Assignment 3

Introduction:

In this assignment, we compare the performance of 2 approaches for natural language inference. Natural language inference is basically a classification problem in which we are given 2 sentences and we have to predict if the sentence pair constitutes entailment / contradiction / neutral. SNLI(Stanford Natural Language Inference) dataset is used in this assignment.

There are 3 major parts for this assignment. These are:

1) Preprocessing
2) Features extraction
3) Classification

Preprocessing is the same for both approaches. We will be using tf_idf features for the first approach and deep features for the second approach. Classifier used in the first approach is logistic regression and neural network in the second approach. Details about each of these steps is provided below. We use accuracy as a metric for comparison of these 2 approaches. Test set contains 9,824 test examples from three classes. This dataset contains examples in almost equal proportion, so accuracy is a good metric for comparing.

Preprocessing:

It is one of the major steps of this assignment. I am using the spaCy library for pre-processing. It takes care of many steps which are required for this assignment like conversion to lowercase, stemming, tokenization etc. There were also sentences which did not have any labels. Those sentences were removed from the dataset. Code file for pre-processing the data is provided. Preprocessed file obtained after this was used for the tf-idf model.

Approach 1 : TF-IDF

TF-IDF stands for term frequency- inverse document frequency. Term frequency is referred to as the number to times a word appears in a document(sentence in this dataset). Term frequency alone is not a good feature, as words like ’a’, ‘the’, ‘of’ etc. are present a lot more times and in every document than other words which are more informative. So we also use inverse document frequency to account for this. IDF is the log term in the formula below. It is log of the ratio of the total number of documents and the total number of documents containing word i. Weight of a word i in document j is calculated using the formula below:

$$W_{i,j} = tf_{i,j} * \log(\frac{N}{df_i})$$

$$tf_{i,j} = number\ of\ occurences\ of\ term\ i\ in\ document\ j$$

$$df_i = number\ of\ documents\ containing\ term\ i$$

$$N = total\ number\ of\ documents$$

For the words like ‘the’, ‘a’ etc, weight will be less as the IDF term will be close to 0 because these words are present in almost all the documents so the ratio inside the log will be close to 1.

We can calculate the features for each sentence using the formula above. I am also removing the stop words before calculating TF-IDF because stop words do not carry important information for classification. I am using bi-gram of the words along with the original words and using TF-IDF to convert these to feature vectors. Using bigram helps us to model some dependency(i.e. We are not considering occurence of all the words independent of others and we will be able to model the dependence of the current word given previous word). After getting TF-IDF feature vectors for both the sentences, I concatenate them and pass it to the classifier. These vectors are then used to learn weights for logistic regression. Many choices of hyperparameter were tried and final hyperparameters selected were based on performance on the validation set. Final hyperparameters used for model are:

Hyperparameter names	Value or Method
min_df	5
max_df	0.99
ngram	(1,2)
Penalty for logistic regression	L2

Min_df=5 means that if the word is not in atleast 5 documents(sentences) then it will not be included in vocabulary. This is done because those words are probably misspelled and it decreases the size of vocabulary. Max_df=0.99 means that if word is present in more than 0.99*total_documents then it is not included in vocabulary.

