ECS7001P - NN & NLP ASSIGNMENT 1: EMBEDDINGS, TEXT CLASSIFICATION, AND MACHINE TRANSLATION

Animesh Devendra Chourey – 210765551

Queen Mary University of London – March 15, 2022

Part A: Word Embeddings with Word2Vec

1. Preprocessing the training corpus

We perform basic pre-processing steps like removing the digits, stopwords and special characters, converting the words into lower cases. Also the sentences having less than 3 words are discarded.

Output:

```
The new length of the preprocessed output:-  13651
```

2. Creating the corpus vocabulary and preparing the dataset

Two dictionaries are created -

(a) *word2idx* : It stores word as the key and unique integer as value in {key,value} pair.

(b) *idx2word* : It stores unique integer as the key and word as value in {key,value} pair.

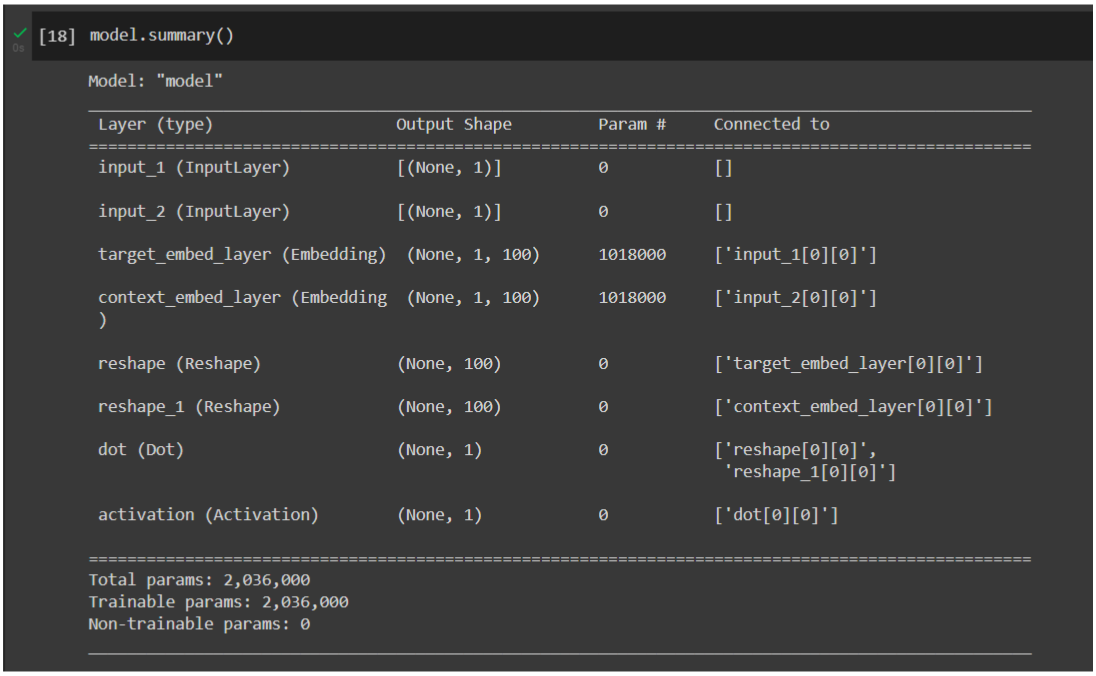
A list *sents_as_ids* is created which stores sentences as a list of integers for every corresponding word in the sentences.

Output :

```
Number of unique words: 10180

Sample word2idx:  [('sense', 0), ('sensibility', 1), ('jane', 2),
                   ('austen', 3), ('the', 4), ('family', 5), ('dashwood', 6),
                   ('long', 7), ('settled', 8), ('sussex', 9)]

Sample idx2word: [(0, 'sense'), (1, 'sensibility'), (2, 'jane'), (3, 'austen'),
                  (4, 'the'), (5, 'family'), (6, 'dashwood'), (7, 'long'),
                  (8, 'settled'), (9, 'sussex')]

Sample sents_as_id: [[0, 1, 2, 3], [41, 72, 6, 201, 619, 35, 620, 296, 621]]
```

3. Building the skip-gram neural network architecture

The code in the target embedding is reused to create context embeddings. The only thing changed there is that inout layer for context is passed to the context embedding layer. While creating the output layer for this architecture we used sigmoid function as activation function because we need a binary output. In the end, we use *mean squared error* as loss function and *rmsprop* as an optimizer.



```
[18] model.summary()

Model: "model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_1 (InputLayer)           [(None, 1)]          0           []

 input_2 (InputLayer)           [(None, 1)]          0           []

 target_embed_layer (Embedding)  (None, 1, 100)      1018000     ['input_1[0][0]']

 context_embed_layer (Embedding  (None, 1, 100)      1018000     ['input_2[0][0]']
 )

 reshape (Reshape)              (None, 100)          0           ['target_embed_layer[0][0]']

 reshape_1 (Reshape)            (None, 100)          0           ['context_embed_layer[0][0]']

 dot (Dot)                      (None, 1)            0           ['reshape[0][0]',
                                                                  'reshape_1[0][0]']

 activation (Activation)        (None, 1)            0           ['dot[0][0]']

==================================================================================================
Total params: 2,036,000
Trainable params: 2,036,000
Non-trainable params: 0
__________________________________________________________________________________________________
```

4. Training the models

1. *What would the inputs and outputs to the model be?*

   Ans. Inputs are the one-hot vector representation of the words. Outputs of the model is the vector representation of the probability of each word being chosen as the next word.

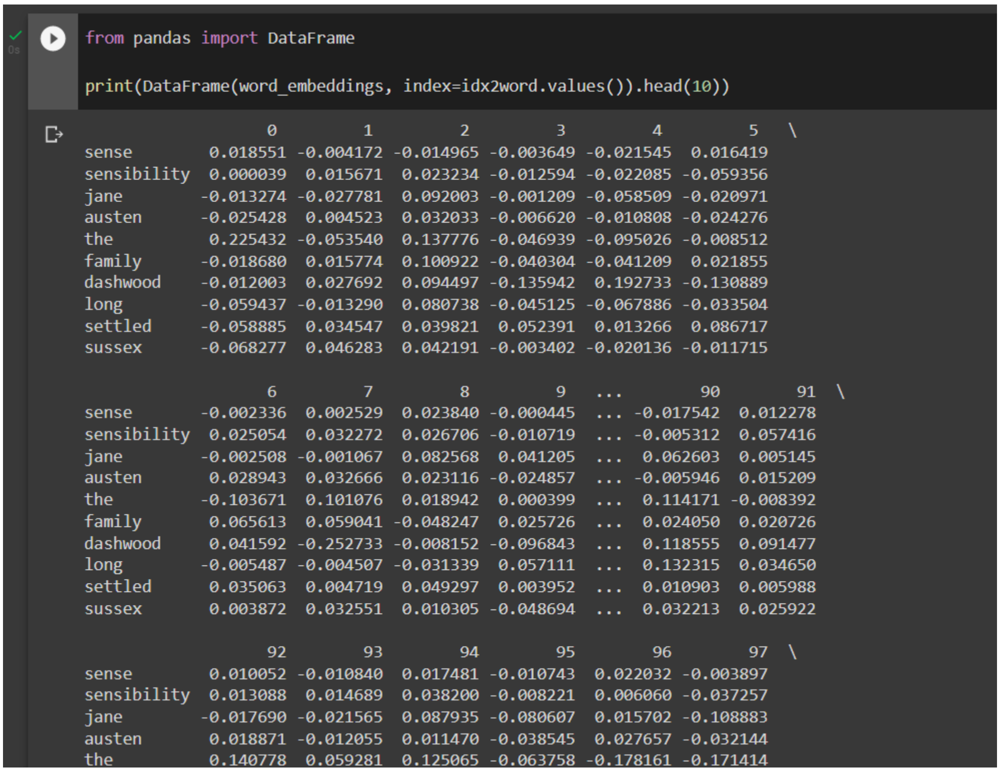
2. *How would you use the Keras framework to create this architecture?*

   Ans. To create the model using keras framework we would create an input layer having two inputs each for the target word and context word. Embeddings for both the inputs after which we need a reshpaing layer for the embeddings. Both the embeddings need to be multiplied using the dot product. Finally an output layer having activation as softmax function.

3. *What are the reasons this training approach is considered inefficient?*

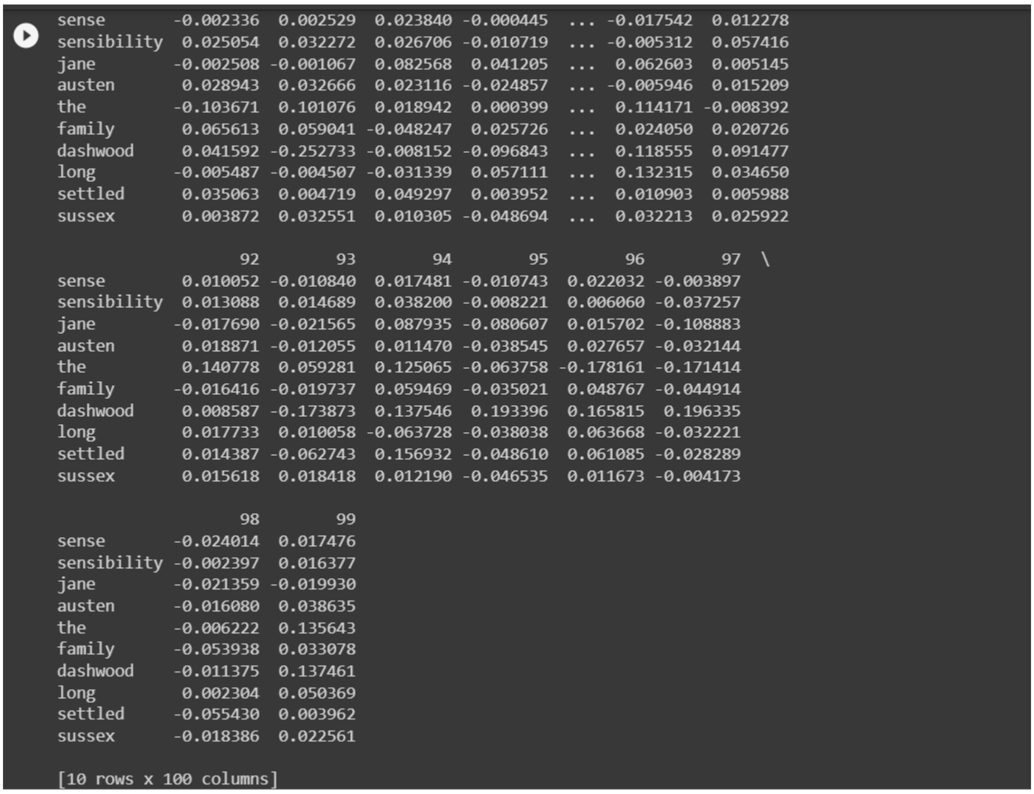
   Ans. This training approach can be considered inefficient because it is computationally costly.The words represented here do not capture the contextual meaning because of which it does not capture semantic relationship very well. Also, the probability of choosing the common word is higher as compared to a rarer word getting selected which can be considered inefficient. We need a dataset that is specifically customized and created for word2vec.

5. Getting the word embeddings



```
from pandas import DataFrame

print(DataFrame(word_embeddings, index=idx2word.values()).head(10))
```

```
                    0         1         2         3         4         5  \
sense        0.018551 -0.004172 -0.014965 -0.003649 -0.021545  0.016419   
sensibility  0.000039  0.015671  0.023234 -0.012594 -0.022085 -0.059356   
jane        -0.013274 -0.027781  0.092003 -0.001209 -0.058509 -0.020971   
austen      -0.025428  0.004523  0.032033 -0.006620 -0.010808 -0.024276   
the          0.225432 -0.053540  0.137776 -0.046939 -0.095026 -0.008512   
family      -0.018680  0.015774  0.100922 -0.040304 -0.041209  0.021855   
dashwood    -0.012003  0.027692  0.094497 -0.135942  0.192733 -0.130889   
long        -0.059437 -0.013290  0.080738 -0.045125 -0.067886 -0.033504   
settled     -0.058885  0.034547  0.039821  0.052391  0.013266  0.086717   
sussex      -0.068277  0.046283  0.042191 -0.003402 -0.020136 -0.011715   

                    6         7         8         9  ...        90        91  \
sense       -0.002336  0.002529  0.023840 -0.000445  ... -0.017542  0.012278   
sensibility  0.025054  0.032272  0.026706 -0.010719  ... -0.005312  0.057416   
jane        -0.002508 -0.001067  0.082568  0.041205  ...  0.062603  0.005145   
austen       0.028943  0.032666  0.023116 -0.024857  ... -0.005946  0.015209   
the         -0.103671  0.101076  0.018942  0.000399  ...  0.114171 -0.008392   
family       0.065613  0.059041 -0.048247  0.025726  ...  0.024050  0.020726   
dashwood     0.041592 -0.252733 -0.008152 -0.096843  ...  0.118555  0.091477   
long        -0.005487 -0.004507 -0.031339  0.057111  ...  0.132315  0.034650   
settled      0.035063  0.004719  0.049297  0.003952  ...  0.010903  0.005988   
sussex       0.003872  0.032551  0.010305 -0.048694  ...  0.032213  0.025922   

                   92        93        94        95        96        97  \
sense        0.010052 -0.010840  0.017481 -0.010743  0.022032 -0.003897   
sensibility  0.013088  0.014689  0.038200 -0.008221  0.006060 -0.037257   
jane        -0.017690 -0.021565  0.087935 -0.080607  0.015702 -0.108883   
austen       0.018871 -0.012055  0.011470 -0.038545  0.027657 -0.032144   
the          0.140778  0.059281  0.125065 -0.063758 -0.178161 -0.171414   
```

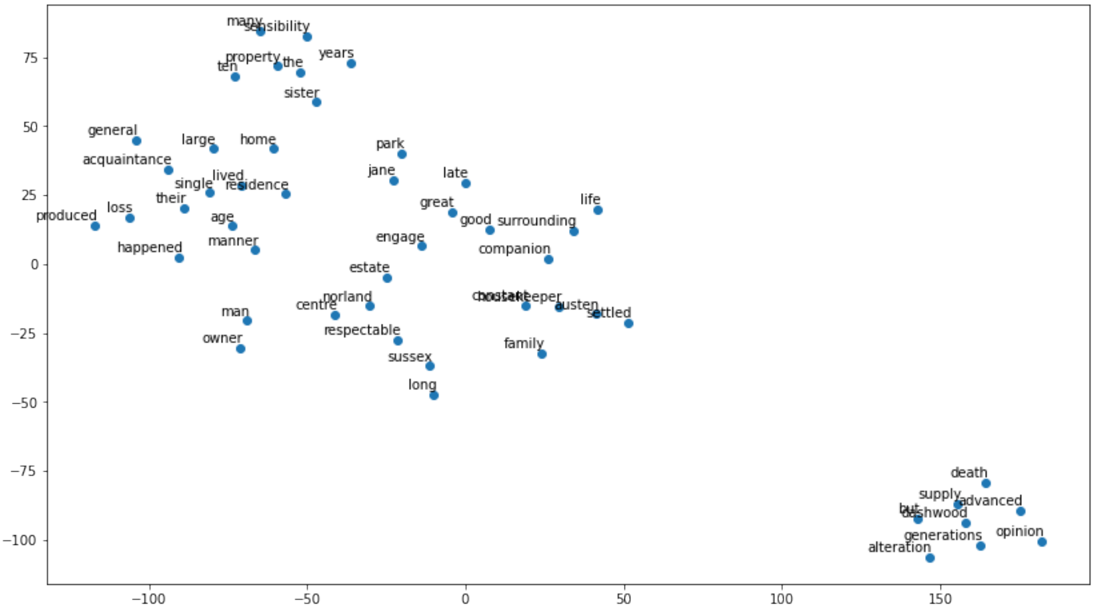


```
sense       -0.002336  0.002529  0.023840 -0.000445  ... -0.017542  0.012278   
sensibility  0.025054  0.032272  0.026706 -0.010719  ... -0.005312  0.057416   
jane        -0.002508 -0.001067  0.082568  0.041205  ...  0.062603  0.005145   
austen       0.028943  0.032666  0.023116 -0.024857  ... -0.005946  0.015209   
the         -0.103671  0.101076  0.018942  0.000399  ...  0.114171 -0.008392   
family       0.065613  0.059041 -0.048247  0.025726  ...  0.024050  0.020726   
dashwood     0.041592 -0.252733 -0.008152 -0.096843  ...  0.118555  0.091477   
long        -0.005487 -0.004507 -0.031339  0.057111  ...  0.132315  0.034650   
settled      0.035063  0.004719  0.049297  0.003952  ...  0.010903  0.005988   
sussex       0.003872  0.032551  0.010305 -0.048694  ...  0.032213  0.025922   

                   92        93        94        95        96        97  \
sense        0.010052 -0.010840  0.017481 -0.010743  0.022032 -0.003897   
sensibility  0.013088  0.014689  0.038200 -0.008221  0.006060 -0.037257   
jane        -0.017690 -0.021565  0.087935 -0.080607  0.015702 -0.108883   
austen       0.018871 -0.012055  0.011470 -0.038545  0.027657 -0.032144   
the          0.140778  0.059281  0.125065 -0.063758 -0.178161 -0.171414   
```

```
family      -0.016416 -0.019737  0.059469 -0.035021  0.048767 -0.044914   
dashwood     0.008587 -0.173873  0.137546  0.193396  0.165815  0.196335   
long         0.017733  0.010058 -0.063728 -0.038038  0.063668 -0.032221   
settled      0.014387 -0.062743  0.156932 -0.048610  0.061085 -0.028289   
sussex       0.015618  0.018418  0.012190 -0.046535  0.011673 -0.004173   

                   98        99  
sense       -0.024014  0.017476  
sensibility -0.002397  0.016377  
jane        -0.021359 -0.019930  
austen      -0.016080  0.038635  
the         -0.006222  0.135643  
family      -0.053938  0.033078  
dashwood    -0.011375  0.137461  
long         0.002304  0.050369  
settled     -0.055430  0.003962  
sussex      -0.018386  0.022561  

[10 rows x 100 columns]
```

