CSA 250 Deep Learning

Assignment 3

Python Dependencies and Requirements

- python (v3.7.4)
- tensorflow (v2.1)
- Other libraries used: Please see “requirements.txt”

Instructions to run

The ‘main.py’ is the interface to the program. It is programmed to run in two modes – train mode and test mode. The classifiers implemented in this program are

- Logistic regression classifier using TF-IDF features
- Deep model classifiers for text such as GRU, LSTM and SumEmbeddings
- BERT-based classifiers (Only implementation code is available. Unable to train and test due to scarcity of resources)

The ‘main.py’ file takes two optional command line argument, one to specify the mode of execution – whether to train or test model, and another one to specify the model architecture to be used. The ‘main.py’, when executed without any arguments, enters into testing the logistic regression model and the SumEmbeddings deep model, and produces the output files ‘deep_model.txt’ and ‘tfidf.txt’ respectively.

The ‘main.py’ when executed with the (optional argument) ‘--train-model’ enters into training mode and saves the models after training. The model to be used for training/testing can be specified with the (optional argument) ‘--model-name’. Model name can be one among the following: BiGRU, BiLSTM, SumEmbeddings, and BERT.

- Train mode : python main.py --train-model --model-name <model_name>
- Test mode : python main.py --model-name <model_name>

Program details

This program is an implementation of the task of Natural Language Inference (NLI). In this task, we are given two sentences called premise and hypothesis. We are supposed to determine whether the ”hypothesis” is true (entailment), false (contradiction), or undetermined (neutral) provided that the ”premise” is

true. For this project, we have used the Stanford Natural Language Inference (SNLI) dataset[1]. We use the files 'snli_1.0_train.jsonl' for training the model and 'snli_1.0_test.jsonl' for testing the model. From each entry in these files, we consider the fields corresponding to *gold_label*, *sentence1* and *sentence2*. *sentence1* serves as the premise and *sentence2* serves as the hypothesis and *gold_label* serves as the relationship label.

The program has following components:

'main.py'	This python file acts as a client file for the user to interact
input	This folder contains the raw input files and the processed input files required for the program
model	This folder contains the different saved models
results	This folder contains the plots, output file for each model trained
utils	This is a python package consisting of python files for data loading-storing and plotting purposes
TF-IDF	This is a python package consisting of python files for training and testing logistic regression model
deep_model	This is a python package consisting of python files for training and testing BiGRU, BiLSTM, SumEmbeddings, and BERT.

Results

The task of Natural Language Inference (NLI) was implemented using logistic regression and deep network models. The best performing models' accuracy are listed below.

Model		Accuracy
Logistic regression		63.38%
Deep model	BiGRU	78.58%
	BiLSTM	76.38%
	SumEmbeddings	80.19%

1 https://nlp.stanford.edu/projects/snli/

Data Preparation

The downloaded SNLI dataset has to be extracted and the files 'snli_1.0_train.jsonl' and 'snli_1.0_test.jsonl' have to be saved in the './input/' directory. After that *generate_meta_input()* function from *utils* package has to be executed to generate cleaned and processed data. This can be done by uncommenting the corresponding code snippet in the main function when the program is run for the first time. This function takes care of text preprocessing which involves:

- removal of HTML tags
- removal of extra white spaces
- conversion to lower case
- removal of accented characters
- expand contractions
- stop words removal
- removal of punctuation
- removal of special character
- removal/conversion of numbers
- lemmatization
- tokenization (using Spacy)

Each line in the JSON file is read and the data is loaded to a dictionary, from which the sentences and labels are extracted. These sentences are preprocessed and tokens of each sentence is stored as a sublist of a main list. The labels of each sentence-pair is also stored as a list. These lists are stored as pickle files at './input/data_pickles/' directory so that they need not be processed again and again during training and testing time. Hence, three files (each for training and testing) are created – for sentence1, sentence2 and gold_labels.

Embedding

GloVe embedding[2] from Stanford has been used throughout this project to embed the words to vectors. GloVe embedding[3] with 6 billion token is used for embedding. The corresponding file has to be download and extracted to './input/embedding/' directory.

2 https://nlp.stanford.edu/projects/glove/

3 https://nlp.stanford.edu/data/glove.6B.zip

Model architecture – Logistic Regression

Logistic regression model was trained using TF-IDF (Term Frequency-Inverse Document Frequency) features obtained using *sklearn* python library. The pickled data lists are first loaded and the tokens of sentence1 and sentence2 is merged together to make a string. During this process, entries in train and test dataset without any gold_labels (label of "-") was encountered. As of writing this report, it was chosen to ignore those entries and delete them from the dataset as they do not add any information.

During training

After creating the dataset corpus, a TF-IDF Vectorizer class object was instantiated and was fit over the training dataset corpus to learn vocabulary and IDF from training set. The fit vectorizer is then stored as a pickle file at './model/TFIDF.pickle' to recover it during testing time. This was done so that, the same vectorizer, and hence the same vocabulary and IDF, will be used during testing as well. This was done to maintain the same "important words" and the same input sample dimension among the training and testing dataset.

The logistic regression model is trained with the TF-IDF features using sklearn LogisticRegression class instance. The model is trained (fit) using L-BFGS (Limited memory - Broyden–Fletcher–Goldfarb–Shanno algorithm) solver with a maximum iteration limit of 1000. The trained (fit) model is saved at './models/LR.pickle' for future uses and testing.

During testing

The data is processed the same way as for training (mentioned above), but instead of using a new vectorizer, we load the previously saved vectorizer from the pickle file. The TF-IDF features are generated and they are used for predicting. The results of prediction are converted from class labels to contraction/neutral/entailment and are written to a file at './tfidf.txt'.

Different methods of generating the TF-IDF features was tested and their corresponding accuracies are listed below. Here the word vectors indicate the TF-IDF vectors of sentence1 and sentence2.

Method	Accuracy
Euclidean distance between vectors	45.78%.
Dot product between vectors	46.44%.
Hadamard (element-wise) product between vectors	47.95%.
TF-IDF vector of concatenated sentence	55.57%
Concatenation of TF-IDF vectors	63.38%

General architecture for deep recurrent network models

Most of the deep models (except that of BERT-based) share a similar architecture which differ in the type of recurrent layer used. Hence, the general details are listed in this section to avoid redundancy. The model-specific details, plots and results are listed in the later section. Few files are duplicated across modules, but that allows the standalone usage of that module and facilitates the easy resusability of code.

The first step towards implementing a deep model for text is to convert each atomic discrete entity in the input (words or characters) into real vectors from $\mathbf{R}^d$ so that their semantics are captured meaningfully. For this purpose, GloVe embedding has been used in this project. Different pretrained GloVe embeddings are available and the embedding chosen for this project is the one with 6 billion tokens trained over Wikipedia corpus. The GloVe embedding is loaded into a dictionary with word-vector as key-value pair. This dictionary is stored as a pickle file at ‘./input/embeddings/glove_dict.pickle’ for fast access in successive runs.

During the processing of dataset, entries without any gold_labels (label of “-”) was encountered. As of writing this report, it was chosen to ignore those entries and delete them from the dataset as they do not add any information.

