Assignment 1: Word Representation, Text Classification, Machine Translation, and Pre-Trained Transformers

Part A:

1. Downloading the Corpus

Sanity Check

Note that the corpus size is 16498 when run in Jupyter and 16463 in Colab.

This training corpus contains 16498 sentences. The following print statement should return 16498.

```
[2] print(len(austen))

16463
```

```
[3] austen[0]

['[', 'Sense', 'and', 'Sensibility', 'by', 'Jane', 'Austen', '1811', ']']
```

2. Preprocessing the Training Corpus

Sanity Check

```
['[', 'Sense', 'and', 'Sensibility', 'by', 'Jane', 'Austen', '1811', ']']
The new length of the preprocessed output:  12498
```

```
normalized_corpus[0]

['sense', 'sensibility', 'jane', 'austen']
```

```
sample = austen[:2] + austen[100:102]
preprocessed_sample = preprocess_corpus(sample)


print(len(sample), sample)
print()
print(len(preprocessed_sample), preprocessed_sample)

['[', 'Sense', 'and', 'Sensibility', 'by', 'Jane', 'Austen', '1811', ']']
4 [['[', 'Sense', 'and', 'Sensibility', 'by', 'Jane', 'Austen', '1811', ']'], ['CHAPTER', '1'], ['But', ',', 'then', ',', 'if', 'Mrs', '.', 'Dashwood', 'should', 'live', 'fifteen', 'ye

2 [['sense', 'sensibility', 'jane', 'austen'], ['mrs', 'dashwood', 'live', 'fifteen', 'years', 'shall', 'completely', 'taken']]
```

3. Creating the Corpus Vocabulary and Preparing the Data

Sanity Check

```
[9] print("Number of unique words:",len(word2idx))

Number of unique words: 10040
```

```
print("\nSample word2idx:",list(word2idx.items())[:10])

Sample word2idx: [('engaging', 1), ('coldly', 2), ('excellencies', 3), ('sweetest', 4), ('cassino', 5), ('fulfil', 6), ('accumulation', 7), ('solidly', 8), ('natural', 9), ('redeem', 1(
```

```
[11] print("\nSample idx2word:",list(idx2word.items())[:10])

Sample idx2word: [(1, 'engaging'), (2, 'coldly'), (3, 'excellencies'), (4, 'sweetest'), (5, 'cassino'), (6, 'fulfil'), (7, 'accumulation'), (8, 'solidly'), (9, 'natural'), (10, 'redeem'
```

```
[12] print("\nSample sents_as_id:", prepareSentsAsId(preprocessed_sample))

Sample sents_as_id: [[438, 6885, 5070, 6141], [2053, 7434, 6231, 5231, 2686, 2628, 7499, 8743]]
```

4. Generating training instances

Sanity Check

```
(jane (5070), everybody (5172)) -> 0
(jane (5070), sense (438)) -> 1
(sensibility (6885), sense (438)) -> 1
(sensibility (6885), doors (1098)) -> 0
(sensibility (6885), rapturous (9994)) -> 0
(sense (438), jane (5070)) -> 1
(sensibility (6885), austen (6141)) -> 1
(sense (438), unanswerable (9839)) -> 0
(austen (6141), sense (438)) -> 1
(jane (5070), palanquins (9773)) -> 0
(austen (6141), jane (5070)) -> 1
(austen (6141), sensibility (6885)) -> 1
(jane (5070), sensibility (6885)) -> 1
(austen (6141), tumbling (1797)) -> 0
(jane (5070), stationing (2058)) -> 0
(austen (6141), conjurer (3176)) -> 0
(sense (438), ing (216)) -> 0
(sense (438), austen (6141)) -> 1
(sensibility (6885), jane (5070)) -> 1
(sense (438), lucy (9373)) -> 0
(austen (6141), amongst (7346)) -> 0
(sense (438), sensibility (6885)) -> 1
(jane (5070), austen (6141)) -> 1
(sensibility (6885), share (7124)) -> 0
```

5. Building the Skip-gram Neural Network Architecture

Layer (type)	Output Shape	Param #	Connected to
input_3 (InputLayer)	[(None, 1)]	0	[]
input_4 (InputLayer)	[(None, 1)]	0	[]
target_embed_layer (Embedding)	(None, 1, 100)	1004100	['input_3[0][0]']
context_embed_layer (Embedding)	(None, 1, 100)	1004100	['input_4[0][0]']
reshape_2 (Reshape)	(None, 100)	0	['target_embed_layer[0][0]']
reshape_3 (Reshape)	(None, 100)	0	['context_embed_layer[0][0]']
dot_1 (Dot)	(None, 1)	0	['reshape_2[0][0]', 'reshape_3[0][0]']
dense_1 (Dense)	(None, 1)	2	['dot_1[0][0]']

```
Total params: 2,008,202
Trainable params: 2,008,202
Non-trainable params: 0
```

6. Training the Model

Q1: What would the inputs and outputs to the model be?

Input is a one-hot vector representing the word we are inputting.

Output of a window size of k is 2k values distributions (I.e., 2k output neurons). Then, for each position, we get a probability distribution for each word in the vocabulary. Then, we select the word which has the highest probability.

Q2: How would you use the Keras framework to create this architecture?

Assuming a one-hot vector input, we have an Input layer

Then use an Embedding layer

Flatten the output of the Embedding layer

Then use a Dense layer with softmax activation

Q3: What are the reasons this training approach is considered inefficient?

There are a lot weights that need to be updated and as such a lot of data is required. Also the weights of non activated neurons are updated on each iteration, which is computationally inefficient. So, a better way is to use negative sampling and only updating a subset of relevant weights in the training.

7. Getting the Word Embeddings

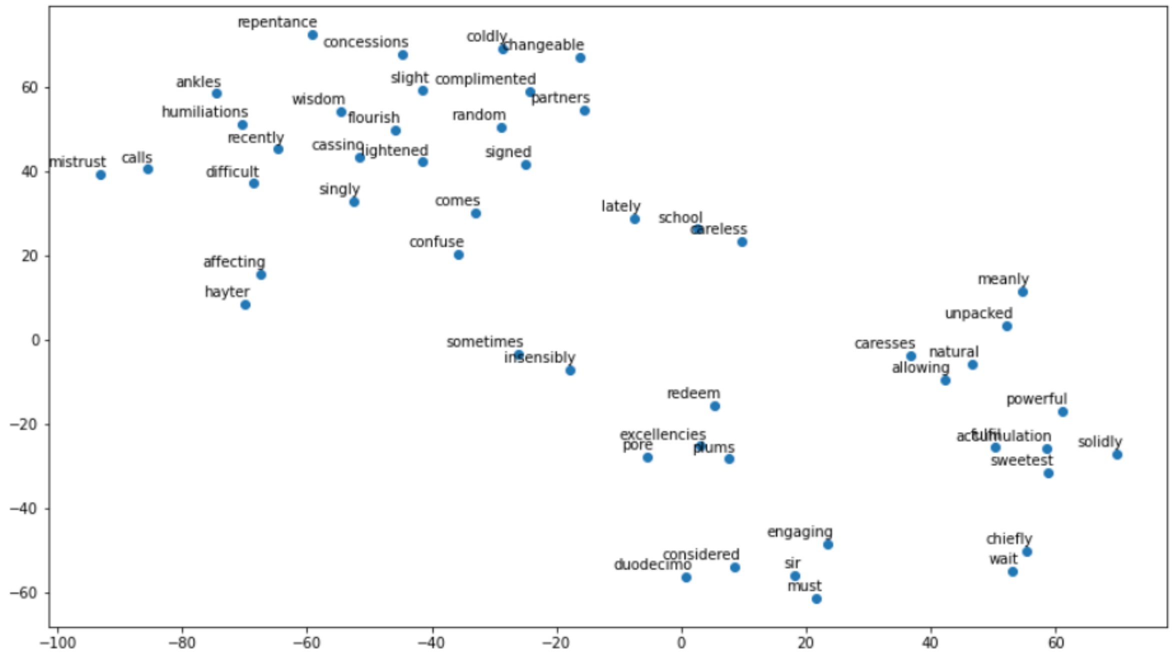
```
                     0         1         2         3         4         5  \
engaging      0.017765  0.000005  0.022112 -0.020767 -0.013893 -0.016174
coldly       -0.027600  0.020438  0.049420  0.010220  0.023247  0.017857
excellencies -0.018321 -0.004841  0.032120  0.032563  0.006279  0.017992
sweetest     -0.008631  0.026240  0.020887  0.020410 -0.013650  0.018767
cassino      -0.012359  0.018434  0.027713  0.039947  0.016284  0.025528
fulfil       -0.002769  0.006105  0.027198 -0.008085 -0.007095 -0.008833
accumulation -0.030529  0.014008  0.019657  0.021920  0.016112  0.012265
solidly      -0.031012  0.019528 -0.008223  0.018680  0.023148 -0.009726
natural      -0.014126  0.018897 -0.003177  0.005826  0.029550  0.023559
redeem       -0.140309 -0.016170  0.112158 -0.064119  0.037389 -0.008252

                     6         7         8         9  ...        90        91  \
engaging      0.012935 -0.023116 -0.002303  0.002457  ... -0.021339  0.005936
coldly       -0.023826  0.012182  0.016692 -0.004947  ... -0.020355 -0.023555
excellencies -0.020537  0.006367  0.008557 -0.030948  ... -0.010404 -0.036386
sweetest     -0.014266  0.017115  0.010169  0.002724  ...  0.001609  0.003035
cassino      -0.019760  0.017783  0.011593 -0.051846  ... -0.039112  0.000120
fulfil       -0.011802  0.012419  0.014846 -0.026387  ... -0.037336 -0.023763
accumulation -0.019975  0.022388 -0.005133 -0.016820  ...  0.006897  0.004993
solidly      -0.030244  0.027006 -0.014429 -0.017336  ...  0.002122  0.009850
natural      -0.027112  0.008908  0.012467 -0.029214  ... -0.035930 -0.000452
redeem       -0.022950 -0.047449  0.045108 -0.050998  ... -0.032899  0.064864

                    92        93        94        95        96        97  \
engaging     -0.008643  0.005085  0.014789 -0.008110  0.014664  0.008601
coldly        0.064867  0.025325 -0.036700 -0.035499  0.051797  0.026862
excellencies -0.007756  0.018761 -0.027996  0.002799  0.029025  0.003498
sweetest      0.033030  0.020545  0.004448 -0.007108  0.013759  0.013011
cassino       0.043426  0.044663 -0.038196 -0.031046  0.035141  0.034378
fulfil        0.006534  0.028483 -0.020342 -0.000624  0.025773  0.028632
accumulation  0.000928 -0.007928 -0.013274 -0.008226  0.036075  0.034024
solidly      -0.009751  0.017625  0.005126 -0.008710  0.026560  0.027365
natural       0.012698  0.028183 -0.008566 -0.016718  0.022607  0.010403
redeem       -0.020144 -0.024476  0.003181 -0.055986 -0.010962 -0.024218

                    98        99
engaging      0.014338  0.000190
coldly        0.035130  0.010713
excellencies  0.010206  0.029503
sweetest      0.026801  0.011844
cassino       0.007945  0.032199
fulfil        0.020781  0.018778
accumulation  0.032888  0.002504
solidly       0.010209  0.025139
natural       0.004158 -0.005703
redeem       -0.037390 -0.011966
```