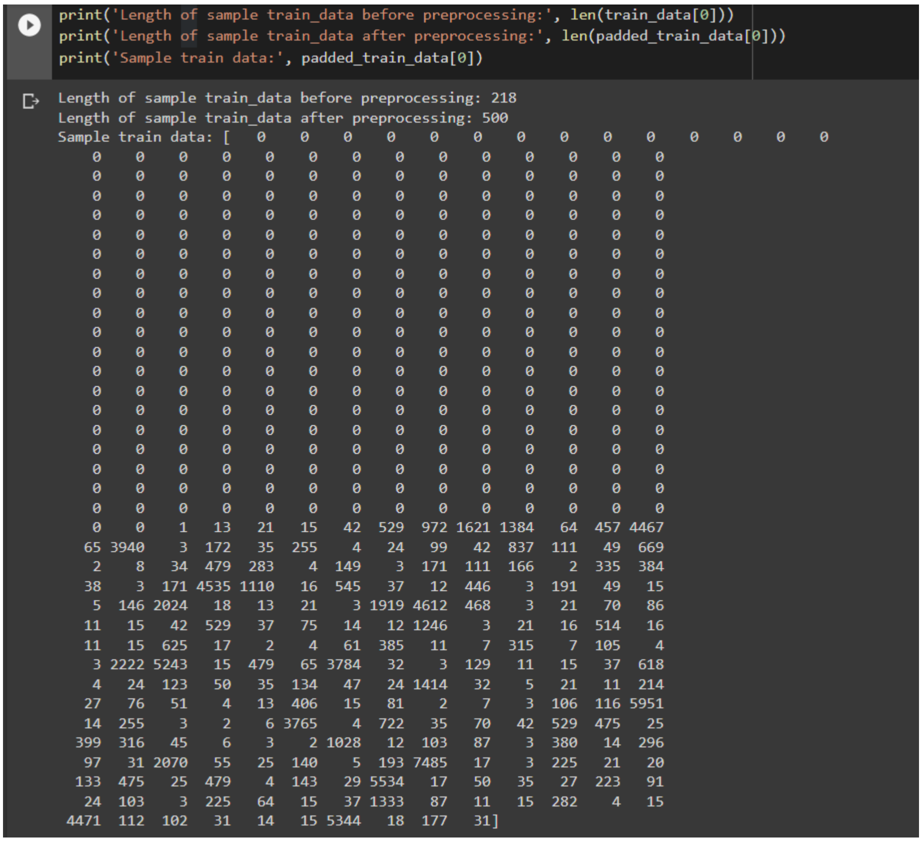
6. Exploring and visualizing your word embeddings using t-SNE

Firstly corresponding to each word, a row is extracted and these rows have similarity measure between the word in consideration and the word in each column. From this list each word is extracted using idx2word. Now a dictionary is created such that the key represents the extracted similar word and value represents the similarity measure. Finally, the dictionary is sorted in the descending order according to the similarity measure and the top 5 items are displayed.



Part B: Using LSTMs for Text Classification

1. Section 2, Readying the inputs for the LSTM



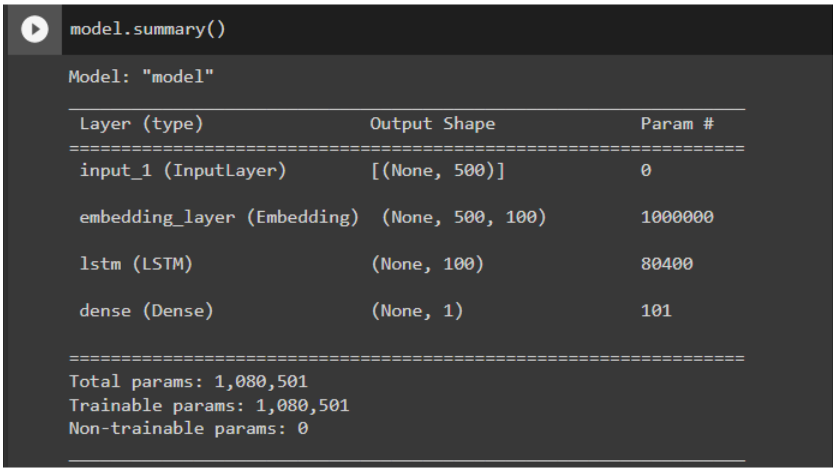
```
print('Length of sample train_data before preprocessing:', len(train_data[0]))
print('Length of sample train_data after preprocessing:', len(padded_train_data[0]))
print('Sample train data:', padded_train_data[0])
```

```
Length of sample train_data before preprocessing: 218
Length of sample train_data after preprocessing: 500
Sample train data: [   0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    1   13   21   15   42  529  972 1621 1384   64  457 4467
   65 3940    3  172   35  255    4   24   99   42  837  111   49  669
    2    8   34  479  283    4  149    3  171  111  166    2  335  384
   38    3  171 4535 1110   16  545   37   12  446    3  191   49   15
    5  146 2024   18   13   21    3 1919 4612  468    3   21   70   86
   11   15   42  529   37   75   14   12 1246    3   21   16  514   16
   11   15  625   17    2    4   61  385   11    7  315    7  105    4
    3 2222 5243   15  479   65 3784   32    3  129   11   15   37  618
    4   24  123   50   35  134   47   24 1414   32    5   21   11  214
   27   76   51    4   13  406   15   81    2    7    3  106  116 5951
   14  255    3    2    6 3765    4  722   35   70   42  529  475   25
  399  316   45    6    3    2 1028   12  103   87    3  380   14  296
   97   31 2070   55   25  140    5  193 7485   17    3  225   21   20
  133  475   25  479    4  143   29 5534   17   50   35   27  223   91
   24  103    3  225   64   15   37 1333   87   11   15  282    4   15
 4471  112  102   31   14   15 5344   18  177   31]
```

The training and the test data is padded by pre-padding them with zeros so that each vector is of same length i.e. 500.

2. Building the model

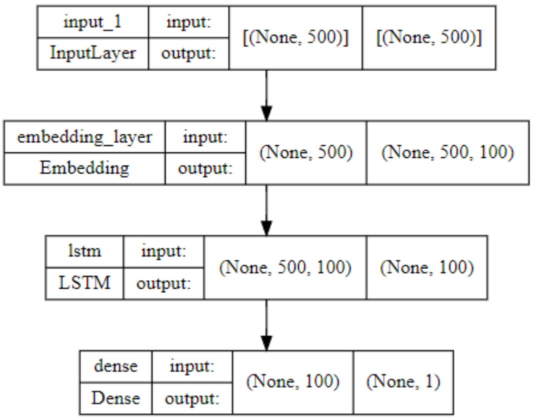
The input layer takes the padded training samples derived previously. The first hidden layer is the embedding layer which takes in the input length as 500 which is corresponding to the length of each training sequence and the input layer dimension will be vocabulary size. We want the words to be embedded in a vector space of size 100. After this LSTM layer is created and then a fully connected layer is added which uses sigmoid function as we expect a binary output.



```
model.summary()
```

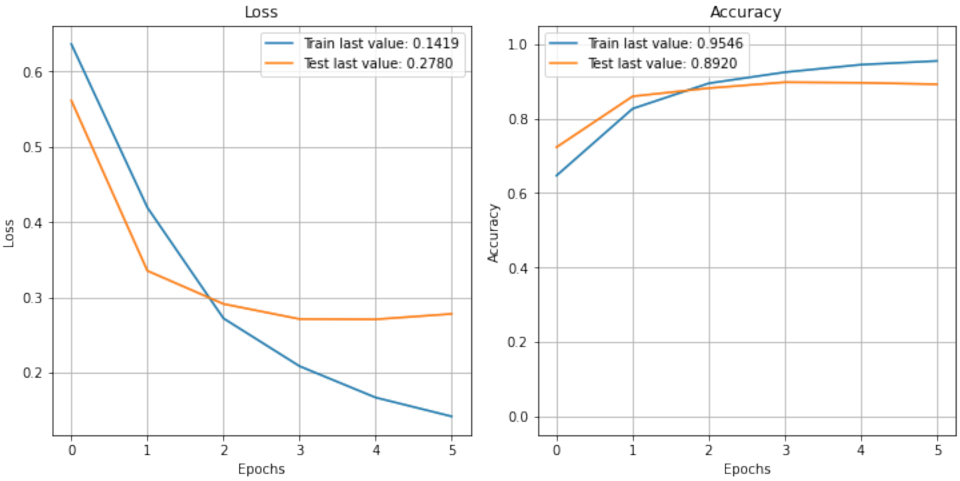
```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_1 (InputLayer)        [(None, 500)]             0         
                                                                 
 embedding_layer (Embedding)  (None, 500, 100)         1000000   
                                                                 
 lstm (LSTM)                 (None, 100)               80400     
                                                                 
 dense (Dense)               (None, 1)                 101       
                                                                 
=================================================================
Total params: 1,080,501
Trainable params: 1,080,501
Non-trainable params: 0
_________________________________________________________________
```

Architecture is as follows:



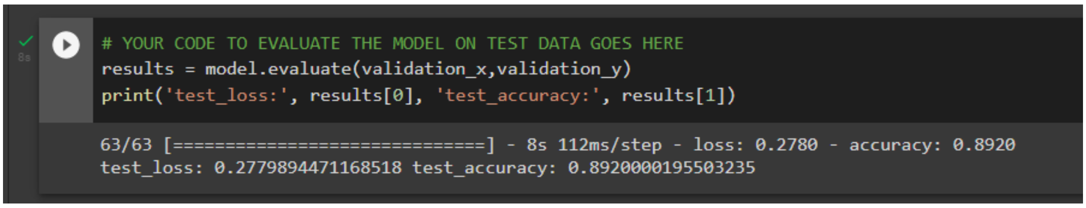
3. Section 4, training the model

During the training we stop whenever the performance on the validation dataset start to degrade. This is known as *Early Stopping*. From the accuracy plot we can see that the accuracy on the test set starts going down after the first epoch only. This means that our model is overfitting which we have to overcome. Therefore, in this situation the model should only be trained only for one epoch.



4. Evaluating the model on the test data

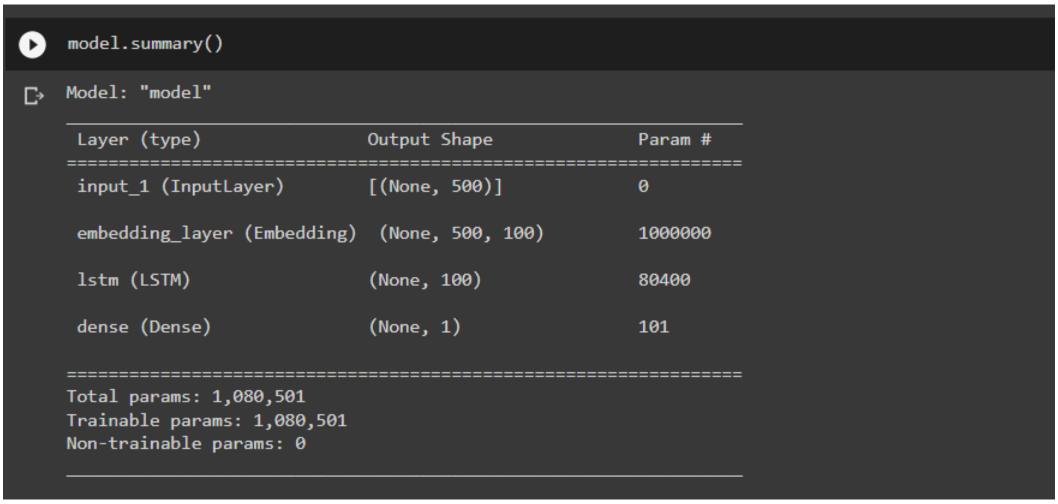
The model has achieved an accuracy of **89.2%** and loss of **27.7%** on the test dataset.



```
# YOUR CODE TO EVALUATE THE MODEL ON TEST DATA GOES HERE
results = model.evaluate(validation_x,validation_y)
print('test_loss:', results[0], 'test_accuracy:', results[1])
```

```
63/63 [==============================] - 8s 112ms/step - loss: 0.2780 - accuracy: 0.8920
test_loss: 0.2779894471168518 test_accuracy: 0.8920000195503235
```

5. Section 6, extracting the word embeddings



```
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 500)]             0

 embedding_layer (Embedding)  (None, 500, 100)         1000000

 lstm (LSTM)                 (None, 100)               80400

 dense (Dense)               (None, 1)                 101

=================================================================
Total params: 1,080,501
Trainable params: 1,080,501
Non-trainable params: 0
_________________________________________________________________
```

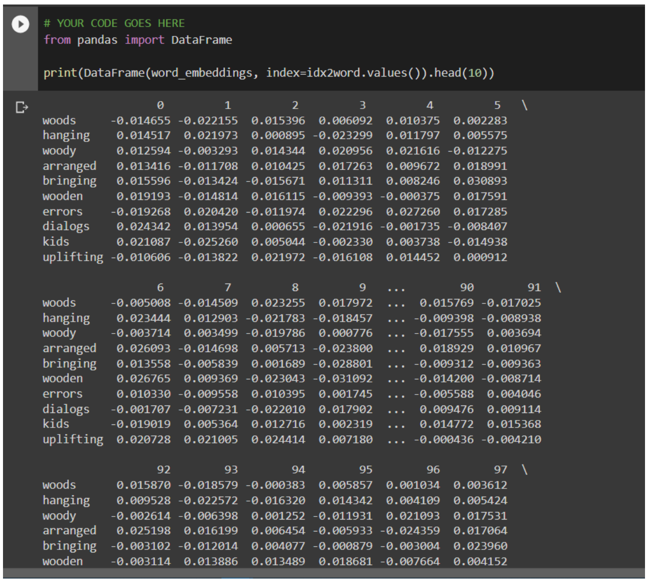
6. Visualizing the reviews

```
<START> this film was just brilliant casting location scenery story direction
everyone's really suited the part they played and you could just imagine
being there robert <UNK> is an amazing actor and now the same being director
<UNK> father came from the same scottish island as myself so i loved the fact
there was a real connection with this film the witty remarks throughout the
film were great it was just brilliant so much that i bought the film as soon
as it was released for <UNK> and would recommend it to everyone to watch and
the fly fishing was amazing really cried at the end it was so sad and you
know what they say if you cry at a film it must have been good and this
definitely was also <UNK> to the two little boy's that played the <UNK> of
```

```
norman and paul they were just brilliant children are often left out of the
<UNK> list i think because the stars that play them all grown up are such a
big profile for the whole film but these children are amazing and should be
praised for what they have done don't you think the whole story was so lovely
because it was true and was someone's life after all that was shared with us all
```

