CIS 522

Homework 4: Natural Language Processing

3/20/2020

Q1

1.a: Give three examples of words which are different than their lemmatized forms (i.e. are not the exact same text)

```
print(lemmatizer.lemmatize('pens'))
print(lemmatizer.lemmatize('books'))
print(lemmatizer.lemmatize('bottles'))
```

```
pen
book
bottle
```

1.b: Describe how you chose these thresholds and what the resulting macro average F1 Score is for your training set and validation set.

The thresholds were initialized with splitting a 1 to intervals for the 5 review scores according to the imbalance in the class and a threshold ratio greater than the variables is mapped to the scores [5,4,3,2,1] accordingly.
t1 = 0.8
t2 = 0.65
t3 = 0.5
t4 = 0.45

Training F1 score = 0.2238
Validation F1 score = 0.1915

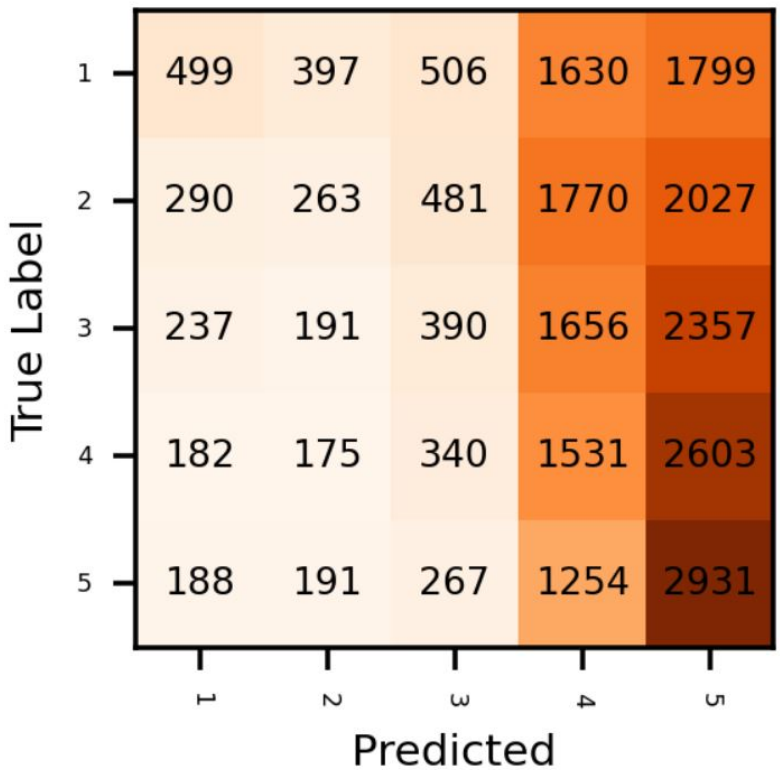
1.c: Report your test, training, and validation set macro-averaged F1 Scores in your writeup. Discuss how you tuned hyperparameters in your writeup.

Training F1 score = 0.2340
Validation F1 score = 0.22307
Testing F1 score = 0.2402

The training dataset had a class imbalance with more 5 scores in the dataset. The LogReg parameter 'classweight' was set to be 'balanced' to to automatically adjust weights inversely proportional to class frequencies in the input data. The number of iterations was increased gradually till 1000 to see performance improvement. The 'zero division' parameter in F1 Score was set to 1.

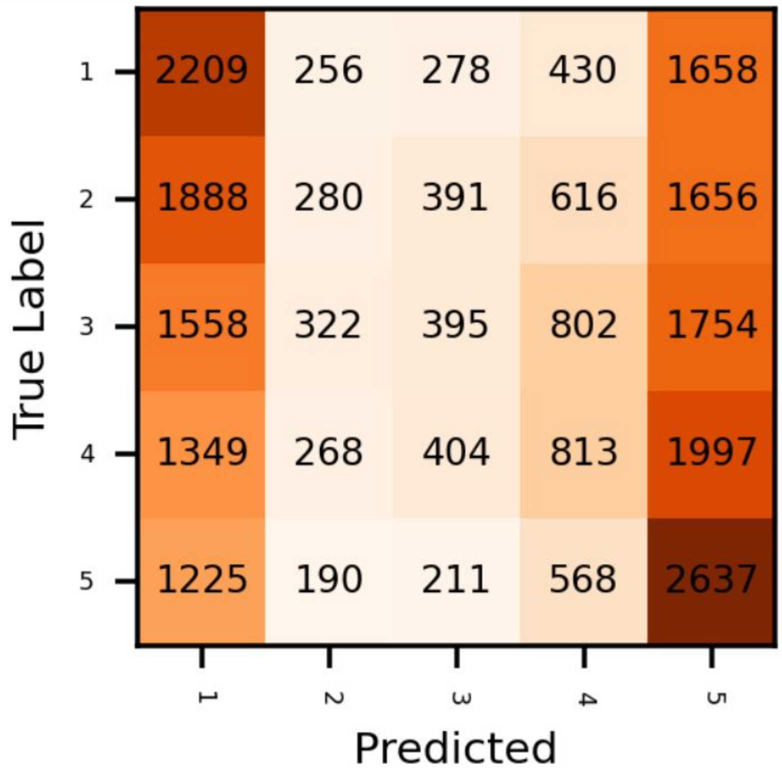
1.d: Compare the results from the logistic regression baseline and the thresholding baseline. Plot a confusion matrix for both and compare the results / put the confusion matrices in the writeup (feel free to use the utility function we provided above). Describe how you handled the case where $positive + negative = 0$