8. Measuring Similarity Between Word Pairs
9. Exploring and Visualizing your Word Embeddings using t-SNE

Sanity Check

```
Term: think
Most similar words: ['deliver', 'devise', 'musical', 'murmurs', 'repetition']
Term: thought
Most similar words: ['rash', 'affirmative', 'plea', 'resembling', 'conceited']
Term: mr
Most similar words: ['overpowering', 'personal', 'urging', 'approver', 'admiring']
Term: friend
Most similar words: ['performers', 'enchanting', 'smiling', 'climbing', 'refrain']
Term: love
Most similar words: ['slept', 'repetition', 'fellows', 'procure', 'incidental']
Term: disdain
Most similar words: ['surpassed', 'wilfully', 'truest', 'extent', 'hating']
```

Plt.annotate



Part B:

1. Getting the Dataset

Sanity Check:

Each instance in the training data is a list of word indices representing the words in a movie review.

Each label is 1 if that review is positive, else 0.

```
[ ] print('Sample review:', train_data[0])

    Sample review: [1, 13, 21, 15, 42, 529, 972, 1621, 1384, 64, 457, 4467, 65, 3940, 3,
```

```
[ ] print('\n Sample label:', train_labels[0])

    Sample label: 1
```

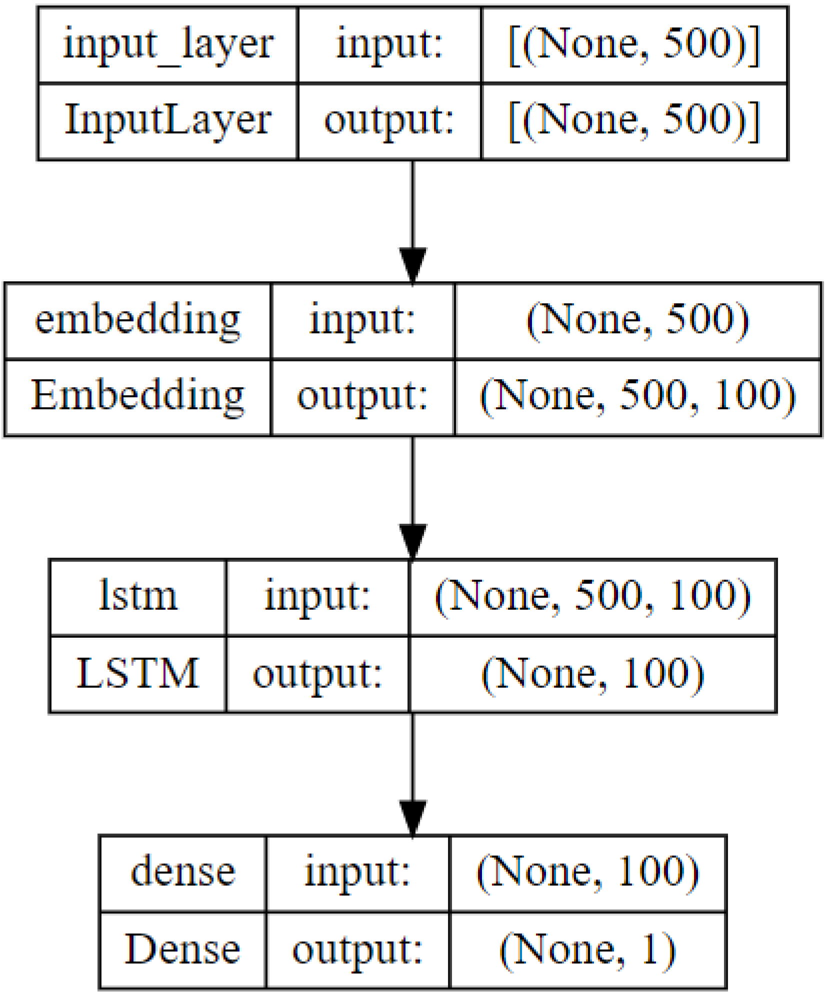
2. Readying the Inputs for the LSTM

Sanity Check:

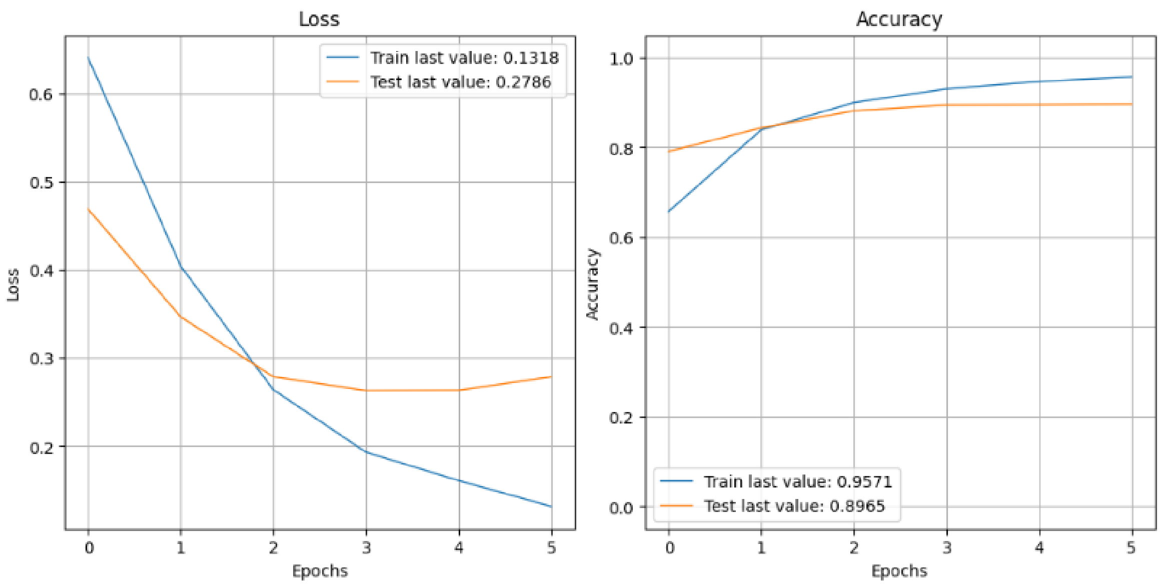
```
Length of sample train_data before preprocessing: 218
Length of sample train_data after preprocessing: 500
Sample train data: [   0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    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]
```

3. Building the Model

Sanity Check



4. Training the Model



Based on the accuracy plot, what do you think the optimal stopping point for your model should have been?

Where the accuracy on the test set was the highest, which is at the final epoch.