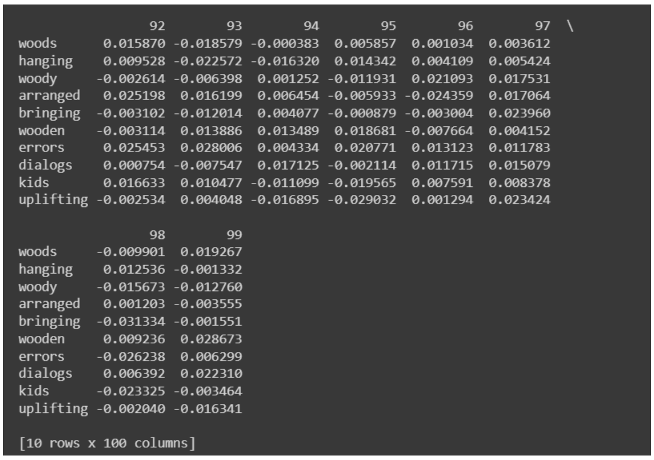
7. Visualizing the word embeddings

Word embedding for the first 10 words:



```
# YOUR CODE GOES HERE
from pandas import DataFrame

print(DataFrame(word_embeddings, index=idx2word.values()).head(10))
```

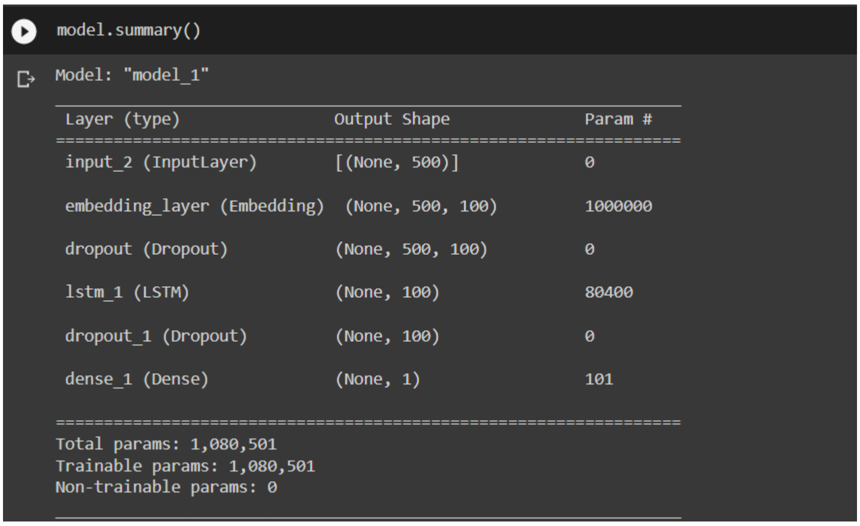
```
                  0         1         2         3         4         5  \
woods     -0.014655 -0.022155  0.015396  0.006092  0.010375  0.002283   
hanging    0.014517  0.021973  0.000895 -0.023299  0.011797  0.005575   
woody      0.012594 -0.003293  0.014344  0.020956  0.021616 -0.012275   
arranged   0.013416 -0.011708  0.010425  0.017263  0.009672  0.018991   
bringing   0.015596 -0.013424 -0.015671  0.011311  0.008246  0.030893   
wooden     0.019193 -0.014814  0.016115 -0.009393 -0.000375  0.017591   
errors    -0.019268  0.020420 -0.011974  0.022296  0.027260  0.017285   
dialogs    0.024342  0.013954  0.000655 -0.021916 -0.001735 -0.008407   
kids       0.021087 -0.025260  0.005044 -0.002330  0.003738 -0.014938   
uplifting -0.010606 -0.013822  0.021972 -0.016108  0.014452  0.000912   

                  6         7         8         9  ...        90        91  \
woods     -0.005008 -0.014509  0.023255  0.017972  ...  0.015769 -0.017025   
hanging    0.023444  0.012903 -0.021783 -0.018457  ... -0.009398 -0.008938   
woody     -0.003714  0.003499 -0.019786  0.000776  ... -0.017555  0.003694   
arranged   0.026093 -0.014698  0.005713 -0.023800  ...  0.018929  0.010967   
bringing   0.013558 -0.005839  0.001689 -0.028801  ... -0.009312 -0.009363   
wooden     0.026765  0.009369 -0.023043 -0.031092  ... -0.014200 -0.008714   
errors     0.010330 -0.009558  0.010395  0.001745  ... -0.005588  0.004046   
dialogs   -0.001707 -0.007231 -0.022010  0.017902  ...  0.009476  0.009114   
kids      -0.019019  0.005364  0.012716  0.002319  ...  0.014772  0.015368   
uplifting  0.020728  0.021005  0.024414  0.007180  ... -0.000436 -0.004210   

                 92        93        94        95        96        97  \
woods      0.015870 -0.018579 -0.000383  0.005857  0.001034  0.003612   
hanging    0.009528 -0.022572 -0.016320  0.014342  0.004109  0.005424   
woody     -0.002614 -0.006398  0.001252 -0.011931  0.021093  0.017531   
arranged   0.025198  0.016199  0.006454 -0.005933 -0.024359  0.017064   
bringing  -0.003102 -0.012014  0.004077 -0.000879 -0.003004  0.023960   
wooden    -0.003114  0.013886  0.013489  0.018681 -0.007664  0.004152   
```



```
                 92        93        94        95        96        97  \
woods      0.015870 -0.018579 -0.000383  0.005857  0.001034  0.003612   
hanging    0.009528 -0.022572 -0.016320  0.014342  0.004109  0.005424   
woody     -0.002614 -0.006398  0.001252 -0.011931  0.021093  0.017531   
arranged   0.025198  0.016199  0.006454 -0.005933 -0.024359  0.017064   
bringing  -0.003102 -0.012014  0.004077 -0.000879 -0.003004  0.023960   
wooden    -0.003114  0.013886  0.013489  0.018681 -0.007664  0.004152   
errors     0.025453  0.028006  0.004334  0.020771  0.013123  0.011783   
dialogs    0.000754 -0.007547  0.017125 -0.002114  0.011715  0.015079   
kids       0.016633  0.010477 -0.011099 -0.019565  0.007591  0.008378   
uplifting -0.002534  0.004048 -0.016895 -0.029032  0.001294  0.023424   

                 98        99  
woods     -0.009901  0.019267  
hanging    0.012536 -0.001332  
woody     -0.015673 -0.012760  
arranged   0.001203 -0.003555  
bringing  -0.031334 -0.001551  
wooden     0.009236  0.028673  
errors    -0.026238  0.006299  
dialogs    0.006392  0.022310  
kids      -0.023325 -0.003464  
uplifting -0.002040 -0.016341  

[10 rows x 100 columns]
```

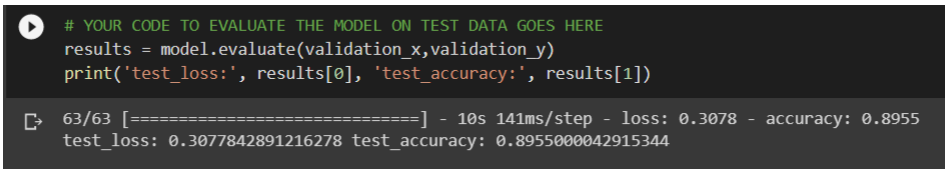
8. Section 9

1. *Create a new model that is a copy of the model step 3. To this new model,add two dropout layers, one between the embedding layer and the LSTM layer and another between the LSTM layer and the output layer. Repeat steps 4 and 5 for this model. What do you observe?*



```
model.summary()
```

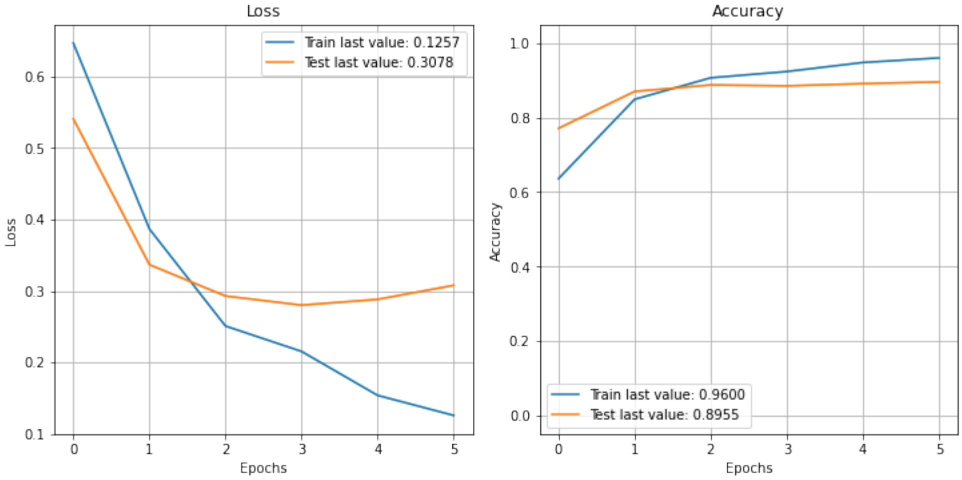
```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_2 (InputLayer)        [(None, 500)]             0         
                                                                 
 embedding_layer (Embedding)  (None, 500, 100)         1000000   
                                                                 
 dropout (Dropout)           (None, 500, 100)          0         
                                                                 
 lstm_1 (LSTM)               (None, 100)               80400     
                                                                 
 dropout_1 (Dropout)         (None, 100)               0         
                                                                 
 dense_1 (Dense)             (None, 1)                 101       
                                                                 
=================================================================
Total params: 1,080,501
Trainable params: 1,080,501
Non-trainable params: 0
_________________________________________________________________
```



```
# YOUR CODE TO EVALUATE THE MODEL ON TEST DATA GOES HERE
results = model.evaluate(validation_x,validation_y)
print('test_loss:', results[0], 'test_accuracy:', results[1])
```

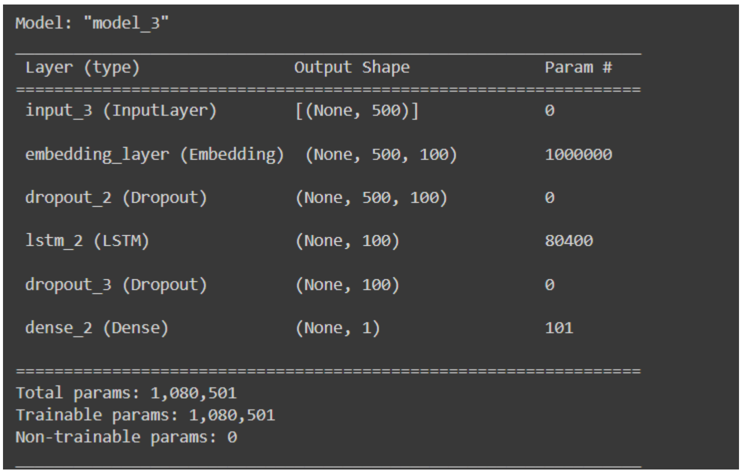
```
63/63 [==============================] - 10s 141ms/step - loss: 0.3078 - accuracy: 0.8955
test_loss: 0.3077842891216278 test_accuracy: 0.8955000042915344
```

When comparing with the previous model we can see that the accuracy has slightly increased from **89.2%** to **89.5%**. However, the loss on the test data has also increased from **27.7%** to **30.7%**. As we can see from the plots below, the learning of this model happens till the second epoch, whereas in the case of previous model learning happens only till epoch 1. Here, after epoch 2 we can say that the model starts overfitting. After adding dropouts we can say that the model converges slightly better here.

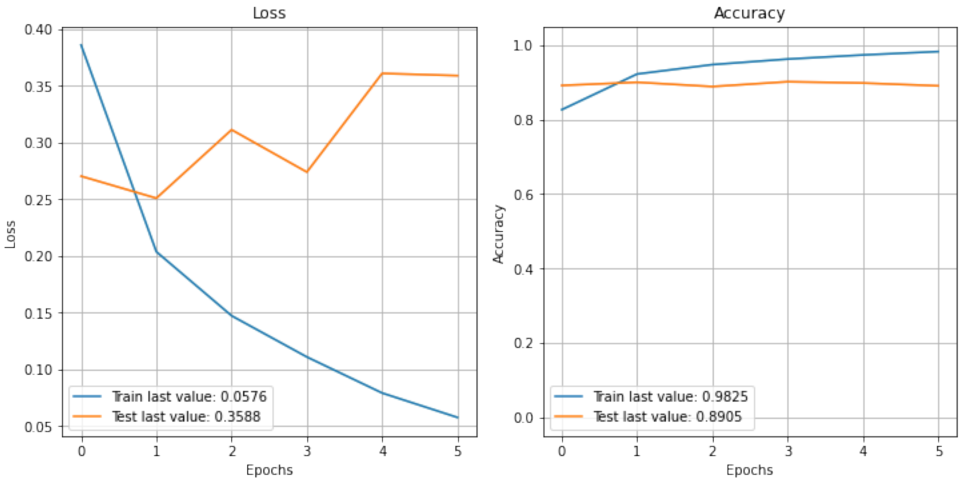


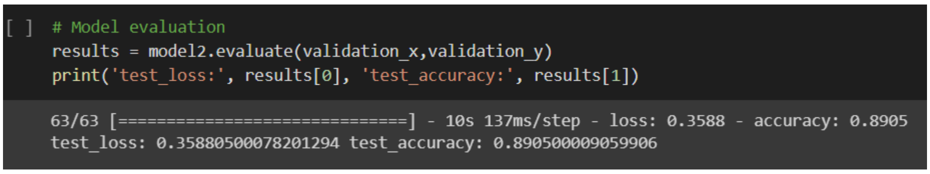
2. *Experiment with training the model with batch sizes of 1, 32, len(training_data). What do you observe?*

    (a) *Training the model with batch size of 1:*



```
Model: "model_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 500)]             0

 embedding_layer (Embedding)  (None, 500, 100)         1000000

 dropout_2 (Dropout)         (None, 500, 100)          0

 lstm_2 (LSTM)               (None, 100)               80400

 dropout_3 (Dropout)         (None, 100)               0

 dense_2 (Dense)             (None, 1)                 101

=================================================================
Total params: 1,080,501
Trainable params: 1,080,501
Non-trainable params: 0
_________________________________________________________________
```

