ECS7001 – NN & NLP

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

Sparsh Verma – 220996233

Part A: Word Embedding with Word2Vec

1. Processing the training corpus:

```
[ ] len(normalized_corpus)

    12498

[ ] normalized_corpus[10]

    'therefore succession norland estate really important sisters fortune independent might arise father inherit
    ing property could small'
```

2. Creating the corpus vocabulary and preparing the dataset:

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

    Sample word2idx:  [('agricultural', 1), ('enumeration', 2), ('share', 3), ('habitation', 4), ('impending', 5), ('sample', 6), ('unexhilarating', 7), ('boils', 8), ('clearest', 9), ('note', 10)]

[ ] print('\nSample idx2word:', list(idx2word.items())[:10])

    Sample idx2word: [(1, 'agricultural'), (2, 'enumeration'), (3, 'share'), (4, 'habitation'), (5, 'impending'), (6, 'sample'), (7, 'unexhilarating'), (8, 'boils'), (9, 'clearest'), (10, 'note')]

[ ] print('\nSample normalized corpus:', normalized_corpus[:3])

    Sample normalized corpus: ['sense sensibility jane austen', 'family dashwood long settled sussex', 'estate large residence norland park centre property many generations lived respectable manner engage general good opinion surrounding

[ ] print('\nAbove sentence as a list of ids:' , sents_as_ids[:30])

    Above sentence as a list of ids: [[8252, 5848, 4618, 2226], [8480, 870, 2603, 2529, 4989], [9449, 4461, 84, 8766, 1738, 5557, 4215, 9790, 6997, 75, 2129, 3415, 9197, 8346, 9845, 4549, 3801, 3649], [3125, 6446, 9449, 8214, 4846, 75, 1
```

3. Building the skip-gram neural network architecture:

```
[ ] model.summary()

    Model: "model"
    ____________________________________________________________________________________________________
     Layer (type)                   Output Shape         Param #     Connected to
    ====================================================================================================
     input_2 (InputLayer)           [(None, 1)]          0           []

     input_3 (InputLayer)           [(None, 1)]          0           []

     target_embed_layer (Embedding)  (None, 1, 100)      1003900     ['input_2[0][0]']

     context_embed_layer (Embedding  (None, 1, 100)      1003900     ['input_3[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]']

     dense (Dense)                  (None, 1)            2           ['dot[0][0]']

