Project 3
Deep Learning E0 250 (CSA)
Textual Entailment (Stanford Language Inference Dataset)
Logistic Regression & LSTM / BiLSTM

Rahul John Roy (M.Tech CDS) - 16703

1. The Problem.

The problem is to do a textual entailment classification on the Stanford Language Inference Dataset using two different models:

- Logistic Regression (LG) using TF-IDF vectors
- A deep neural network architecture (LSTM and BiLSTM)

Given a ‘premise’ and a ‘hypothesis’, the task is to determine whether the ”hypothesis” is true (entailment), false (contradiction), or undetermined (neutral) given that the ”premise” is true.

In the Logistic Regression model, Text Frequency - Inverse Document Frequency has been used to generate the vectorized form of the words.
In the Deep Learning model, PyTorch has an inbuilt vocabulary generator function for NLP datasets. This can be used or, pre-trained vectors like GloVe can also be used for the same.

2. Structure.

- **Requirements.**

The following libraries are required to be pre-installed for working of the code

- **NLTK**
- **Spacy (**With the English language loaded**)**
- **Torchtext**

NLTK was used for pre-processing of the SNLI data obtained from the link attached in the project instructions. Spacy is used for tokenizing the sentences for the dataset that is inbuilt in PyTorch. Torchtext[1] is used for generating the iterable for the dataset to be passed into the training and the testing phase into the model. It contains the data processing utilities and popular datasets in NLP.

[1] https://torchtext.readthedocs.io/en/latest/#

The configuration used was the following:

- Python 3.6.9
- Pytorch 1.4.0

The training was done on Google Colab with the runtime chosen as GPU and partly on a personal laptop with a Nvidia MX250 GPU (2 GB)
The dataset for the LG model is stored in the Dataset folder in the code directory. The LSTM model downloads the dataset during runtime from torchtext during runtime.

- **Organisation.**

The following files were part of this project and the organisation of the folder is also explained below:

1. **main.py** - The main file of this project. It downloads the test dataset from the inbuilt APIs in torchtext and tests it on pre-trained models that have been saved. It generates two output files - **LSTM.txt** and **TFIDF_Log_Regression.txt**. These have the output classes predicted by the two networks and also the accuracy of each model in the testing phase. It creates the testing model by importing the corresponding class from the two python files **bilstm.py** and **tfidf_snli.py**.
2. **bilstm.py** - This file contains the implementation of the LSTM network considered in this project. The code for training also is given in the code for and the training is done on the SNLI dataset from the APIs in PyTorch. This is downloaded during runtime.The trained models are stored in the respective folder with the parameters in the model name.
3. **tfidf_snli.py** - This file contains the implementation of the LG model using the tfidf features.The model and the vectorizers are stored as pickle files and loaded by main.py during testing.
4. **pre_processing_snli.py** - This file does the preprocessing of the dataset for the LG model. The training, testing and validation data are loaded separately, and stored as .csv files for use later.
5. **Dataset** - This folder holds the processed and unprocessed test datasets. The train dataset has not been uploaded here, since Github regulates the maximum size of files to 100 Mb.
6. **Models** - This folder holds the trained models for LG and the DNN models.

.

3. Methodology.

- **Pre-processing for LG model.**

  The pre-processing for the logistic regression model has been carried out with the NLTK library. The txt file for train, validation and test are read separately and processed separately. The files are read into a dataframe as tab separated files and then the next steps are done.

  - Initially the blank columns in the 'sentence1' and 'sentence2' columns are found out and the whole row is removed from the dataset.
  - Then stopwords in the nltk library are downloaded and the Tokenizer, Stemmer and Lemmatizer libraries are also downloaded.
  - Various steps like removal of non-ASCII words, changing to lowercase, punctuation removal, replacing numbers with corresponding words, stopword removal, stemming and lemmatization of words is done using separate functions for each.
  - The final output is saved as a csv file. This will help in compression of the file and requires less bandwidth for upload and download of the same.

- **TFIDF Logistic Regression.**

  The first task is to build the vocabulary for the words in the sentences. For this, both the premise and the hypothesis were concatenated into a single column, and the transformation is done on this.

  - The vocabulary is built using the **CountVectorizer** library in sklearn, and the default parameters were chosen for this. The new concatenated column was used to fit the vectorizer and build the vocabulary.
  - The two columns, ie, premise and hypothesis are then transformed into vectors by the transform operation using the earlier fit model.
  - Then **tfidfTransformer** is imported and a fit_transform is done for both the columns separately. This can be done separately as the vocabulary has been derived from the concatenated column, and TFIDF is in turn a simple normalization technique.
  - The final resultant vectors for all the sentences will be the same, thus bringing in uniformity.
  - Various methods like element-wise multiplication, addition, and subtraction of the premise and the hypothesis vectors were carried out to generate the final training vector. The results of the same have been reported.
  - Parameters like maximum iterations, learning rate and the tolerance were varied and checked.

- Finally the vectorizers and the tfidftransformer are saved in the form of pickle files, which can be loaded for vectorizing the test dataset.