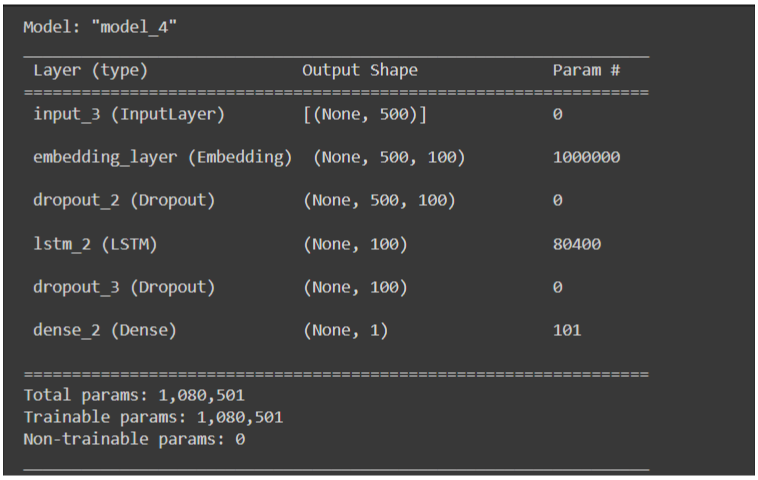




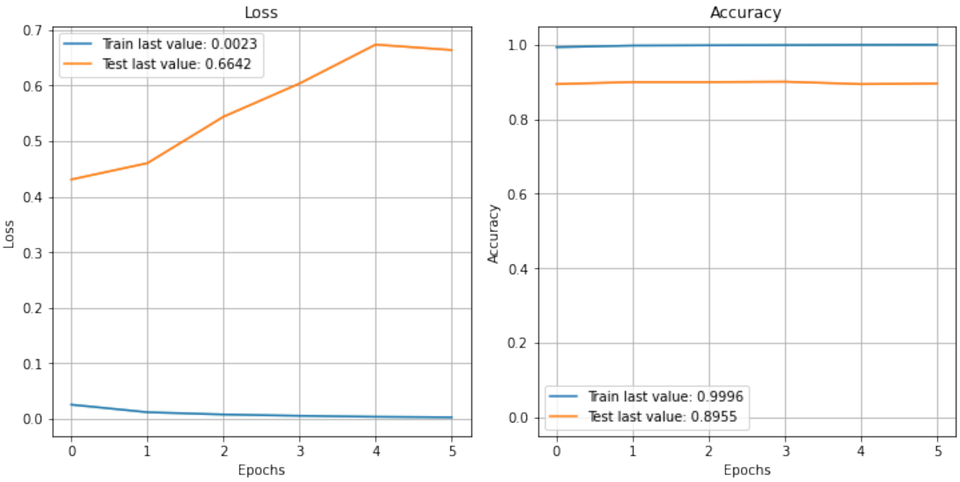
```
[ ]  # Model evaluation
     results = model2.evaluate(validation_x,validation_y)
     print('test_loss:', results[0], 'test_accuracy:', results[1])

     63/63 [==============================] - 10s 137ms/step - loss: 0.3588 - accuracy: 0.8905
     test_loss: 0.35880500078201294 test_accuracy: 0.890500009059906
```

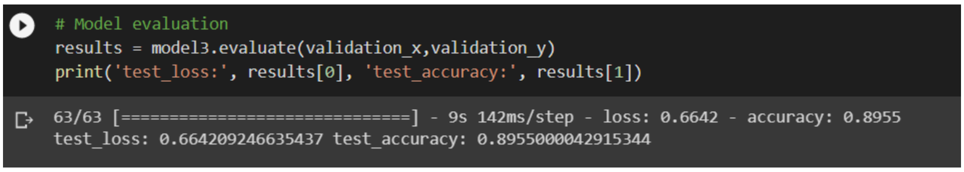
When using the batch size of 1, the parameters are being updated with every iteration. Given how large the number of trainable parameters are, the computational time rockets up exponentially. Training model with batch size=1 takes around 7 hrs to run which is an extremely high overtime head and the model is also overfitting on the dataset and is not able to generalize well on the test data as we can see from the graph above. Since the batch size is selected as 1 the whole dataset is passed to the model at once. Therefore the model takes so much time to run as it has to process whole dataset all at once. Thus for these reasons, this should not be the right way to train a model.

(b) *Training the model with batch size of 32:*



```
Model: "model_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 500)]             0

 embedding_layer (Embedding)  (None, 500, 100)         1000000

 dropout_2 (Dropout)         (None, 500, 100)          0

 lstm_2 (LSTM)               (None, 100)               80400

 dropout_3 (Dropout)         (None, 100)               0

 dense_2 (Dense)             (None, 1)                 101

=================================================================
Total params: 1,080,501
Trainable params: 1,080,501
Non-trainable params: 0
_________________________________________________________________
```





```
# Model evaluation
results = model3.evaluate(validation_x,validation_y)
print('test_loss:', results[0], 'test_accuracy:', results[1])
```

```
63/63 [==============================] - 9s 142ms/step - loss: 0.6642 - accuracy: 0.8955
test_loss: 0.664209246635437 test_accuracy: 0.8955000042915344
```

The results have not much improved in this case also. Training with batch size of 32 also takes plenty time although not as much as compared to batch size=1. Surprisingly, the train and test accuracy has remained almost similar and has only changed that is barely noticeable. The model is not improving and here in this case also the model has been overfitting and generalizing well as we can see from the test loss from the graph above.

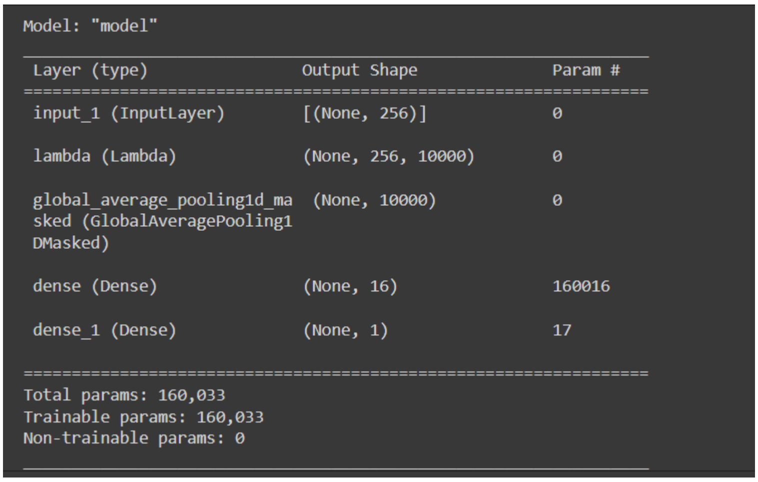
(c) *Training the model with batch size of len(training_data):*

Training the model with batch size equal to the size of training data is practically impossible. The model is not able to train even for one epoch as the memory usage for creating such large batch is extremely intensive. The model crashes on its first epoch.

Part C: Comparing Classification Models

1. Build a neural network classifier using one-hot word vectors (Model 1), and train and evaluate it

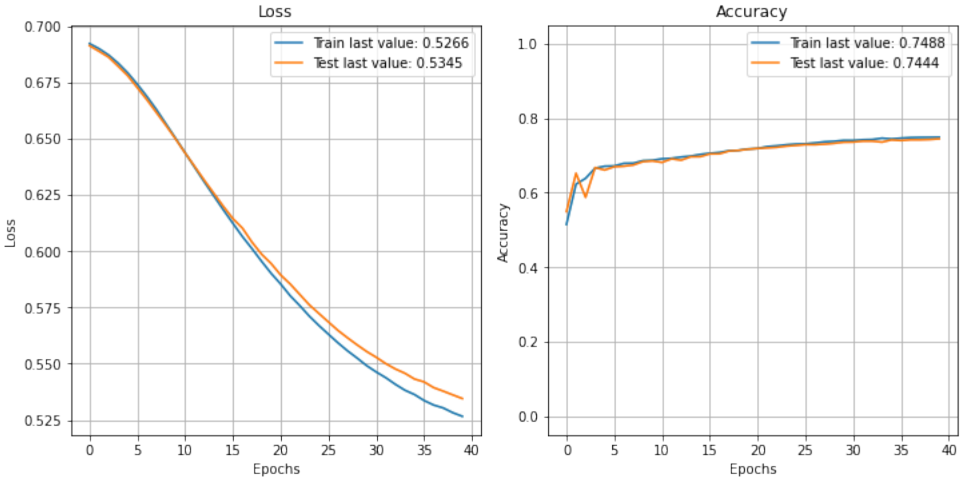
Firstly we add an input layer of shape 256 corresponding to the maximum length of padded sequence. Next we add a layer which represents one-hot representation that uses 10000 as vocab size and 100 as embedding size. Then we add a pooling layer after which a dense layer is added. Finally, an output layer with sigmoid activation function is used as we expect a binary output.



```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 256)]             0

 lambda (Lambda)             (None, 256, 10000)        0

 global_average_pooling1d_ma  (None, 10000)            0
 sked (GlobalAveragePooling1
 DMasked)

 dense (Dense)               (None, 16)                160016

 dense_1 (Dense)             (None, 1)                 17

=================================================================
Total params: 160,033
Trainable params: 160,033
Non-trainable params: 0
_________________________________________________________________
```

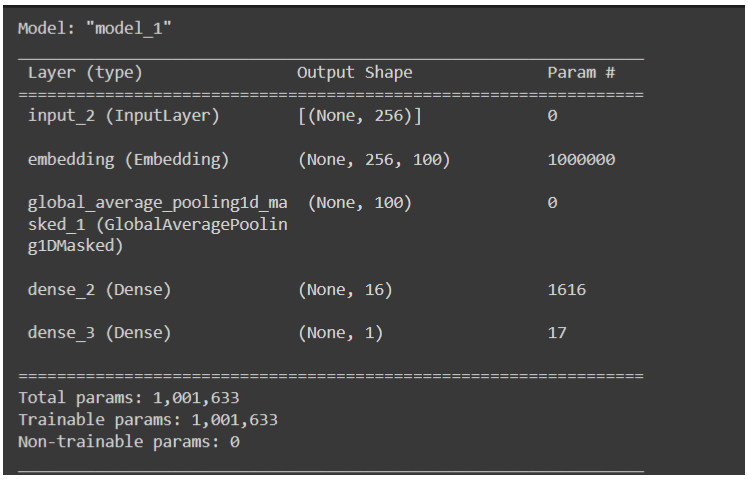
The model is compiled using adam optimizer and binary cross entropy as loss function.

As seen in the plot below the graph achieves an accuracy of **74%** on the test set.



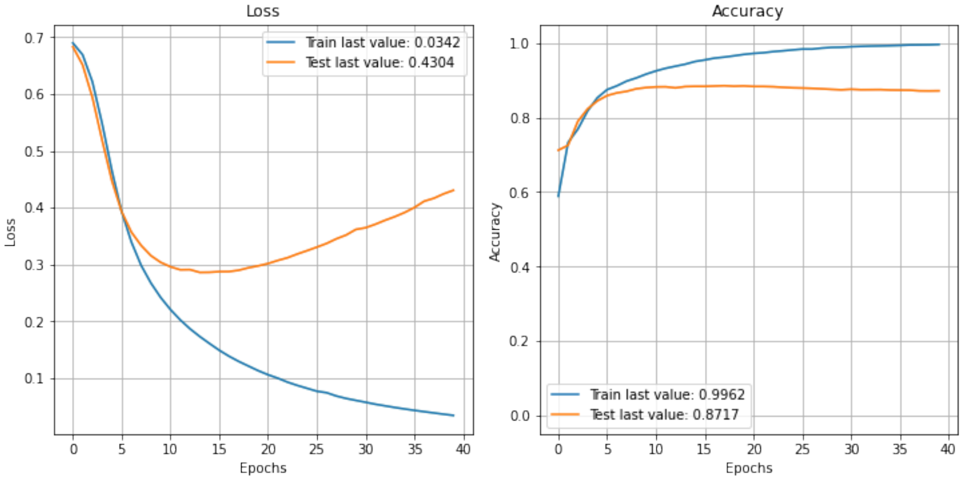
2. Modify your model to use a word embedding layer instead of one-hot vectors (Model 2), and to learn the values of these word embedding vectors along with the model

Same model is used as before, with the only exception being that instead of using one-hot representation for words embeddings is used created by the embedding layer. Same parameters are passed as before to every layer. Embedding layer also has similar parameters passed as we did with one-hot layer before.



```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 256)]             0

 embedding (Embedding)       (None, 256, 100)          1000000

 global_average_pooling1d_ma  (None, 100)              0
 sked_1 (GlobalAveragePoolin
 g1DMasked)

 dense_2 (Dense)             (None, 16)                1616

 dense_3 (Dense)             (None, 1)                 17

=================================================================
Total params: 1,001,633
Trainable params: 1,001,633
Non-trainable params: 0
_________________________________________________________________
```

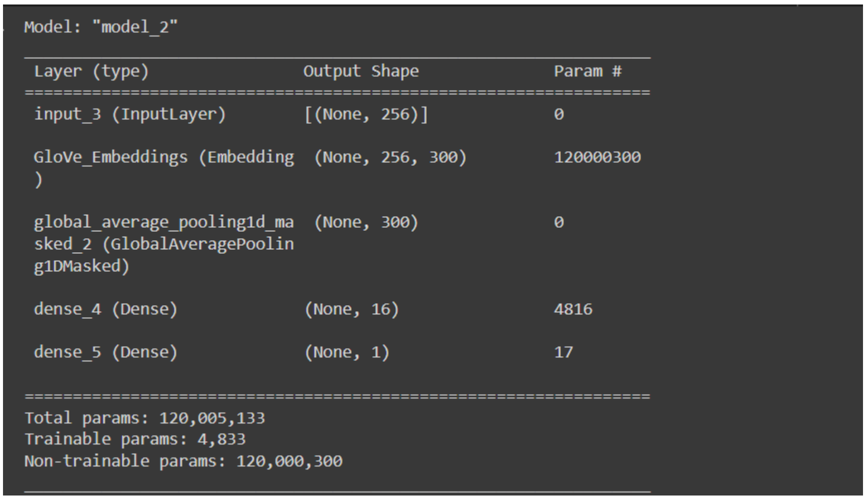
The accuracy obtained as seen from the graph below is **87.1%**. Using embedding layer has significantly improved the performance.



3. Adapt your model to load and use pre-trained word embeddings instead (Model 3); train and evaluate it and compare the effect of freezing and fine-tuning the embeddings

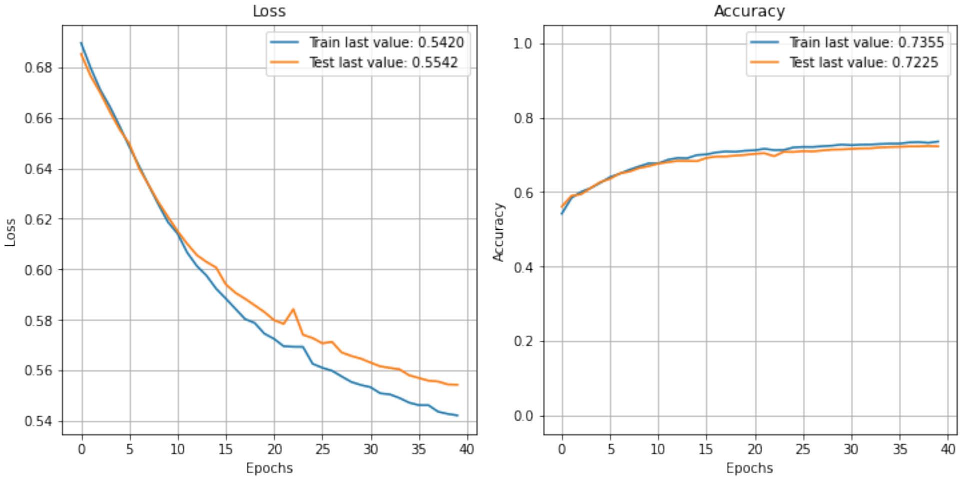
1. *Model 3-1: Neural bag of words using pre-trained word embeddings:*

    (a) Here another embedding is used called Glove embedding where embeddings are pre-trained. As we want the pre-trained word embeddings, the parameter *isTrainable* is set to False.