During training

The next step is to construct a weight matrix for embedding layer, such that each column of this matrix represents an embedding vector. After a dataset corpus is generated, we use Tokenizer from Tensorflow Keras to assign every word an positive integer. The i^{th} column of the embedding matrix is initialized with the embedding vector of the word with integer value of i assigned by the Tokenizer. This Tokenizer is stored as a pickle file at ‘./model/tokenizer.pickle’ to preserve the word-integer mapping, so that it can be used during testing. It is then used to convert the each input data entry to a sequence of numbers, which are then fed into the model for training.

Model

The model has two input layers, one defined for the premise and another for hypothesis. The input layers are followed by an embedding layer initialized with the weight matrix previously generated. This layer converts the sequence of integers of each sentence into a sequence of embedding vectors. At this stage the shape of the input is tensor is (NUM_SAMPLES x MAX_SEQ_LEN x EMBEDDING_DIM) for premise and hypothesis, where NUM_SAMPLES represents the number of samples, MAX_SEQ_LEN represents the maximum length of sequence (defined in the program), and EMBEDDING_DIM represents the embedding vector dimension. A time distributed translation

Dense layer is applied through the time axis (here, the word sequence axis), so that the same Dense layer is applied to each time-step of the input. The outputs are processed through a model-specific layer, and is then normalized.

The normalized vectors for premise and hypothesis are then concatenated, and are subjected to three units of 600D Dense layers with ReLU activation. L2 regularizer is used alongside to avoid over fitting. The final layer consists of a 3-way softmax classification, which classifies into one of entailment, neutral, or contraction.

The optimizer used was ‘RMSprop’ and the loss function used was ‘categorical_crossentropy’. Three callbacks were implemented for the better training of the model, which are EarlyStopping, ModelCheckpoint, ReduceLROnPlateau. EarlyStopping callback monitors a specified quantity (here, validation accuracy) and stops training when that quantity stops improving. ModelCheckpoint callback keeps track and saves the best performing model during training. Once training is completed, we load the weights from the saved best performing model. ReduceLROnPlateau callbacks reduces the learning rate by a factor, when the value of quantity being monitored plateaus. Once the training is completed the model is stored at ‘./model/’ directory as a h5 file.

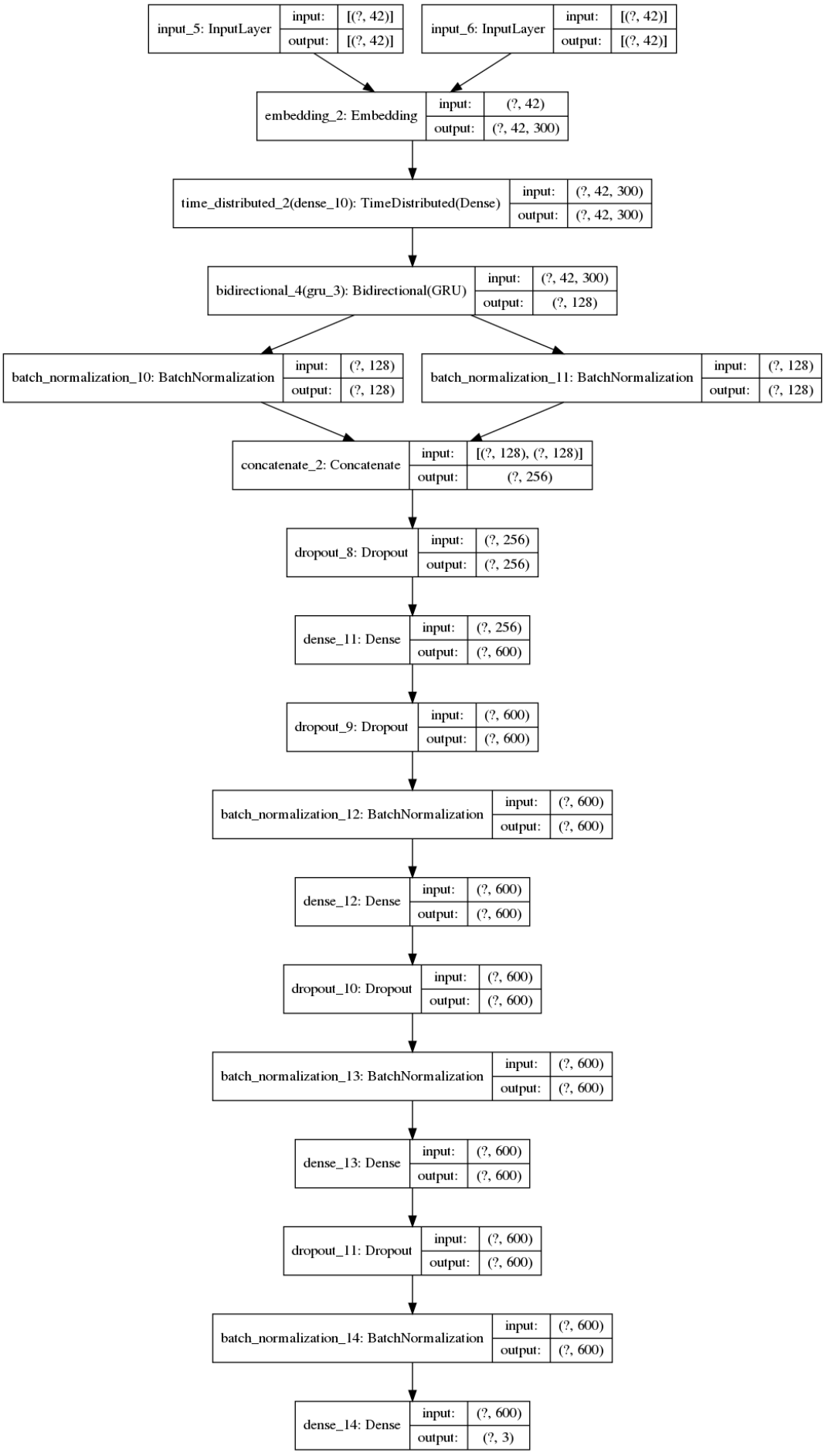
During testing

The Tokenizer that was stored as a pickle file is loaded and used to convert the input samples to sequences of integers. The deep model is loaded from the ‘./model/’ directory and is fed with the converted test data. The predicted integer labels are converted back to relationship labels – entailment, neutral, or contraction – and are written to an output file. The test accuracy and test loss are printed on screen and plots are generated for analyzing the performance of the model.