Accuracy: 68 percent

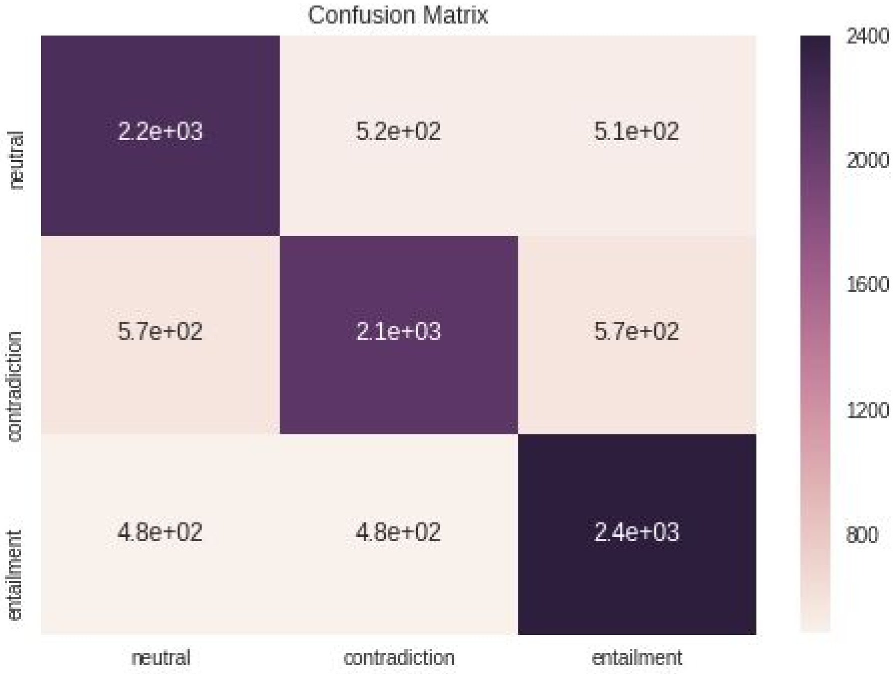


Fig. Confusion Matrix for TF-IDF

From the confusion matrix above, we can see that we make most errors on the contradiction examples. But errors made in each class are pretty much balanced and our classifier is not skewed.

Approach 2: Deep Model

In this approach, deep learning models are used for generating features and classifying. Preprocessed text is converted to one-hot vectors, and these vectors are passed to the model to create a feature vector that represents information about both sentences which is then used for classification. Mentioned steps are shown in the figure below.