- **LSTM and BiLSTM.**

- The dataset for train, test and validation was obtained from torchtext.datasets. This provides a wide range of functionality as the iterators can be built with just one line of code. Tokenizing is also done with spacy, the data.Field accepts this as a parameter.
- The vocabulary is built using the build_vocab function in torchtext, and this returns the sentence converted into a matrix.
- The model has an embedding layer, followed by a Relu and a dropout layer, which was added to prevent overfitting. This is followed by an LSTM module with 3 hidden layers. The output from the LSTM was fed into a sequential dense network with three hidden layers for the classification task.
- The encoding for the premise and the hypothesis was generated separately and then finally concatenated and fed into the dense network for classification.
- Parameters like batch size of the dataset, the embedding dimension, epochs for training were varied.
- Also a Bi-directional LSTM was also trained by changing the bidirectional parameter in the LSTM layer. The parameters were varied for the same and tested.
- The optimizers used were Adam, RMSProp and Adagrad. Others were not tried out as in the previous projects, the best results came out of these two.
- Learning rates were also varied and the results have been reported.
- The loss function used was cross entropy, and the learning curves were plotted for every trial and some of them have been shown in the observations below.
- Pre-trained embeddings like GloVE could also have been used in the model, and the requres_grad parameter set to False. But this requires good bandwidth for downloading and uploading the vectors every time for the trials. In this project the initial embeddings were generated by the vocabulary generator in the torchtext library.
- As the training progresses, the embedding weights for the input vectors are learned. The hope is that, say if the premise and the hypothesis are

entailed, then the embeddings get close together, and vice-versa for contradiction.

- The two vectors of the premise and the hypothesis are concatenated and then passed into the classification dense network Other methods like addition and subtraction will lead to skewed and sparse vectors respectively for entailed sentences. Thus, keeping in mind that sparse vectors would not help the network to learn, concatenation was used here.

4. Results and Observations.

➢ **TFIDF Logistic Regression**

- **Trial 1.**

Parameter perturbed: **Pre-Processing Done or not**
Preparation of training vectors: Subtraction
Max Iterations: 300
Tolerance: 0.001

Processing Done/Not	Test Accuracy
No	**57.82**
Yes	53.45

We can see that, when trained on the dataset without preprocessing, we are getting better results, by about 4%. For the subsequent trials, we have changed the preparation of the training vectors.

- **Trial 2.**

Parameter perturbed: Preparation of the vectors
Pre-processing: **Yes**
Max Iterations: 300
Tolerance: 0.001

Preparation of Vectors	Test Accuracy
Addition	43.48
Subtraction	**53.45**
Multiplication	32.4

We can see that subtraction of the premise and the hypothesis vectors gives the best results.

- **Trial 3.**
  Parameter perturbed: Preparation of the vectors
  Pre-processing: **No**
  Max Iterations: 300
  Tolerance: 0.001

Preparation of Vectors	**Test Accuracy**
Addition	50.46
Subtraction	**57.82**
Multiplication	32.37

  It can be noted that, in case of multiplication, there is no significant improvement in the test accuracy. In case of preparation of vectors by addition and subtraction, an improvement of 7% and 4% is seen respectively.
  Thus it is noticed that, with no processing, it leads to a significant improvement in accuracy.

- **Trial 4.**
  The parameters L1 ratio and penalty parameters in the scikit learn parameters were changed. But it was observed that upon giving an L2 penalty, the model never converged even with a maximum epochs of 500. Also with no L1 ratio and a tolerance of 0.001, the model converged at around 260 epochs (2.5 minutes).

- **Observations.**
  With no preprocessing, we get better results. Subtraction of the vectors of premise and hypothesis gave the best results compared to addition and multiplication.
  So for the next deep neural network, **we have chosen to not do preprocessing and just tokenize using Spacy**.

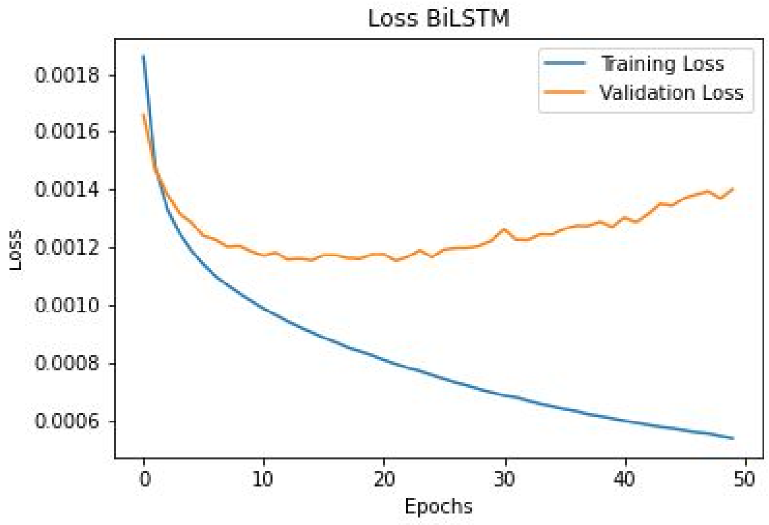
Changing the learning rate parameters led to non convergence of the model. This says that the data is very sensitive to perturbations.

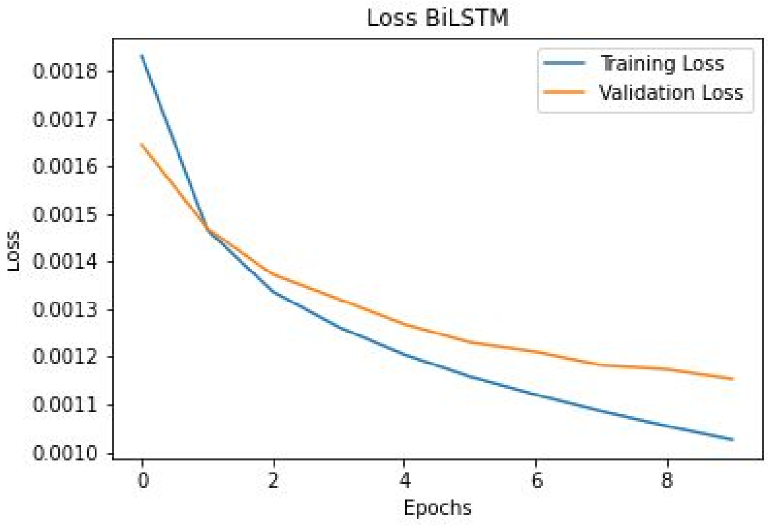
- ➢ **LSTM Model**

- ○ **Trial 1. (LSTM - Optimal Epochs)**

The LSTM model was trained for 50 epochs, and the learning curves for both training and validation was plotted. This was done to inspect if the model was overfitting in any manner, and the results were interesting.