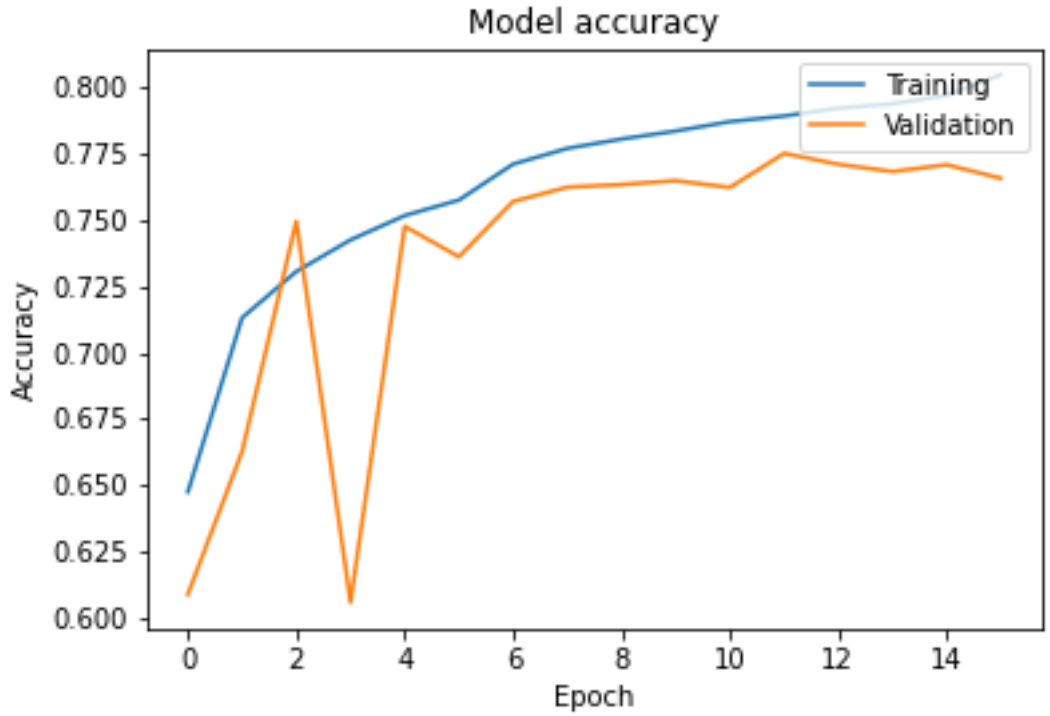
In the following plots, the class label and relationship label mapping is as follows.