    ====================================================================================================
    Total params: 2,007,802
    Trainable params: 2,007,802
    Non-trainable params: 0
    ____________________________________________________________________________________________________
```

4. Training the models: The input is the numeric/vector representation of textual data/word/string. The output is a single vector representation for each word which gives us the probability of being chosen as the next word. An Input layer with two inputs, target word and context word. And embedding of the above layer and the transforming the embedding layer. Finally taking the dot product. The words which are presented here do not capture the contextual meaning internally thus the result we get does not capture semantic relationship very well. We need rich dataset such as which is artificially created word2vec.
5. Getting the word embedding:

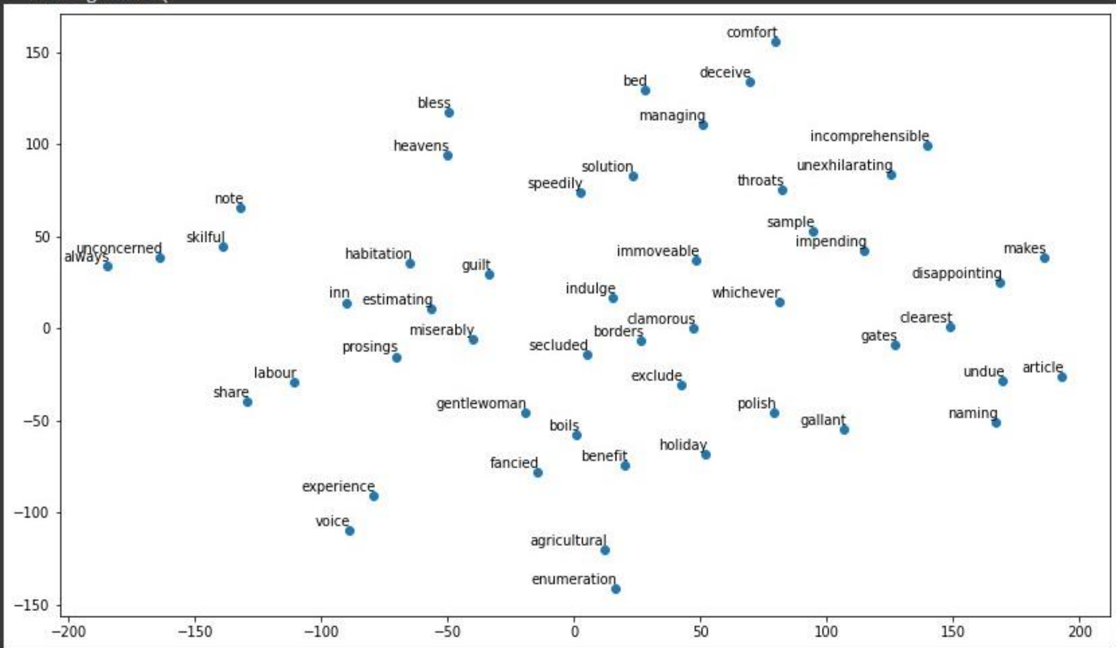
```
import pandas as pd
pd.DataFrame(word_embeddings, index=list(idx2word.values())).head(10)
```

	0	1	2	3	4	5	6	7	8	
agricultural	-0.050827	-0.000346	-0.014999	-0.024274	0.032542	0.013817	-0.059165	-0.021513	0.007517	0.01
enumeration	-0.016764	0.018393	-0.014047	-0.036145	0.014646	0.050686	-0.029557	-0.052179	0.004057	0.0C
share	0.132191	-0.105709	0.113350	-0.014739	0.095044	0.149274	0.371555	-0.087336	-0.030958	0.15
habitation	-0.037607	-0.037124	-0.007126	0.004274	-0.063412	0.021302	0.039118	-0.023034	-0.126960	0.07
impending	-0.029901	0.000949	0.006445	-0.062367	0.006158	-0.009940	-0.060981	-0.010933	-0.046151	0.06
sample	0.024184	-0.001557	-0.035046	-0.005544	0.041931	0.028603	-0.042631	-0.012820	-0.020909	0.03
unexhilarating	0.002843	-0.037773	-0.048014	-0.072212	0.001170	-0.003389	-0.016003	-0.048277	-0.013797	-0.0C
boils	-0.002897	0.008363	-0.012954	-0.002966	0.006714	0.008126	-0.022727	-0.021139	-0.025995	0.01
clearest	-0.004789	-0.003221	-0.019409	-0.041609	0.034804	0.018842	-0.073023	-0.033172	-0.040464	0.1C
note	-0.256145	-0.093022	0.057054	-0.118053	-0.234950	-0.081610	0.269425	0.102358	-0.107117	0.16

10 rows × 100 columns

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

```
/usr/local/lib/python3.8/dist-packages/sklearn/manifold/_t_sne.py:790: FutureWarning: The default learning ra
  warnings.warn(
/usr/local/lib/python3.8/dist-packages/sklearn/manifold/_t_sne.py:982: FutureWarning: The PCA initialization
  warnings.warn(
```



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[1]), type(train_data[1]))
    print('Length of sample train_data after preprocessing:', len(preprocessed_train_data[0]), type(preprocessed

    Length of sample train_data before preprocessing: 189 <class 'list'>
    Length of sample train_data after preprocessing: 500 <class 'numpy.ndarray'>
```

2. Building the model:

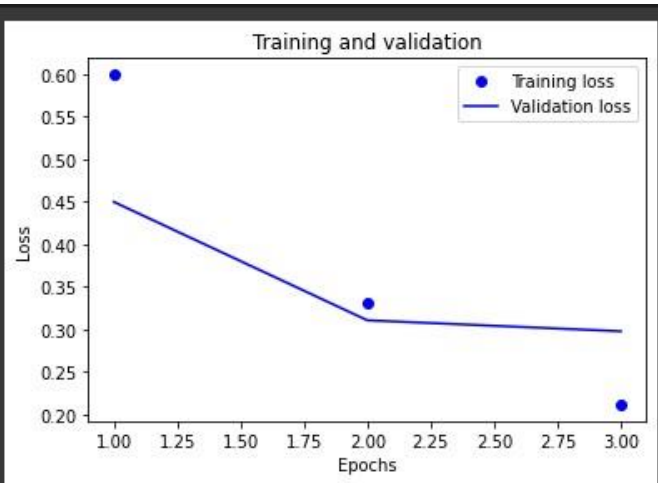
```
[ ] print(model.summary())

    Model: "sequential_1"
    _________________________________________________________________
     Layer (type)                Output Shape              Param #
    =================================================================
     embedding_1 (Embedding)     (None, 500, 100)          1000000

     lstm_1 (LSTM)               (None, 100)               80400

     dense_2 (Dense)             (None, 1)                 101

    =================================================================
    Total params: 1,080,501
    Trainable params: 1,080,501
    Non-trainable params: 0
    _________________________________________________________________
    None
```