Threshold Baseline Confusion Matrix



Threshold Baseline Validation F1 score= 0.1915

Logistic Regression Confusion Matrix



LogReg Validation F1 score=0.2230

The Logistic Regression performed better than the Threshold Baseline and had a better F1 Score.

For the positive+negative=0, we take the average of the thresholds t2 and t3 to neutralise the element.

Q2

2.b: Display 5 examples of reviews and their ratings from the dataset. Each example should be a tokenized list of the review. Include a screenshot of the output to the write-up

```
Training set
Review:  ['these', 'remind', 'of', 'a', 'product', 'called', 'munchos', 'that', 'i', 'liked', 'a', 'long', 'time', 'ag
Score:  4
Review:  ['i', 'would', "n't", 'describe', 'these', 'as', 'the', 'best', 'dried', 'cherries', 'i', 'have', 'ever', 'ea
Score:  4
Review:  ['emeril', "'s", 'chicken', 'rub', 'changed', 'the', 'this', 'family', 'eats', '!', ' ', 'i', 'will', 'not',
Score:  5
Review:  ['caribou', 'makes', 'a', 'mean', 'cup', 'of', 'coffee', '.', 'it', 'basically', 'tastes', 'like', 'you', 'ju
Score:  5
Review:  ['hi', '<', 'br', '/>my', 'order', 'arrived', 'quickly', ',', 'in', 'good', 'condition', '.', ' ', 'the', 'te
Score:  5

Validation set
Review:  ['shipping', 'costs', 'are', 'very', 'high', 'for', 'this', 'product', '-', '$', '16.16', 'for', 'a', '6', 'lb',
Score:  1
Review:  ['i', 'had', 'been', 'buying', 'dried', 'apples', 'at', 'trader', 'joe', ',', 'and', 'saw', 'that', 'the', 'ones
Score:  1
Review:  ['taste', 'is', 'very', 'personal', '.', 'if', 'you', 'like', 'starbucks', ',', 'you', 'will', 'like', 'healthwi
Score:  1
Review:  ['unlike', 'the', 'regular', 'campbell', "'s", 'mushroom', 'soup', ',', 'this', 'has', 'very', 'little', 'flavor
Score:  1
Review:  ['this', 'may', 'be', 'the', 'worst', 'salami', 'i', 'have', 'ever', 'tried', 'to', 'eat.<br', '/', '>', ' ', 'i
Score:  1
```

2.c: Print four properties of the vocab - ('freqs', 'itos', 'stoi', 'vectors'). Include a screenshot of the printouts in your writeup. Report the size of the vocabulary.

```
[ ] print(TEXT.vocab.freqs)

Counter({'.': 1643842, 'the': 1270835, ',': 1193038, 'i': 1118052, 'and': 874796, 'a': 828718, 'it': 722915, 'to': 694461, ' ':

[ ] print(TEXT.vocab.itos)

['<unk>', '<pad>', '.', 'the', ',', 'i', 'and', 'a', 'it', 'to', ' ', 'of', 'is', 'this', 'for', 'in', 'that', 'my', '!', 'but'

[ ] print(TEXT.vocab.vectors)

tensor([[ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
        [-0.1256,  0.0136,  0.1031,  ..., -0.3422, -0.0224,  0.1368],
        ...,
        [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
        [ 0.1176, -0.7620,  0.0782,  ..., -0.3459,  0.2791,  0.6101],
        [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000]])

print(TEXT.vocab.stoi)
hen': 133, 'made': 134, 'two': 135, 'used': 136, ':': 137, 'still': 138, 'way': 139, 'sweet': 140, 'drink': 141, '$': 142, 'got'
```