Below we can see two plots, one when the model was trained for 50 epochs, and the other for 10 epochs.

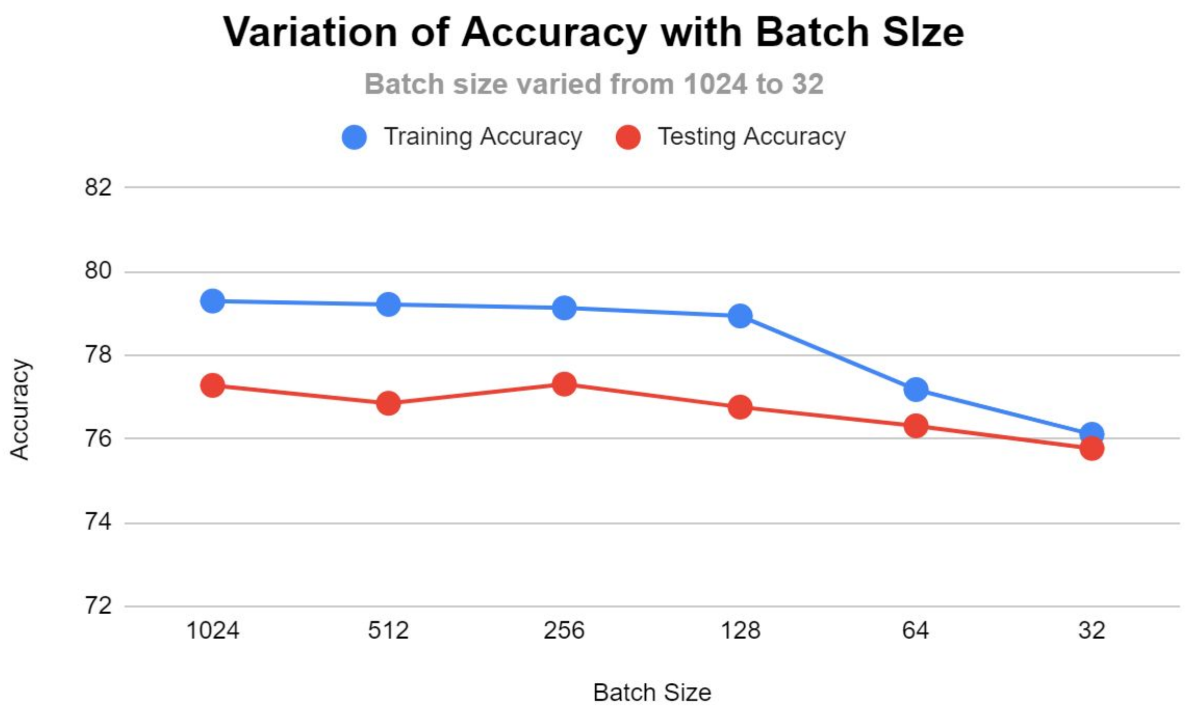




Till the 10th epoch, we can see the validation loss curve follows the training loss. After this the validation loss starts rising, indicating overfitting. Thus, the optimal epoch for subsequent training was fixed at 10 for LSTMs.

- **Trial 2. (LSTM - Optimal Batch size)**
  Parameter Perturbed: Batch Size of the training data
  Epochs: 10
  Embedding dim: 150
  Dropout ratio: 0.2
  Hidden layer dim: 256
  Learning rate: 0.001
  Bidirectional: False

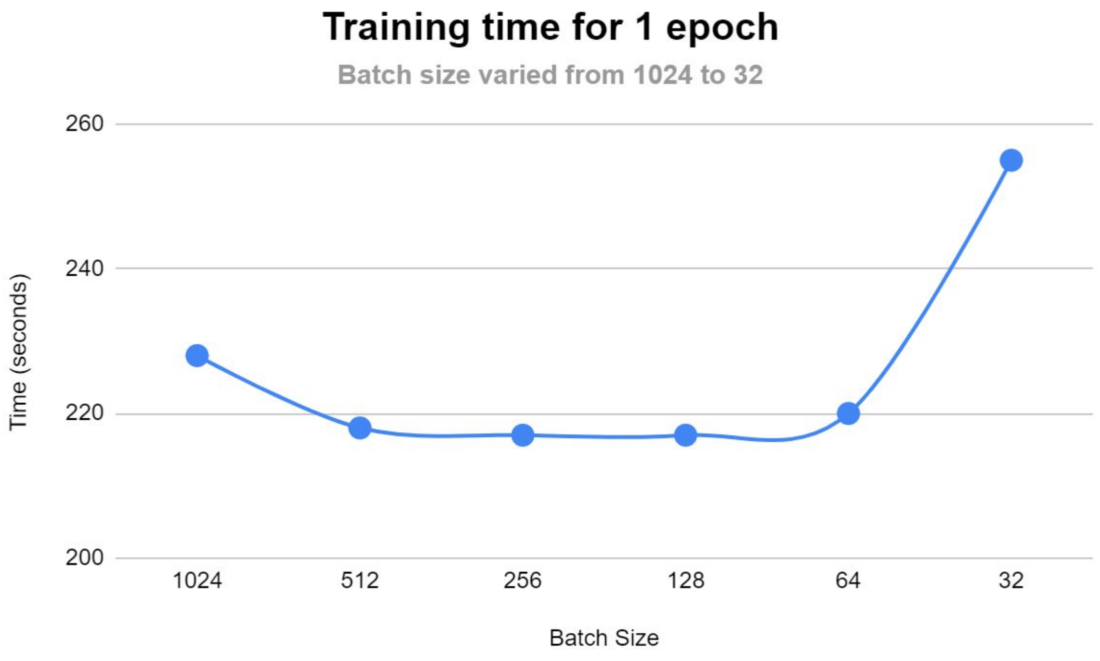
  The following graph shows the variation of the batch size with respect to the training and the testing accuracies.