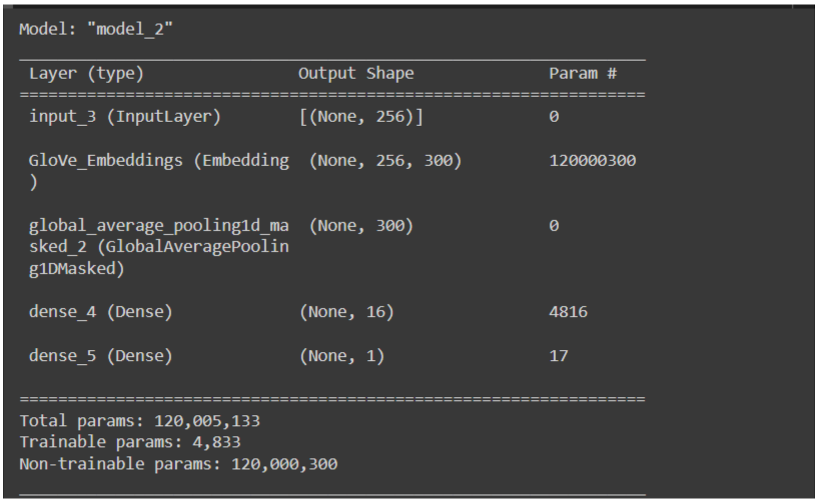


```
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 256)]             0

 GloVe_Embeddings (Embedding  (None, 256, 300)         120000300
 )

 global_average_pooling1d_ma  (None, 300)              0
 sked_2 (GlobalAveragePoolin
 g1DMasked)

 dense_4 (Dense)             (None, 16)                4816

 dense_5 (Dense)             (None, 1)                 17

=================================================================
Total params: 120,005,133
Trainable params: 4,833
Non-trainable params: 120,000,300
_________________________________________________________________
```

The accuracy obtained here is **72.2%**

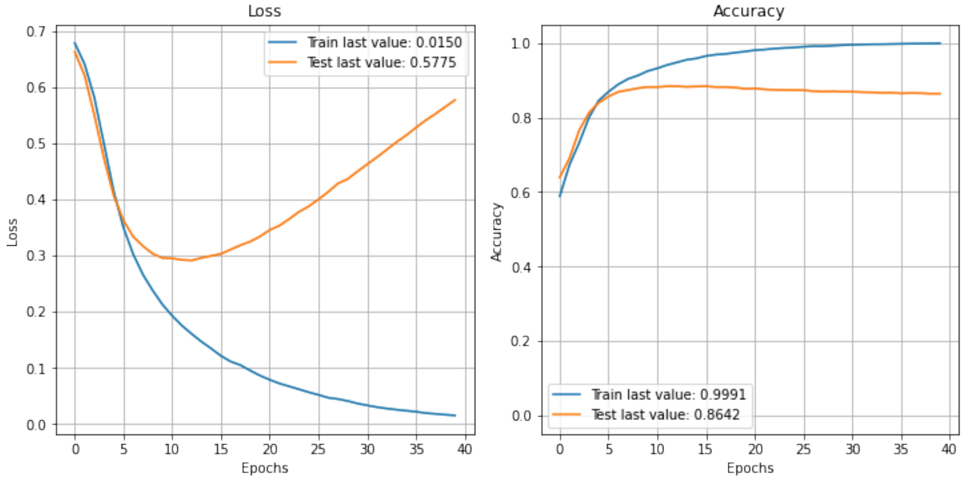


(b) As we want to compare the pre-trained with the fine tuned ones, in this section the parameter *isTrainable* is set to True this time while creating the embedding layer.



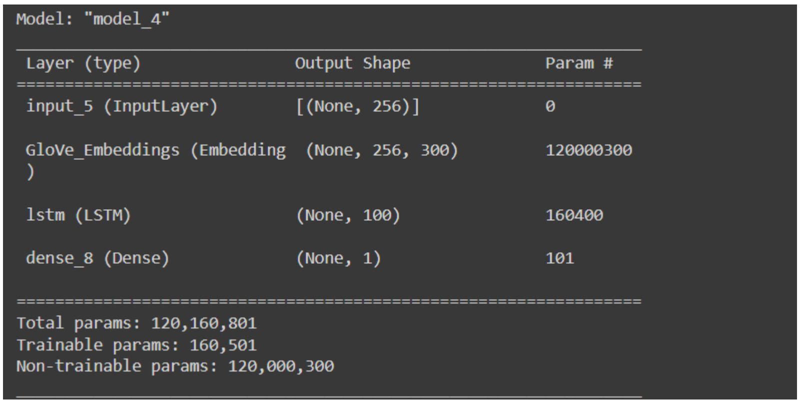
```
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 256)]             0

 GloVe_Embeddings (Embedding  (None, 256, 300)         120000300
 )

 global_average_pooling1d_ma  (None, 300)              0
 sked_2 (GlobalAveragePoolin
 g1DMasked)

 dense_4 (Dense)             (None, 16)                4816

 dense_5 (Dense)             (None, 1)                 17

=================================================================
Total params: 120,005,133
Trainable params: 4,833
Non-trainable params: 120,000,300
_________________________________________________________________
```

The accuracy here has significantly improved with going up to **86%**.



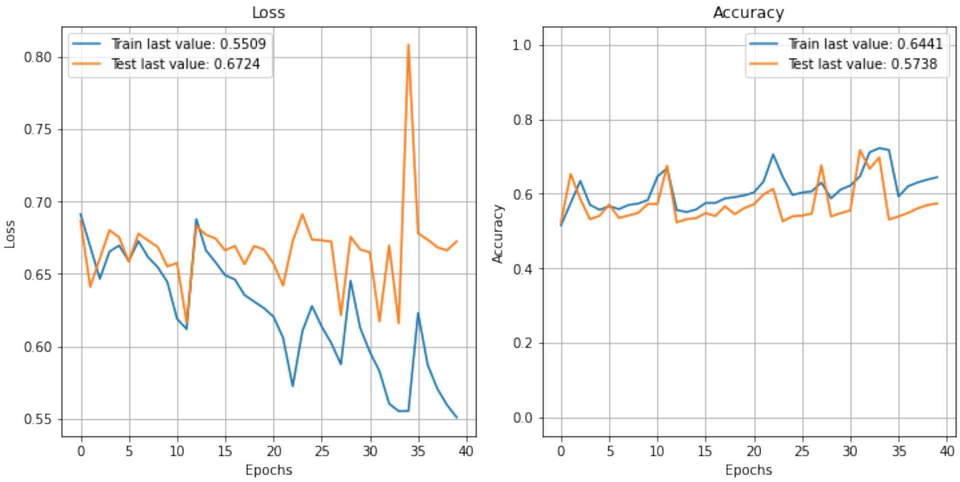
2. *Model 3-2: LSTM with pre-trained word embeddings:*

   The similar model is used as used in previous lab, the only difference here being that the embeddings has been replaced with the Glove word embeddings.



```
Model: "model_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_5 (InputLayer)        [(None, 256)]             0

 GloVe_Embeddings (Embedding  (None, 256, 300)         120000300
 )

 lstm (LSTM)                 (None, 100)               160400

 dense_8 (Dense)             (None, 1)                 101

=================================================================
Total params: 120,160,801
Trainable params: 160,501
Non-trainable params: 120,000,300
_________________________________________________________________
```

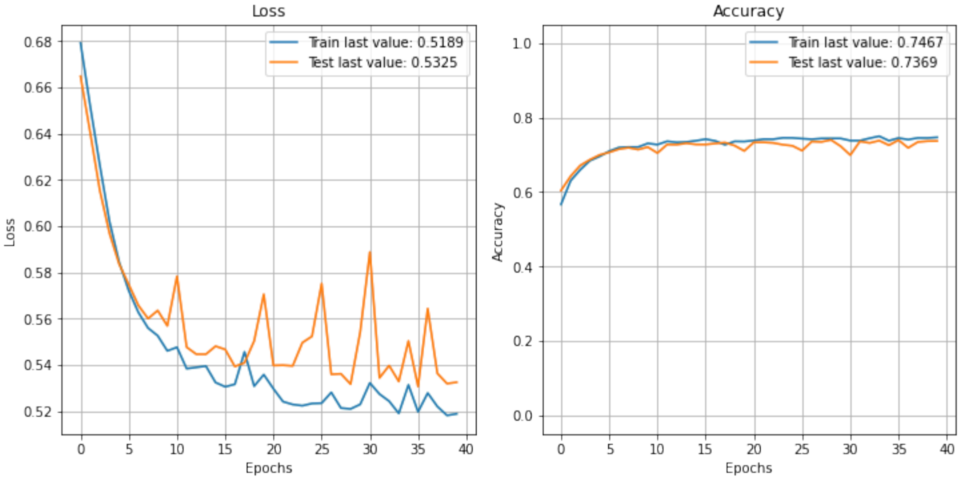
Simply replacing the lab 2 model embeddings with pre-trained word embeddings (GloVe) will cause performance to drop significantly. The reason for that is because glove pre-trained embeddings are not data specific to our domain here. They can be fine tuned in order to make the embeddings trainable so that they can model from the data specific domain better. Also in order to avoid over fitting we can try early stopping.



4. One way to improve the performance is to add another fully-connected layer to your network. Try this (Model 4) and see if it improves the performance. If not, what can you do to improve it?

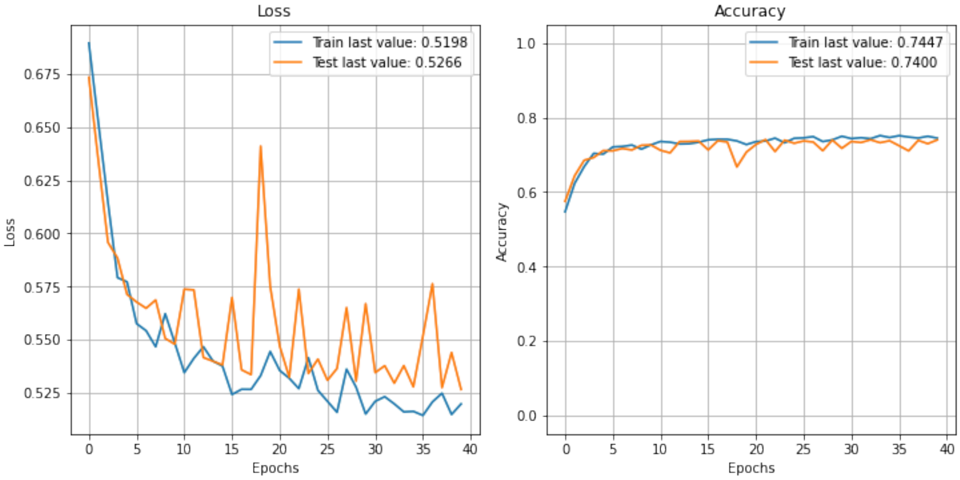
1. *Adding one extra dense layer:*

   We reuse the model used in section 3-1. The only change made here is that one more dense layer is added. After doing so, from the graph below we can see that the performance has increased slightly when compared to 3-1.



2. *Adding two extra dense layers:*

   In this section, instead of adding one, two dense layers have been added in comparison to section 3-1. After doing so, from the graph below we can see that this situation also makes the performance slightly better when compared to section 3-1.

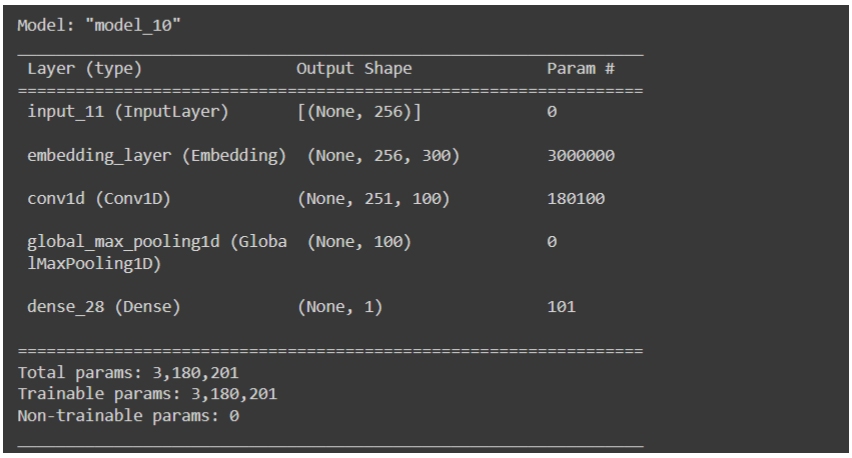


After adding one extra layer and two extra layers slightly improved the accuracy because the network is able to learn more complex functions as the models are now more deeper networks. When comparing the loss with section 3-1, both the models here have showed slight improvement and have decreased the loss percentage. Adding extra layers has definitely helped in both the cases and they are able to generalise the data more efficiently.

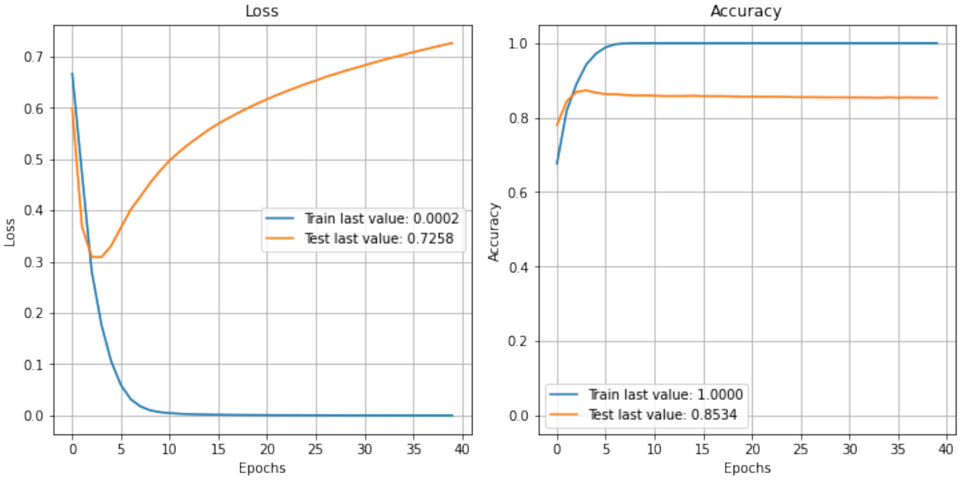
5. Build a CNN classifier (Model 5), and train and evaluate it. Then try adding extra convolutional layers, and conduct training and evaluation

1. Model 5-1: Basic CNN model for Text Classification:

   While we continue to train pre-trained glove word embeddings, we create a CNN model by adding a convolutional layer to the network. For the added convolutional layer, there are 100 output filters with each filter having a size of 6.

