Midterm Documentation

CSYE 7374

Derek Hung

Mitesh Puthran

Introduction:

As knowledge become more and more advanced in the modern world especially in the world of cognitive computing and AI, there are a need modernize many aspects of life. One of the needs especially in the business world is the creation of the sentiment engine tool. A sentiment engine is a tool that classifies data based on the opinion, emotions, and attitudes. In this project itself, we take the data from two subtasks and using keras and tensorflow will test and calculate the sentiment based on the text in the file itself. The first subtask contains data from twitter such as the tweets, the source, the cash tag, and finally the golden standard sediment score. The second subtask contains data based on news articles such the company, the title, and the sediment score. After the optimal way of finding the right sediment score is found, a cosine score is found based on the golden standard vector and predicted score vector. Finally, on the final day before the presentation/ a new data set will be released and the neural network that was used for the assignment would be used to predict the data set's sentiment value.

Data Used:

The. json files used for training and testing neural network used for our project for the first subtask is located at:

https://bitbucket.org/ssix-project/semeval-2017-task-5-subtask-1/src/beadeb1fd0f9b8093e4828a198a92e651a4e10c6/Microblog_Trainingdata.json?at=master&fileviewer=file-view-default

and the second subtask:

https://bitbucket.org/ssix-project/semeval-2017-task-5-subtask-2/src/2fb645e839b7fb9923d2402d8ee817242360993f/Headline_Trainingdata.json?at=master&fileviewer=file-view-default

To create the embedding layer for the first subtask, we used pretrained word vector files. For the first subtask, we used pretrained word vector file glove.27.twitter.txt which includes word vectors for 27 billion words specifically used in twitter tweets. For the second subtask, we used the pretrained word vector file GoogleNews-vectors-negative300.bin which contains pretrained word vectors for 100 billion words specifically used in news headlines.

In this project we used Keras with Tensorflow GPU. This allows the training of the data to become much faster than using normal Tensorflow especially with a large training set.

Analyses:

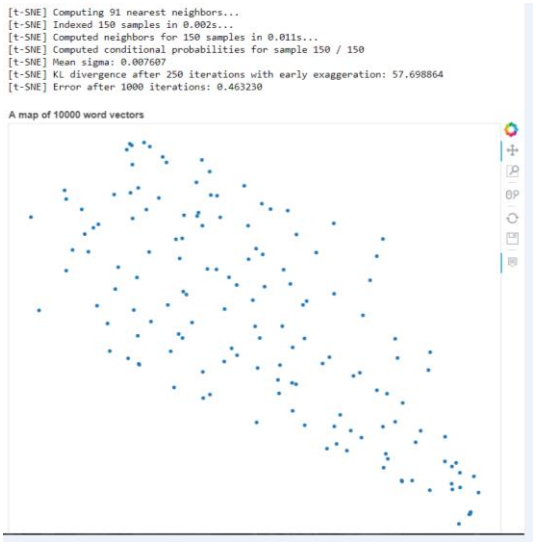
Converted Sentiment Score Analyses:

To clearly see if our data based on accuracy we converted the sentiment value into three types of values: -1,0,1. -1 means that the sentiment value is "negative". From the original sentiment score, it has a range of -1 to -0.3. 0 means that the sentiment value is "neutral". It has a range of -0.3 to 0.3 from the original range. 1 means that the sentiment value is "negative". It means it has a range from 0.3 to 1 from the original range.

The reason why three types of values were chosen instead of two is because of unlike reviews, neutral tweets and news articles are common. For example, some tweets can be used to explain a schedule or activity. This is a departure from reviews where they are meant to be more polarizing since they are trying to inform people of whether or not one should go see or buy a specific product.

Self-Trained Embedded layer Analyses:

The first step we tried to do and experiment was to try to create our own embedding layer specifically the firs that uses the spans for the first subtask. The reason why was that to ensure that our code flexible so there would be no need to download pretrained word vectors for the embedding layer. Using the word2Vector from the genism library, we created our own embedding layer from our own testing data to create the word vectors.




```
In [214]: w2v.most_similar('rally')

Out[214]: [(',', 0.8812909126281738),
           ('$', 0.8780311346054077),
           ('year', 0.875724196434021),
           ('the', 0.8730244636535645),
           ('on', 0.8684260845184326),
           ('and', 0.8660690188407898),
           ('is', 0.8649418354034424),
           ('to', 0.8627430200576782),
           ('will', 0.8609170913696289),
           ('a', 0.8608945608139038)]

In [215]: w2v.most_similar('good')

Out[215]: [('to', 0.9209266304969788),
           (',', 0.9166077971458435),
           ('a', 0.9165568351745605),
           ('$', 0.9139012098312378),
           ('on', 0.9078933596611023),
           ('the', 0.9067692756652832),
           ('is', 0.9049218893051147),
           ('will', 0.9023064970970154),
           ('year', 0.902186930179596),
           ('.', 0.9011276960372925)]
```

As seen above, because of the low amount of test data needed to be used to create a word vector the resulting trained word vector is inaccurate therefore unsuitable to be used for our embedded

layer. It was finally concluded that we needed to import pretrained word vectors for each subtask after this analysis.