5. Evaluating the Model on the Test Data

```
782/782 [==============================] - 27s 30ms/step - loss: 0.3362 - accuracy: 0.8698
test_loss: 0.3361652195453644 test_accuracy: 0.8697999715805054
```

6. Extracting the Word Embeddings

```
Shape of word_embeddings: (10000, 100)

Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (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
_________________________________________________________________
```

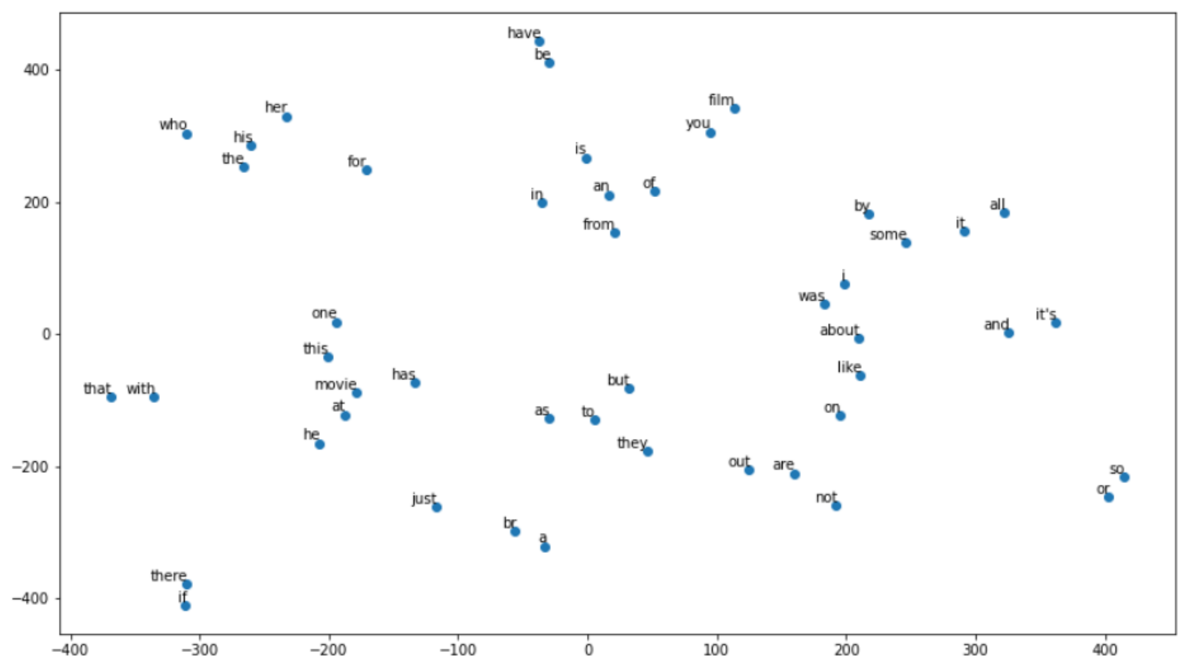
7. Visualizing the Reviews

```
<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
```

8. Visualizing the Word Embeddings

	0	1	2	3	4	5	6	7	8	9	...	90	91	92	93	94	95	96
woods	0.034749	-0.022929	-0.042274	0.003660	0.012328	0.019674	0.047157	0.030400	0.039966	0.005881	...	-0.025695	-0.020659	-0.029948	-0.021821	-0.003663	-0.001473	-0.038691
hanging	0.023770	0.005806	-0.026436	0.023907	-0.036522	-0.003042	0.006303	0.054684	0.021125	-0.000722	...	-0.001673	0.021840	0.005051	-0.039812	-0.014943	-0.031275	-0.044489
woody	0.012521	-0.047144	-0.030312	0.001971	-0.032417	0.008225	-0.027092	-0.019926	-0.009127	0.031912	...	0.002320	-0.051157	-0.002887	-0.041049	-0.020325	0.026693	0.051277
arranged	0.056459	0.052256	0.010988	0.041696	0.042100	-0.034009	0.054902	-0.020743	-0.012765	-0.016781	...	-0.042161	0.016000	-0.027051	0.008761	0.022420	-0.030188	-0.010200
bringing	0.037957	0.017603	-0.037762	0.003273	-0.046980	-0.029365	-0.018095	0.026762	-0.019261	0.013654	...	0.036962	-0.037561	-0.001162	0.003658	0.032474	-0.008875	-0.039150
wooden	0.034470	-0.021360	0.003335	0.041142	-0.032598	0.026591	0.040554	-0.046954	-0.032404	0.025215	...	0.000053	0.031198	0.041692	0.027852	0.034170	0.008012	-0.023703
errors	0.043822	0.024764	0.047095	-0.036585	-0.034633	-0.013739	0.037724	-0.042581	-0.006961	-0.019361	...	0.043611	0.040472	-0.036316	0.026268	-0.022406	-0.001418	-0.038634
dialogs	0.008102	-0.046629	-0.037880	-0.024437	0.014995	0.050349	0.022416	-0.043939	-0.011451	0.002020	...	-0.017627	0.009910	0.038459	0.029377	0.031789	-0.049039	-0.044007
kids	-0.012254	-0.008369	-0.035314	0.030103	-0.018072	-0.021425	0.023406	0.050894	0.006287	-0.021122	...	-0.008116	-0.005464	-0.031098	-0.039549	0.030379	-0.015896	0.032091
uplifting	0.030923	0.008830	0.025629	0.024759	-0.010428	0.028655	0.034355	0.014114	0.003080	0.043952	...	0.005992	-0.012695	0.010913	-0.028248	-0.024437	0.012366	-0.035219

9. Plot your Word Embeddings using t-SNE



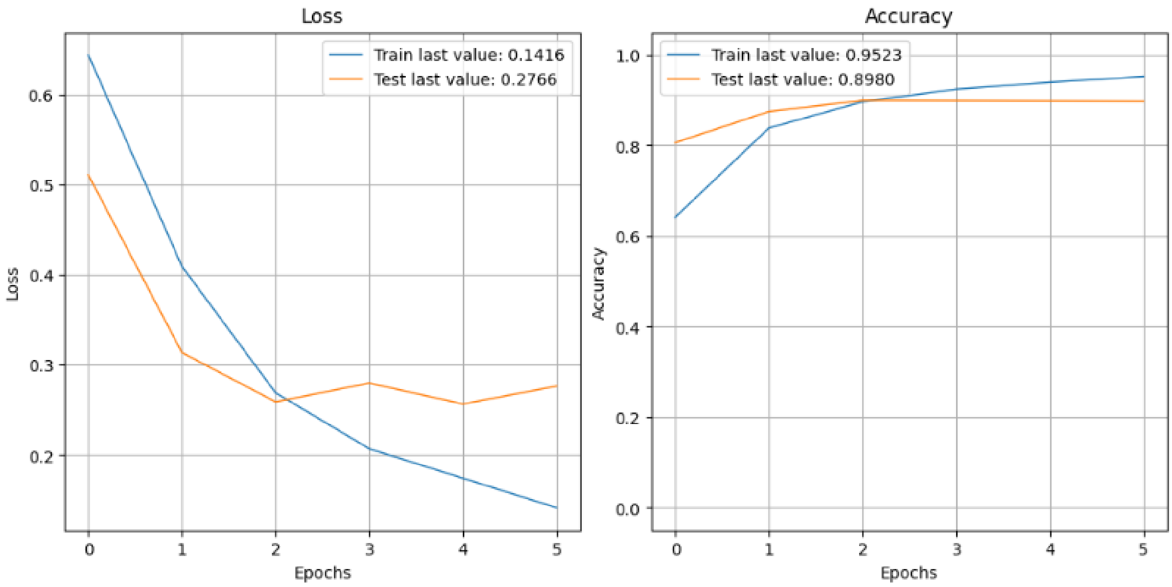
10. Questions

Question 1: What do you observe?

model.summary() is:

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (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
_________________________________________________________________
```

Loss and Accuracy Plots;



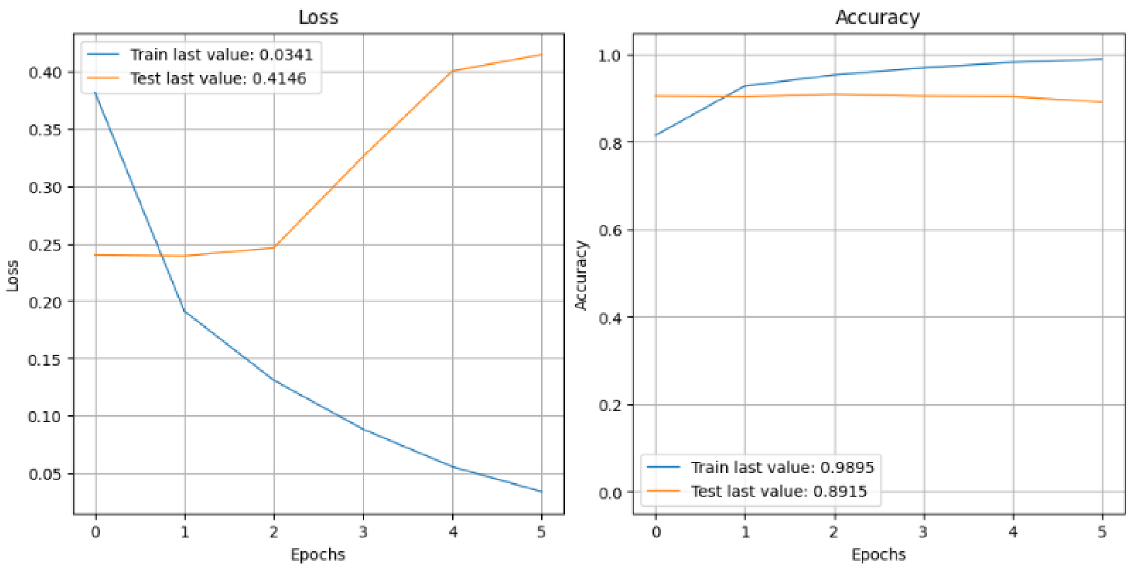
Test Loss and Accuracy:

```
782/782 [==============================] - 21s 23ms/step - loss: 0.3483 - accuracy: 0.8703
test_loss: 0.3482823073863983 test_accuracy: 0.8703200221061707
```

This model has a marginally higher accuracy than the non-dropout model but also a higher loss.

The loss on the Validation set is lowest at Epoch 4, which has a validation accuracy of 89.85%.

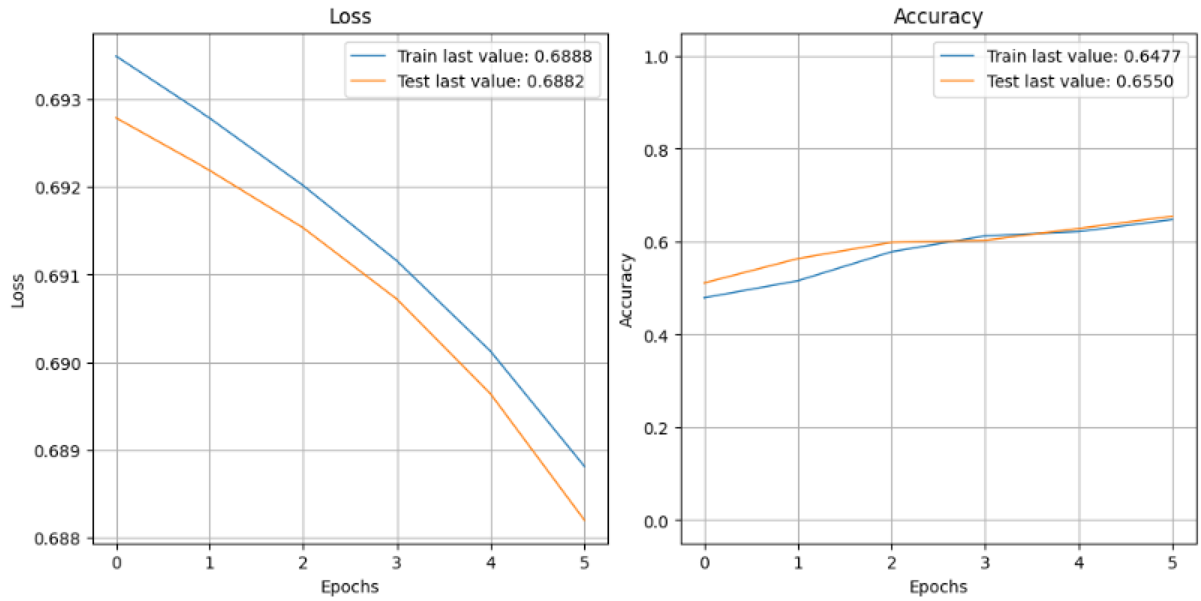
Question 2:
What do you Observe?
Batch Size 1:




```
782/782 [==============================] - 17s 20ms/step - loss: 0.4525 - accuracy: 0.8732
test_loss: 0.45245346426963806 test_accuracy: 0.873199999332428
```

Overfits very quickly and as such the Test loss nearly doubles from epoch 2 to 4. But the Test accuracy still the highest out of the rest. The validation accuracy is highest at epoch 2 at 90.90%. Training this model was very slow.

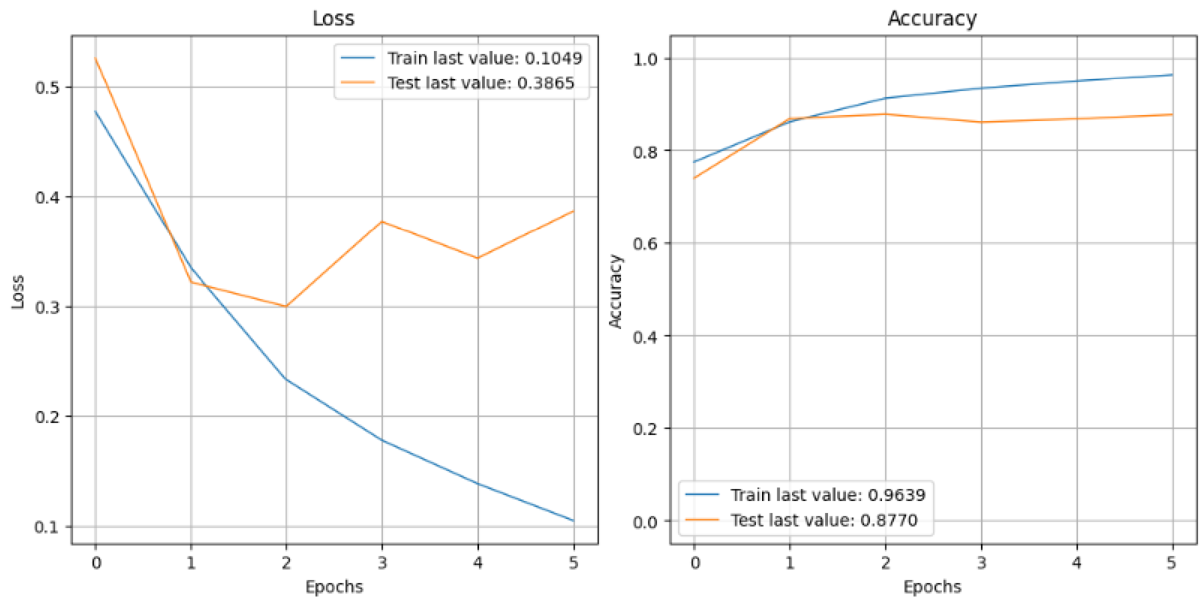
Batch Size len(train_data):




```
782/782 [==============================] - 23s 27ms/step - loss: 0.6880 - accuracy: 0.6456
test_loss: 0.6879885196685791 test_accuracy: 0.6456400156021118
```

The Loss decreases very slowly and the test accuracy stays very low. This is most likely due to very little stochasisity in the training due to the batch size being really high. This model performs very badly.

Batch Size 32:



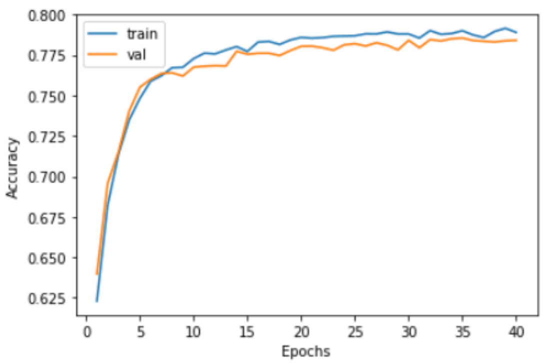

```
782/782 [==============================] - 22s 27ms/step - loss: 0.4342 - accuracy: 0.8619
test_loss: 0.434171199798584 test_accuracy: 0.8618800044059753
```

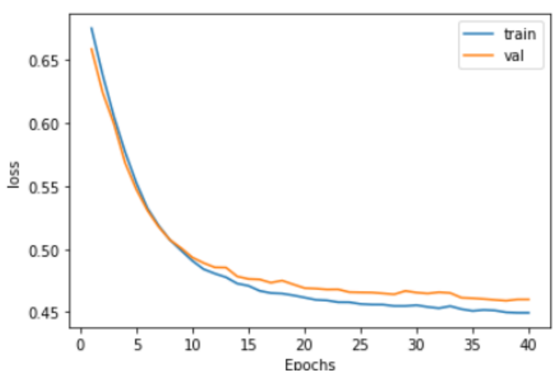
The model overfits and the test accuracy increases after epoch 2. The Test accuracy is still very high. So, when the batch size is very high, the model doesn't have much stochasisity and as such cannot jump out of local optima. When decreasing the batch size, the amount of stochasisity increases but the training time also increases.

Part C:

1. Model 1

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 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
_________________________________________________________________
```



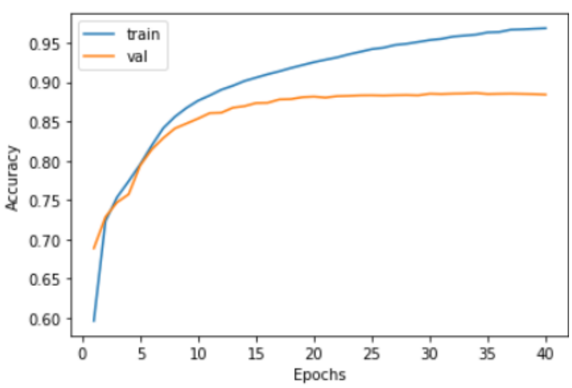


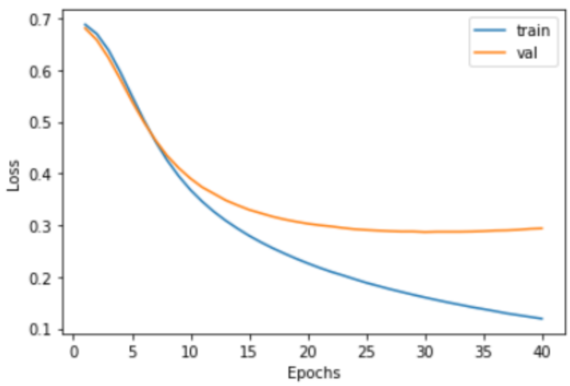

```
782/782 [==============================] - 28s 34ms/step - loss: 0.4613 - accuracy: 0.7789
```

Note that this accuracy/loss is higher/lower than expected due to the fact I used TPU for my computations.