  We can see from the graph that batch size of 1024 produces the best accuracies. It constantly keeps decreasing as the batch size decreases. The line goes more steeper as the batch size decreases.

  Below we can see the training time for one epoch for different batch sizes.

The optimal batch size with regards to training time seems to be 256 or 128.Batch sizes of 64 and 32 show rise in the training time.

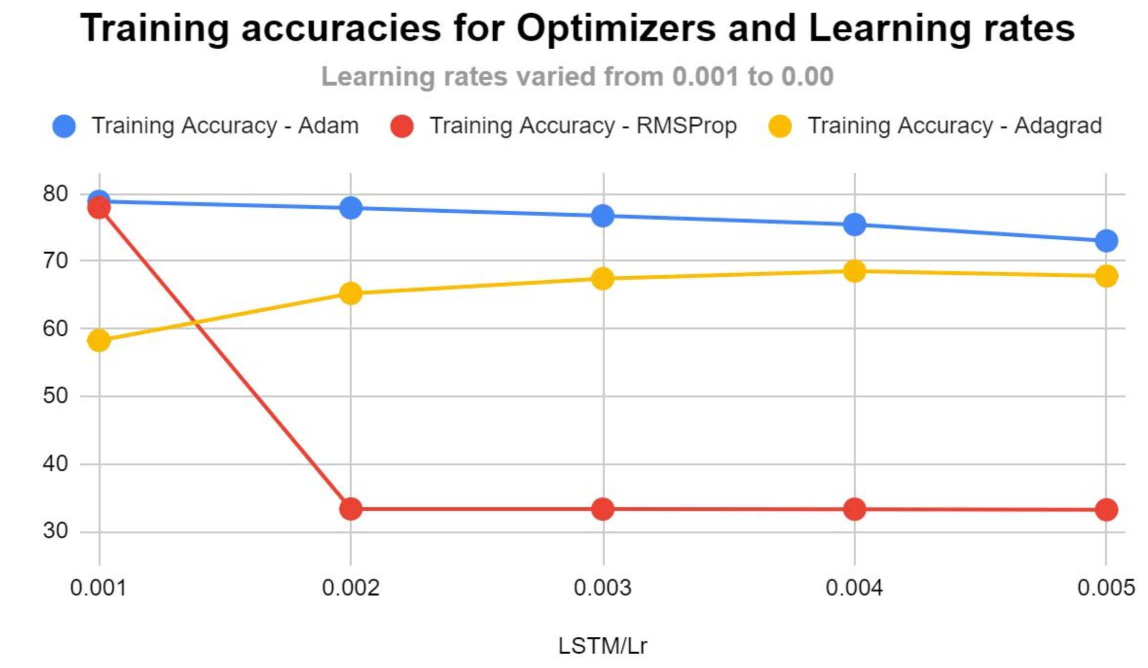


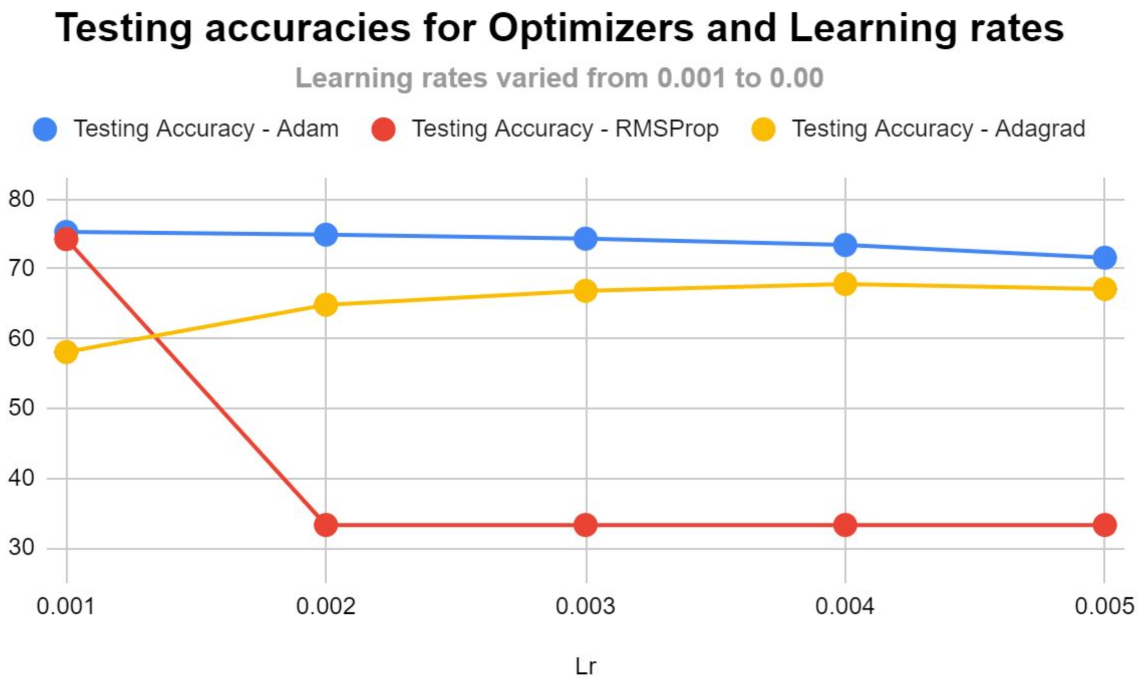
On the whole, we can see that a batch size of 1024 or 512 is the most appropriate with regards to accuracy and training time (slightly more than 256 and 128).