Subtask 1 Neural Network Type Analyses:

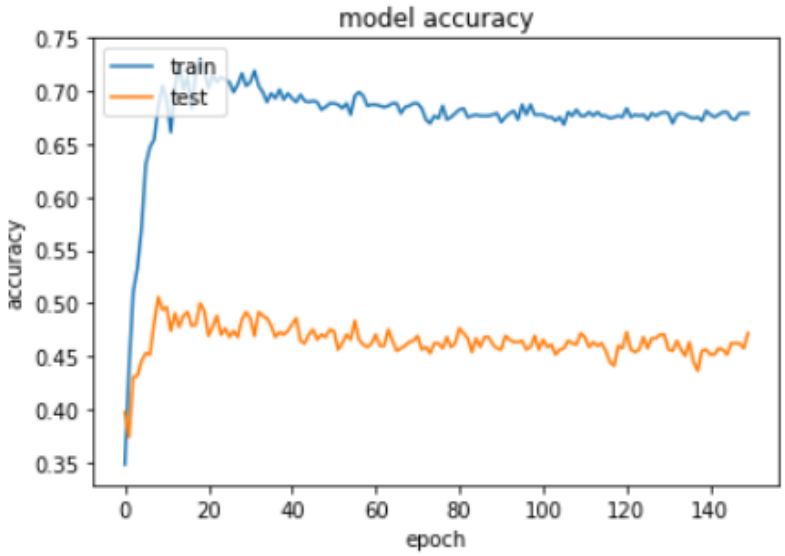
In order to choose the right class of neural networks, three types of classes were chosen to test subtask 1 and 2 , MLP,CNN, and LSTM. The MLP, CNN and LSTM would be run and based on the model accuracy and also loss data, a certain class of neural network would be chosen to be used to find the cosine score and also used for the final dataset.

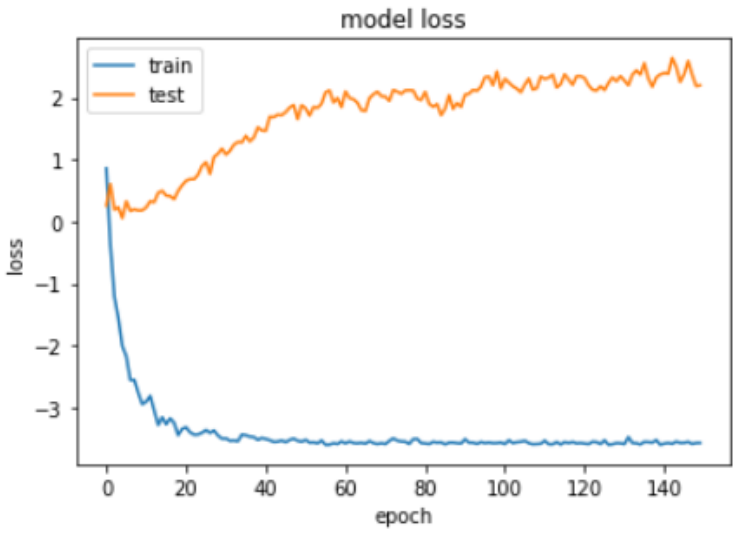
MLP:

For the MLP since the results have a negative number, we chose the activation function to be equal to "tanh" for the output layer and "linear" for the input layer. The code was run with 150 epochs with a batch of 32.

```
]: model_mlp.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 300)               0
_________________________________________________________________
embedding_1 (Embedding)      (None, 300, 200)          325400
_________________________________________________________________
flatten_1 (Flatten)          (None, 60000)             0
_________________________________________________________________
dense_1 (Dense)              (None, 25)                1500025
_________________________________________________________________
dropout_1 (Dropout)          (None, 25)                0
_________________________________________________________________
dense_2 (Dense)              (None, 25)                650
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 26
=================================================================
Total params: 1,826,101
Trainable params: 1,500,701
Non-trainable params: 325,400
_________________________________________________________________
```