2. Model 2

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, None, 256)         2560000

 global_average_pooling1d_ma  (None, 256)              0
 sked_1 (GlobalAveragePoolin
 g1DMasked)

 dense_2 (Dense)             (None, 1)                 257

=================================================================
Total params: 2,560,257
Trainable params: 2,560,257
Non-trainable params: 0
_________________________________________________________________
```



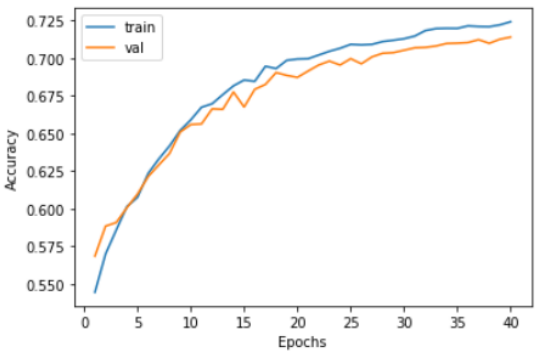


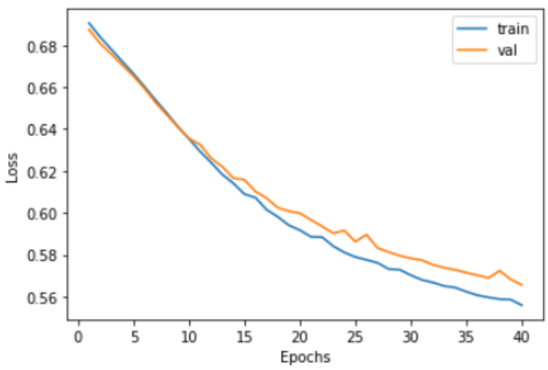

```
782/782 [==============================] - 21s 25ms/step - loss: 0.3105 - accuracy: 0.8746
```

By using a word embedding layer, the accuracy of the model went up by 10%

3. Model 3
    a. Model 3-1

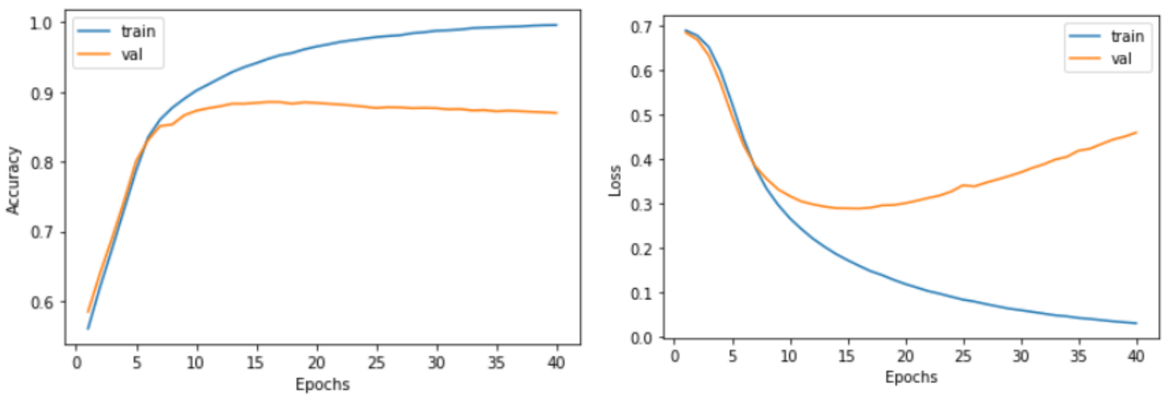
```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 GloVe_Embeddings (Embedding  (None, 256, 300)         120000300
 )

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

 dense_3 (Dense)             (None, 16)                4816

 dense_4 (Dense)             (None, 1)                 17

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






```
782/782 [==============================] - 21s 26ms/step - loss: 0.5670 - accuracy: 0.7089
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 GloVe_Embeddings (Embedding  (None, 256, 300)         120000300 
 )                                                               
                                                                 
 global_average_pooling1d_ma  (None, 300)              0         
 sked_3 (GlobalAveragePoolin                                     
 g1DMasked)                                                      
                                                                 
 dense_5 (Dense)             (None, 16)                4816      
                                                                 
 dense_6 (Dense)             (None, 1)                 17        
                                                                 
=================================================================
Total params: 120,005,133
Trainable params: 120,005,133
Non-trainable params: 0
_________________________________________________________________
```

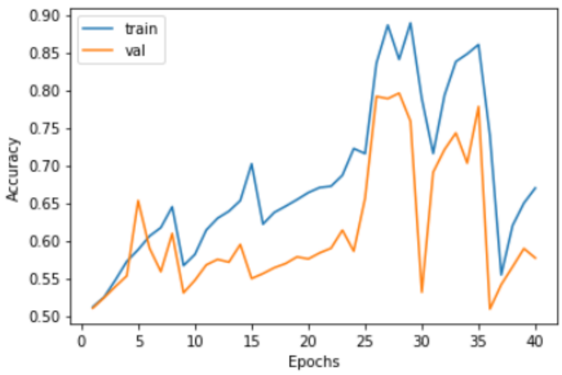


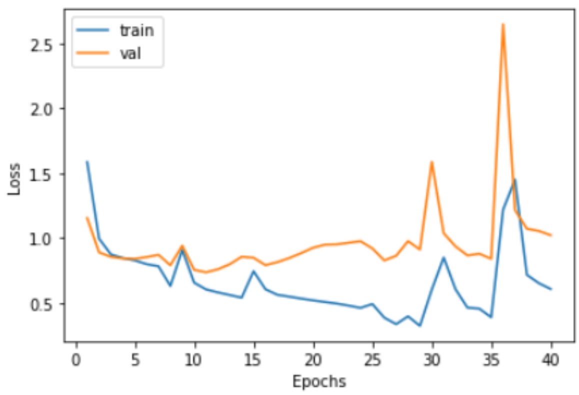

```
782/782 [==============================] - 26s 27ms/step - loss: 0.4908 - accuracy: 0.8551
```

So, from the above we can see that using a fine-tuning a pre-trained word embedding results in a better performance than not fine-tuning it. Interestingly, fine-tuning the pretrained word embeddings overfits very quickly. The highest val accuracy is 88.56% which actually outperforms Model 2, training our own word embedding.

b. Model 3-2

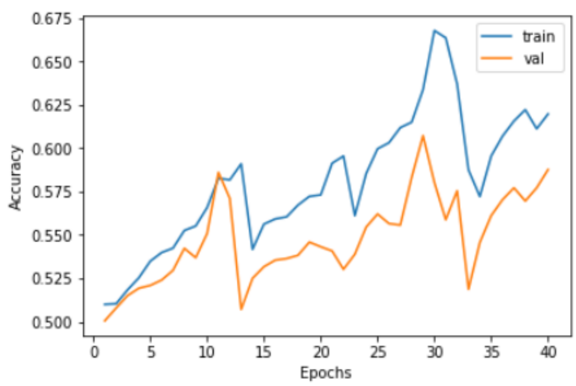
```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 GloVe_Embeddings (Embedding  (None, 256, 300)         120000300
 )

 lstm (LSTM)                 (None, 100)               160400

 dense_7 (Dense)             (None, 1)                 101

=================================================================
Total params: 120,160,801
Trainable params: 120,160,801
Non-trainable params: 0
_________________________________________________________________
```

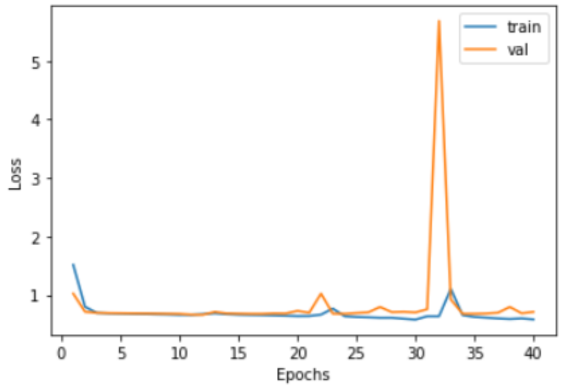





```
782/782 [==============================] - 24s 29ms/step - loss: 1.0130 - accuracy: 0.5714
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 GloVe_Embeddings (Embedding  (None, 256, 300)         120000300
 )

 lstm_1 (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
_________________________________________________________________
```