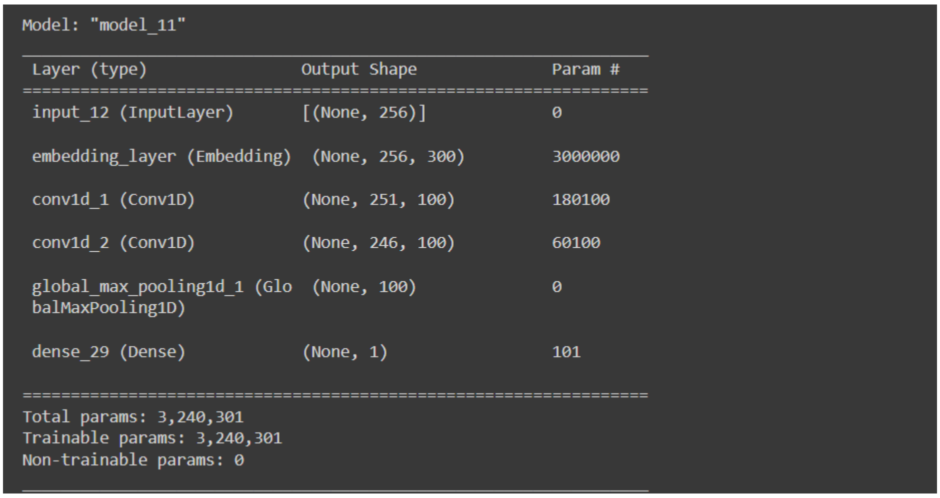


```
Model: "model_10"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_11 (InputLayer)       [(None, 256)]             0

 embedding_layer (Embedding)  (None, 256, 300)         3000000

 conv1d (Conv1D)             (None, 251, 100)          180100

 global_max_pooling1d (Globa  (None, 100)              0
 lMaxPooling1D)

 dense_28 (Dense)            (None, 1)                 101

=================================================================
Total params: 3,180,201
Trainable params: 3,180,201
Non-trainable params: 0
_________________________________________________________________
```

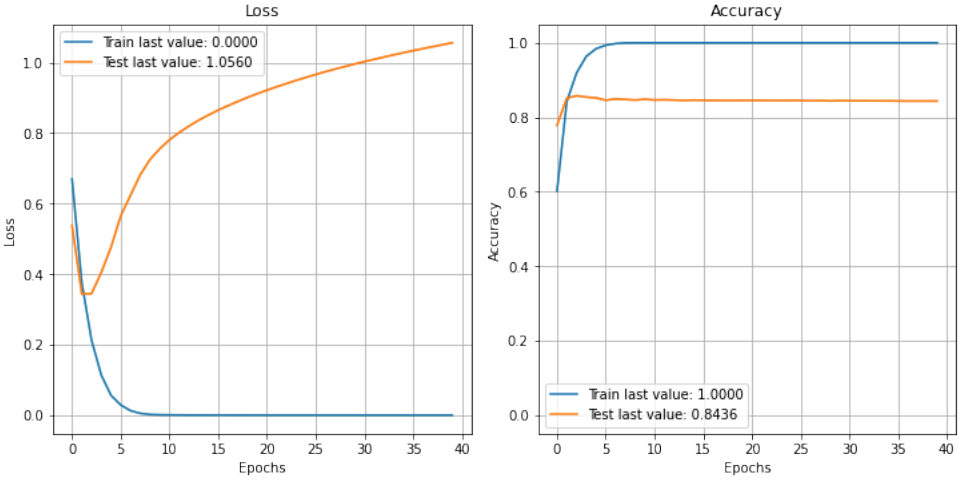


2. Model 5-2: Adding extra convolutional layer:

   In this section, instead of adding one convolutional layer, two convolutional layers are added to the network. Everthing else in the network is used as similar as before.



```
Model: "model_11"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_12 (InputLayer)       [(None, 256)]             0

 embedding_layer (Embedding)  (None, 256, 300)         3000000

 conv1d_1 (Conv1D)           (None, 251, 100)          180100

 conv1d_2 (Conv1D)           (None, 246, 100)          60100

 global_max_pooling1d_1 (Glo  (None, 100)              0
 balMaxPooling1D)

 dense_29 (Dense)            (None, 1)                 101

=================================================================
Total params: 3,240,301
Trainable params: 3,240,301
Non-trainable params: 0
_________________________________________________________________
```



After plotting the graphs we can clearly see that even after adding extra layer CNN does not perform well and gives us lower accuracy in comparison to the models used previously. CNN here is more prone to overfitting as compared to denser neural averaging network model. Complexity of the model is increased because the weights are increased when more layers are added and a deeper CNN is constructed. This leads to overfitting and the models performing extremely poorly on the test data.

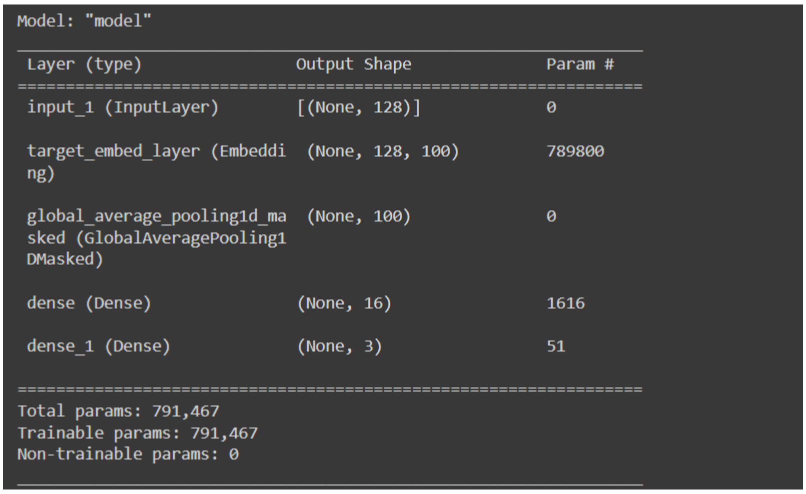
Part D: A Real Text Classification Task

1. Preprocess the data, to adapt the models from Parts C

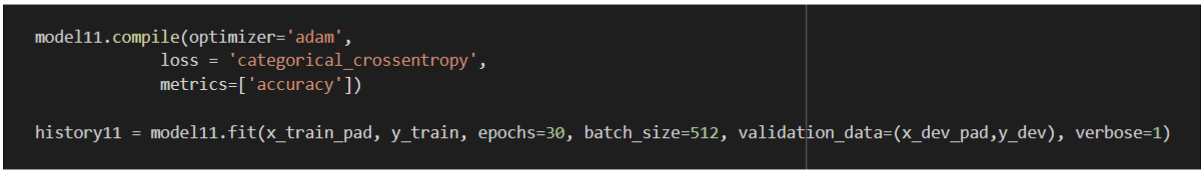
To perform the pre-processing on the data, we have used corresponding columns from train dev and test. The text in the review has been tokenized using the *text_to_word_sequence()* function. *word_index* dictionary is used for converting the tokens present in the reviews and aspect to corresponding integer. If we encounter a token that is not present in *word_index* from the dev and test dataset, that token is considered as unknown.

2. Adapt your models without pre-trained word embeddings in Part C to this task (Model 1); train and evaluate it

1. *Model 1-1: Neural bag of words without pre-trained word embeddings:*



```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_1 (InputLayer)        [(None, 128)]             0         
                                                                 
 target_embed_layer (Embeddi  (None, 128, 100)         789800    
 ng)                                                             
                                                                 
 global_average_pooling1d_ma  (None, 100)              0         
 sked (GlobalAveragePooling1                                     
 DMasked)                                                        
                                                                 
 dense (Dense)               (None, 16)                1616      
                                                                 
 dense_1 (Dense)             (None, 3)                 51        
                                                                 
=================================================================
Total params: 791,467
Trainable params: 791,467
Non-trainable params: 0
_________________________________________________________________
```



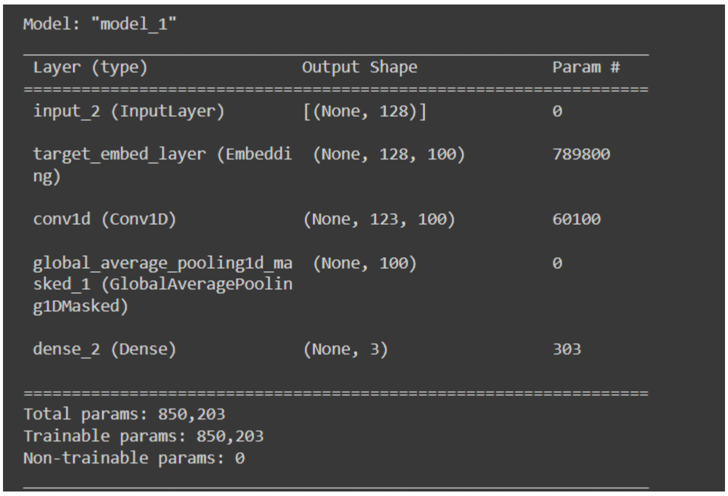
```
model11.compile(optimizer='adam',
              loss = 'categorical_crossentropy',
              metrics=['accuracy'])

history11 = model11.fit(x_train_pad, y_train, epochs=30, batch_size=512, validation_data=(x_dev_pad,y_dev), verbose=1)
```

After evaluating the model following results were obtained:

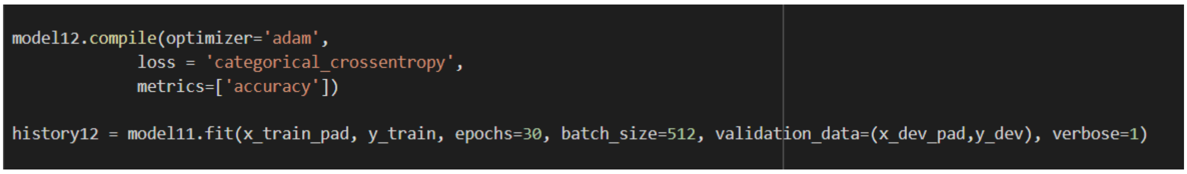
```
loss: 0.9175584316253662 accuracy: 0.582335352897644
```

2. *Model 1-2: CNN or LSTM without pre-trained word embeddings:*

   Here, instead of using neural bag of words model we have used CNN model. However the CNN model has not performed well when compared to the previous model.



```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 128)]             0

 target_embed_layer (Embeddi  (None, 128, 100)         789800
 ng)

 conv1d (Conv1D)             (None, 123, 100)          60100

 global_average_pooling1d_ma  (None, 100)              0
 sked_1 (GlobalAveragePoolin
 g1DMasked)

 dense_2 (Dense)             (None, 3)                 303

=================================================================
Total params: 850,203
Trainable params: 850,203
Non-trainable params: 0
_________________________________________________________________
```



```
model12.compile(optimizer='adam',
              loss = 'categorical_crossentropy',
              metrics=['accuracy'])

history12 = model11.fit(x_train_pad, y_train, epochs=30, batch_size=512, validation_data=(x_dev_pad,y_dev), verbose=1)
```

After evaluating the model following results were obtained:

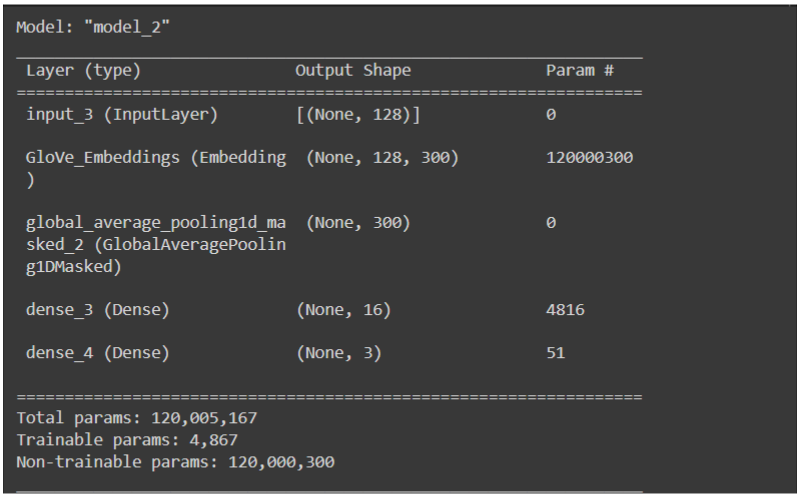
```
loss: 1.2465183734893799 accuracy: 0.5681137442588806
```

The accuracy has receded little bit coming down to **58%** in comparison to the previous model's **56%**.

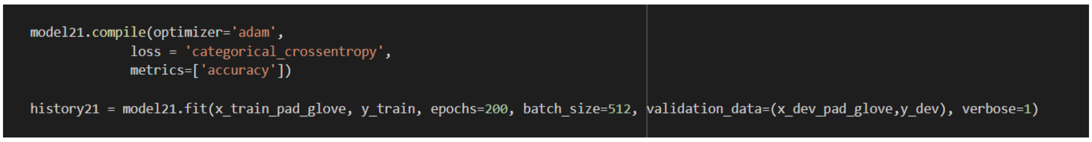
3. Adapt your models with pre-trained word embeddings in Part C to this task (Model 2); train and evaluate it

Partitioning of reviews and aspects is done on each train, dev, and test dataset using the *generate_review_aspect_Glove()* function. Every string in the data is tokenized, and then finding the indexes from glove embeddings in *wordToIndex* corresponding to the tokens to create two lists *review_int* and *aspect_int*. In the end we concatenate the list of indexes for texts and aspects and pad the resulting sequences.

1. *Model 2-1: Neural bag of words using pre-trained word embeddings:*



```
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_3 (InputLayer)        [(None, 128)]             0         
                                                                 
 GloVe_Embeddings (Embedding  (None, 128, 300)         120000300 
 )                                                               
                                                                 
 global_average_pooling1d_ma  (None, 300)              0         
 sked_2 (GlobalAveragePoolin                                     
 g1DMasked)                                                      
                                                                 
 dense_3 (Dense)             (None, 16)                4816      
                                                                 
 dense_4 (Dense)             (None, 3)                 51        
                                                                 
=================================================================
Total params: 120,005,167
Trainable params: 4,867
Non-trainable params: 120,000,300
_________________________________________________________________
```



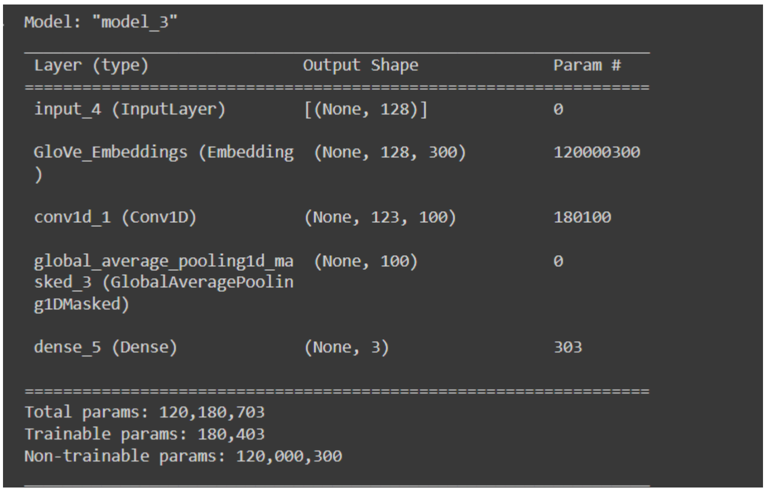
```
model21.compile(optimizer='adam',
            loss = 'categorical_crossentropy',
            metrics=['accuracy'])

history21 = model21.fit(x_train_pad_glove, y_train, epochs=200, batch_size=512, validation_data=(x_dev_pad_glove,y_dev), verbose=1)
```

After evaluating the model following results were obtained:

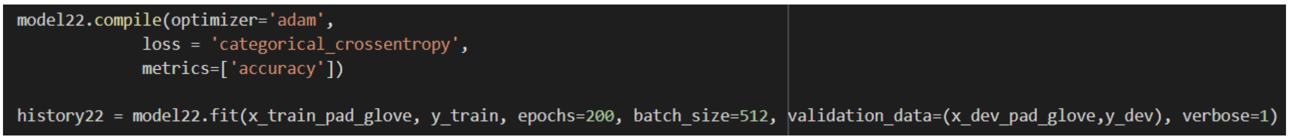
`loss: 0.8814734816551208 accuracy: 0.56886225938797`

As seen from the results below the ”glorot_uniform” initialization does not improve the performance of this model significantly.