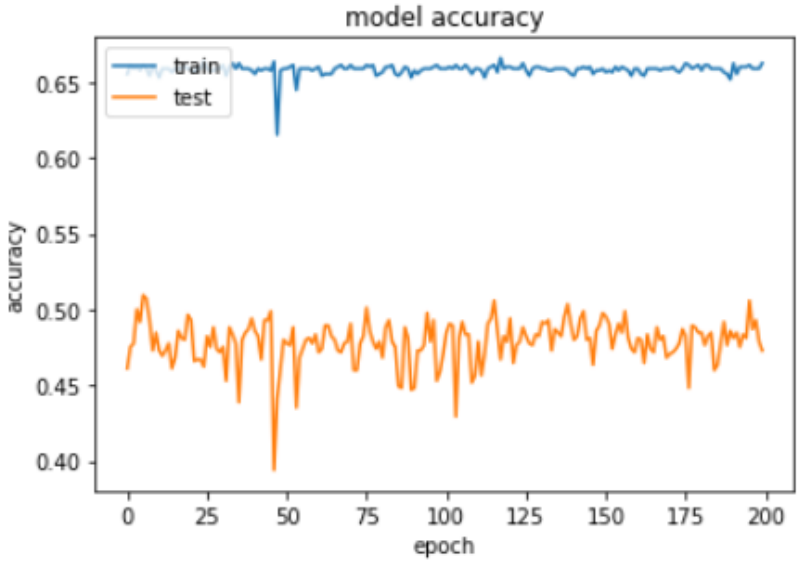


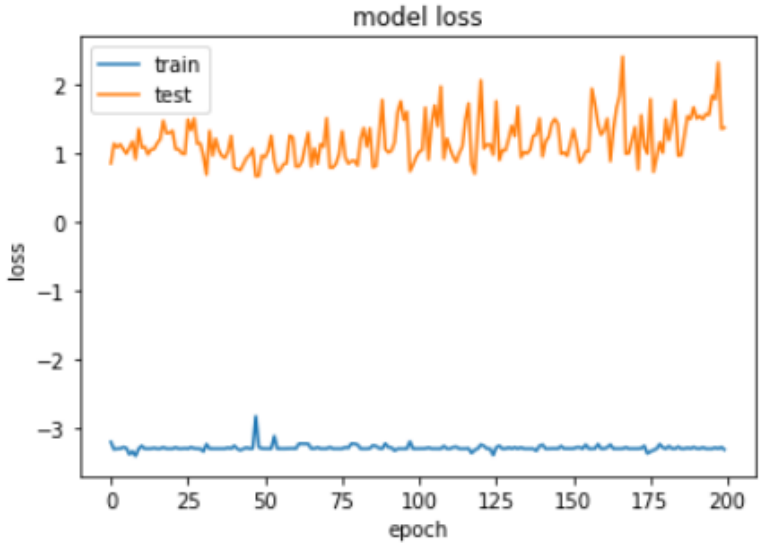
CNN:

For the CNN, it contains around five layers with around 200 epochs with a batch of 32. Like the MLP the output layer consists of activation function tanh with the linear for the input layers.

```
: model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_8 (InputLayer)         (None, 300)               0
_________________________________________________________________
embedding_1 (Embedding)      (None, 300, 200)          325400
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 296, 128)          128128
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 59, 128)           0
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 55, 128)           82048
_________________________________________________________________
max_pooling1d_2 (MaxPooling1 (None, 11, 128)           0
_________________________________________________________________
conv1d_3 (Conv1D)            (None, 7, 128)            82048
_________________________________________________________________
max_pooling1d_3 (MaxPooling1 (None, 1, 128)            0
_________________________________________________________________
flatten_8 (Flatten)          (None, 128)               0
_________________________________________________________________
dense_30 (Dense)             (None, 128)               16512
_________________________________________________________________
dense_31 (Dense)             (None, 1)                 129
=================================================================
Total params: 634,265
Trainable params: 308,865
Non-trainable params: 325,400
_________________________________________________________________
```