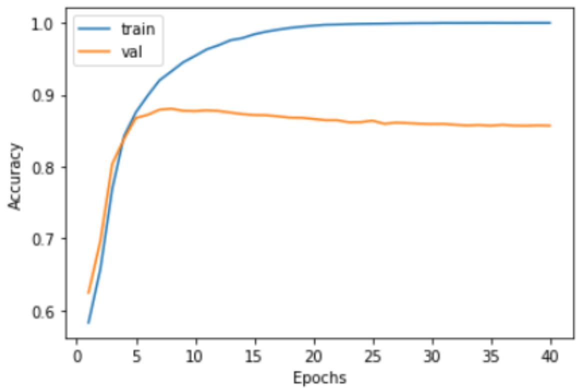


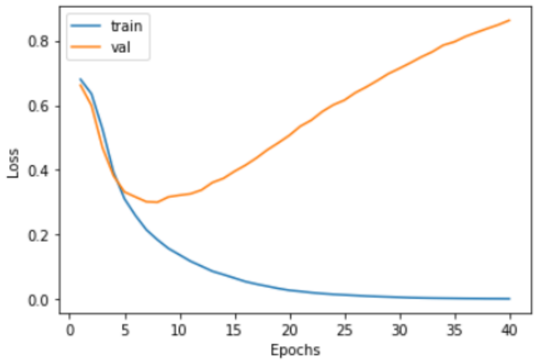

```
782/782 [==============================] - 24s 29ms/step - loss: 0.7186 - accuracy: 0.5864
```

Interestingly using a pre-trained word embedding with and LSTM doesn't perform too well. The non-tweakable pre-trained word embedding doesn't perform well, as expected. But surprisingly the tweakable pre-trained word embedding doesn't perform too well either. This could be due to the fact that I used a TPU to train which caused the model to overfit quickly (as the model does reach a validation accuracy of 79.59%). Another explanation could be that the learning rate is too high. The standard ADAM optimizer's learning rate may be too high for this situation, especially since the embedding is pre-trained.

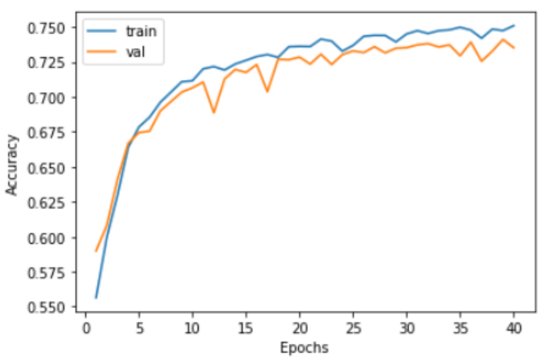
4. Model 4

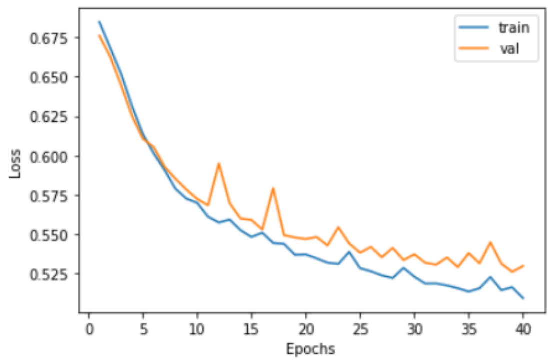
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 GloVe_Embeddings (Embedding  (None, 256, 300)         120000300 
 )                                                               
                                                                 
 global_average_pooling1d_ma  (None, 300)              0         
 sked (GlobalAveragePooling1                                     
 DMasked)                                                        
                                                                 
 dense (Dense)               (None, 100)               30100     
                                                                 
 dense_1 (Dense)             (None, 16)                1616      
                                                                 
 dense_2 (Dense)             (None, 1)                 17        
                                                                 
=================================================================
Total params: 120,032,033
Trainable params: 120,032,033
Non-trainable params: 0
_________________________________________________________________
```






```
782/782 [==============================] - 20s 24ms/step - loss: 0.9167 - accuracy: 0.8450

Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 GloVe_Embeddings (Embedding  (None, 256, 300)         120000300
 )

 global_average_pooling1d_ma  (None, 300)              0
 sked_1 (GlobalAveragePoolin
 g1DMasked)

 dense_3 (Dense)             (None, 100)               30100

 dense_4 (Dense)             (None, 16)                1616

 dense_5 (Dense)             (None, 1)                 17

=================================================================
Total params: 120,032,033
Trainable params: 31,733
Non-trainable params: 120,000,300
_________________________________________________________________
```

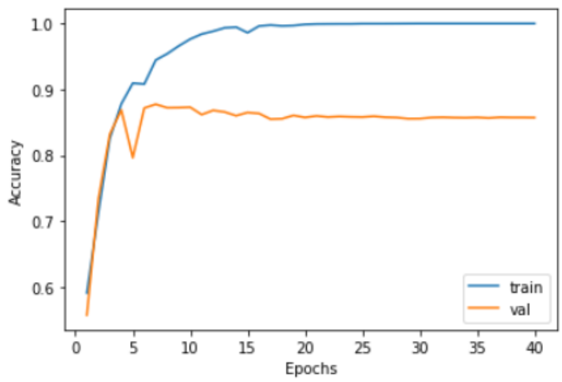


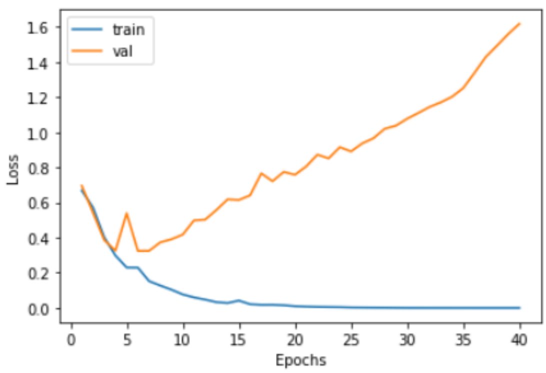



```
782/782 [==============================] - 20s 25ms/step - loss: 0.5345 - accuracy: 0.7306
```

Both models above have a pre-trained embedding with an extra dense layer. The first model’s pre-trained embedding is fine-tunable and as such results in a better performance, as expected. It achieves a final validation accuracy of 84.50%. We can see that this model starts overfitting early, with its highest validation accuracy being 88.06%. The non fine-tunable model performs much better just using an LSTM layer, as earlier This has an accuracy of 73.06%.

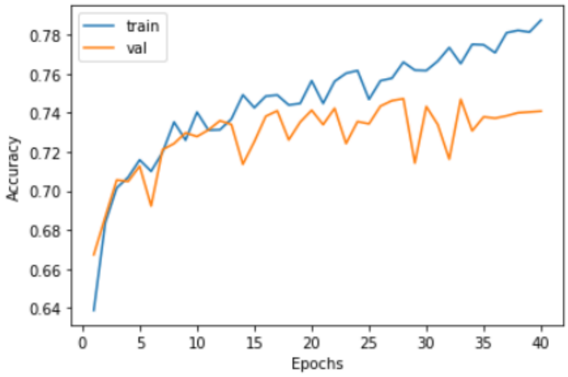
```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 GloVe_Embeddings (Embedding  (None, 256, 300)         120000300
 )

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

 dense_6 (Dense)             (None, 300)               90300

 dense_7 (Dense)             (None, 300)               90300

 dense_8 (Dense)             (None, 100)               30100

 dense_9 (Dense)             (None, 16)                1616

 dense_10 (Dense)            (None, 1)                 17

=================================================================
Total params: 120,212,633
Trainable params: 120,212,633
Non-trainable params: 0
_________________________________________________________________
```

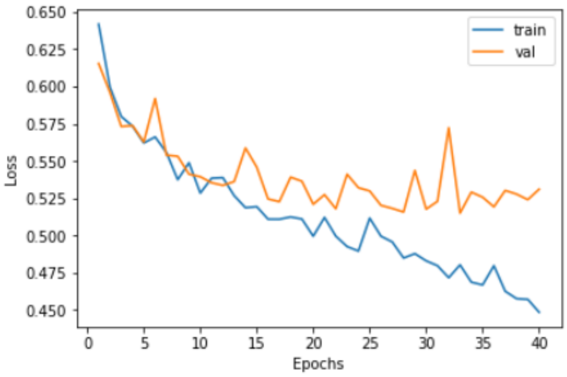