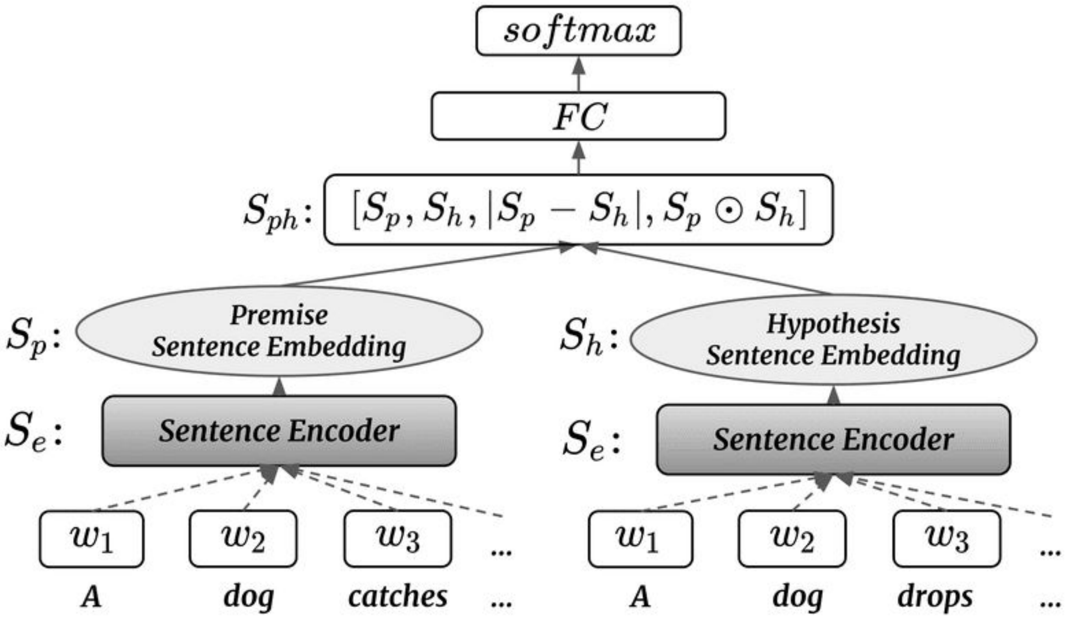


Fig 1) High-level architecture

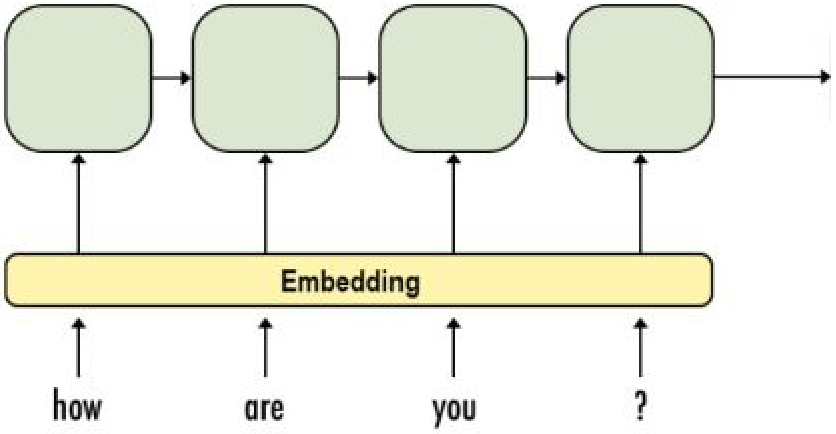
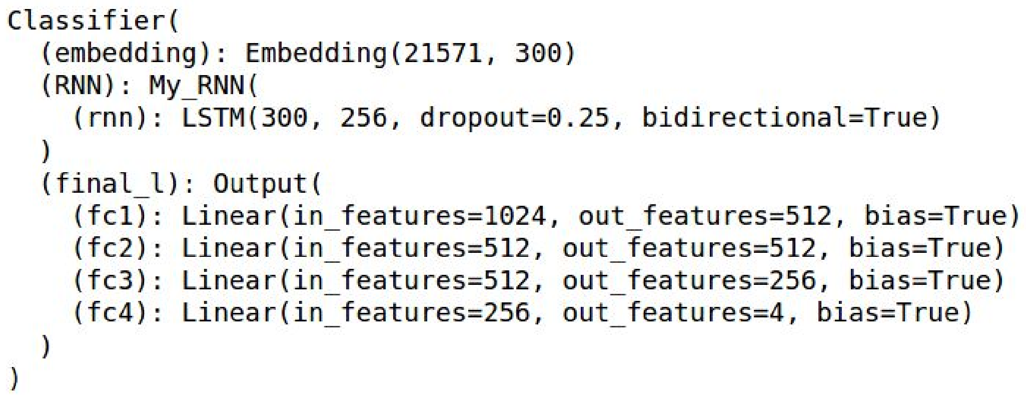


Fig 2) Model to get sentence embedding

I am using recurrent neural network for creating sentence embedding because they have fixed hidden layer size and hidden layer in RNN captures all of the information regarding previous inputs required for classification. Also weights of RNN are shared for all time steps which results in statistical efficiency. Figure 2 shows how words are converted to sentence embeddings. Each word is a one-hot vector, which is passed through an embedding layer which is just a lower dimensional representation of the word. Embedding layer is like a dictionary i.e. when we multiply one-hot vector to this layer weights, then we are basically selecting the corresponding column from the weight matrix. The embedding layer has the same weights for each step just like the LSTM layer. These embeddings are then passed to LSTM at each time step to get sentence embedding(which is basically the output of the hidden layer at last timestep). After getting embeddings for both the sentences, they are concatenated and passed through a fully connected neural network to get the output as shown in figure1. I will only discuss the concatenation of the 2 sentence embeddings in this assignment.

I have not used any pre-trained word embeddings(like Word2Vec) as embedding such as Word2Vec is trained based on context words and may not be great for NLI tasks. E.g. words hot and cold may appear in similar context and therefore will be very close which is not desirable for this task. One can also fine-tune the embeddings to get around these problems. I am using bidirectional LSTM for this model. LSTM because it is able to learn long-term dependencies. This is because they can selectively forget, input and output data using different gates and do not suffer from vanishing gradient problem like vanilla RNN's. And bidirectional LSTM because they can represent better sentence embedding for classification and the perform better(see model selection pdf). After passing sentences to the model I get feature vector of size 4*hidden_state_size, one factor of 2 comes from concatenation sentence embeddings and another factor of 2 comes from using bidirectional LSTM. This is then passed through 3 hidden layer neural network to produce the final output.



```
Classifier(
  (embedding): Embedding(21571, 300)
  (RNN): My_RNN(
    (rnn): LSTM(300, 256, dropout=0.25, bidirectional=True)
  )
  (final_l): Output(
    (fc1): Linear(in_features=1024, out_features=512, bias=True)
    (fc2): Linear(in_features=512, out_features=512, bias=True)
    (fc3): Linear(in_features=512, out_features=256, bias=True)
    (fc4): Linear(in_features=256, out_features=4, bias=True)
  )
)
```

Fig. Final Model

Hyperparameter name	Value or Method
Loss	Cross-Entropy
Regularization	Weight decay of 10^{-4}
Batch Size	128
Learning Rate	0.005
Optimizer	Adam
Momentum	0.9

Epochs	35
Scheduler	Step scheduler with gamma=0.55, steps=5
Dropout LSTM	0.25
Dropout FC layers	0.3
Embedding dimension	300
Hidden state dimension(LSTM)	256
Hidden layer 1 dimension(FC)	512
Hidden layer 2 dimension(FC)	512
Hidden layer 2 dimension(FC)	256
Activation function	Relu
Output Layer	Softmax

I tried some different dimensions for the hidden state of LSTM, increasing the dimension did not increase accuracy much, this may be due to the hidden state dimension of 256 being able to capture all the useful information. I have also provided weights for network with a hidden state dimension of 350 which performs better(accuracy of 78.1 percent) than 256 dimension model but is slower. For the rest of the assignment I will talk about the 256 dimension model. Fully connected layer has 3 layers, because the models with lesser layers performed poorly. Dimension of layers and number of layers are set using accuracy on validation set as metric.