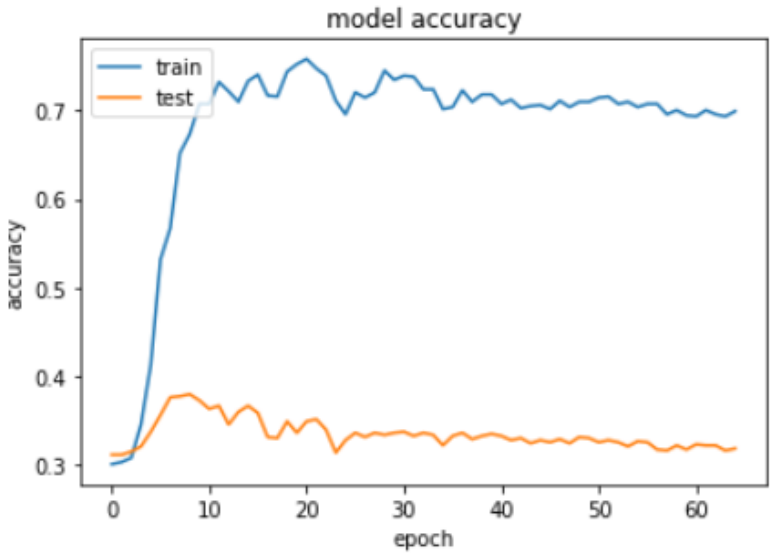
As seen above, CNN was considered the worse to be used for this project. This was expected since CNNs are used mainly on sight to discover certain features. Bothe Model accuracy and model losss shows a high overshooting.

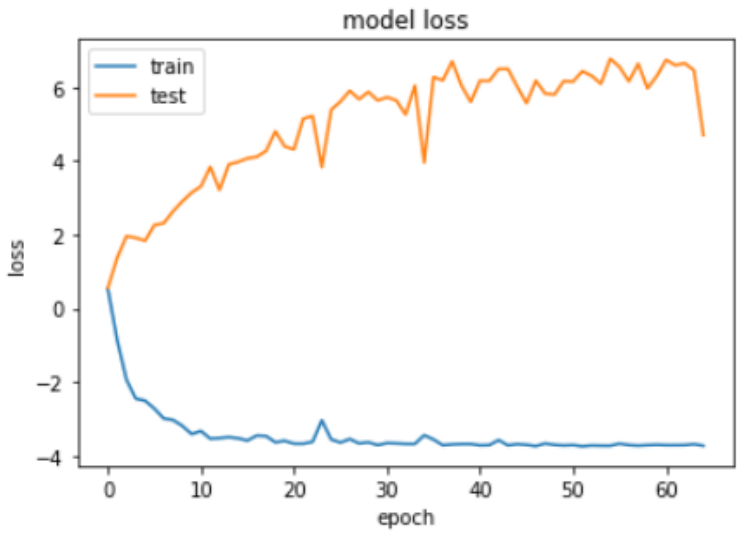
LSTM:

For the LSTM because of time constraints, an epoch of 65 was chosen. It also contains a dropout rate of 0.2.

```
|: model_lstm.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_10 (Embedding)     (None, None, 128)         2560000
_________________________________________________________________
lstm_9 (LSTM)                (None, 128)               131584
_________________________________________________________________
dense_32 (Dense)             (None, 1)                 129
=================================================================
Total params: 2,691,713
Trainable params: 2,691,713
Non-trainable params: 0
_________________________________________________________________
```



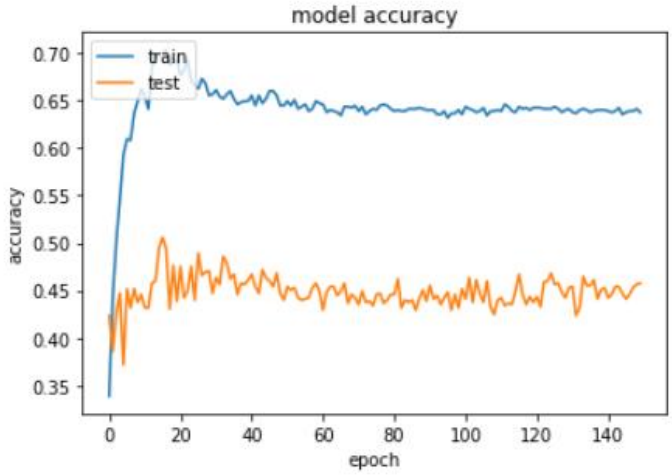


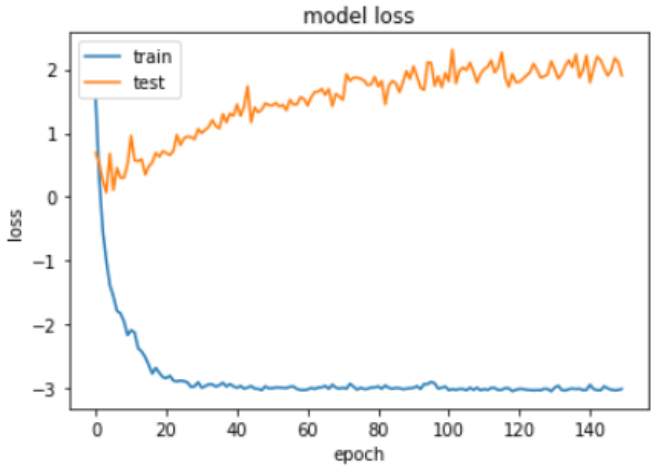
As seen above for the MLP, CNN, and LSTM class, there seems like the MLP seems to be the right choice for the job. Even though there is a large overfitting, it has highs valued accuracy with a value of 50. Also based on the loss graph and the accuracy graph it seems to be still be able to tunable to increase its accuracy even more.