3. Section 4, training the model:

```
[ ] history = model.fit(preprocessed_train_data,
                        train_labels,
                        epochs=3,
                        batch_size=512,
                        validation_split=0.08,
                        verbose=1)

    Epoch 1/3
    45/45 [==============================] - 128s 3s/step - loss: 0.6106 - accuracy: 0.6895 - val_loss: 0.4576 -
    Epoch 2/3
    45/45 [==============================] - 124s 3s/step - loss: 0.3661 - accuracy: 0.8527 - val_loss: 0.3373 -
    Epoch 3/3
    45/45 [==============================] - 123s 3s/step - loss: 0.2550 - accuracy: 0.9038 - val_loss: 0.3089 -
```

4. Evaluating the model on the test data (section 5):

```
results = model.evaluate(processed_test_data, test_labels)

782/782 [==============================] - 62s 80ms/step - loss: 0.3219 - accuracy: 0.8651

[ ] print(results)

    [0.3218930661678314, 0.865119993686676]
```



5. Section 6, extracting the word embeddings:

```
[82] embed_layer = model.get_layer('embedding').get_weights()[0]
```

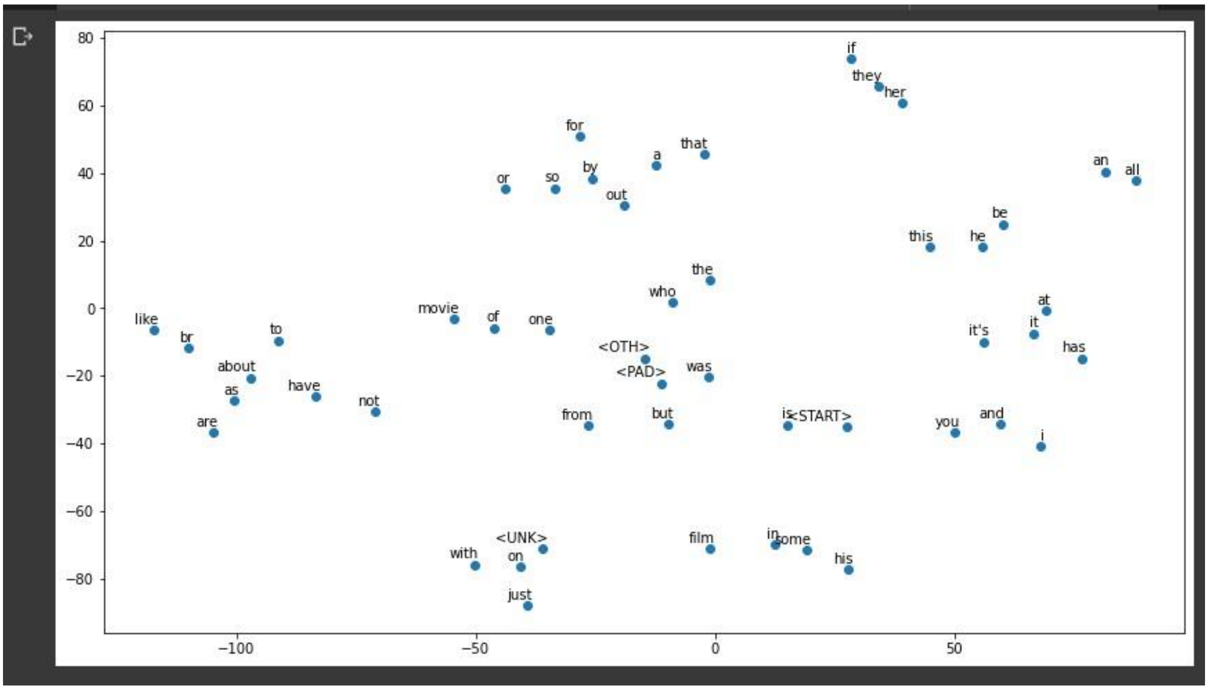
```
[83] print('Shape of word_embeddings:', embed_layer.shape)

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

```
[84] print(model.summary())

     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
     _________________________________________________________________
     None
```

6. Visualizing the reviews:

```
[86] idx2word = {v: k for k,v in word2idx.items()}
     print(' '.join(idx2word[idx] for idx in train_data[0]))