I also tried 3 different values for the embedding size which were 200, 250 and 300, but the embedding size of 300 worked best for this dataset. I also tried different batch sizes from 64 to 256, increasing it by a size of 2 each time. There was not much difference in the accuracy, but training was faster for larger batch sizes. This is because I am using BucketIterator in pytorch as this arranges the examples of similar length together and results in better parallel computation. So I am using the final batch size of 128 so that there are more weight updates than batch size of 256 in one epoch and batch size 128 is little faster than batch size 64.

Difference between the train and validation loss was a lot ,so dropout was added to prevent it from overfitting.

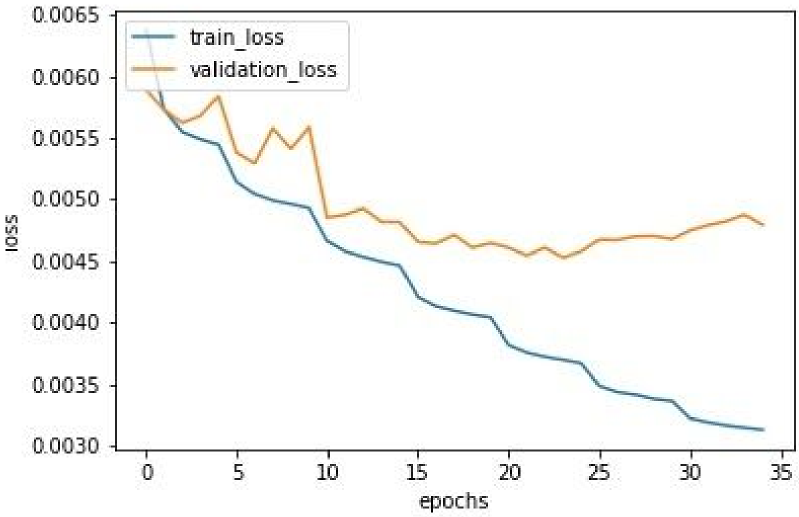


Fig 3) Training and Validation Loss vs Epochs

We can clearly see from figure 3 that model overfits after 35 epochs. So I am using early stopping criteria and saving weights for which model has best performance on the validation set. Increasing the regularization(e.g. Dropout, L2 penalty) can also be tried to prevent overfitting. Learning rate for this model is chosen(among 4 predetermined ones) by performance on a validation set after the first 3 epochs. Due to this, the curve starts with less loss on the y-axis, as I am getting accuracy of 67 percent after the first epoch. Also using the learning rate scheduler helps to model to learn faster. Without using learning rate scheduler, accuracy would remain around 74 percent and would not increase.

Accuracy: 77.6 percent

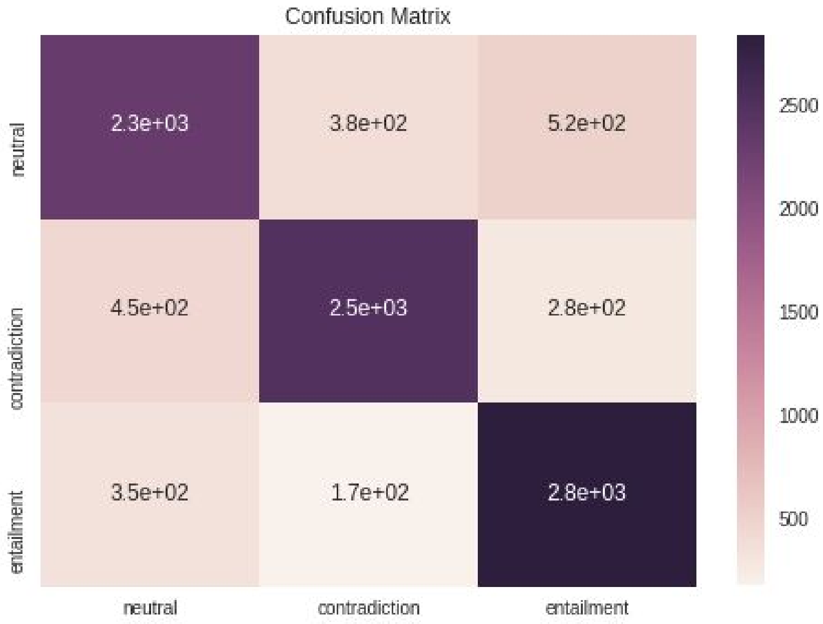


Fig. Confusion Matrix for Deep Model

As we can see from the confusion matrix above, the model struggles most with neutral sentences and classifies them as entailment many times.