```
782/782 [==============================] - 20s 25ms/step - loss: 1.7369 - accuracy: 0.8431
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 GloVe_Embeddings (Embedding  (None, 256, 300)         120000300 
 )                                                               
                                                                 
 global_average_pooling1d_ma  (None, 300)              0         
 sked_3 (GlobalAveragePoolin                                     
 g1DMasked)                                                      
                                                                 
 dense_11 (Dense)            (None, 300)               90300     
                                                                 
 dense_12 (Dense)            (None, 300)               90300     
                                                                 
 dense_13 (Dense)            (None, 100)               30100     
                                                                 
 dense_14 (Dense)            (None, 16)                1616      
                                                                 
 dense_15 (Dense)            (None, 1)                 17        
                                                                 
=================================================================
Total params: 120,212,633
Trainable params: 212,333
Non-trainable params: 120,000,300
_________________________________________________________________
```





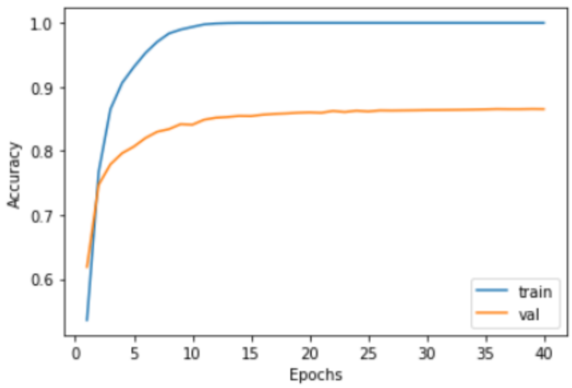

```
782/782 [==============================] - 20s 25ms/step - loss: 0.5394 - accuracy: 0.7349
```

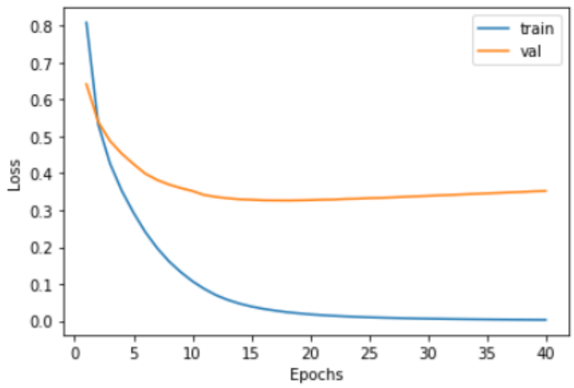
Both models above use the pre-trained word embedding with several hidden layers. As expected again, the fine-tunable word embedding performs better than the non-finetunable one. The fine-tunable model has a final validation accuracy of 84.31%, which is lower than the same fine-tunable model with less hidden layers. I think this is because the model overfits very quickly. Also, the non-fine-tunable model has a final validation accuracy of 73.49%, which is only marginally higher than the non-fine-tunable model with less hidden layers.

5. Model 5
    a. Model 5-1

Here I use pre-trained word embeddings.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 GloVe_Embeddings (Embedding  (None, 256, 300)         120000300
 )

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

 global_max_pooling1d (Globa  (None, 100)              0
 lMaxPooling1D)

 dense (Dense)               (None, 1)                 101

=================================================================
Total params: 120,180,501
Trainable params: 120,180,501
Non-trainable params: 0
_________________________________________________________________
```



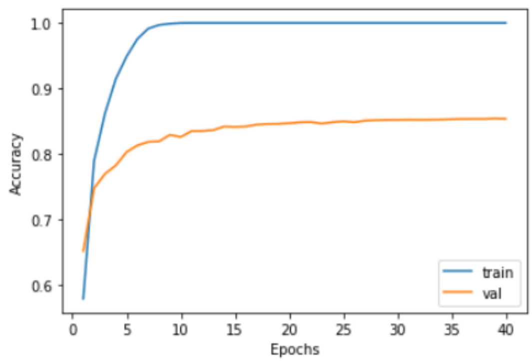


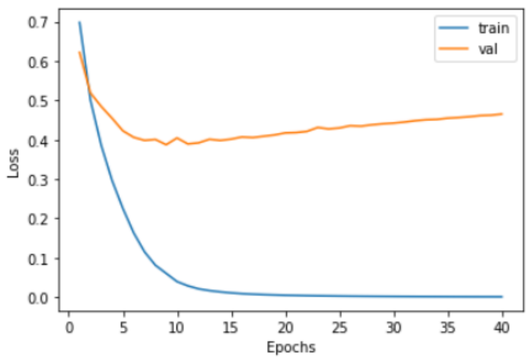

```
782/782 [==============================] - 18s 22ms/step - loss: 0.3548 - accuracy: 0.8638
```

Here I have only used a fine-tunable Model. This has a final validation accuracy of 86.38%, which is also the highest.

    b. Model 5-2

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 GloVe_Embeddings (Embedding  (None, 256, 300)         120000300
 )

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

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

 global_max_pooling1d_1 (Glo  (None, 100)              0
 balMaxPooling1D)

 dense_1 (Dense)             (None, 1)                 101

=================================================================
Total params: 120,240,601
Trainable params: 120,240,601
Non-trainable params: 0
_________________________________________________________________
```






```
782/782 [==============================] - 18s 22ms/step - loss: 0.4557 - accuracy: 0.8504
```

This final model has 2 Convolution layers with a fine-tunable pre-trained embedding layer. Interestingly, it doesn't perform as well (final validation accuracy of 85.04%) as only using 1 convolution layer. This is likely due to overfitting and due to the number of parameters, getting stuck in a local optimal. This could also be due to the fact that with a convolution, we are capturing the relationship in subsequences rather than the whole sequence.

Part D:

1. Implementing the Encoder
    a. Firstly I created the Embedding lookups which just involves creating 2 embedding layers, one for the source, another for the target.
        i. The source embedding will be configured for the source text (I.e., we use self.vocab_source_size for the input dimensions into the source embedding)
        ii. I set mask_zero=True to ensure that padding is ignored
    b. Now that the embeddings are created, I pass the inputs of the encoder (source_words, and target_words) to the correct embedding.
        i. At the same time, I have created and used a Dropout layer, which receives the output of the embedding layer as input
    c. I now create the encoder LSTM, which has return_state=True so the encoder has access to the hidden state and cell state (which are used to initialize the initial hidden states of the Training 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, mask_zero=True)
embeddings_target = Embedding(self.vocab_target_size, self.embedding_size, mask_zero=True)

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

target_words_embeddings = Dropout(rate = 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_lstm = LSTM(self.hidden_size, recurrent_dropout=self.hidden_dropout_rate, return_sequences=True, return_state=True)

encoder_outputs, encoder_state_h, encoder_state_c = encoder_lstm(source_words_embeddings)
"""
End Task 1
"""
```

2. Implementing the Decoder and the Inference Loop
    a. In this step, I get the initial states for the Inference Decoder from the final hidden states of the Training Decoder. This is passed to the decoder_lstm
    b. Add attention if needed by using the decoder_attention trained in the Training Decoder.
    c. Then, simply pass the decoder outputs with or without attention to the decoder dense layer (From Training Decoder). This is the output of the model.

```
"""
Task 2 decoder for inference

Start
"""
# Task 1 (a.) Get the decoded outputs
print('\nPutting together the decoder states')
# get the initial 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
"""
```

```
It looks like you forgot to detokenize your test data, which may hurt your score.
If you insist your data is detokenized, or don't care, you can suppress this message with the `force` parameter.
Model BLEU score: 5.53
Time used for evaluate on dev set: 0 m 10 s
Starting training epoch 8/10
240/240 [==============================] - 30s 124ms/step - loss: 1.4371 - accuracy: 0.3923
Time used for epoch 8: 0 m 29 s
Evaluating on dev set after epoch 8/10:
That's 100 lines that end in a tokenized period ('.')
It looks like you forgot to detokenize your test data, which may hurt your score.
If you insist your data is detokenized, or don't care, you can suppress this message with the `force` parameter.
Model BLEU score: 5.66
Time used for evaluate on dev set: 0 m 9 s
Starting training epoch 9/10
240/240 [==============================] - 30s 125ms/step - loss: 1.4123 - accuracy: 0.3964
Time used for epoch 9: 0 m 30 s
Evaluating on dev set after epoch 9/10:
That's 100 lines that end in a tokenized period ('.')
It looks like you forgot to detokenize your test data, which may hurt your score.
If you insist your data is detokenized, or don't care, you can suppress this message with the `force` parameter.
Model BLEU score: 5.72
Time used for evaluate on dev set: 0 m 10 s
Starting training epoch 10/10
240/240 [==============================] - 30s 123ms/step - loss: 1.3927 - accuracy: 0.3992
Time used for epoch 10: 0 m 29 s
Evaluating on dev set after epoch 10/10:
That's 100 lines that end in a tokenized period ('.')
It looks like you forgot to detokenize your test data, which may hurt your score.
If you insist your data is detokenized, or don't care, you can suppress this message with the `force` parameter.
Model BLEU score: 5.67
Time used for evaluate on dev set: 0 m 10 s
Training finished!
Time used for training: 6 m 46 s
Evaluating on test set:
That's 100 lines that end in a tokenized period ('.')
It looks like you forgot to detokenize your test data, which may hurt your score.
If you insist your data is detokenized, or don't care, you can suppress this message with the `force` parameter.
Model BLEU score: 6.14
Time used for evaluate on test set: 0 m 10 s
```


```
Model BLEU score: 6.14
Time used for evaluate on test set: 0 m 10 s
```


The BLEU score is low as expected, with no attention, this model doesn't perform too well. Without Attention this model performs badly due to the fact it cannot determine which parts of the input are relevant when making a prediction. This means that each parts of the input is weighed the same even if they are not relevant.