2. *Model 2-2: CNN or LSTM with pre-trained word embeddings:*



```
Model: "model_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_4 (InputLayer)        [(None, 128)]             0         
                                                                 
 GloVe_Embeddings (Embedding  (None, 128, 300)         120000300 
 )                                                               
                                                                 
 conv1d_1 (Conv1D)           (None, 123, 100)          180100    
                                                                 
 global_average_pooling1d_ma  (None, 100)              0         
 sked_3 (GlobalAveragePoolin                                     
 g1DMasked)                                                      
                                                                 
 dense_5 (Dense)             (None, 3)                 303       
                                                                 
=================================================================
Total params: 120,180,703
Trainable params: 180,403
Non-trainable params: 120,000,300
_________________________________________________________________
```



```
model22.compile(optimizer='adam',
            loss = 'categorical_crossentropy',
            metrics=['accuracy'])

history22 = model22.fit(x_train_pad_glove, y_train, epochs=200, batch_size=512, validation_data=(x_dev_pad_glove,y_dev), verbose=1)
```

After evaluating the model following results were obtained:

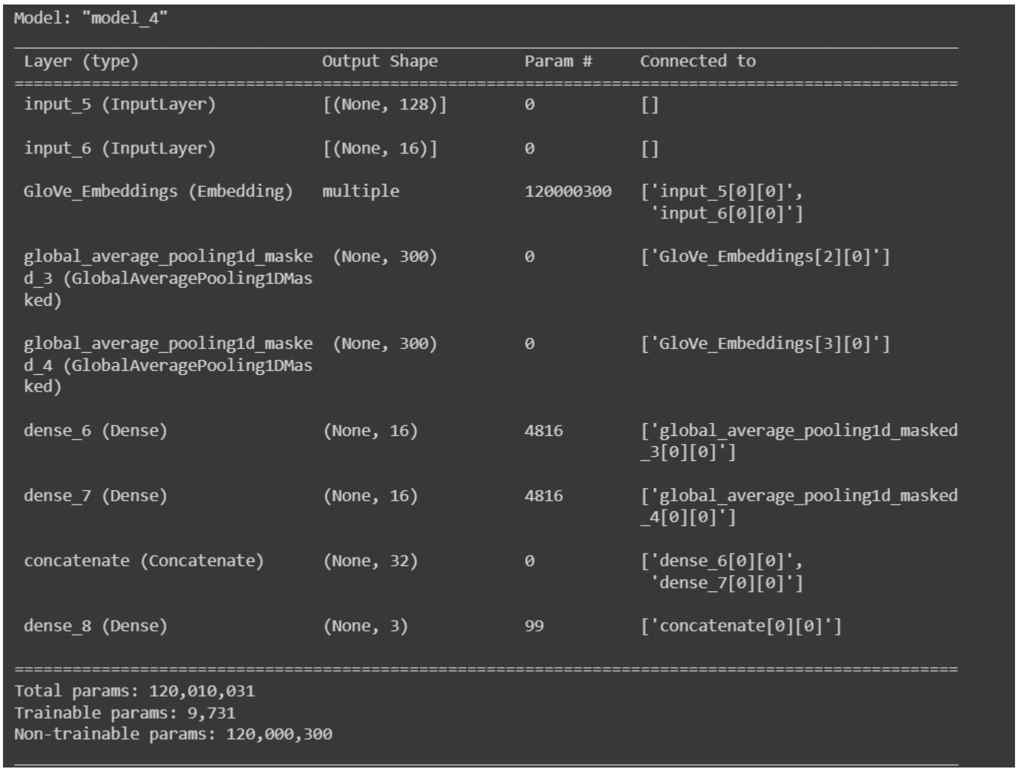
```
loss: 0.8446930050849915 accuracy: 0.6482036113739014
```

Using CNN model with pre-word embedding improves the performance with accuracy going up to **64%**

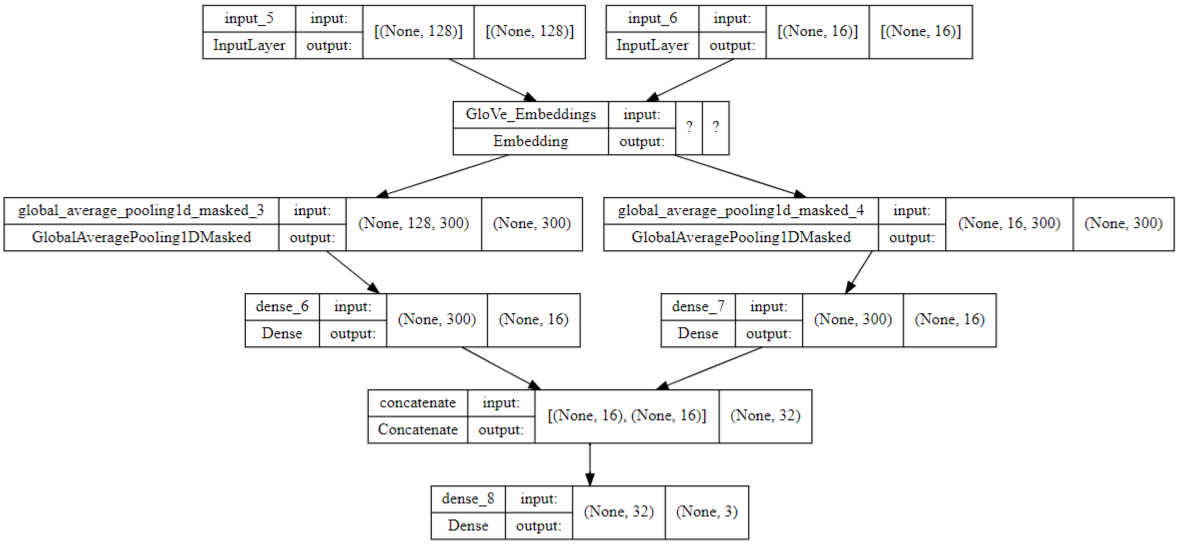
4. Build and evaluate two more classifiers with multiple input (Model 3: separate inputs for text and aspect)

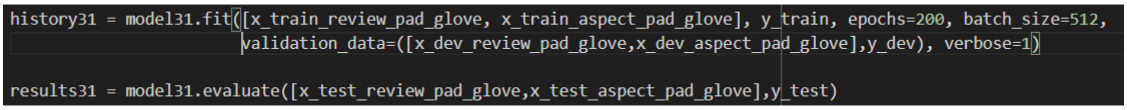
Till now we have used concatenated list of indices for text and their aspects. In this section, we will pass them as separate inputs to a set of multiple different layers. These layers will be merging together right before the output layer.

1. *Model 3-1 Neural bag of words model with multiple-input:*



```
Model: "model_4"
 __________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_5 (InputLayer)           [(None, 128)]        0           []

 input_6 (InputLayer)           [(None, 16)]         0           []

 GloVe_Embeddings (Embedding)   multiple             120000300   ['input_5[0][0]',
                                                                  'input_6[0][0]']

 global_average_pooling1d_maske  (None, 300)         0           ['GloVe_Embeddings[2][0]']
 d_3 (GlobalAveragePooling1DMas
 ked)

 global_average_pooling1d_maske  (None, 300)         0           ['GloVe_Embeddings[3][0]']
 d_4 (GlobalAveragePooling1DMas
 ked)

 dense_6 (Dense)                (None, 16)           4816        ['global_average_pooling1d_masked
                                                                 _3[0][0]']

 dense_7 (Dense)                (None, 16)           4816        ['global_average_pooling1d_masked
                                                                 _4[0][0]']

 concatenate (Concatenate)      (None, 32)           0           ['dense_6[0][0]',
                                                                  'dense_7[0][0]']

 dense_8 (Dense)                (None, 3)            99          ['concatenate[0][0]']

==================================================================================================
Total params: 120,010,031
Trainable params: 9,731
Non-trainable params: 120,000,300
 __________________________________________________________________________________________________
```



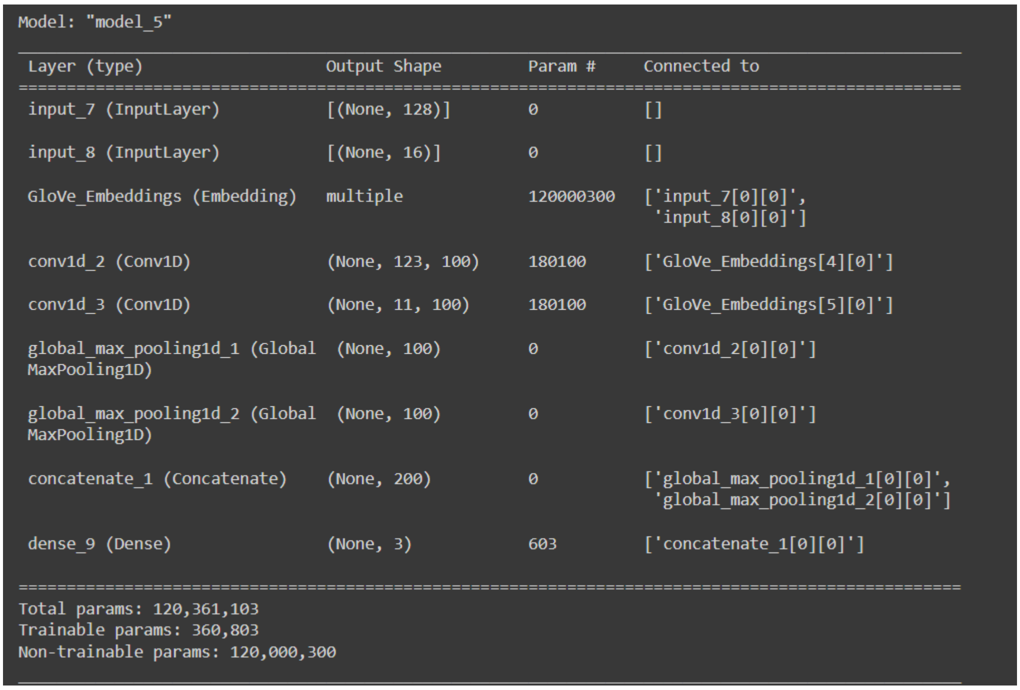


```
history31 = model31.fit([x_train_review_pad_glove, x_train_aspect_pad_glove], y_train, epochs=200, batch_size=512,
                        validation_data=([x_dev_review_pad_glove,x_dev_aspect_pad_glove],y_dev), verbose=1)

results31 = model31.evaluate([x_test_review_pad_glove,x_test_aspect_pad_glove],y_test)
```

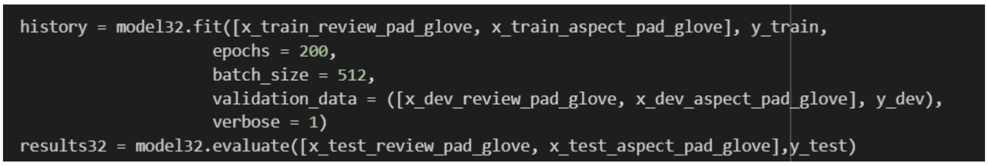
After evaluating the model following results were obtained:

`loss: 0.782159686088562 accuracy: 0.652694582939148`

2. *Model 3-2 CNN or LSTM model with multiple-input:*



```
Model: "model_5"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_7 (InputLayer)           [(None, 128)]        0           []

 input_8 (InputLayer)           [(None, 16)]         0           []

 GloVe_Embeddings (Embedding)   multiple             120000300   ['input_7[0][0]',
                                                                  'input_8[0][0]']

 conv1d_2 (Conv1D)              (None, 123, 100)     180100      ['GloVe_Embeddings[4][0]']

 conv1d_3 (Conv1D)              (None, 11, 100)      180100      ['GloVe_Embeddings[5][0]']

 global_max_pooling1d_1 (Global  (None, 100)         0           ['conv1d_2[0][0]']
 MaxPooling1D)

 global_max_pooling1d_2 (Global  (None, 100)         0           ['conv1d_3[0][0]']
 MaxPooling1D)

 concatenate_1 (Concatenate)    (None, 200)          0           ['global_max_pooling1d_1[0][0]',
                                                                  'global_max_pooling1d_2[0][0]']

 dense_9 (Dense)                (None, 3)            603         ['concatenate_1[0][0]']

==================================================================================================
Total params: 120,361,103
Trainable params: 360,803
Non-trainable params: 120,000,300
__________________________________________________________________________________________________
```



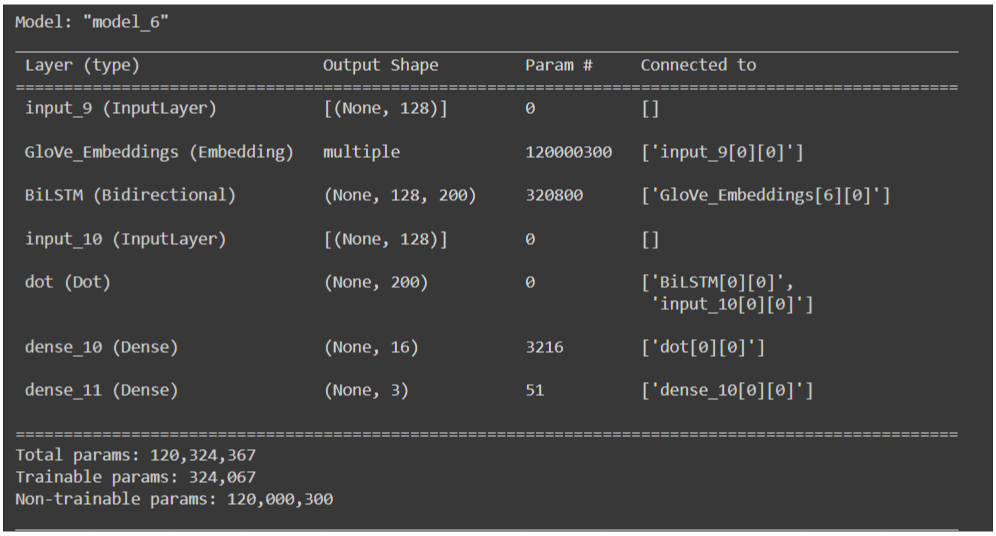
```
history = model32.fit([x_train_review_pad_glove, x_train_aspect_pad_glove], y_train,
                      epochs = 200,
                      batch_size = 512,
                      validation_data = ([x_dev_review_pad_glove, x_dev_aspect_pad_glove], y_dev),
                      verbose = 1)
results32 = model32.evaluate([x_test_review_pad_glove, x_test_aspect_pad_glove],y_test)
```

After evaluating the model following results were obtained:

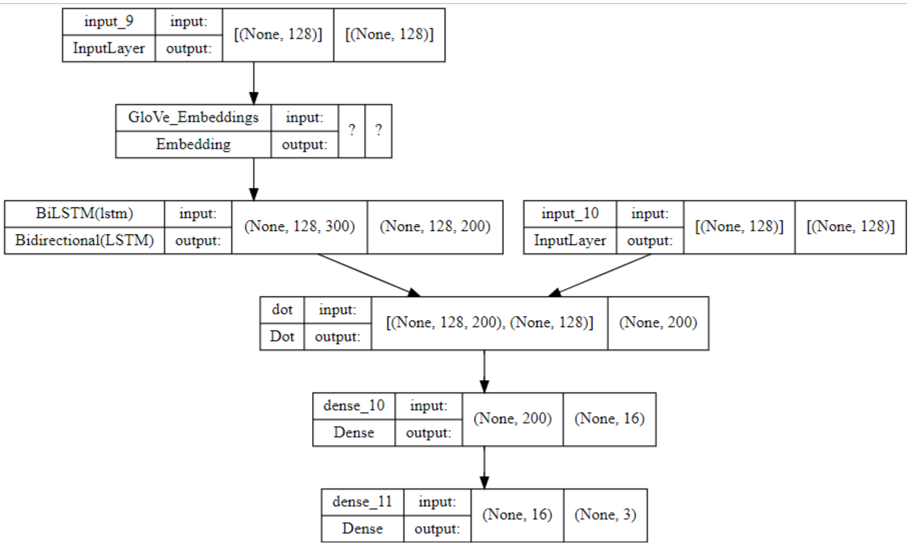
`loss: 1.2963393926620483 accuracy: 0.6444610953330994`

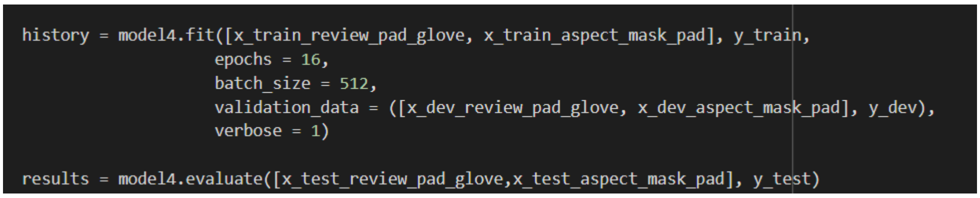
The accuracy does not increase much when compared to the model without pre-trained word embeddings. In this version, the ”glorot_uniform” initialization method does not improve model performance significantly. In pre-trained word embeddings, we cannot directly use the index data, we convert them from text tokens to GLOVE word index. Plus we are using only single input.