- **Trial 3. (LSTM - Optimal Learning Rate and Optimizer)**
  Three optimizers were tried out: Adam, RMSProp and Adagrad. The learning rates were varied from 0.001 to 005, in steps of 0.001. The results for the trials are reported below in the form of plots (next page).

  We can observe that Adam gives the best results with learning rate as 0.001. The training time for all three optimizers was the same at around 200 seconds.
  That is not the case with RMSProp. At a learning rate of 0.001, it gives a slightly lower accuracy than Adam. After that, the accuracy is stuck at around the lower 33%. This says that more fine tuning is required in case of RMSProp, and lower learning rates might work out better.
  In the case of Adagrad, a learning rate of 0.004 seems to work the best, with the curve tapering lower on both sides of 0.004.

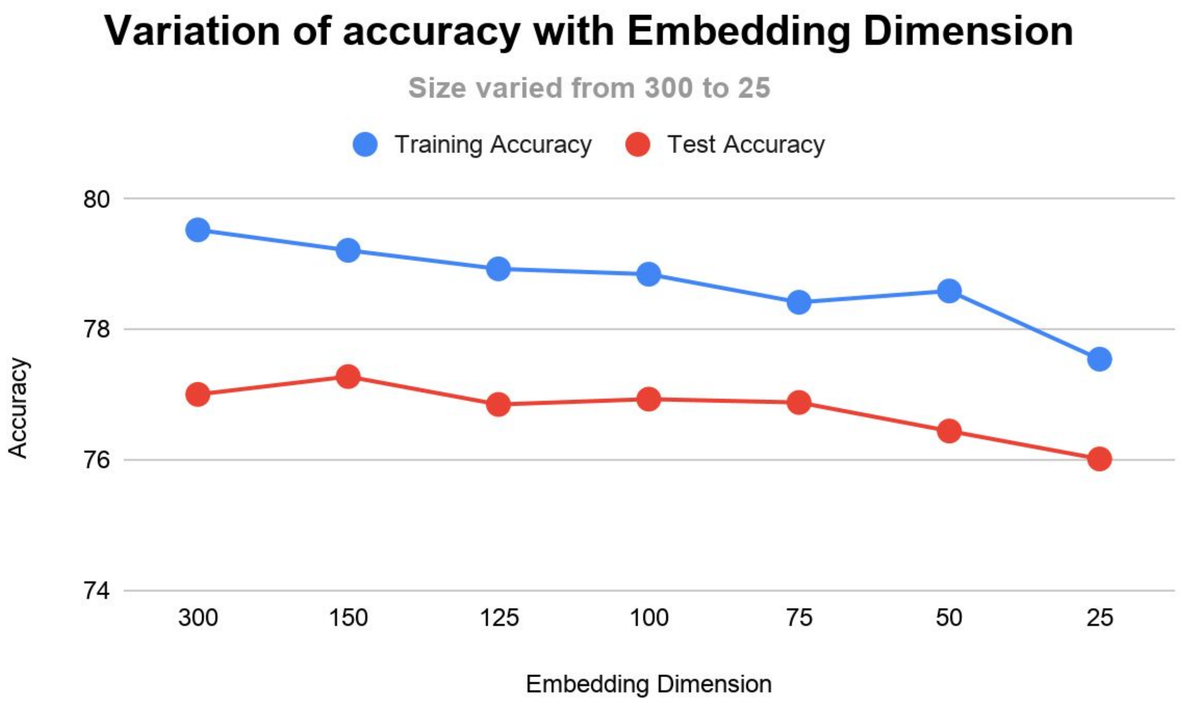
Overall, we can see Adam gives consistent results as seen in the previous projects, and accommodates well for higher learning rates as well. Adam with lr = 0.001 has been chosen as the optimizer for further trials.





- **Trial 4. (LSTM - Optimal Embedding dimension)**
  Embedding dimension was varied from 300 to 25, and the following graph reports the results for training and testing accuracies of the same.

  An interesting thing to note is that the training accuracy decreases as the embedding dimension decreases. The highest testing accuracy occurs for an embedding dimension of 150. After this, both accuracies start decreasing.



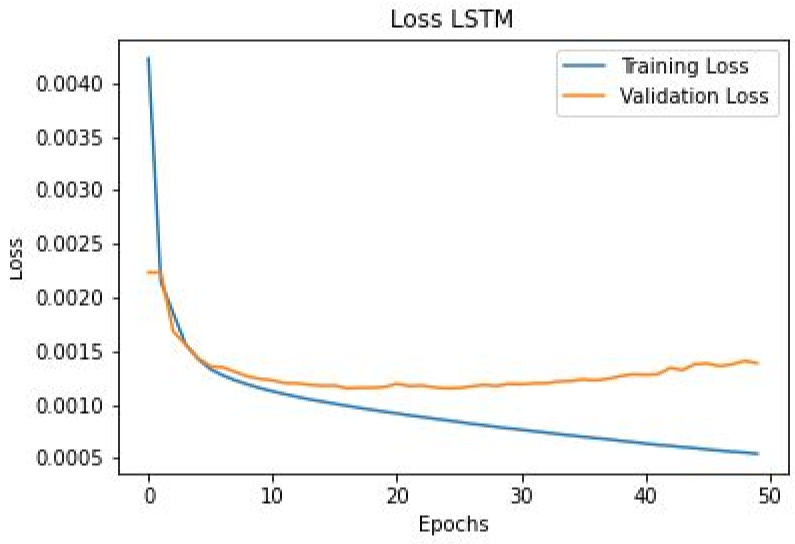
Thus, an embedding dimension of 150 has been chosen for the final model.