Out of all the three classes the LSTMs have the slowest training time. This makes sense since LSTMS are based on recurrent neural networks which use their internal memory to process arbitrary sequences of inputs. Surprisingly, they were not the best choice as seen above since they are used in many language applications such as translations.

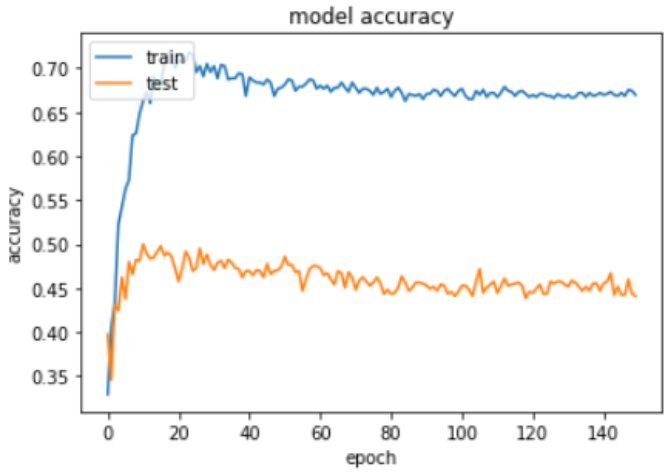
Tuning the MLP:

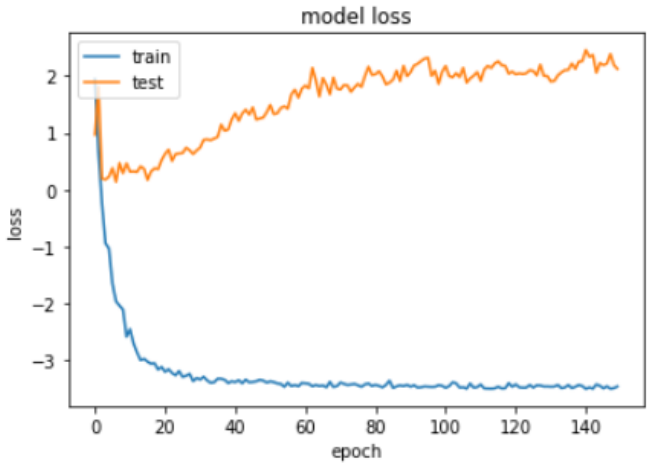
As seen above in the loss graph, it seems there seems to be a problem of overfitting in between the trained data and predicted data. Also in the loss graph, there at least shown in the curve of the trained data, there seems to be a need to lower the learning rate. The first step we did to tune the MLP was to change the dropout rate for 0.5 to 0.3