5. Build and evaluate the classifier extracting information from LSTM (Model 4)



```
Model: "model_6"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_9 (InputLayer)           [(None, 128)]        0           []

 GloVe_Embeddings (Embedding)   multiple             120000300   ['input_9[0][0]']

 BiLSTM (Bidirectional)         (None, 128, 200)     320800      ['GloVe_Embeddings[6][0]']

 input_10 (InputLayer)          [(None, 128)]        0           []

 dot (Dot)                      (None, 200)          0           ['BiLSTM[0][0]',
                                                                  'input_10[0][0]']

 dense_10 (Dense)               (None, 16)           3216        ['dot[0][0]']

 dense_11 (Dense)               (None, 3)            51          ['dense_10[0][0]']

==================================================================================================
Total params: 120,324,367
Trainable params: 324,067
Non-trainable params: 120,000,300
__________________________________________________________________________________________________
```





```
history = model4.fit([x_train_review_pad_glove, x_train_aspect_mask_pad], y_train,
                    epochs = 16,
                    batch_size = 512,
                    validation_data = ([x_dev_review_pad_glove, x_dev_aspect_mask_pad], y_dev),
                    verbose = 1)

results = model4.evaluate([x_test_review_pad_glove,x_test_aspect_mask_pad], y_test)
```

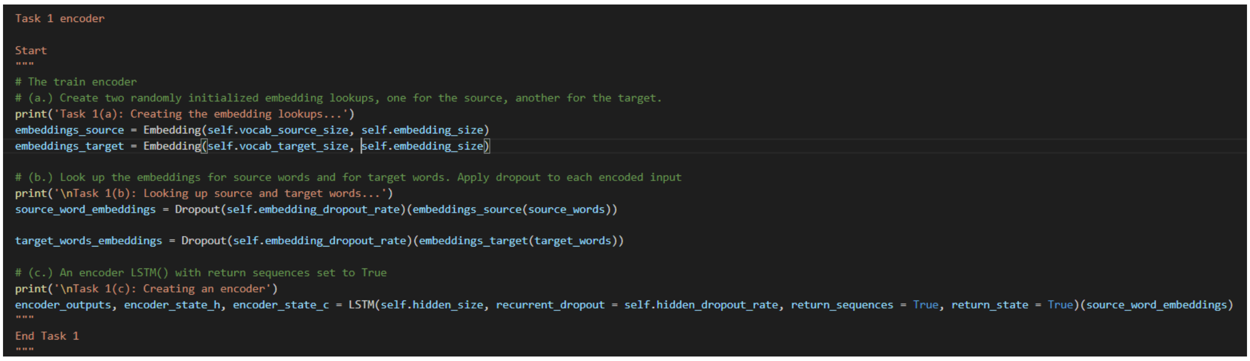
After evaluating the model following results were obtained:

```
[1.1583406925201416, 0.711077868938446]
```

Part E: Neural Machine Translation

1. Task 1: Implementing the encoder

In this section we create a simple encoder model. The model consists of embedding layer, a dropout layer and a LSTM layer. The embedding layer converts the sentences to an embedded list of word arrays. After that dropout layer shuts down the selected number of neurons. After that the LSTM layer which is an RNN layer is used to pass the input sentences in several consecutive time steps. In each of those time steps the model learns from the input words. This learnt information is stored in the hidden vector which is used to preserve the information form the previous time steps. Finally the information is passed on to the decoder.



```
Task 1 encoder

Start
"""
# The train encoder
# (a.) Create two randomly initialized embedding lookups, one for the source, another for the target.
print('Task 1(a): Creating the embedding lookups...')
embeddings_source = Embedding(self.vocab_source_size, self.embedding_size)
embeddings_target = Embedding(self.vocab_target_size, self.embedding_size)

# (b.) Look up the embeddings for source words and for target words. Apply dropout to each encoded input
print('\nTask 1(b): Looking up source and target words...')
source_word_embeddings = Dropout(self.embedding_dropout_rate)(embeddings_source(source_words))

target_words_embeddings = Dropout(self.embedding_dropout_rate)(embeddings_target(target_words))

# (c.) An encoder LSTM() with return sequences set to True
print('\nTask 1(c): Creating an encoder')
encoder_outputs, encoder_state_h, encoder_state_c = LSTM(self.hidden_size, recurrent_dropout = self.hidden_dropout_rate, return_sequences = True, return_state = True)(source_word_embeddings)
"""
End Task 1
"""
```

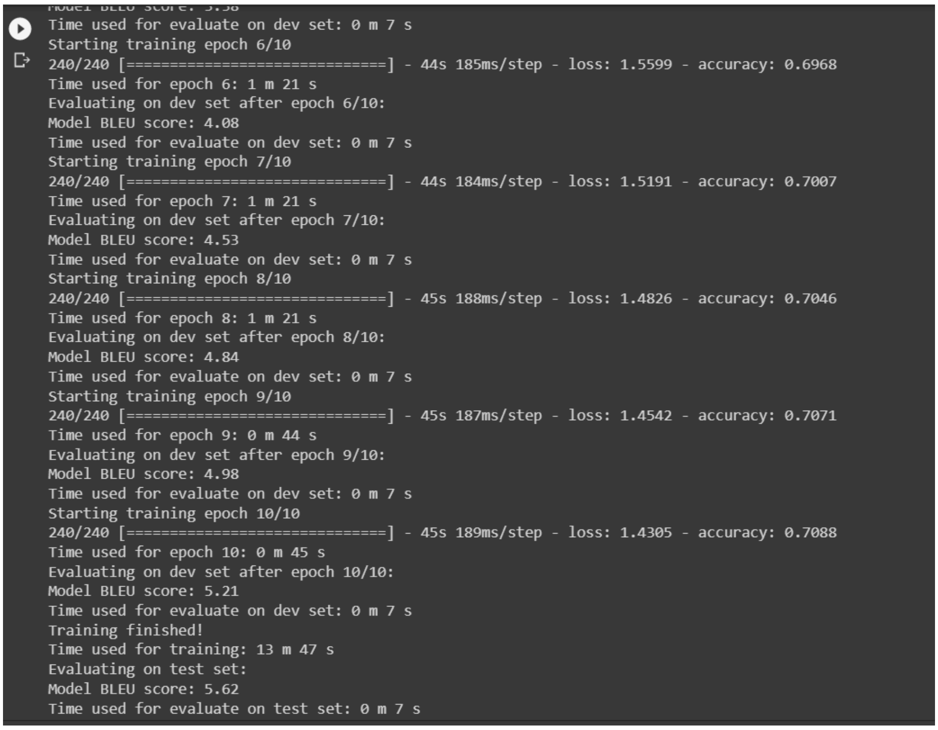
2. Task 2: Implementing the decoder

The outputs given by the encoder will be received by the decoder's input layer to interpret. The decoder will provide a series of outputs that can be used to predict the upcoming sequences. The training and inference processes needs to be considered while designing the decoder. In the training scenario, all the tokens that make up the sentence in a single step will be processed. In the inference process, one token at a time will be processed.

```
"""
Task 2 decoder for inference

Start
"""
# Task 1 (a.) Get the decoded outputs
print('\n Putting together the decoder states')
# get the inititial states for the decoder, decoder_states
# decoder states are the hidden and cell states from the training stage
decoder_states = [decoder_state_input_h, decoder_state_input_c]
# use decoder states as input to the decoder lstm to get the decoder outputs, h, and c for test time inference
decoder_outputs_test,decoder_state_output_h, decoder_state_output_c = decoder_lstm(target_words_embeddings, initial_state = decoder_states)



# Task 1 (b.) Add attention if attention

if self.use_attention:
    decoder_outputs_test = decoder_attention([encoder_outputs_input, decoder_outputs_test])


# Task 1 (c.) pass the decoder_outputs_test (with or without attention) to the decoder dense layer

decoder_outputs_test = decoder_dense(decoder_outputs_test)

"""
End Task 2
"""
```

BLEU Score:-

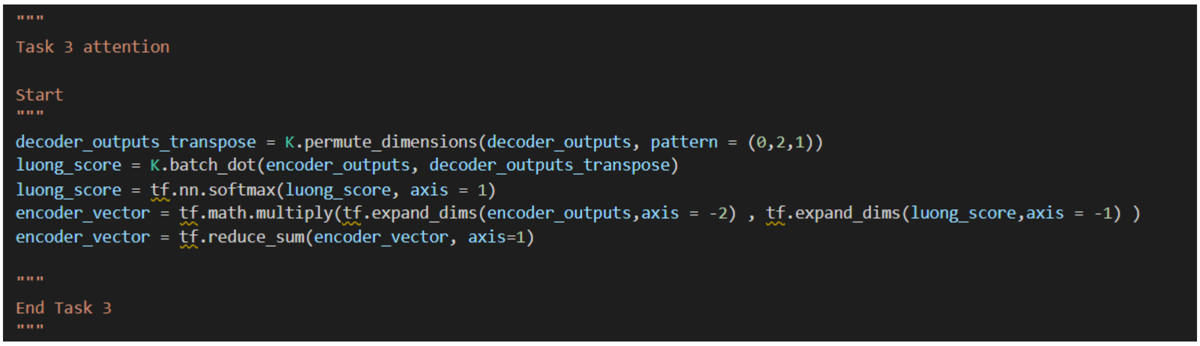
```
Model BLEU score: 5.62
```



```
Time used for evaluate on dev set: 0 m 7 s
Starting training epoch 6/10
240/240 [==============================] - 44s 185ms/step - loss: 1.5599 - accuracy: 0.6968
Time used for epoch 6: 1 m 21 s
Evaluating on dev set after epoch 6/10:
Model BLEU score: 4.08
Time used for evaluate on dev set: 0 m 7 s
Starting training epoch 7/10
240/240 [==============================] - 44s 184ms/step - loss: 1.5191 - accuracy: 0.7007
Time used for epoch 7: 1 m 21 s
Evaluating on dev set after epoch 7/10:
Model BLEU score: 4.53
Time used for evaluate on dev set: 0 m 7 s
Starting training epoch 8/10
240/240 [==============================] - 45s 188ms/step - loss: 1.4826 - accuracy: 0.7046
Time used for epoch 8: 1 m 21 s
Evaluating on dev set after epoch 8/10:
Model BLEU score: 4.84
Time used for evaluate on dev set: 0 m 7 s
Starting training epoch 9/10
240/240 [==============================] - 45s 187ms/step - loss: 1.4542 - accuracy: 0.7071
Time used for epoch 9: 0 m 44 s
Evaluating on dev set after epoch 9/10:
Model BLEU score: 4.98
Time used for evaluate on dev set: 0 m 7 s
Starting training epoch 10/10
240/240 [==============================] - 45s 189ms/step - loss: 1.4305 - accuracy: 0.7088
Time used for epoch 10: 0 m 45 s
Evaluating on dev set after epoch 10/10:
Model BLEU score: 5.21
Time used for evaluate on dev set: 0 m 7 s
Training finished!
Time used for training: 13 m 47 s
Evaluating on test set:
Model BLEU score: 5.62
Time used for evaluate on test set: 0 m 7 s
```

3. Adding attention

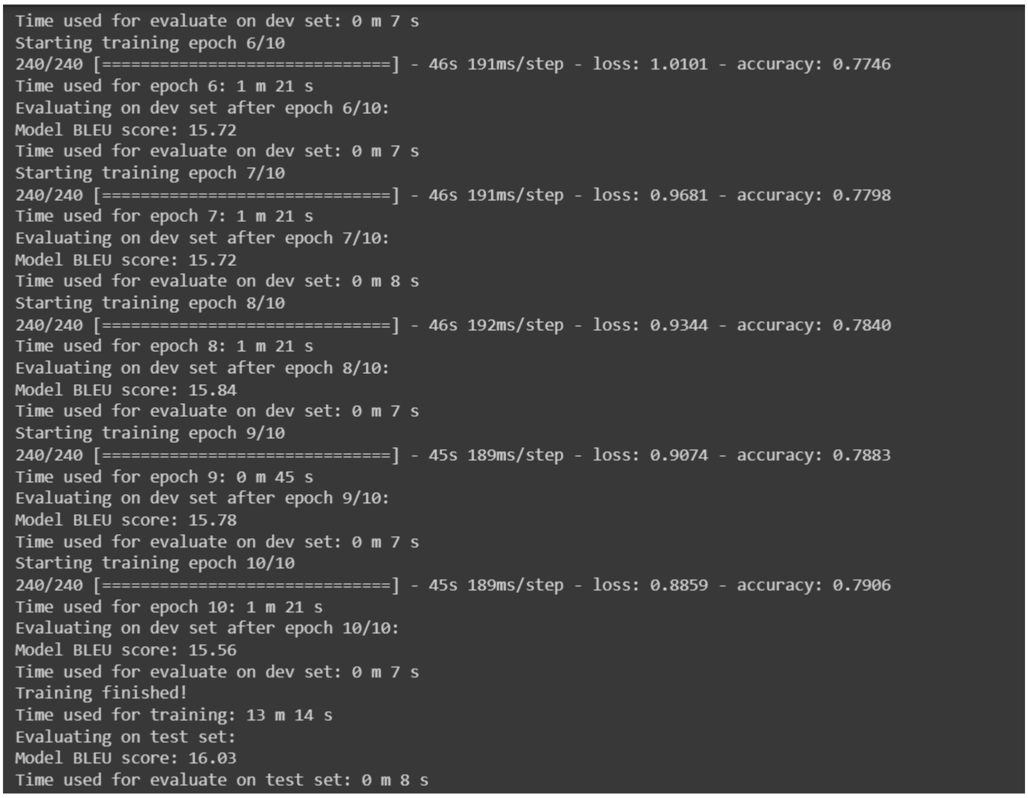
In extended semantic sentences, the prior NMT is unable to extract significant contextual linkages, becuase of which there will be an impact on model's performance. In order to overcome this, attention is added to the network to improve the model's accuracy. When predicting the output at each time step in the output sequence, the decoder will focus on a certain section of the input sentence and then relate it to elements in the output sequence.



```
"""
Task 3 attention

Start
"""
decoder_outputs_transpose = K.permute_dimensions(decoder_outputs, pattern = (0,2,1))
luong_score = K.batch_dot(encoder_outputs, decoder_outputs_transpose)
luong_score = tf.nn.softmax(luong_score, axis = 1)
encoder_vector = tf.math.multiply(tf.expand_dims(encoder_outputs,axis = -2) , tf.expand_dims(luong_score,axis = -1) )
encoder_vector = tf.reduce_sum(encoder_vector, axis=1)

"""
End Task 3
"""
```

BLEU Score:-

```
Model BLEU score: 16.03
```



```
Time used for evaluate on dev set: 0 m 7 s
Starting training epoch 6/10
240/240 [==============================] - 46s 191ms/step - loss: 1.0101 - accuracy: 0.7746
Time used for epoch 6: 1 m 21 s
Evaluating on dev set after epoch 6/10:
Model BLEU score: 15.72
Time used for evaluate on dev set: 0 m 7 s
Starting training epoch 7/10
240/240 [==============================] - 46s 191ms/step - loss: 0.9681 - accuracy: 0.7798
Time used for epoch 7: 1 m 21 s
Evaluating on dev set after epoch 7/10:
Model BLEU score: 15.72
Time used for evaluate on dev set: 0 m 8 s
Starting training epoch 8/10
240/240 [==============================] - 46s 192ms/step - loss: 0.9344 - accuracy: 0.7840
Time used for epoch 8: 1 m 21 s
Evaluating on dev set after epoch 8/10:
Model BLEU score: 15.84
Time used for evaluate on dev set: 0 m 7 s
Starting training epoch 9/10
240/240 [==============================] - 45s 189ms/step - loss: 0.9074 - accuracy: 0.7883
Time used for epoch 9: 0 m 45 s
Evaluating on dev set after epoch 9/10:
Model BLEU score: 15.78
Time used for evaluate on dev set: 0 m 7 s
Starting training epoch 10/10
240/240 [==============================] - 45s 189ms/step - loss: 0.8859 - accuracy: 0.7906
Time used for epoch 10: 1 m 21 s
Evaluating on dev set after epoch 10/10:
Model BLEU score: 15.56
Time used for evaluate on dev set: 0 m 7 s
Training finished!
Time used for training: 13 m 14 s
Evaluating on test set:
Model BLEU score: 16.03
Time used for evaluate on test set: 0 m 8 s
```