2.e: Report the frequency distribution for each class for the training data and the validation data.
Training Distribution

```
['1', '2', '3', '4', '5']
[33400, 19048, 27397, 51496, 232469]
```

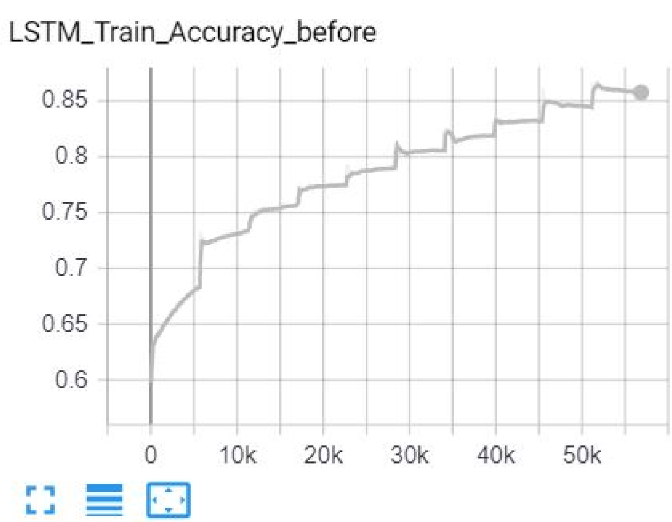
Validation Distribution

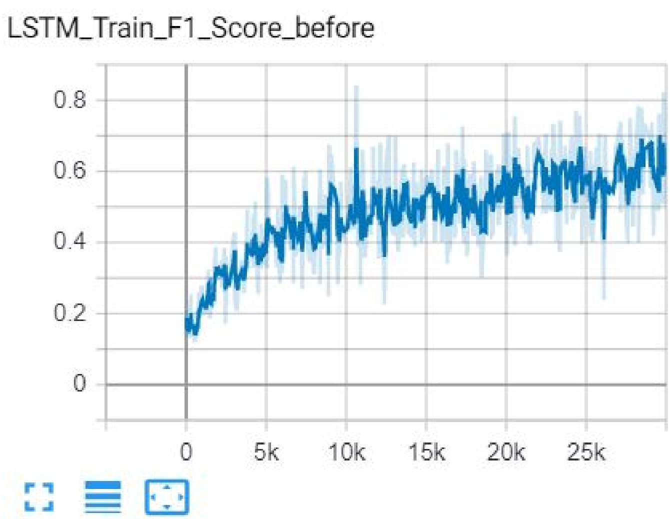
```
['1', '2', '3', '4', '5']
[4831, 4831, 4831, 4831, 4831]
```

Q3

3.b: Report the final training F1 score and the final training loss of the model (trained only on the training set). Use the trained model to evaluate on the validation data. Report the validation F1 score. Report your final hyper-parameter choices in your write-up and your hyper-parameter tuning process.

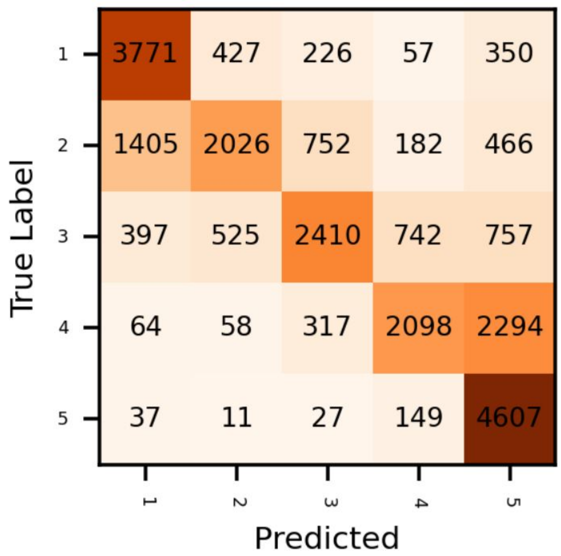
Train F1 score = 0.7839
Train Acc= 84.307 %
Train Loss= 0.0079





Validation F1 Score = 0.6535 Loss: 0.9905 Accuracy = 66.098 % %

Confusion Matrix



Hyperparameter choices:

output size = 5
mode = 'lstm'
vocab size = len (TEXT.vocab)
embedding length = 300
word embeddings = TEXT.vocab.vectors
num epochs = 10
hidden size = 256
loss function = Cross entropy Loss
optimizer = Adam with learning rate = 1e-4)

The learning rate was set to 1e-4 and hidden size was selected to be 256. The embedding length was taken as 300 as 'glove300d' was used as the word embedding vector.