     <START> this film was just brilliant casting location scenery story direction everyone's really suited the pa
```

7. Visualizing the word embeddings:

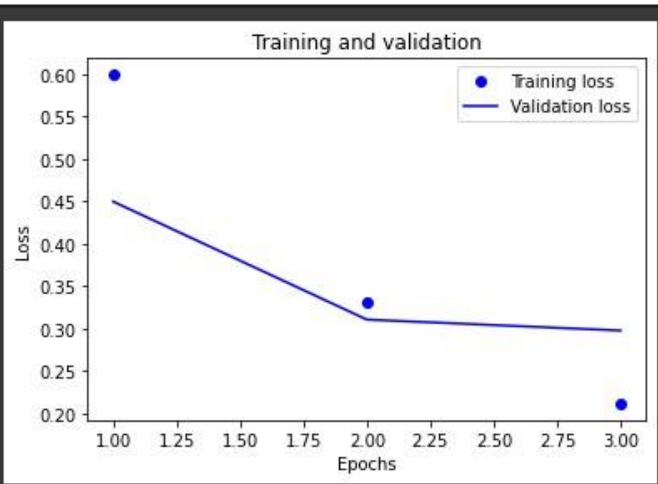


8. Section 9:

```
model2 = Sequential()
EMBED_SIZE = 100
# model.add()
model2.add(Embedding(VOCAB_SIZE,EMBED_SIZE,input_length=MAXIMUM_LENGTH))
model2.add(Dropout(0.2))
model2.add(LSTM(100, activation='tanh'))
model2.add(Dropout(0.2))
model2.add(Dense(1,activation='sigmoid',input_shape=(1,)))
print(model2.summary())
```

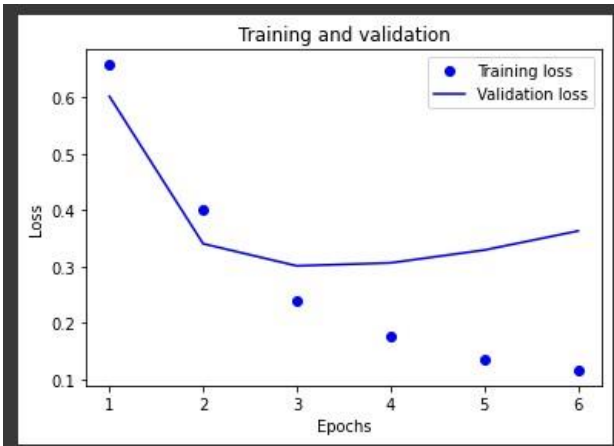
```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, 500, 100)          1000000

 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
_________________________________________________________________
None
```

Model with dropout with 3 epoch



Model with dropout with 6 epoch

```
[95] results = model2.evaluate(processed_test_data, test_labels)
     print(results)