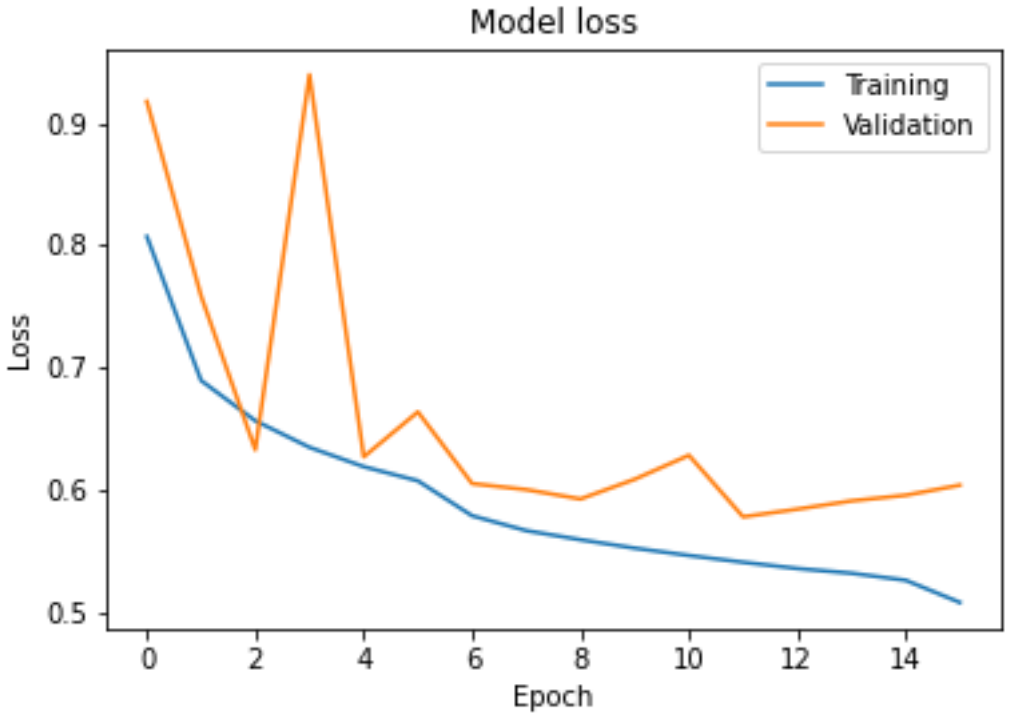
Contraction	0
Neutral	1
Entailment	2

Model architecture – BiGRU

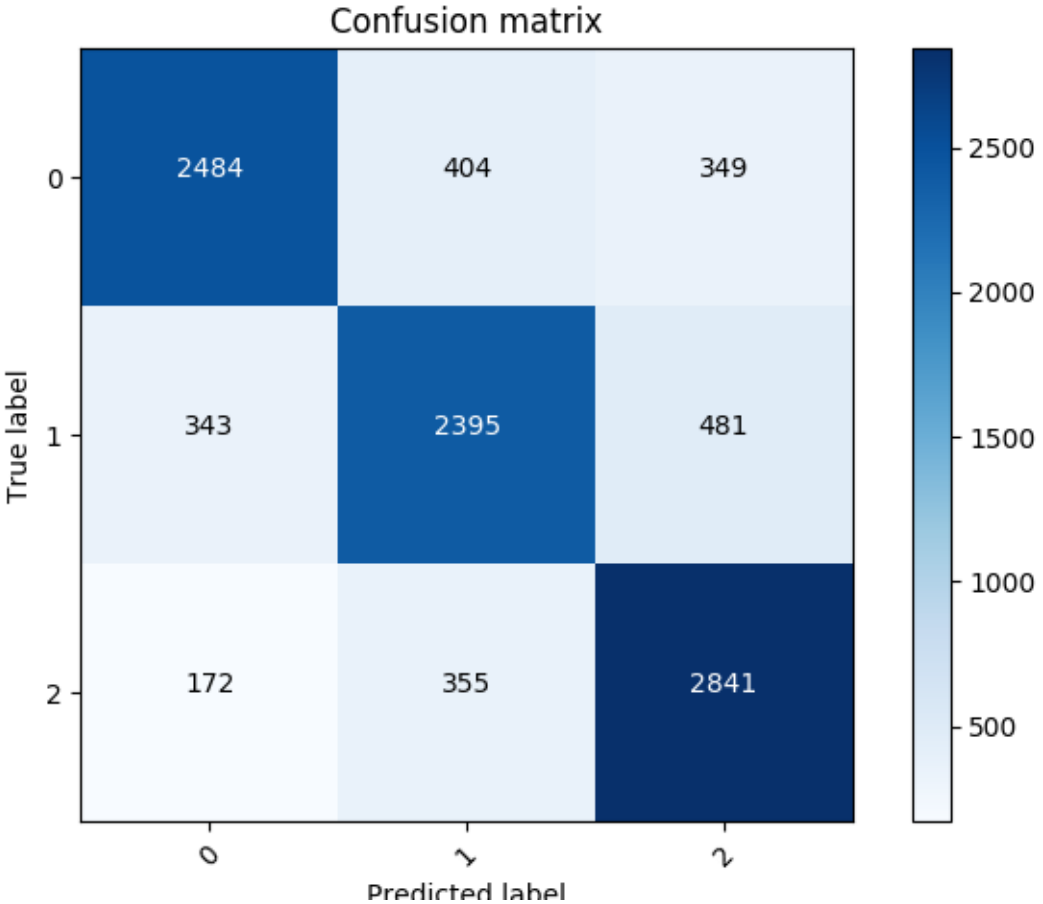


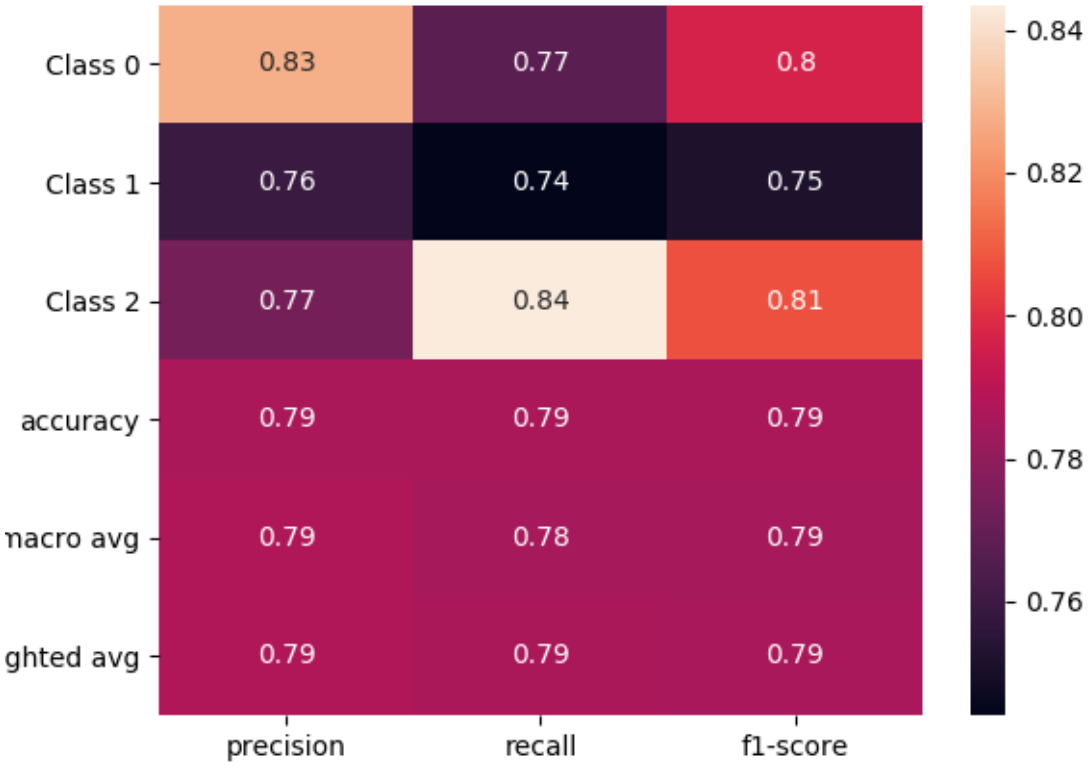
The model architecture is shown in the image above. A Bidirectional Gated Reccurent Unit (BiGRU) of 64 units has been used as the model-specific layer. The model performs with an accuracy of 78.58% and the output text file and plots are saves at ‘./results/BiGRU/’.





The model performs the best in classifying ‘entailment’ relationships and the worst in ‘neutral’ relationships.