Adding Attention:

Attention was also added to the model according to “Text Classification Research with Attention-based Recurrent Neural Networks” paper. This type of attention is very different from the one that is used in encoder-decoder model.

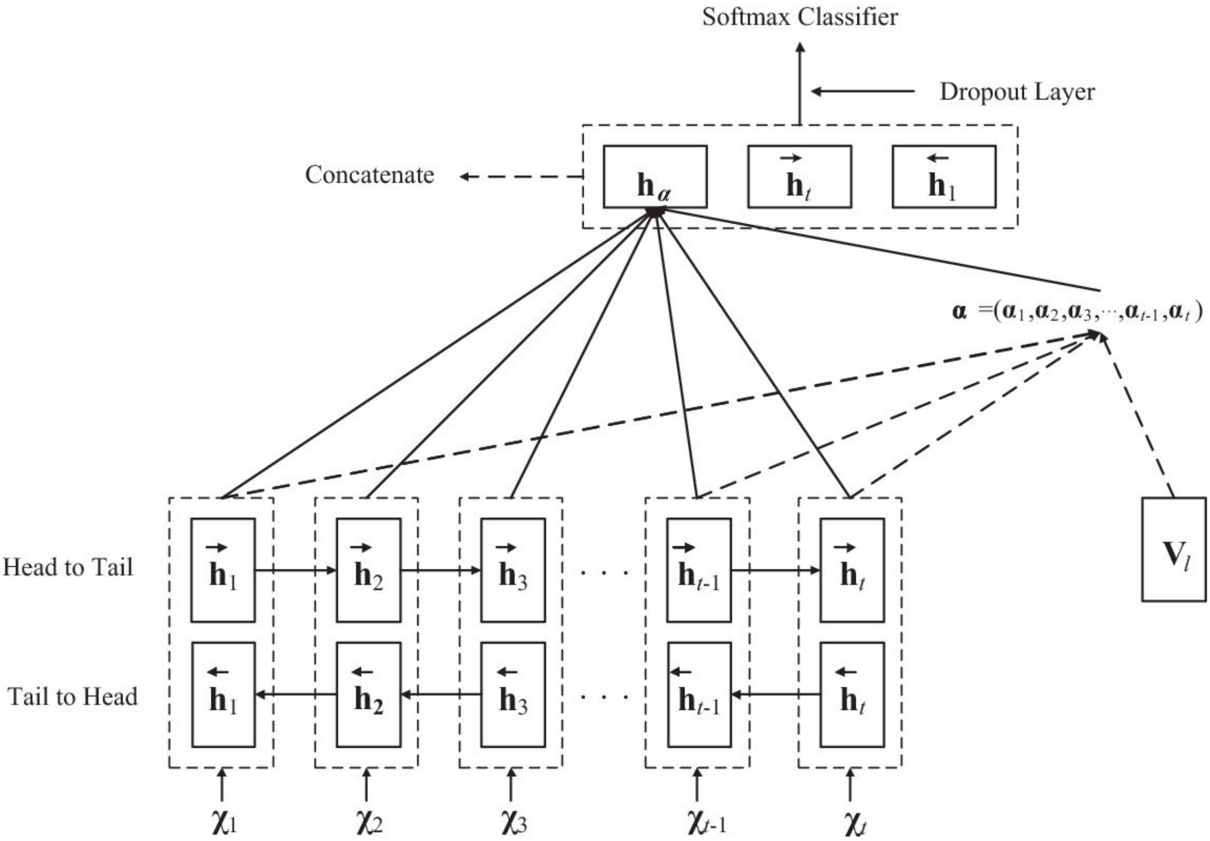

Softmax Classifier
Dropout Layer
Concatenate
$\mathbf{h}_\alpha$
$\overrightarrow{\mathbf{h}}_t$
$\overleftarrow{\mathbf{h}}_1$
$\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \alpha_3, \cdots, \alpha_{t-1}, \alpha_t)$
Head to Tail
Tail to Head
$\overrightarrow{\mathbf{h}}_1$ $\overrightarrow{\mathbf{h}}_2$ $\overrightarrow{\mathbf{h}}_3$ $\overrightarrow{\mathbf{h}}_{t-1}$ $\overrightarrow{\mathbf{h}}_t$
$\overleftarrow{\mathbf{h}}_1$ $\overleftarrow{\mathbf{h}}_2$ $\overleftarrow{\mathbf{h}}_3$ $\overleftarrow{\mathbf{h}}_{t-1}$ $\overleftarrow{\mathbf{h}}_t$
$\mathbf{V}_l$
χ_1 χ_2 χ_3 χ_{t-1} χ_t