3. Adding Attention
   Assume 0-indexing on dim.
   a. I calculate the Luong score by doing batch_dot on the encoder_outputs and decoder_outputs on the second dim
   b. Then, softmax on the first dim.

c. Thereafter, expand the last dim of the luong_score
d. Subsequently, expand the second to last dim of the encoder_Outputs
e. Then, do elementwise multiplication fo the luong_score and encoder_outputs
f. FInally, sum along the second dim (index 1).

This gives the correct luong_score.

```
"""
Task 3 attention

Start
"""
luong_score = K.batch_dot(encoder_outputs,decoder_outputs, (2))
luong_score = K.softmax(luong_score, 1)
luong_score = K.expand_dims(luong_score,-1)
encoder_outputs = K.expand_dims(encoder_outputs, 2)
encoder_vector = luong_score * encoder_outputs
encoder_vector = K.sum(encoder_vector, 1)

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

```
That's 100 lines that end in a tokenized period ('.')
It looks like you forgot to detokenize your test data, which may hurt your score.
If you insist your data is detokenized, or don't care, you can suppress this message with the `force` parameter.
Model BLEU score: 14.91
Time used for evaluate on dev set: 0 m 10 s
Starting training epoch 8/10
240/240 [==============================] - 30s 124ms/step - loss: 0.9598 - accuracy: 0.5366
Time used for epoch 8: 0 m 29 s
Evaluating on dev set after epoch 8/10:
That's 100 lines that end in a tokenized period ('.')
It looks like you forgot to detokenize your test data, which may hurt your score.
If you insist your data is detokenized, or don't care, you can suppress this message with the `force` parameter.
Model BLEU score: 15.08
Time used for evaluate on dev set: 0 m 10 s
Starting training epoch 9/10
240/240 [==============================] - 31s 128ms/step - loss: 0.9308 - accuracy: 0.5460
Time used for epoch 9: 0 m 30 s
Evaluating on dev set after epoch 9/10:
That's 100 lines that end in a tokenized period ('.')
It looks like you forgot to detokenize your test data, which may hurt your score.
If you insist your data is detokenized, or don't care, you can suppress this message with the `force` parameter.
Model BLEU score: 15.11
Time used for evaluate on dev set: 0 m 10 s
Starting training epoch 10/10
240/240 [==============================] - 30s 127ms/step - loss: 0.9070 - accuracy: 0.5531
Time used for epoch 10: 0 m 30 s
Evaluating on dev set after epoch 10/10:
That's 100 lines that end in a tokenized period ('.')
It looks like you forgot to detokenize your test data, which may hurt your score.
If you insist your data is detokenized, or don't care, you can suppress this message with the `force` parameter.
Model BLEU score: 14.65
Time used for evaluate on dev set: 0 m 10 s
Training finished!
Time used for training: 6 m 58 s
Evaluating on test set:
That's 100 lines that end in a tokenized period ('.')
It looks like you forgot to detokenize your test data, which may hurt your score.
If you insist your data is detokenized, or don't care, you can suppress this message with the `force` parameter.
Model BLEU score: 14.98
Time used for evaluate on test set: 0 m 10 s
```

```
Model BLEU score: 14.98
Time used for evaluate on test set: 0 m 10 s
```

As expected, this model performs better than using no attention. Attention works by giving different weights to different parts of the input. This means that it can give a high weight to the most relevant parts.

Part E

1. Data Processing

```
x_dev_aspect_int[0]:
[ 101 8974  102    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0]
x_dev_aspect_masks[0]:
[1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
x_dev_review_int[0]:
[  101  2044  1037  3232  1997  8974  1010  1996 18726  1011  1011  1045
  2066  1996 27940  1013 24792  2621  4897  1998  1996 13675 11514  6508
 26852  1011  1011  2175  2091  2307  1012   102     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0]
x_dev_review_masks[0]:
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

```
print(x_dev_int[0])
print(x_dev_masks[0],'\n')
print(x_dev_int_np[0])
print(x_dev_masks_np[0]) # sentence + aspect
```

```
[  101  2044  1037  3232  1997  8974  1010  1996 18726  1011  1011  1045
  2066  1996 27940  1013 24792  2621  4897  1998  1996 13675 11514  6508
 26852  1011  1011  2175  2091  2307  1012   102  8974   102     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     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 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

[  101  2044  1037  3232  1997  8974  1010  1996 18726  1011  1011  1045
  2066  1996 27940  1013 24792  2621  4897  1998  1996 13675 11514  6508
 26852  1011  1011  2175  2091  2307  1012   102  8974   102     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     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 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

```
print(tokenize(train_review[0] + "[SEP]" + train_aspect[0], tokenizer))
print(train_review[0] + " [SEP] " + train_aspect[0])
print(tokenize(train_review[0],tokenizer))
```

```
(array([  101,  1996, 25545,  2003,  2025,  2569,  2012,  2035,  2021,
        2037,  2833,  1998,  6429,  7597,  2191,  2039,  2005,  2009,
        1012,   102, 25545,   102,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0], dtype=int32), array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int32), array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int32))
the decor is not special at all but their food and amazing prices make up for it. [SEP] decor
(array([  101,  1996, 25545,  2003,  2025,  2569,  2012,  2035,  2021,
        2037,  2833,  1998,  6429,  7597,  2191,  2039,  2005,  2009,
        1012,   102,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0], dtype=int32), array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int32), array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int32))
```

2. Basic classifiers using BERT

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

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

 tf_distil_bert_for_sequence_cl  TFSequenceClassifie  66955779   ['input_token[0][0]',
 assification (TFDistilBertForS  rOutput(loss=None,               'masked_token[0][0]']
 equenceClassification)         logits=(None, 3),
                                 hidden_states=None
                                , attentions=None)

==================================================================================================
Total params: 66,955,779
Trainable params: 66,955,779
Non-trainable params: 0
__________________________________________________________________________________________________
```

```
42/42 [==============================] - 7s 91ms/step - loss: 1.0286 - accuracy: 0.8174
[1.0286388397216797, 0.817365288734436]
```

```
Model: "Model2_BERT"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_1 (InputLayer)           [(None, 128)]        0           []

 input_2 (InputLayer)           [(None, 128)]        0           []

 tf_distil_bert_model (TFDistil  TFBaseModelOutput(l  66362880   ['input_1[0][0]',
 BertModel)                     ast_hidden_state=(N               'input_2[0][0]']
                                one, 128, 768),
                                 hidden_states=None
                                , attentions=None)

 global_average_pooling1d_maske  (None, 768)         0           ['tf_distil_bert_model[0][0]']
 d (GlobalAveragePooling1DMaske
 d)

 dense (Dense)                  (None, 16)           12304       ['global_average_pooling1d_masked
                                                                 [0][0]']

 dense_1 (Dense)                (None, 3)            51          ['dense[0][0]']

==================================================================================================
Total params: 66,375,235
Trainable params: 66,375,235
Non-trainable params: 0

42/42 [==============================] - 8s 93ms/step - loss: 0.6841 - accuracy: 0.8204
[0.6841117143630981, 0.8203592896461487]
```

The second model with distilBERT, a dense layer and a masked pooling layer performs better than just using distilBERT for classification, but only just. Interestingly, the second model starts overfitting very quickly with the validation accuracy plateuing quickly.

For lab 4, we didn't do any sentiment analysis???

3. Advanced classifier using BERT

```
Model: "model_1"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_3 (InputLayer)           [(None, 128)]        0           []

 input_4 (InputLayer)           [(None, 128)]        0           []

 tf_distil_bert_model_1 (TFDist  TFBaseModelOutput(l  66362880   ['input_3[0][0]',
 ilBertModel)                   ast_hidden_state=(N               'input_4[0][0]']
                                one, 128, 768),
                                 hidden_states=None
                                , attentions=None)

 lstm (LSTM)                    (None, 100)          347600      ['tf_distil_bert_model_1[0][0]']

 dense_2 (Dense)                (None, 3)            303         ['lstm[0][0]']

==================================================================================================
Total params: 66,710,783
Trainable params: 66,710,783
Non-trainable params: 0
__________________________________________________________________________________________________
```

```
42/42 [==============================] - 9s 110ms/step - loss: 0.4810 - accuracy: 0.8353
[0.4810287654399872, 0.8353293538093567]
```

Here I used distilBERT with an LSTM layer. It outperforms the models from 2, the basic classifier. This could be because LSTMs perform a lot better at language modelling than just using a Dense Layer. But because of the use of the distrilBERT model, the LSTM version, here, doesn't significantly outperform the non-LSTM versions, with only about 1-1.5% improvement in performance.