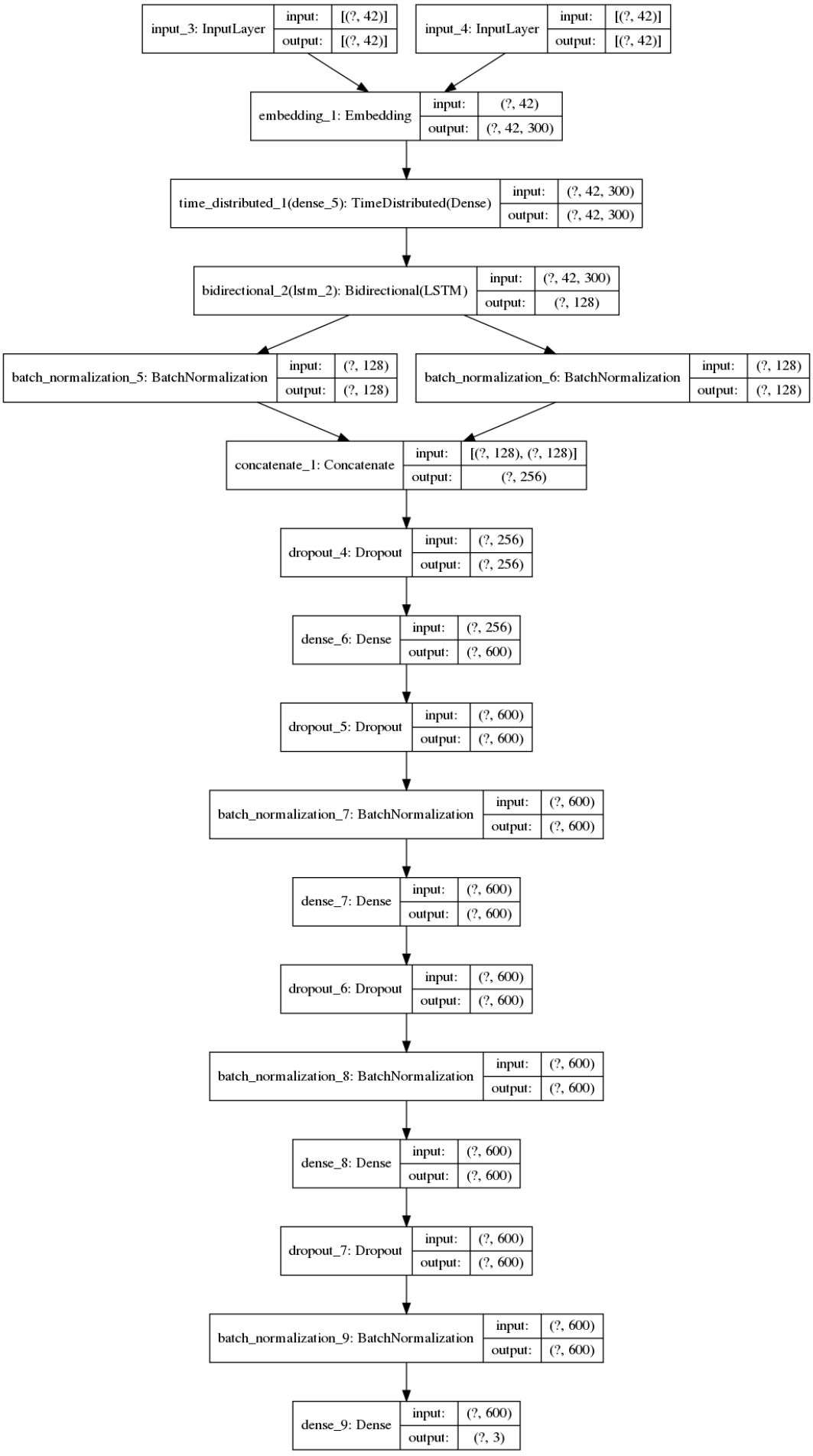


Experiments

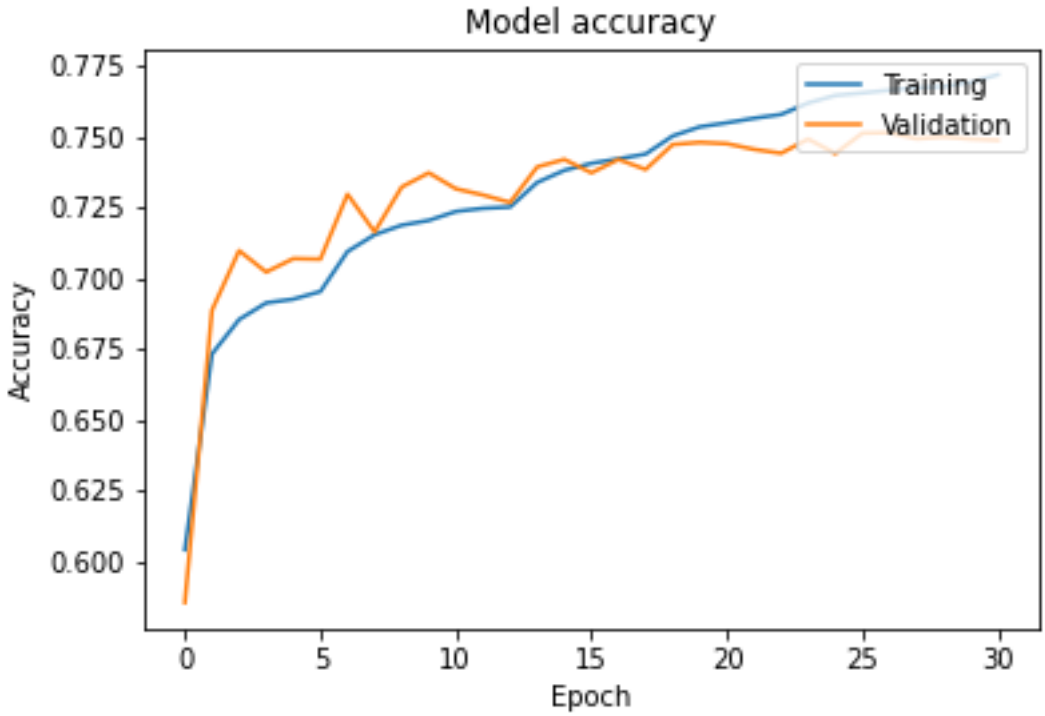
Layers	Optimizer	LR	Regularizer	Accuracy
Bidirectional(GRU(64))	RMSprop	0.01	L2 (4e-6)	**78.58%**
Bidirectional(GRU(128)) Bidirectional(GRU(64))	RMSprop	0.01	L2 (4e-6)	76.17%
Bidirectional(LSTM(128)) Bidirectional(LSTM(64))	RMSprop	0.01	L2 (4e-6)	75.82%
Bidirectional(LSTM(256)) Bidirectional(LSTM(128)) Bidirectional(LSTM(64))	RMSprop	0.01	L2 (4e-6)	75.03%
Bidirectional(GRU(64))	RMSprop	0.01	L1 (4e-6)	74.41%
Bidirectional(GRU(128)) Bidirectional(GRU(64))	RMSprop	0.01	L1 (4e-6)	75.45%
Bidirectional(LSTM(128)) Bidirectional(LSTM(64))	RMSprop	0.01	L1 (4e-6)	74.21%
Bidirectional(LSTM(256)) Bidirectional(LSTM(128)) Bidirectional(LSTM(64))	RMSprop	0.01	L1 (4e-6)	72.67%
Bidirectional(GRU(64))	Adam	0.01	L2 (4e-6)	72.39%
Bidirectional(GRU(128)) Bidirectional(GRU(64))	Adam	0.01	L2 (4e-6)	69.85%
Bidirectional(GRU(64))	Adam	0.01	L1 (4e-6)	75.94%
Bidirectional(GRU(128)) Bidirectional(GRU(64))	Adam	0.01	L1 (4e-6)	75.02%
Bidirectional(GRU(64))	Adagrad	0.01	L2 (4e-6)	75.34%
Bidirectional(GRU(128)) Bidirectional(GRU(64))	Adagrad	0.01	L2 (4e-6)	72.72%
Bidirectional(GRU(64))	Adagrad	0.01	L1 (4e-6)	75.71%
Bidirectional(GRU(128)) Bidirectional(GRU(64))	Adagrad	0.01	L1 (4e-6)	75.34%

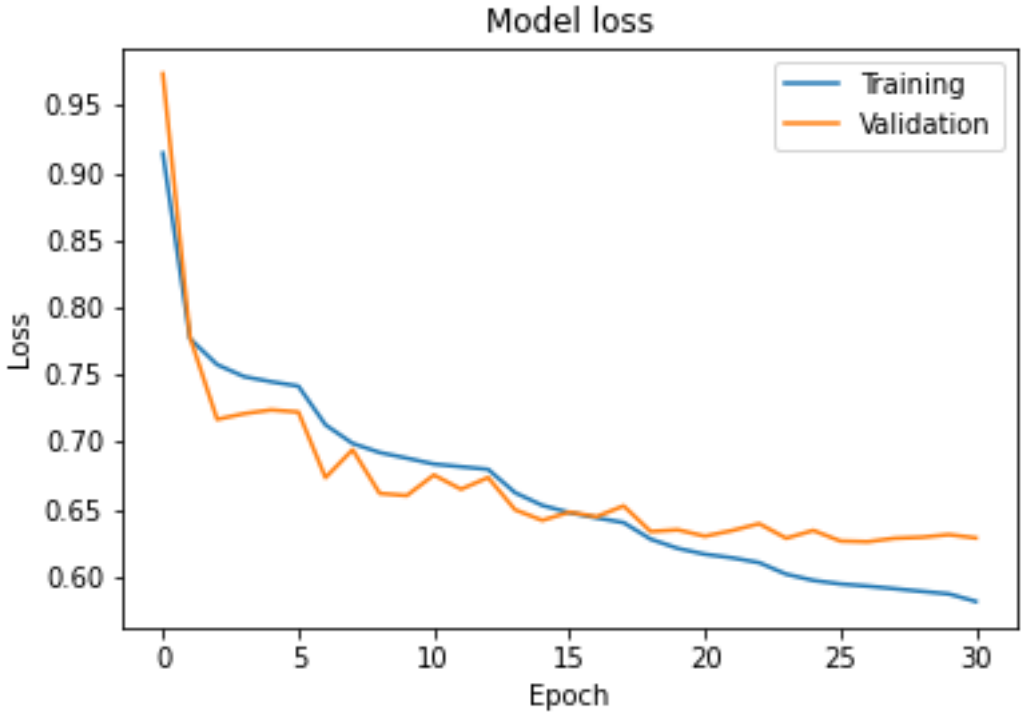
The learning rates and number of epochs were not tuned to a great extent and was initialized with suitable values, since appropriate callbacks were implemented. The learning rates gets reduced in case the training performance plateaus and the training stops if the performance saturates. RMSprop and Adam are the widely used optimizers for NLP and hence was implemented and evaluated. The general structure of the deep learning model in terms of the Dense layers, its units, activation etc was also experimented with, but was omitted from the table due to space constraints.