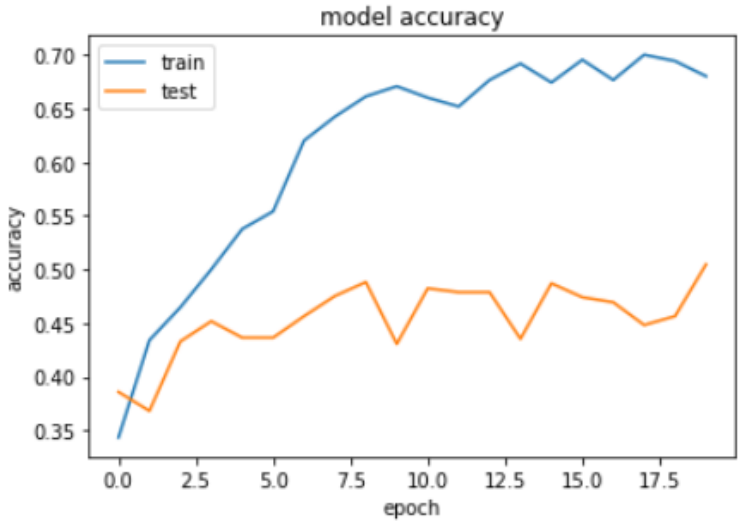


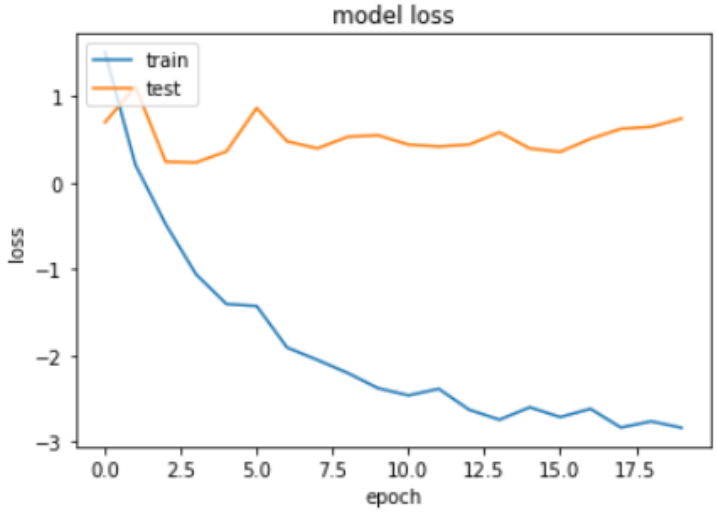
Even though the dropout rate was lowered, the data as shown above only decreased the accuracy, therefore lowering is not the right solution. Another way based on the loss graph is to include a learning rate of 0.0001 in there.





As we could see the loss of the train and test data almost follow the same period in the beginning. However, it starts to overfit. In the Model Accuracy, the test accuracy decreased. Therefore, another plan, one way to increase the accuracy is to lower the epochs. As shown above, the top it seems the accuracy is at its highest in the epoch 20.





As seen above, the resulting graph had a reached the point of the highest accuracy. However, when the test data was checked, it seems like the neural network did not include the -1 which is the sentiment value of “negative”. Therefore, it was finally concluded that the original MLP would be the right neural network class for the job not the modified one.

Subtask 2 Neural Network Type Analyses:

MLP:

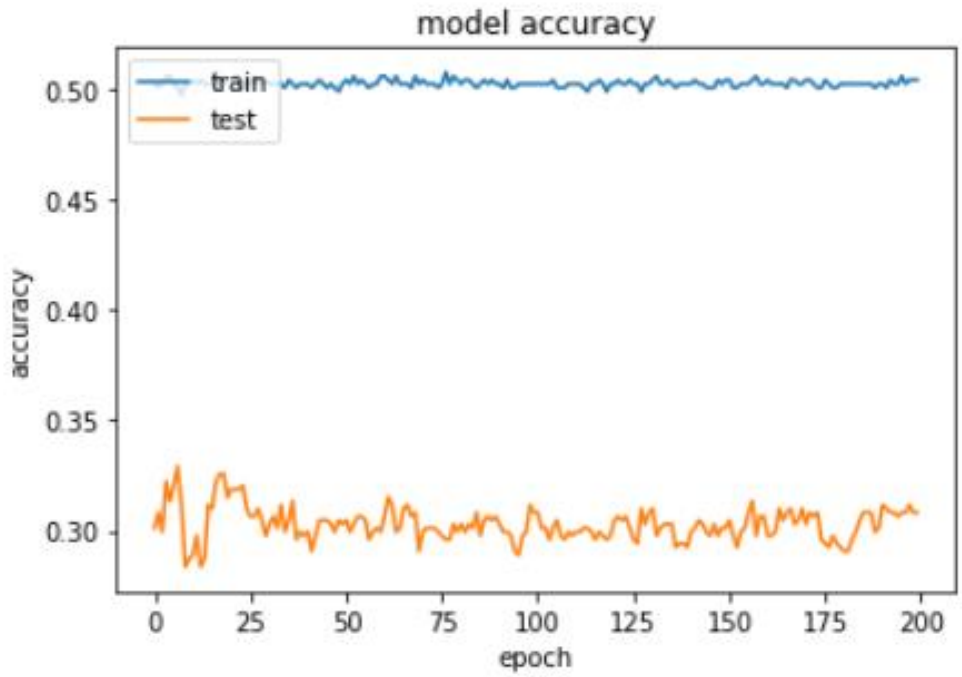
While building the MLP model we took care while choosing the activation functions as we wanted our model to predict the negative values for the sentiment score. Therefore we decided to go for the linear and tanh activation function as others eliminate values less than zero.

```
# train a regular MLP
sequence_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='float32')
embedded_sequences = embedding_layer(sequence_input)
x = Flatten()(embedded_sequences)
x = Dense(25, activation='linear')(x)
x = Dropout(0.5)(x)
x = Dense(25, activation='linear')(x)
preds = Dense(1, activation='tanh')(x)

model_mlp = Model(sequence_input, preds)
model_mlp.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['acc'])
```

```
model_mlp.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 300)               0
_________________________________________________________________
embedding_1 (Embedding)      (None, 300, 300)          819600
_________________________________________________________________
flatten_1 (Flatten)          (None, 90000)             0
_________________________________________________________________
dense_1 (Dense)              (None, 25)                2250025
_________________________________________________________________
dropout_1 (Dropout)          (None, 25)                0
_________________________________________________________________
dense_2 (Dense)              (None, 25)                650
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 26
=================================================================
Total params: 3,070,301
Trainable params: 2,250,701
Non-trainable params: 819,600
```