     782/782 [==============================] - 69s 89ms/step - loss: 0.3751 - accuracy: 0.8704
     [0.3751215934753418, 0.8704400062561035]
```

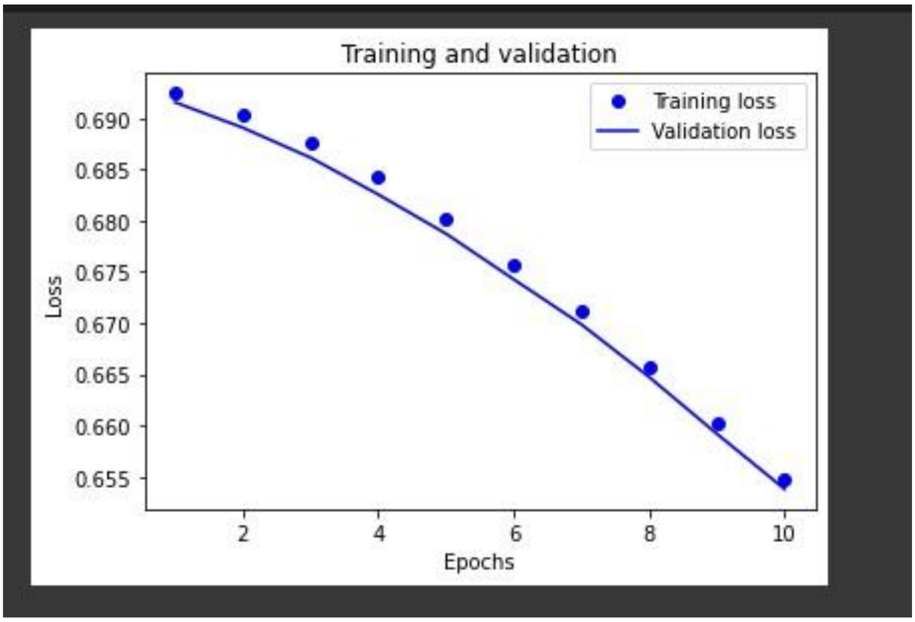


From the above we can see that as we add dropouts and increase epoch, although there is a slight increase in accuracy, the model converges better here.
Smaller the batch size started converging at a local minima very early, as I increase the batch size there was a significant degradation in the quality of the model, as measured by its ability to generalize.

Part C: Comparing Classification Models

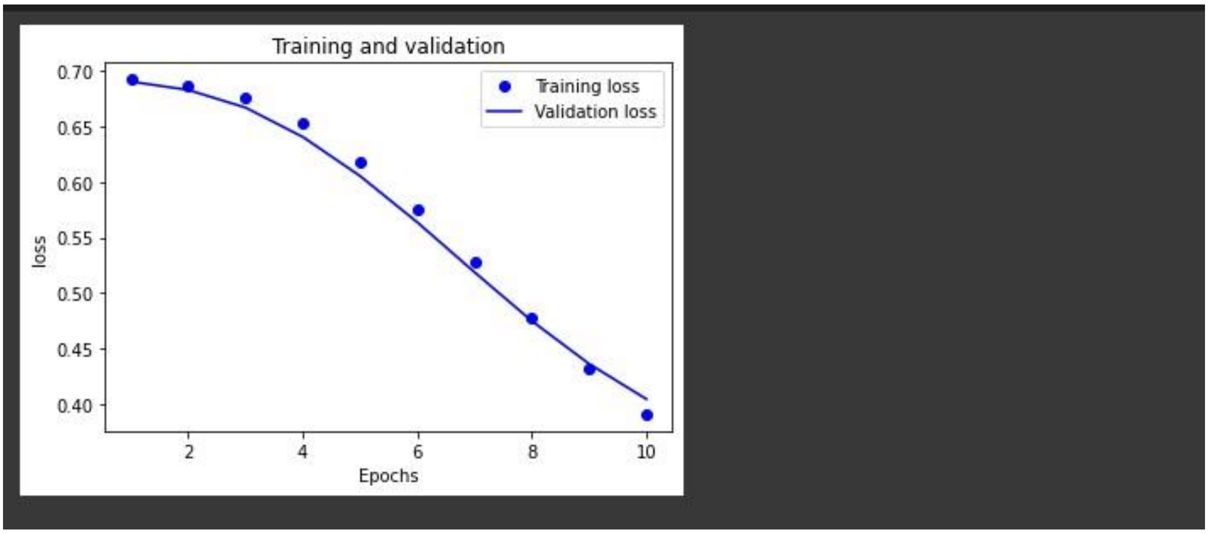
1. Build a neural network classifier using one-hot word vectors, and train and evaluate it:

```
model = Sequential()
model.add(OneHot(VOCAB_SIZE,input_length=MAX_SEQUENCE_LENGTH))
model.add(GlobalAveragePooling1DMasked())
model.add(Dense(16,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
model.summary()

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
_________________________________________________________________
```



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:

```
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
_________________________________________________________________
None
Epoch 1/10
30/30 [==============================] - 2s 31ms/step - loss: 0.6922 - accuracy: 0.5288 - val_loss: 0.6902 -
Epoch 2/10
30/30 [==============================] - 1s 20ms/step - loss: 0.6869 - accuracy: 0.6931 - val_loss: 0.6827 -
Epoch 3/10
30/30 [==============================] - 1s 23ms/step - loss: 0.6751 - accuracy: 0.7213 - val_loss: 0.6668 -
Epoch 4/10
30/30 [==============================] - 1s 25ms/step - loss: 0.6523 - accuracy: 0.7409 - val_loss: 0.6404 -
Epoch 5/10
30/30 [==============================] - 1s 26ms/step - loss: 0.6185 - accuracy: 0.7786 - val_loss: 0.6053 -
Epoch 6/10
30/30 [==============================] - 1s 40ms/step - loss: 0.5758 - accuracy: 0.8130 - val_loss: 0.5636 -
Epoch 7/10
30/30 [==============================] - 1s 42ms/step - loss: 0.5282 - accuracy: 0.8315 - val_loss: 0.5187 -
Epoch 8/10
30/30 [==============================] - 1s 48ms/step - loss: 0.4782 - accuracy: 0.8502 - val_loss: 0.4749 -
Epoch 9/10
30/30 [==============================] - 1s 42ms/step - loss: 0.4316 - accuracy: 0.8651 - val_loss: 0.4363 -
Epoch 10/10
30/30 [==============================] - 1s 23ms/step - loss: 0.3910 - accuracy: 0.8757 - val_loss: 0.4046 -
782/782 [==============================] - 3s 4ms/step - loss: 0.4128 - accuracy: 0.8471
[0.4127701222896576, 0.8471199870109558]
```



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:
   Creating a pre-trained embedding layer:

```
[129] def createPretrainedEmbeddingLayer(wordToGlove, wordToIndex, isTrainable):
          vocabLen = len(wordToIndex) + 1
          embDim = next(iter(wordToGlove.values())).shape[0]

          embeddingMatrix = np.zeros((vocabLen, embDim))
          for word, index in wordToIndex.items():
              embeddingMatrix[index, :] = wordToGlove[word]

          embeddingLayer = Embedding(vocabLen, embDim, embeddings_initializer=Constant(embeddingMatrix), trainable
          return embeddingLayer
```

   Freezing the weights. To create a model:

```
[137] from keras.initializers import Constant
      wordToIndex, indexToWord, wordToGlove = readGloveFile('/content/glove.6B.300d.txt')
      embeddingLayer = createPretrainedEmbeddingLayer(wordToGlove, wordToIndex, isTrainable=True)
```

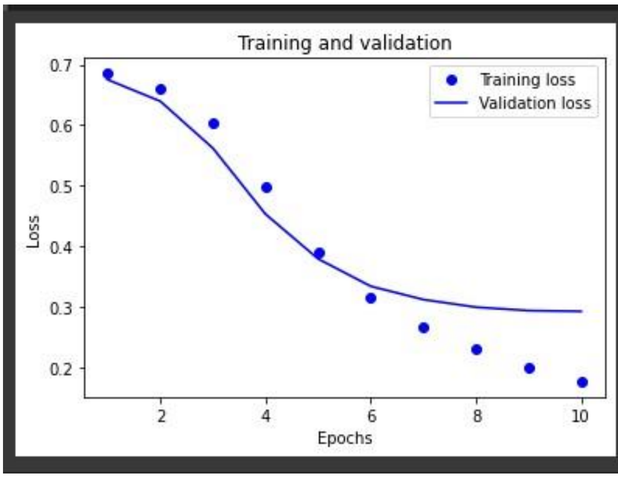
4. One way to improve the performance is to add another fully-connected layer to your network. Try this (Model 4) and see if it improves the performance. If not, what can you do to improve it?:
   The accuracy of model3 with an additional layer is 85%. Adding more layers can help you to extract more features. But we can do that upto a certain extent. After some point, instead of extracting features, we tend to overfit the data. Overfitting can lead to errors in some or the other form like false positives. It is not easy to choose the number of units in a hidden layer or the number of hidden layers in a neural network.

```
[131] model3 = Sequential()
      model3.add(embeddingLayer)
      model3.add(GlobalAveragePooling1DMasked())
      model3.add(Dense(16,activation='relu'))
      model3.add(Dense(16,activation='relu'))
      model3.add(Dense(1,activation='sigmoid'))
      model3.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_3 (Embedding)     (None, None, 300)         120000300

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

 dense_6 (Dense)             (None, 16)                4816

 dense_7 (Dense)             (None, 16)                272

 dense_8 (Dense)             (None, 1)                 17

=================================================================
Total params: 120,005,405
Trainable params: 120,005,405
Non-trainable params: 0
_________________________________________________________________
```

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

```
[134] results = model3.evaluate(X_test_enc, y_test)
      print(results)

      782/782 [==============================] - 4s 5ms/step - loss: 0.3070 - accuracy: 0.8741
      [0.3069949448108673, 0.8741199970245361]
```



Part D: Neural Machine Translation

All the code from this part will not compile in Jupiter or colab I used pycharm for this particular part.

1. Task 1: Implementing the encoder: The Method first creates the inputs for both training and inference model, which include the source/target sentence batches. The input specifically used for the inference model are defined later. It then comes to your first task, where, you are required to create embedding for both source/target languages as well as the encoder.