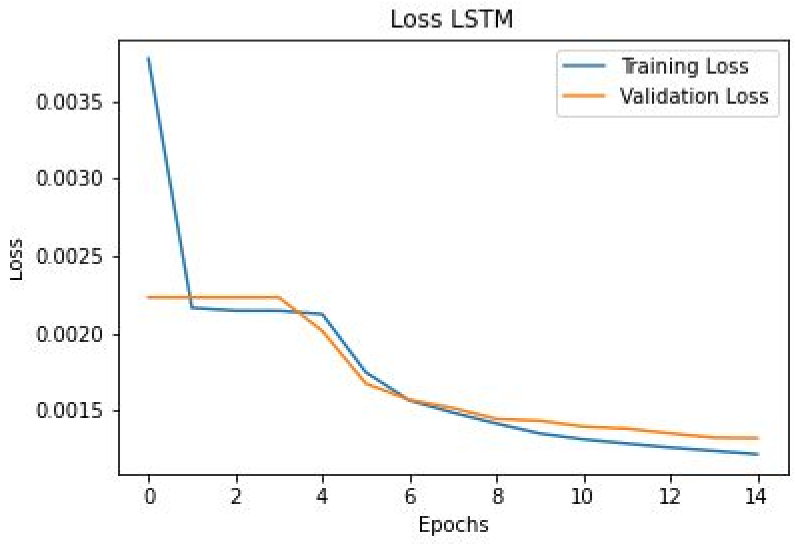
➢ **BiLSTM Model**

○ **Trial 1. (BiLSTM - Optimal Epochs)**

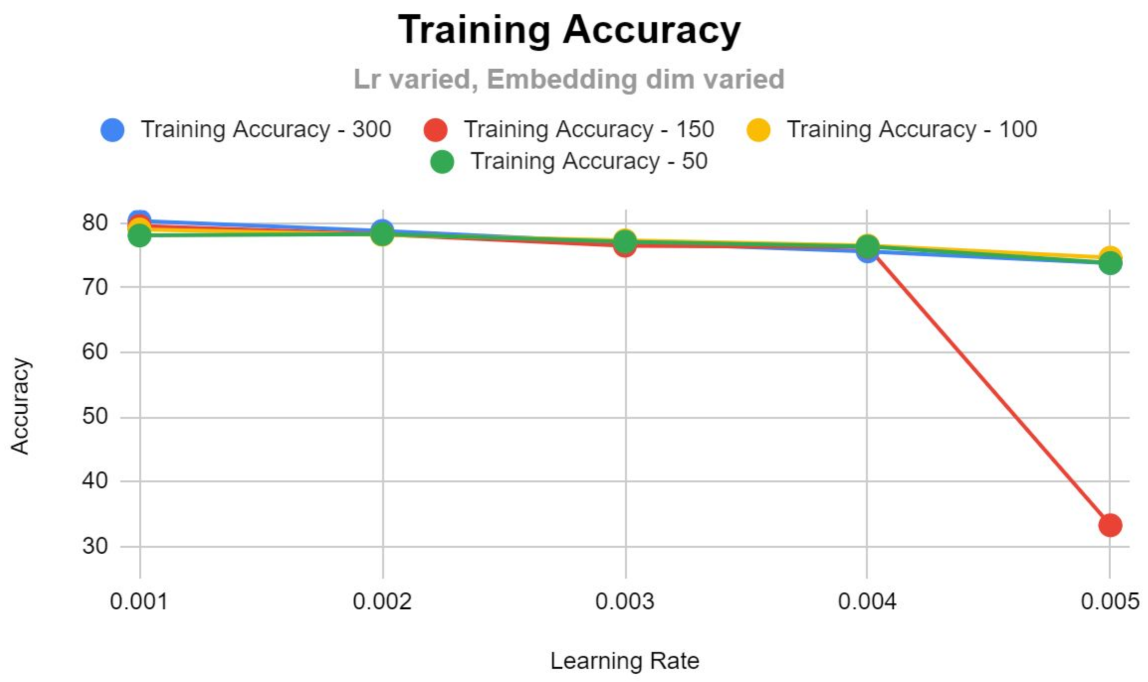
The same method as done with LSTM was chosen, the model was trained for 50 epochs and the learning curves were plotted.