Model architecture – BiLSTM

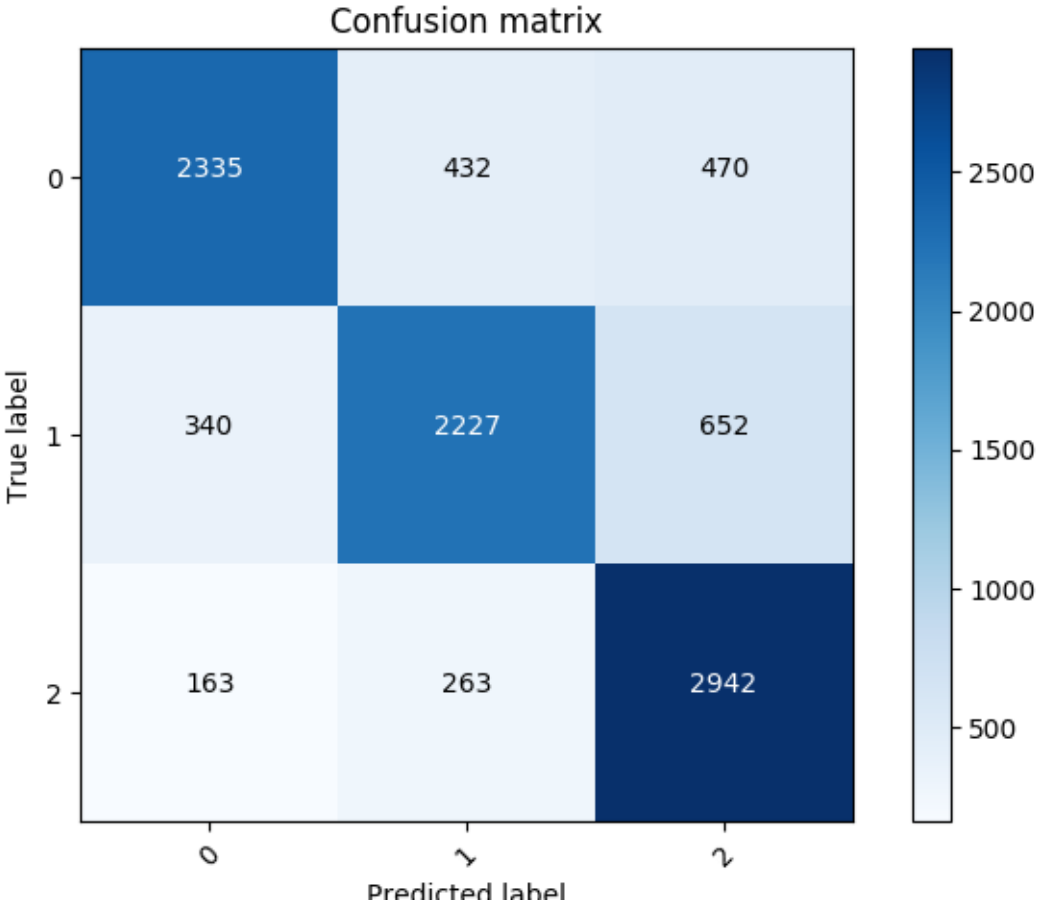


The model architecture is shown in the image above. A Bidirectional Long Short Term Memory (BiLSTM) of 64 units has been used as the model-specific layer. The model performs with an accuracy of 76.38% and the output text file and plots are saves at ‘./results/BiLSTM/’.





The model performs the best in classifying ‘entailment’ relationships and the worst in ‘neutral’ relationships.