On running this model for 200 epochs and batch size of 32 we got training accuracy of around 50% while the validation on the test data was close to 32%



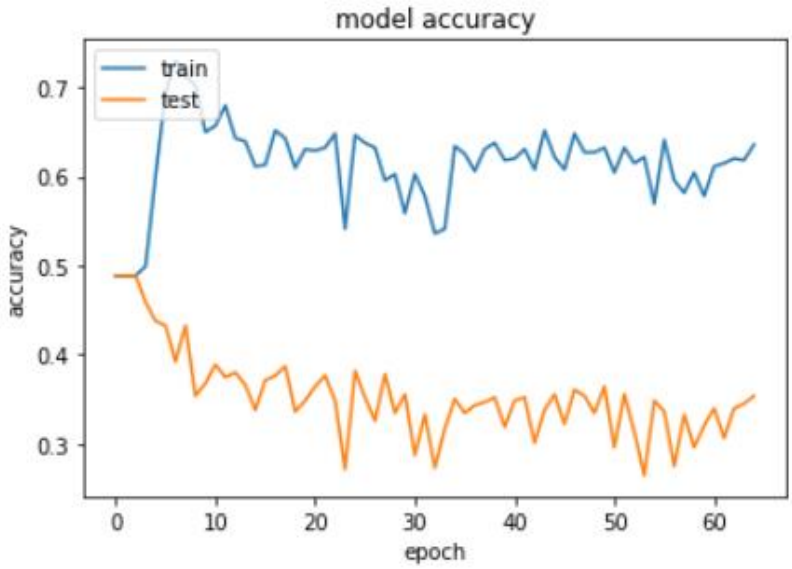
On decreasing the epoch size to 125, we could see that model was not learning anything new as the validation accuracy was stable at 30% throughout.

LSTM:

For the LSTM model, we went with tanh activation function as we wanted the model to predict negative scores. The model also included a dropout rate which was set to 15% which is lower than expected but we had to do it as the size of the data we had was too small.

```
print('Build model...')
batch_size = 32
maxlen = 80

x_train_short = pad_sequences(x_train, maxlen=maxlen)
x_test_short = pad_sequences(x_train, maxlen=maxlen)

model_lstm = Sequential()
model_lstm.add(Embedding(MAX_NB_WORDS, 128))
model_lstm.add(LSTM(128, dropout=0.15, recurrent_dropout=0.2))
model_lstm.add(Dense(1, activation='tanh'))

# try using different optimizers and different optimizer configs
model_lstm.compile(loss='binary_crossentropy',
              optimizer='RMSProp',
              metrics=['accuracy'])

print('Train...')
lstmhistory1 = model_lstm.fit(x_train_short, y_train, batch_size=batch_size, epochs=65,
          validation_data=(x_test_short, y_test))
```

The accuracy attained by the model was around 65% for the training data and around 35% for the validation data. We got better validation accuracy when using RMSProp optimizer rather than Adam.