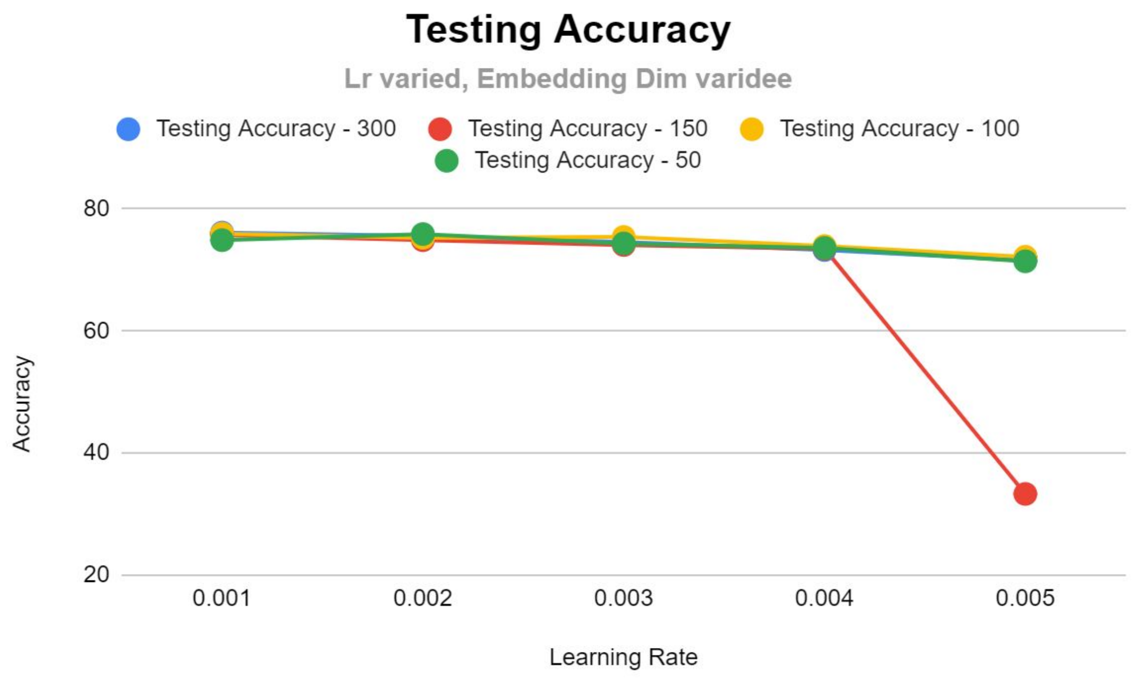
From this it was found that the validation loss curve starts deviating at around 15 epochs, indicating overfitting. The curve for the model trained with 15 epochs has also been reported below.





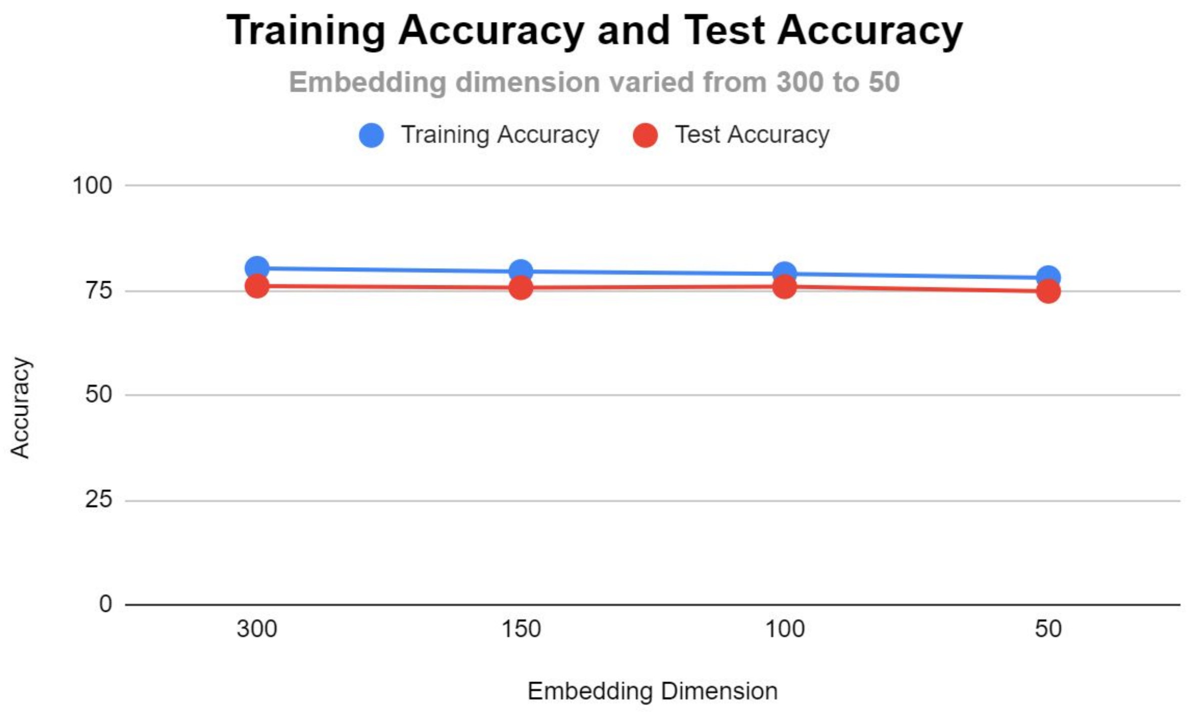
- **Trial 2. (BiLSTM - Optimal Learning Rate and Optimizer)**
  Based on the results from LSTM, we are only using Adam as the optimizer here. The learning rate varied from 0.001 to 0.005 in steps of 0.001. The results have been plotted below.





One thing to notice is that, at a learning rate of 0.005 and an embedding dimension of 150, we can notice a dip in the accuracy to a low 33%. Overall, we can see that a rate of 0.001 and an embedding of size 300 works best for the BiLSTM.

- **Trial 3. (BiLSTM - Optimal Embedding dimension)**
  The embedding dimensions have been varied as 300, 150, 100, 50. This was done as the training time was almost double the ordinary LSTM. The results have been shown below.



From the plots, we can see that the embedding size of 300 is the best in terms of training and testing accuracy.