Fig) Attention Mechanism

As we can see from the figure, we concatenate the attention feature vector along with the last(time-wise) forward-hidden state and backward hidden state before passing it to the fully connected layer for classification. Attention weights are calculated by using batch matrix multiplication of the unsqueezed last(time-wise) hidden states with the output(which is the last(depth-wise) hidden-state of LSTM at each time step) of the LSTM outputs. For details, look at code which is easier to understand for more details. Then these attention weights are multiplied with the hidden states and added to make the attention feature vector which is h_alpha in the figure.

This procedure lets the network to look at the part hidden state at each time step to give output. This makes the network very computationally inefficient. This attention layer does not have any trainable parameters. This also does not increase the accuracy of the network, this may be because the last hidden state contains enough information which is necessary for classifying unlike in the seq-to-seq model which has to output a complete sequence which is hard using just the hidden state of the last layer.

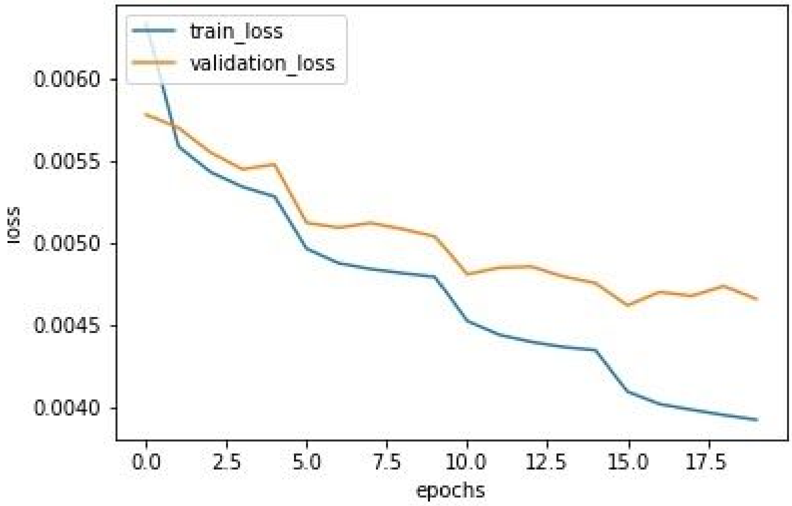


Fig) Training and Validation Loss for model with attention

Accuracy: 76.08 percent after 20 epochs

Conclusion:

We can see that deep model perform much better than the TF-IDF approach because deep models are able to get good representation of sentences using bidirectional LSTM and can learn more sophisticated functions for classification due to the 3-layer fully connected network. But the TF-IDF approach is a good starting point and requires much less computation power. TF-IDF approach may even be better if we have less data.

Experiments:

Many experiments were performed for selecting various hyperparameters. Main results of these experiments are mentioned above. This section contains some more information regarding how some of the model hyperparameters are selected and experiments performed for hyper-parameter selection. For many experiments, just the validation accuracy is available as I was **not saving** train and validation loss curves for **initial experiments**. Training for experiments mentioned below are done only for 15 epochs and accuracy on validation set is calculated, if the accuracy after 15 epochs were greater or comparable to accuracy of final model after 15 epochs on validation set then model was trained for longer. These experiments were only performed once and therefore do not account for variance in accuracy. Also the hyperparameters are considered orthogonal, which is not true (e.g. when increasing the number of layers, we should also increase dropout probability) .So the result could be improved by tuning all the parameters together. Learning rate, batch size for learning these models were fixed and not tuned for particular models. Details for selecting the learning rate, batch size are already mentioned above.

Embedding Size: 3 values of hyper-parameter were tested.

Size	Accuracy
200	76.1
250	75.83
300	76.32

We can see that embedding size of 250 has worse performance which is not justified, this is because these experiments were only performed once and there is variance in accuracy therefore embedding size 200 has better performance than embedding size 250.

Hidden State dimension: 4 values of hyper-parameter were tested.