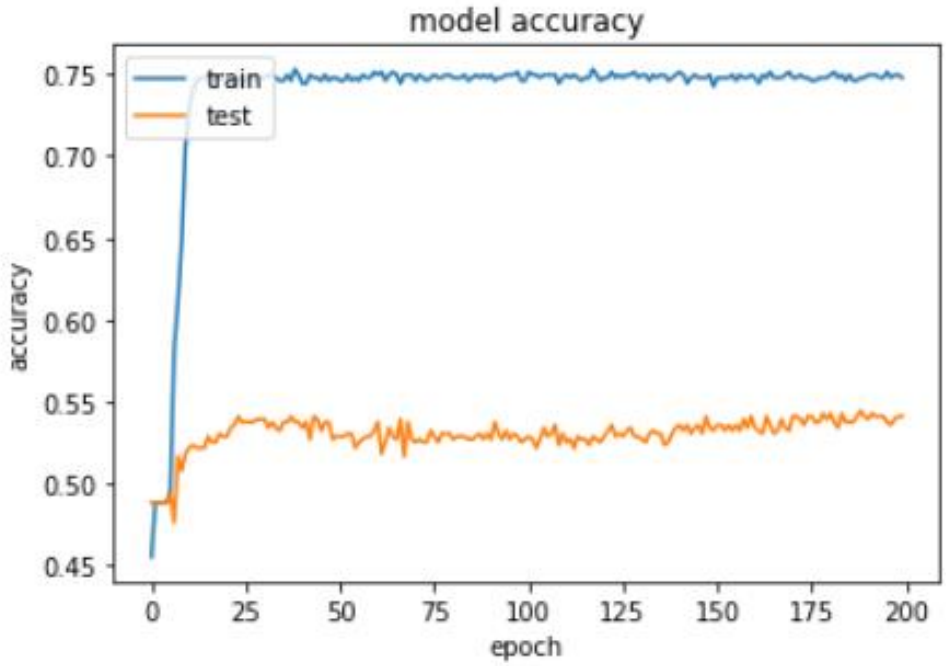
CNN :

While building CNN model we used 5 layers with linear and tanh activation functions. While increasing the epoch up to 200 gave out the best accuracy, increasing it more than that caused the accuracy to plummet. On using 200 epochs and a batch size of 128 we were able to get a testing accuracy of 75% and validation accuracy of 55%

```
# train a 1D convnet with global maxpooling
sequence_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='int32')
embedded_sequences = embedding_layer(sequence_input)
x = Conv1D(128, 5, activation='linear')(embedded_sequences)
x = MaxPooling1D(5)(x)
x = Conv1D(128, 5, activation='linear')(x)
x = MaxPooling1D(5)(x)
x = Conv1D(128, 5, activation='linear')(x)
x = MaxPooling1D(7)(x)  # modified from example since our seq len is 300
x = Flatten()(x)
x = Dense(128, activation='linear')(x)
preds = Dense(1, activation='tanh')(x)

model = Model(sequence_input, preds)
model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['acc'])
```

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_6 (InputLayer)         (None, 300)               0
_________________________________________________________________
embedding_1 (Embedding)      (None, 300, 300)          819600
_________________________________________________________________
conv1d_10 (Conv1D)           (None, 296, 128)          192128
_________________________________________________________________
max_pooling1d_10 (MaxPooling (None, 59, 128)           0
_________________________________________________________________
conv1d_11 (Conv1D)           (None, 55, 128)           82048
_________________________________________________________________
max_pooling1d_11 (MaxPooling (None, 11, 128)           0
_________________________________________________________________
conv1d_12 (Conv1D)           (None, 7, 128)            82048
_________________________________________________________________
max_pooling1d_12 (MaxPooling (None, 1, 128)            0
_________________________________________________________________
flatten_6 (Flatten)          (None, 128)               0
_________________________________________________________________
dense_17 (Dense)             (None, 128)               16512
_________________________________________________________________
```



Cosine Score Analyses:

The cosine score is based on this equation below:

$$cosine(G, P) = \frac{\sum_{i=1}^{n} G_i \times P_i}{\sqrt{\sum_{i=1}^{n} G_i^2} \times \sqrt{\sum_{i=1}^{n} P_i^2}}$$

Gi is the vector of the gold standard score. P is the vector is of the predicted value. The gold standard score is based on the sentiment score given for each subtask to provide accuracy. The cosine similarity from genism library was used to find the cosine score.

The cosine score of subtask 1 is: -0.02749754
The cosine score of subtask 2 is: 0.45

Conclusion:

As the results shown above, the best type of class to use is to use it seems like the result has shown that the MLP is the one that would be used for both subtasks. The MLP has the most accuracy for both of the subtask and also seemed tunable based on the accuracy and loss. However, this is highly surprising, since it was expected that LSTMs would be the most accurate. This is because in most language applications such as translations or handwriting deciphering, LSTMS are used. There might be several reasons for this. One is that the neutral range is needed to be studied more. It is possible that the range for the neutral is smaller than expected, leaving a much inaccurate amount of data seen.

Another reason is that the current LSTM needs to be tuned. This is unlike the just negative or positive sediment score, where there is set range that is known. Another is that there is lack of more data. Language can create a wide range moods based on millions of combinations. In order to collect them all in order to make a accurate assessment of sentiment there needs to be a large dataset. However, even if there is no problem with this, there needs to be more tuning with more parameters to increase the accuracy, even with the parameters shown not commonly used such as the range for neutrality.