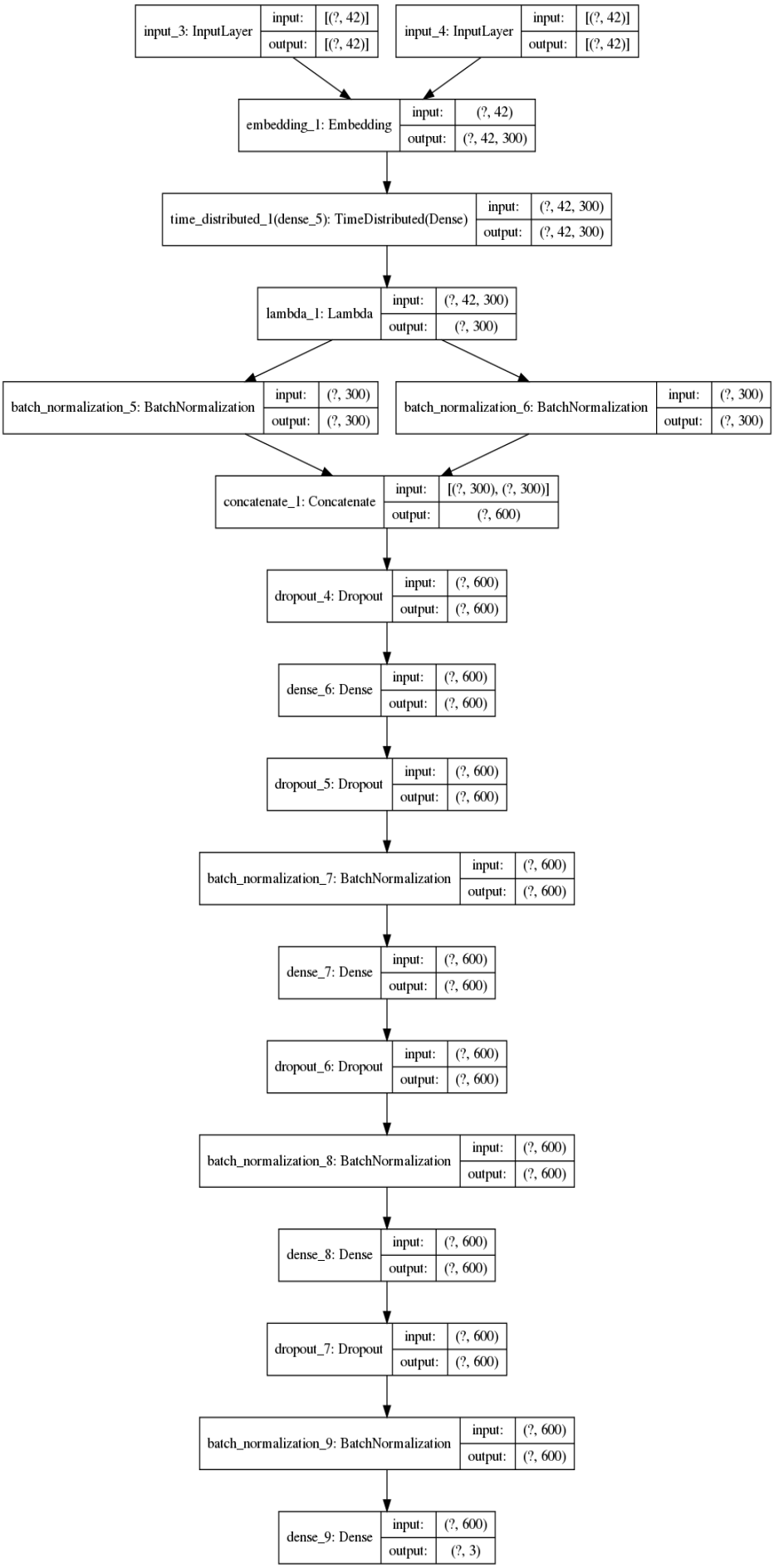


Experiments

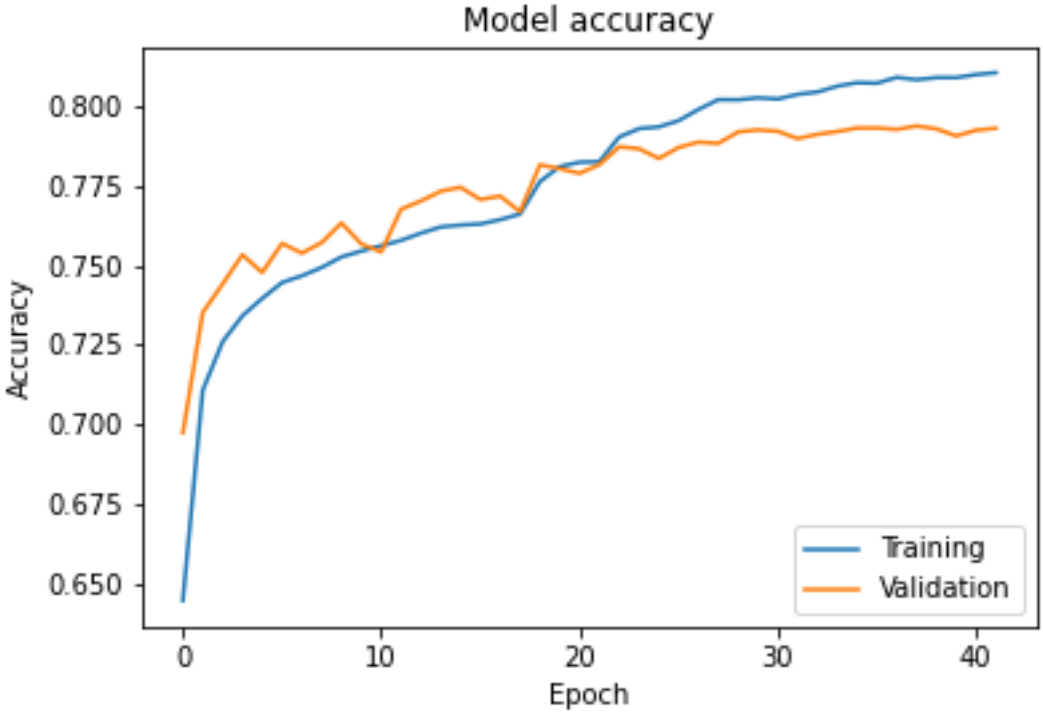
Layers	Optimizer	LR	Regularizer	Accuracy
Bidirectional(LSTM(64))	RMSprop	0.01	L2 (4e-6)	**76.38%**
Bidirectional(LSTM(128))	RMSprop	0.01	L2 (4e-6)	75.69%
Bidirectional(LSTM(128)) Bidirectional(LSTM(64))	RMSprop	0.01	L2 (4e-6)	76.37%
Bidirectional(LSTM(256)) Bidirectional(LSTM(128)) Bidirectional(LSTM(64))	RMSprop	0.01	L2 (4e-6)	74.17%
Bidirectional(LSTM(64))	RMSprop	0.01	L1 (4e-6)	76.14%
Bidirectional(LSTM(128))	RMSprop	0.01	L1 (4e-6)	76.33%
Bidirectional(LSTM(128)) Bidirectional(LSTM(64))	RMSprop	0.01	L1 (4e-6)	76.17%
Bidirectional(LSTM(256)) Bidirectional(LSTM(128)) Bidirectional(LSTM(64))	RMSprop	0.01	L1 (4e-6)	75.87%
Bidirectional(LSTM(64))	Adam	0.01	L2 (4e-6)	75.32%
Bidirectional(LSTM(128)) Bidirectional(LSTM(64))	Adam	0.01	L2 (4e-6)	73.63%
Bidirectional(LSTM(64))	Adam	0.01	L1 (4e-6)	75.27%
Bidirectional(LSTM(128)) Bidirectional(LSTM(64))	Adam	0.01	L1 (4e-6)	74.87%
Bidirectional(LSTM(64))	Adagrad	0.01	L2 (4e-6)	75.86%
Bidirectional(LSTM(128)) Bidirectional(LSTM(64))	Adagrad	0.01	L2 (4e-6)	73.49%
Bidirectional(LSTM(64))	Adagrad	0.01	L1 (4e-6)	74.38%
Bidirectional(LSTM(128)) Bidirectional(LSTM(64))	Adagrad	0.01	L1 (4e-6)	72.96%

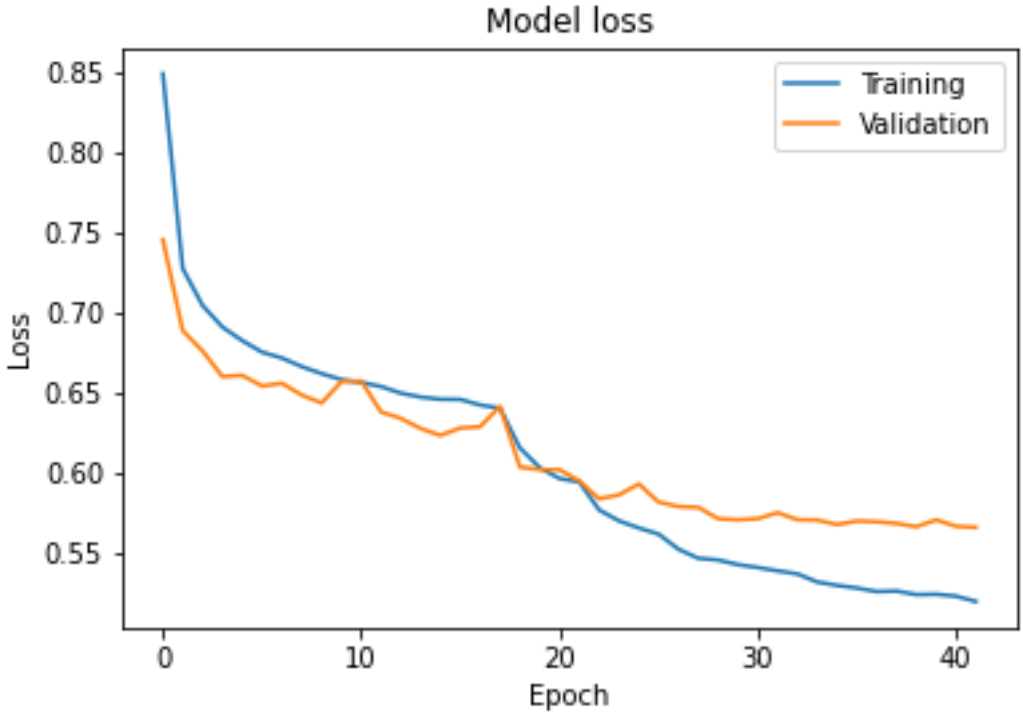
The learning rates and number of epochs were not tuned to a great extent and was initialized with suitable values, since appropriate callbacks were implemented. The learning rates gets reduced in case the training performance plateaus and the training stops if the performance saturates. RMSprop and Adam are the widely used optimizers for NLP and hence was implemented and evaluated. The general structure of the deep learning model in terms of the Dense layers, its units, activation etc was also experimented with, but was omitted from the table due to space constraints.