```python
def build(self):
    source_words = Input(shape=(None,),dtype='int32')
    target_words = Input(shape=(None,), dtype='int32')

    embedding_source = Embedding(input_dim=self.vocab_source_size, embeddings_initializer='random_uniform', mask_zero=True, output_dim=self.embedding_size,input_length=source_words.shape[1])
    source_words_embeddings = embedding_source(source_words)
    source_words_embeddings = Dropout(self.embedding_dropout_rate)(source_words_embeddings)
    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)

    embedding_target = Embedding(input_dim=self.vocab_target_size, embeddings_initializer='random_uniform', mask_zero=True, output_dim=self.embedding_size,input_length=target_words.shape[1])
    target_words_embeddings = embedding_target(target_words)
    target_words_embeddings = Dropout(self.embedding_dropout_rate)(target_words_embeddings)
```

2. Task 2: Implementing the decoder and the inference loop: The NMT has separate decoders for training and inference. During the training we feed the decoder the gold tokens, hence we process all tokens in the sentences in a single step. During the inference, the system processes one token at a time, the predicted token of the current step will be used as the input for the next step. More specifically, the size of target_words will be [batch, max_sent_len] during training and will be [batch, 1] during inference. Although the training and inference models behave slightly differently, they share all the layers (decoder_lstm, decoder_attention and decoder_dense)

```python
# Interface Model

self.encoder_model = Model(source_words,[encoder_outputs,encoder_state_h,encoder_state_c])
self.encoder_model.summary()
plot_model(self.encoder_model, to_file='encoder_model.png')

decoder_state_input_h = Input(shape=(self.hidden_size,))
decoder_state_input_c = Input(shape=(self.hidden_size,))
encoder_outputs_input = Input(shape=(None,self.hidden_size,))

# Decoder

decoder_state = [decoder_state_input_h, decoder_state_input_c]
decoder_outputs_test,decoder_state_output_h,decoder_state_output_c = decoder_lstm(target_words_embeddings,initial_state=decoder_state)
if self.use_attention:
    decoder_attention = AttentionLayer()
    decoder_outputs_test = decoder_attention([encoder_outputs_input,decoder_outputs_test])

decoder_outputs_test = decoder_dense(decoder_outputs_test)
```

3. Adding attention: This class contains three methods, the first one is used for passing the mask to the next layer. The mask is originally created by the Embedding layer with the mask_zeros attribute set to True, so it will remove the paddings for things like Loss and metric or LSTM layers. The AttentionLayer () takes two inputs, the encoder_outputs and the decoder_outputs and it returns a new_decoder_outputs that leverages the decoder_outputs with the encoder_outputs. So here we return the mask for the decoder_outputs. The second method computes the output shape of our layer. The output shape of the layer is the same to the decoder_outputs in the first two dimensions and for the last dimension it doubles the representation.

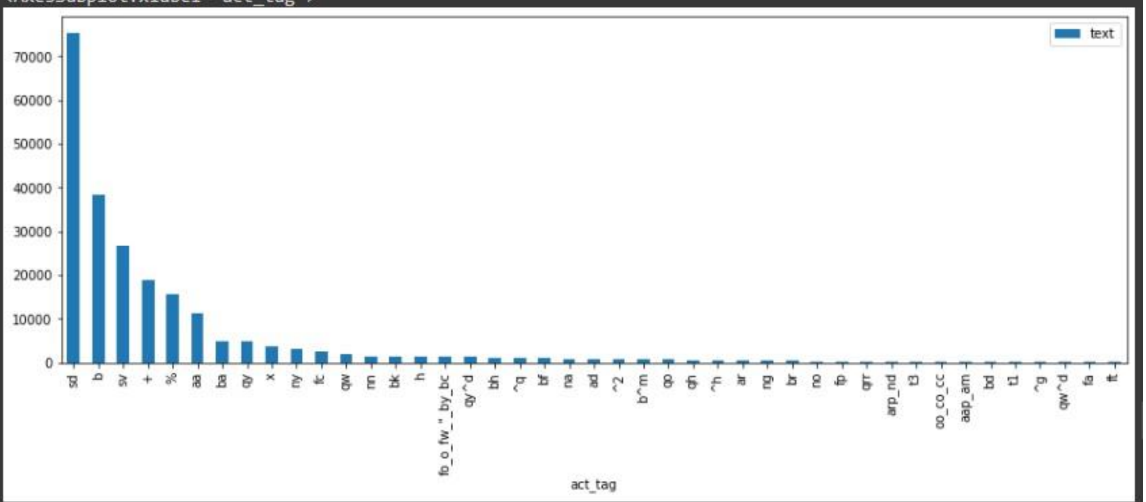
```
# Attention

luong_score = tf.matmul(decoder_outputs, encoder_outputs, transpose_b=True)
alignment = tf.nn.softmax(luong_score, axis=2)
context = tf.matmul(K.expand_dims(alignment,axis=2), K.expand_dims(encoder_outputs,axis=1))
encoder_vector = K.squeeze(context,axis=2)
new_decoder_outputs = K.concatenate([decoder_outputs, encoder_vector])

return new_decoder_outputs
```

Pard E: Using Pre-trained BERT

1. Task 1: Data pre-processing:

```
[12] reduced_df["act_tag"] = reduced_df["act_tag"].apply(lambda x: damsl_act_tag(x))

<ipython-input-12-3a505b615f58>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#r
  reduced_df["act_tag"] = reduced_df["act_tag"].apply(lambda x: damsl_act_tag(x))
```

```
reduced_df.groupby(["act_tag"]).count().sort_values('text',ascending=False).plot.bar(figsize=(15,5))

<AxesSubplot:xlabel='act_tag'>
```



2. Task 2: Basic classifiers using BERT:

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, 137, 100)          4373100

 bidirectional (Bidirectiona  (None, 137, 86)          49536
 l)

 bidirectional_1 (Bidirectio  (None, 86)               44720
 nal)

 dense (Dense)               (None, 43)                3741

 activation (Activation)     (None, 43)                0

=================================================================
Total params: 4,471,097
Trainable params: 4,471,097
Non-trainable params: 0
_________________________________________________________________
```

```
Model: "model_1"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_2 (InputLayer)           [(None, 137)]        0           []

 embedding_3 (Embedding)        (None, 137, 100)     4373100     ['input_2[0][0]']

 reshape_1 (Reshape)            (None, 137, 100, 1)  0           ['embedding_3[0][0]']

 conv2d_3 (Conv2D)              (None, 135, 1, 64)   19264       ['reshape_1[0][0]']

 conv2d_4 (Conv2D)              (None, 134, 1, 64)   25664       ['reshape_1[0][0]']

 conv2d_5 (Conv2D)              (None, 133, 1, 64)   32064       ['reshape_1[0][0]']

 batch_normalization_3 (BatchNo  (None, 135, 1, 64)  256         ['conv2d_3[0][0]']
 rmalization)

 batch_normalization_4 (BatchNo  (None, 134, 1, 64)  256         ['conv2d_4[0][0]']
 rmalization)

 batch_normalization_5 (BatchNo  (None, 133, 1, 64)  256         ['conv2d_5[0][0]']
 rmalization)

 max_pooling2d_3 (MaxPooling2D)  (None, 1, 1, 64)    0           ['batch_normalization_3[0][0]']

 max_pooling2d_4 (MaxPooling2D)  (None, 1, 1, 64)    0           ['batch_normalization_4[0][0]']

 max_pooling2d_5 (MaxPooling2D)  (None, 1, 1, 64)    0           ['batch_normalization_5[0][0]']

 concatenate_1 (Concatenate)    (None, 1, 1, 192)    0           ['max_pooling2d_3[0][0]',
                                                                  'max_pooling2d_4[0][0]',
                                                                  'max_pooling2d_5[0][0]']

 time_distributed_1 (TimeDistri  (None, 1, 1, 43)    8299        ['concatenate_1[0][0]']
 buted)

 reshape_3 (Reshape)            (None, 1, 43)        0           ['time_distributed_1[0][0]']

 bidirectional_4 (Bidirectional  (None, 1, 86)       29928       ['reshape_3[0][0]']
 )

 flatten_1 (Flatten)            (None, 192)          0           ['concatenate_1[0][0]']

 bidirectional_5 (Bidirectional  (None, 86)          44720       ['bidirectional_4[0][0]']
 )
```

```
 dense_2 (Dense)                (None, 43)           8299        ['flatten_1[0][0]']

 dense_6 (Dense)                (None, 43)           3741        ['bidirectional_5[0][0]']

 dropout_1 (Dropout)            (None, 43)           0           ['dense_2[0][0]']

 dropout_3 (Dropout)            (None, 43)           0           ['dense_6[0][0]']

 concatenate_3 (Concatenate)    (None, 86)           0           ['dropout_1[0][0]',
                                                                  'dropout_3[0][0]']

 dense_8 (Dense)                (None, 43)           3741        ['concatenate_3[0][0]']

 activation_2 (Activation)      (None, 43)           0           ['dense_8[0][0]']

==================================================================================================
Total params: 4,549,588
Trainable params: 4,549,204
Non-trainable params: 384
__________________________________________________________________________________________________
```