2 CNN

The pre-processing for data cleanup was the same as in Tasks 1 and 2.

We used CNN-2d model for training and evaluating the data instead of LSTM. CNN is generally used as a feature extractor for images but the idea behind using CNN on text data was to implement different n-grams models using different filter sizes[5]. For our model we use the filter sizes 2,3,4 and 5. Making a filter size greater than 5 is not feasible as that would not work for smaller sentences. The embedding size used is 300, so the different filter sizes would be 2x300, 3x300, 4x300, 5x300. The idea is to identify different bi-grams, tri-grams, 4-grams and 5-grams which are relevant for sentiment analysis. The output is then passed through a max pooling layer where the feature with the maximum value or the most important of the different n-grams for a sentence. The model has 200 filters of 4 different sizes which gives us 800 different n-grams. These are concatenated into a single vector and passed through the linear layer to predict the sentiment.

The rest of the training and evaluating part is similar to Task 2.

3 Results

The loss values from our original skipgram to skipgram with negative sampling improved greatly, starting from loss values of over 9.0 to ones under 0.9. However the training time nearly tripled because of the creation of over double the amount of one-hot vectors needed. In the original Skipgram program only the center word needed to be generated to a one-hot and passed into the model, whereas with negative sampling required both the center and context words to be passed in. Because we also had an increase of word-pairs to train on with the additional 'negative' samples a way to improve this is to look at more efficient ways to generate one-hot encodings.

Unfortunately because negative sampling was computationally expensive so we stopped training after only a few epochs although we may have gotten better embeddings if we continued. However, the loss values were already under 0 so it was better to stop before we started overfitting.

The CNN models were first run using the embeddings from task1 for both Hindi and Bengali data and a slight increase was observed on both datasets when compared to LSTM. Then we used the negative samplings with the CNN model and the combination outperformed the LSTM model good margin.

For the LSTM model using our original embeddings (no negative sampling) we got test accuracies of roughly 74% and 79% for Hindi and Bengali respectively.

For the CNN model using the original embeddings the test accuracy for Hindi was around 77% and the embeddings generated from SGNS model was around 84%.

Likewise, for Bengali, test accuracies of 82% (without negative sampling) and 84% (with negative sampling).

In terms of hyper-parameter tuning, the number of epochs was the one that had the most significant impact. Overall, the LSTM model required more epochs than the CNN. For LSTM we found that about 5 epochs was best before it started overfitting. The CNN only required 2 epochs for the original word embeddings but 3 was best for the embeddings generated by negative sampling. However, the starting training accuracy for the negative sampling was much higher at around 70% whereas the original embeddings started at around 55-60%. If you would like to see the exact outputs from a a couple test runs, including the training accuracies and the loss values after each epoch, please view the accuracies pdf document in the Data folder.

Increasing the number of filters in the CNN model also increased performance but only around half of a percentage.

For transfer learning we used the CNN model and the Hindi embeddings with negative sampling to train the model and tested it on a portion of the Bengali dataset. This yielded a test accuracy of roughly 53% which is not much better than random guessing. This is unsurprising because the word embeddings won't correspond properly to the hindi dataset. However, it would be interesting to see what would happen if tested on a language dataset that was less related to Hindi such as Chinese or French.

We were interested in trying to run the sentiment classifier with Hindi word embeddings for Bengali but it was not possible to do with our model architecture, when we attempted to do it we received high accuracy. The reason for this is because establishing the weights for torchtext requires the word mapped with the weights. When the model encounters unknown word it maps it to an unknown embedding tensor which is trained while doing the sentiment analysis. Thus, when using Hindi words for Bengali all will map to the unknown embeddings which get trained and vice versa. We suspect this was the reason the CNN model took a longer amount of time to train because it was also training the embeddings.

4 Conclusion

It was observed that the test accuracy on the Bengali data was slightly better for all steps (word vectors, LSTM, and CNN) compared to the Hindi data and required less number of epochs. This might be because the total number of tokens in Bengali data set is lesser than the Hindi data set (which we found while making the word, context pairs for the first task) so it takes lesser iterations to train the Bengali data. The Hindi data might also have more noise due to its mixing of English or

other variables, which perhaps could be improved with adjustments to the pre-processing steps. Thus, different languages require different cleaning processes to be effective.

Unsurprisingly the results improved moving from the LSTM to CNN model and for improved word embeddings from negative sampling. However, the differences were small, only a few percentage increase. Applying a sentiment analysis on Bengali after being trained on Hindi yielded low accuracy which suggests that not much transfer learning is taking place and the model was merely guessing.

It would be interesting to apply the models for other classification tasks but on the same language. For example, identifying sentiment of reviews or even hate speech but taken from other sources than what were used for the original data.

References

[1] M.D. Samad, N.D. Khounviengxay, M.A. Witherow, "Effect of Text Processing Steps on Twitter Sentiment Classification using Word Embedding," arXiv preprint arXiv:2007.13027: 1-14, 2020.

[2] Wang, H., Castanon, J. A. (2015). Sentiment expression via emoticons on social media. Retrieved from arXiv:1511.02556

[3] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013b. Distributed representations of words and phrases and their compositionality. In NIPS, pages 3111–3119.

[4]Hochreiter S, Schmidhuber J. Long short-term memory. Neural Comput. 1997 Nov 15;9(8):1735-80. doi: 10.1162/neco.1997.9.8.1735. PMID: 9377276.

[5]Y. Kim. Convolutional Neural Networks for Sentence Classification Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL , page 1746–1751. (2014)

[6] https://github.com/bentrevett/pytorch-sentiment-analysis

[7] McCormick, C. (2016, April 19). Word2Vec Tutorial - The Skip-Gram Model Retrieved from http://www.mccormickml.com