Model architecture – SumEmbeddings

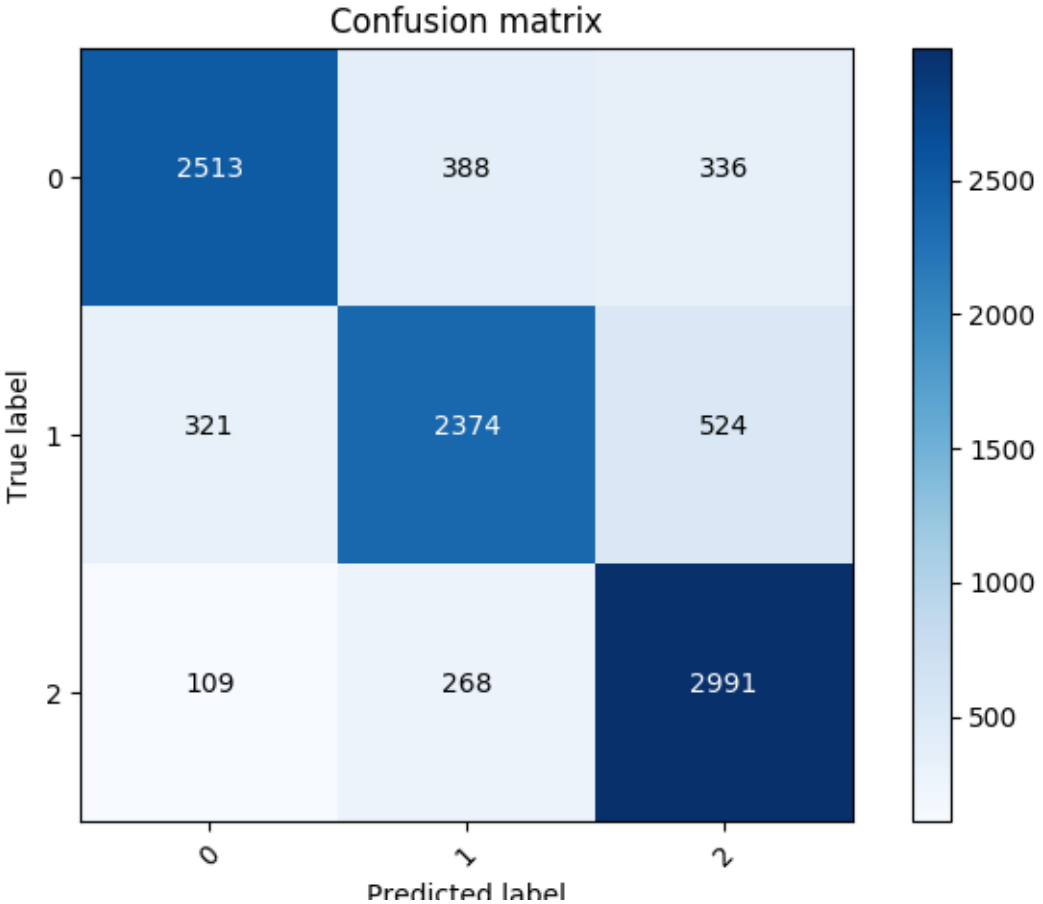


The model architecture is shown in the image above. A SumEmbedding lambda layer, which sums up all the embedding vectors in the sentence is used. The model performs with an accuracy of 80.19% and the output text file and plots are saves at ‘./results/SumEmbeddings/’. Since this is the best performing model the output text files are stored at ‘./deep_model.txt’ as well.





The model performs the best in classifying ‘entailment’ relationships and the worst in ‘neutral’ relationships. It also outperforms the other models in all three classes.



Experiments

Layers	Optimizer	LR	Regularizer	Accuracy
SumEmbedding Lambda	RMSprop	0.01	L2 (4e-6)	**80.19%**
SumEmbedding Lambda	RMSprop	0.01	L2 (4e-5)	78.65%
SumEmbedding Lambda	RMSprop	0.01	L2 (4e-4)	76.70%
SumEmbedding Lambda	RMSprop	0.01	L1 (4e-6)	79.78%
SumEmbedding Lambda	RMSprop	0.01	L1 (4e-5)	72.61%
SumEmbedding Lambda	RMSprop	0.01	L1 (4e-4)	68.39%
SumEmbedding Lambda	RMSprop	0.01	L1L2 (4e-6)	79.65%
SumEmbedding Lambda (with two Dense layers)	RMSprop	0.01	L2 (4e-6)	76.03%
SumEmbedding Lambda (with one Dense layers)	RMSprop	0.01	L2 (4e-6)	78.96%
SumEmbedding Lambda (with just output layer)	RMSprop	0.01	L2 (4e-6)	63.80%
SumEmbedding Lambda	Adam	0.01	L2 (4e-6)	79.94%
SumEmbedding Lambda	Adam	0.01	L1 (4e-6)	80.08%
SumEmbedding Lambda	Adam	0.01	L1L2 (4e-6)	79.78%
SumEmbedding Lambda	Adagrad	0.01	L2 (4e-6)	75.58%
SumEmbedding Lambda	Adagrad	0.01	L1 (4e-6)	75.58%
SumEmbedding Lambda	Adamax	0.01	L2 (4e-6)	80.17%
SumEmbedding Lambda	Adamax	0.01	L1 (4e-6)	79.91%
SumEmbedding Lambda	Adadelta	0.01	L2 (4e-6)	64.31%
SumEmbedding Lambda	Adadelta	0.01	L1 (4e-6)	64.86%

The learning rates and number of epochs were not tuned to a great extent and was initialized with suitable values, since appropriate callbacks were implemented. The learning rates gets reduced in case the training performance plateaus and the training stops if the performance saturates. RMSprop and Adam are the widely used optimizers for NLP and hence was implemented and evaluated. The general structure of the deep learning model in terms of the Dense layers, its units, activation etc was also experimented with, but was omitted from the table due to space constraints.

For hyper parameter tuning, grid search or other auto-tuning were not implemented because of the compute power required.

Model architecture – BERT (Experimental)

An experimental Huggingface transformer based BERT is also implemented in the project. All the codes work and the model trains and tests, but the training process is computationally very expensive. Even after using Google Colab TPU and dividing the data into smaller parts, the training was unable to be completed due to Google usage time restrictions. Hence, the performance analysis or plots are not attached this report.

The Huggingface library is a great tool for implementing transformer based deep learning models. We can directly use pretrained models, by simply importing them into the code. With minor modifications to code, we can use any transformers based approach supported by Huggingface. Due to the scarcity of computing resources, I have not tried using other transformer based approaches.