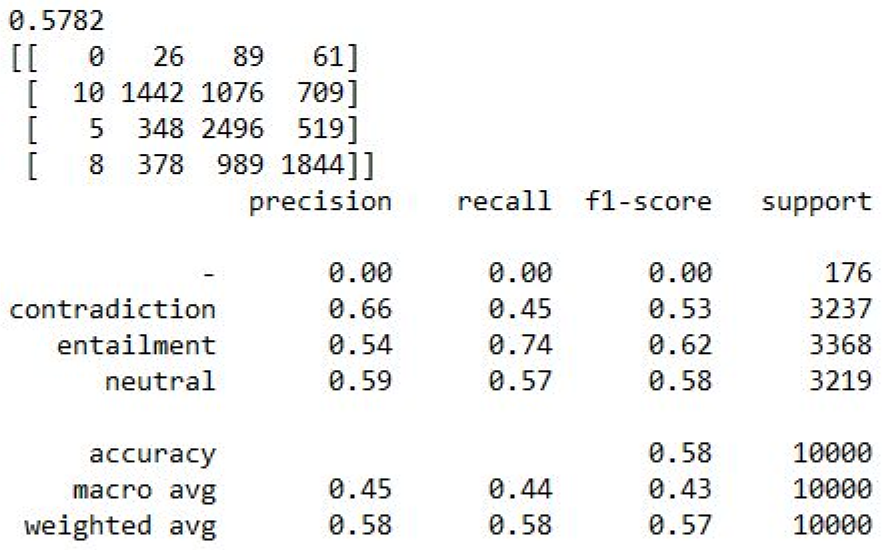
- **Observations:**
  We can see that using a BiLSTM model prevents overfitting till 15 epochs, and the best optimizer is found to be Adam with a learning rate of 0.001. The optimal embedding dimension for LSTM and the BiLSTM models were found to be 150 and 300 respectively. Batch size tests was not carried out with BiLSTM, but the results should not vary that much from LSTM, and it was chosen to be 512.

5. **Conclusions.**
    - **TFIDF Logistic Regression Model.**

The following model with the parameters listed below has been chosen finally for The testing. The results from the same will get written into a .txt file.

- Pre-processing: No
- Max Iterations: 300 (Converged: 266)
- L1 ratio: None
- Tolerance: 0.001
- Penalty: L2
- Vector preparation: Subtraction of premise and hypothesis

Testing Accuracy: 57.82
Confusion Matrix:



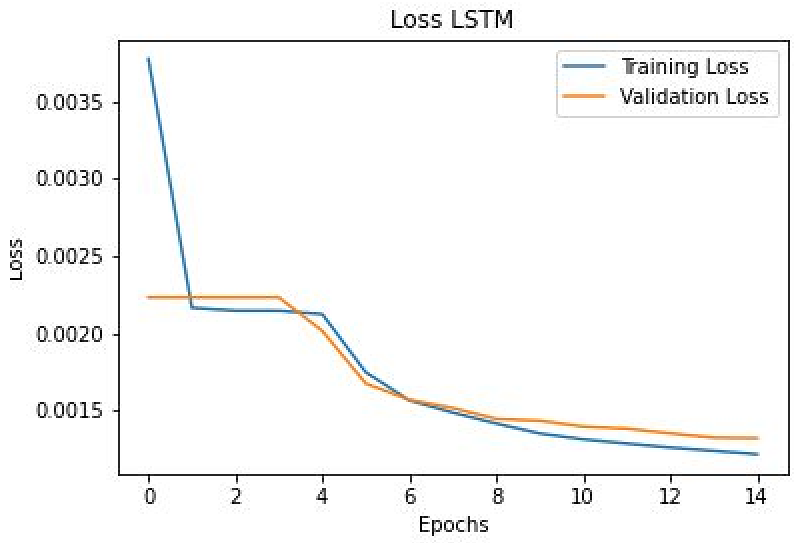
```
0.5782
[[   0   26   89   61]
 [  10 1442 1076  709]
 [   5  348 2496  519]
 [   8  378  989 1844]]
               precision    recall  f1-score   support

            -       0.00      0.00      0.00       176
contradiction       0.66      0.45      0.53      3237
   entailment       0.54      0.74      0.62      3368
      neutral       0.59      0.57      0.58      3219

     accuracy                           0.58     10000
    macro avg       0.45      0.44      0.43     10000
 weighted avg       0.58      0.58      0.57     10000
```

The precision of all the classes (contradiction, entailment and neutral is around the 60% mark. But the recall for entailment is the highest, making the F1-score for entailment as 62%. Thus the LG model is most accurate in predicting entailment. As we are using subtraction in the preparation of final training vectors, this means that, if the premise and hypothesis are entailed, this should ideally lead to a sparse vector, whereas in the case of a contradiction, it should lead to a non-sparse vector. But this will also be the case in neutral class. This makes the end classification a bit hard.
Overall, we get an accuracy of 57.82%, which is not so bad considering that we are directly using the TFIDF vectors, and not learning them.

- **Bidirectional LSTM Model.**
    - Batch Size: 512
    - Embedding size: 300
    - Dropout ratio: 0.2
    - Hidden layer dimension in BiLSTM: 256
    - Optimizer and Learning rate: Adam & 0.001
    - Bidirectional: True
    - Vector Preparation: Concatenation

Testing Accuracy: 75.5904

The following is the learning curve for the model, trained for 15 epochs, on the train dataset, validated on the validation dataset.



We can see that the training and the validation curve follows the same trend, indicating no overfitting. The final loss after training was found out to be 0.001253 and the training and validation accuracy came out to be 80.3326 and 76.11257 respectively. The final test accuracy was reported as 75.5904.

It is also observed that the DNN performs much better than the Logistic Regression model. This result also supports the fact that we learn much better and important features in a deep network as compared to conventional techniques.

6. References.

- https://torchtext.readthedocs.io/en/latest/#
- hps://nlp.stanford.edu/projects/snli/tt
- https://github.com/pytorch/examples/tree/master/snli
- https://pytorchnlp.readthedocs.io/en/latest/_modules/torchnlp/datasets/snli.html