Size	Accuracy
200	75.56
256	76.32
300	75.19
350	76.61

Hidden State of 350 performs best, but the final difference between model with hidden state dimension of 256 and model with hidden state dimension of 350 was very less, so model with hidden state dimension 256 was chosen.

Hidden layers in Fully connected network: 3 values of hyper-parameter was tested.

Size	Accuracy
2	75.07
3	76.32
4	75.14

We see that accuracy decreases using 4 hidden layers, but this may be due to overfitting and could probably be avoided by increasing dropout probability.

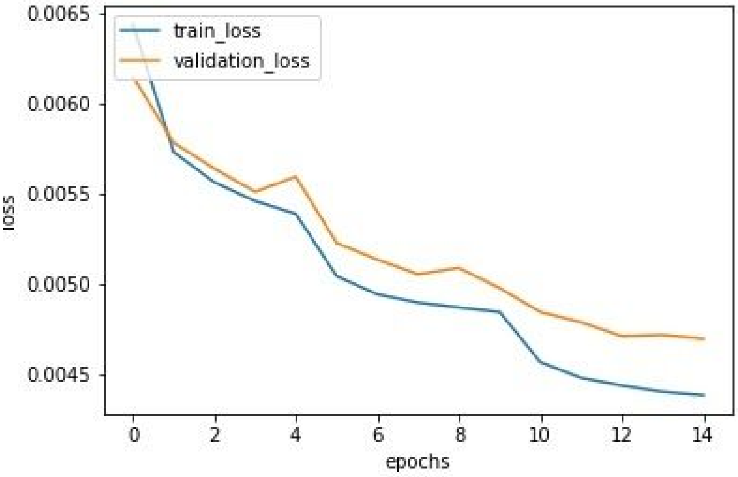
Learning rate, Batch size : Details are mentioned above.

LSTM or Bi-LSTM:

Method	Accuracy
LSTM	74.15
Bi-LSTM	76.32

We can see that Bi-LSTM works better than LSTM which is expected.

Train and Validation Loss graphs for some experiments:



Fig, Embedding dimension 200

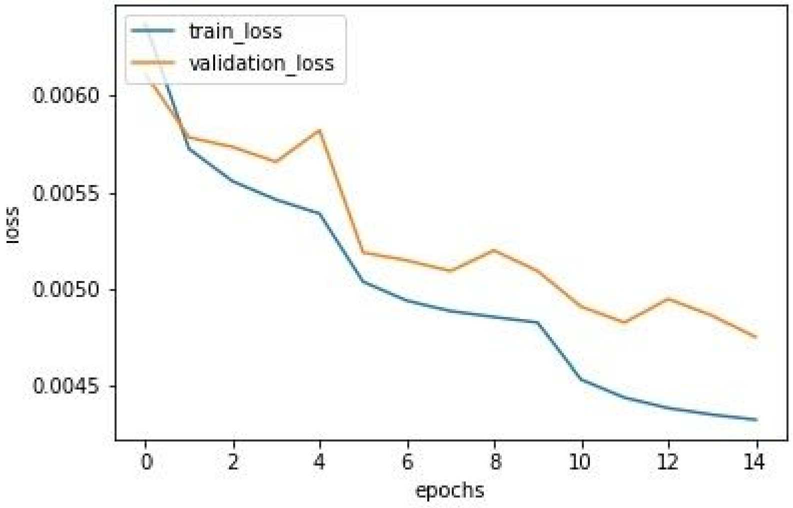


Fig. Embedding dimension 250

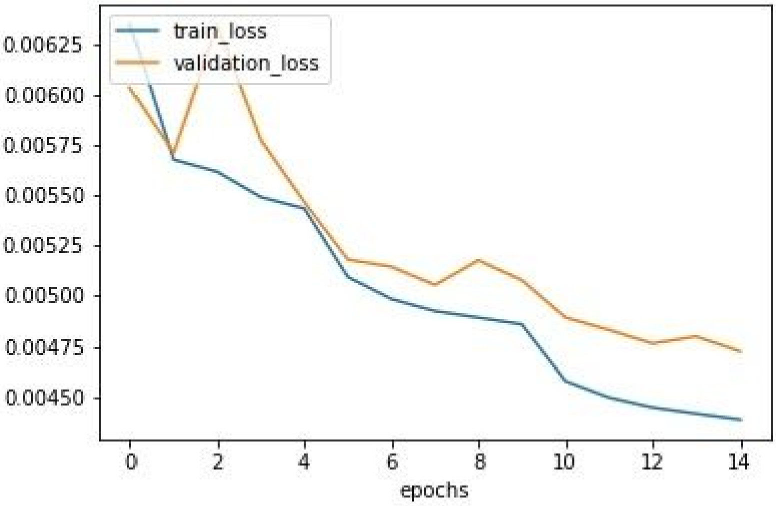


Fig. Hidden state dimension 200

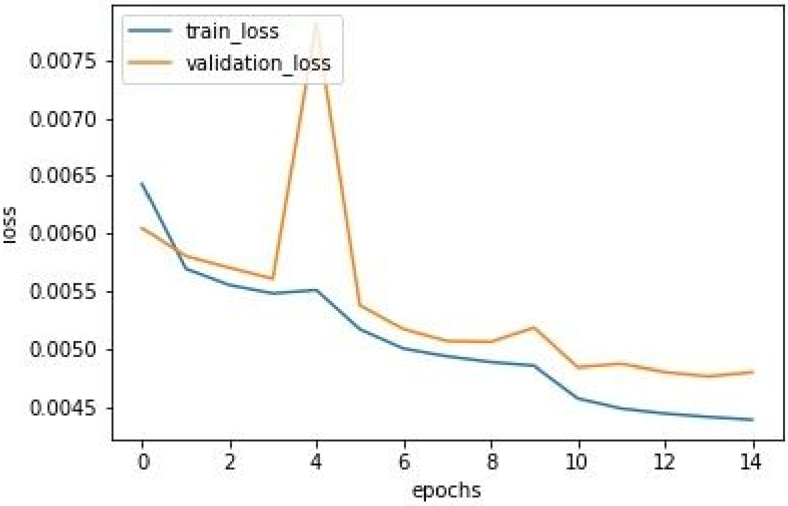


Fig. Hidden state dimension 300

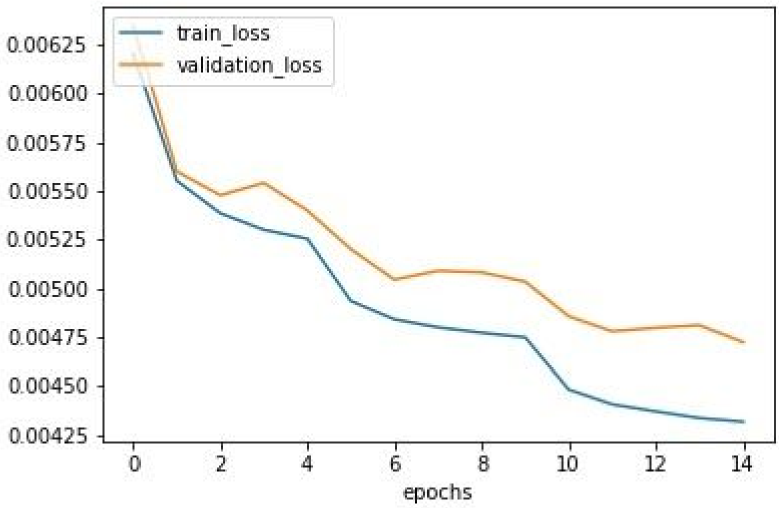


Fig. Hidden layer=2 in FC layers

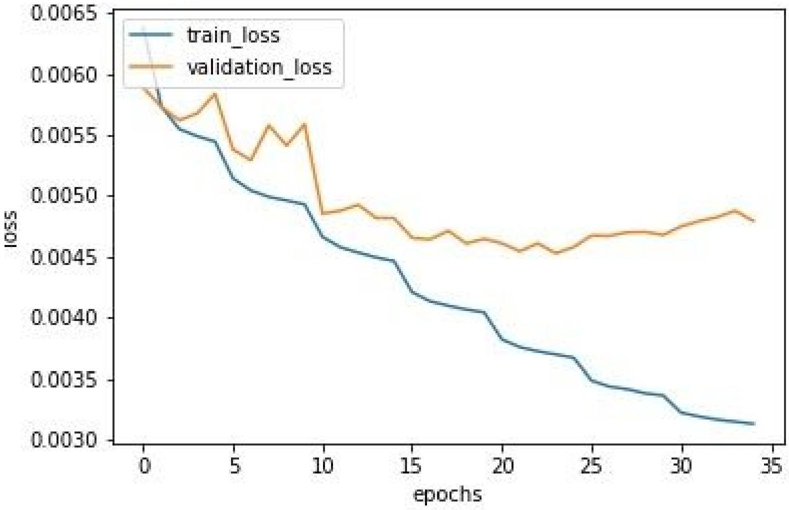


Fig. Hidden layer=3, Embedding dim=300,
Hidden state dim=256