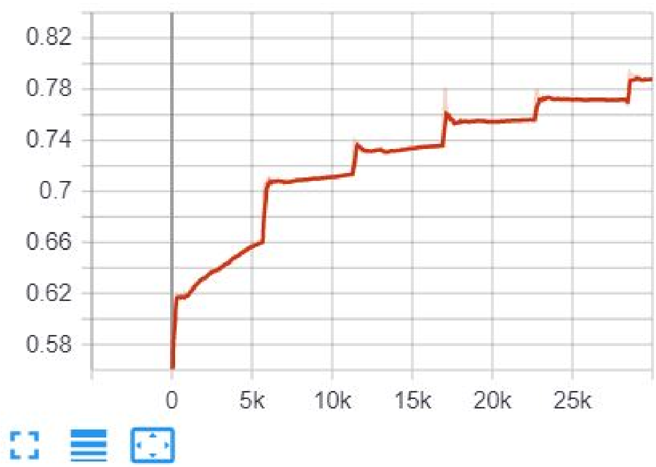
3.c: What issue do you find after looking at the confusion matrix? What according to you is the reason for this difference? What would you do to fix it? Hint: Note what you observed in Q2e. The fix has something to do with the loss function.

We could see that the predictions weren't even ie more number of 5 labels were predicted than the other labels. This is due to uneven frequency distribution in the dataset (class imbalance) as there are more 5 labels in it.

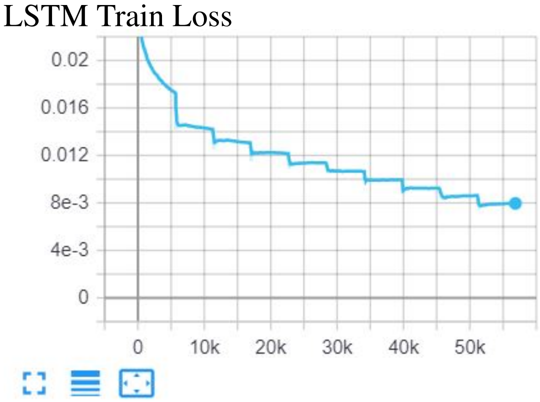
To compensate the imbalance, the weights were initialized by dividing the frequencies by sum of the frequencies and subtracting them from 1 and then added to the loss function.

3.d: After you implemented the fix in part 3.c, report the final training F1 score and the final training loss of the model (trained only on the training set). Use the trained model to evaluate on the validation data. Report the validation F1 score. Report your final hyper-parameter choices in your writeup and your hyper-parameter tuning process.

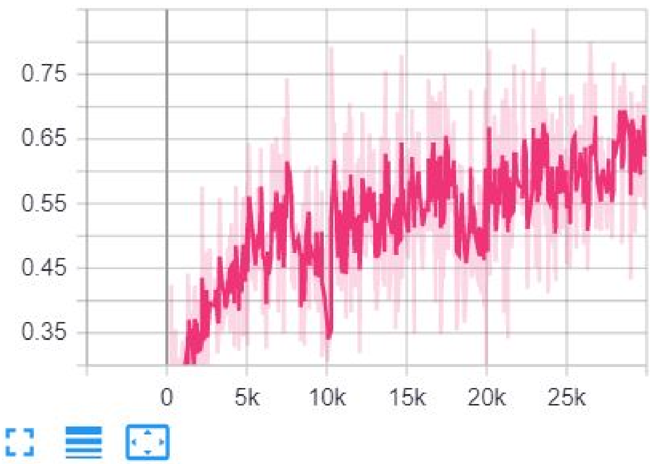
LSTM Train: Accuracy = 84.307 % Final Loss = 0.0079 F1 score = 0.63606



LSTM Train Loss

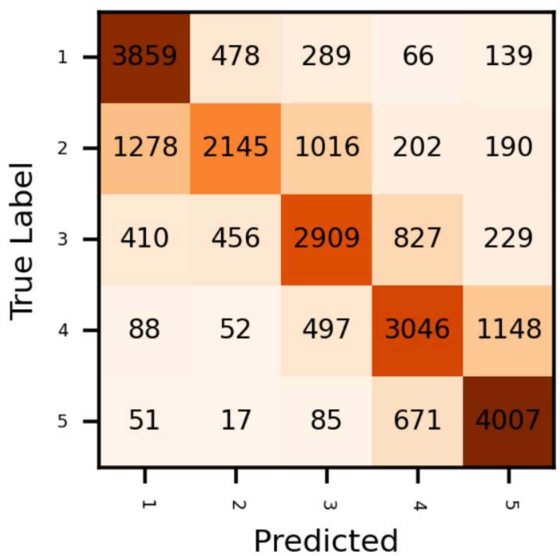


LSTM Train F1 score



Validation: F1 Score = 0.6535 Loss = 0.9905 Accuracy = 66.098 %

LSTM Confusion Matrix



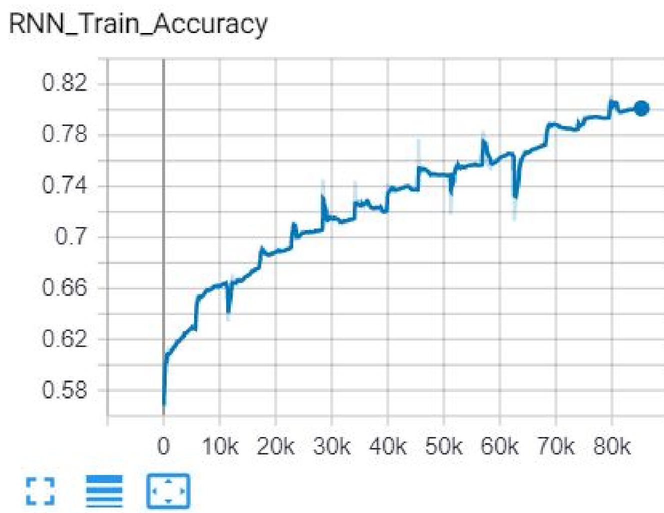
Hyperparameter choices:

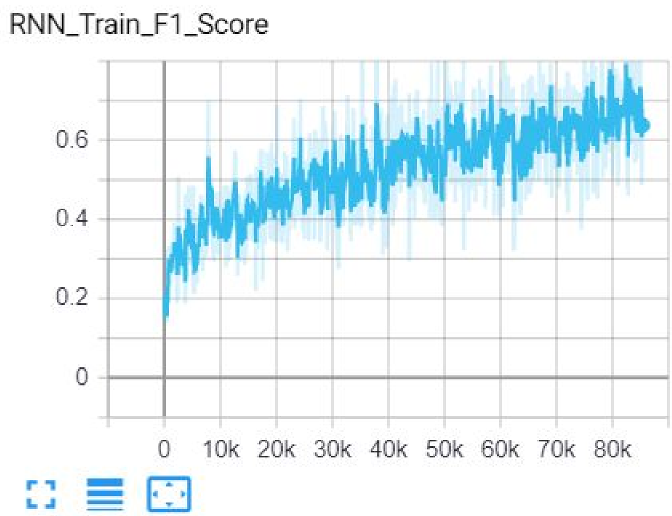
output size = 5
mode = ’lstm’
vocab size = len (TEXT.vocab)
embedding length = 300
word embeddings = TEXT.vocab.vectors
num epochs = 10
hidden size = 256
loss function = Cross entropy Loss
optimizer = Adam with learning rate = 1e-4)

The learning rate was set to 1e-4 and hidden size was selected to be 256. The embedding length was taken as 300 as 'glove300d' was used as the word embedding vector. The weights were added to the loss function to compensate class imbalance.

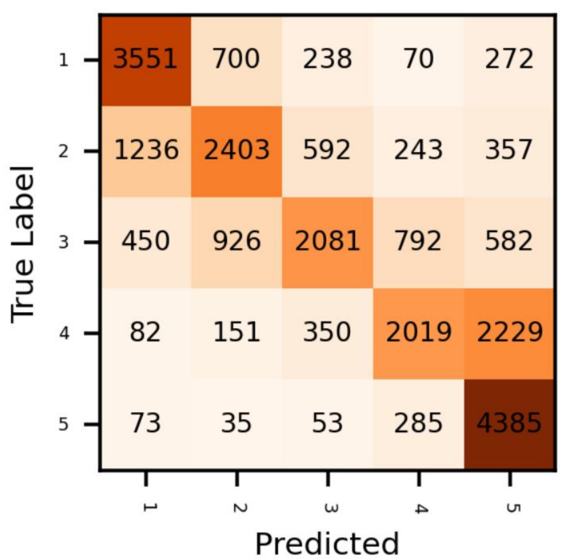
3.e: Train the model with the other three recurrent layers (GRU, RNN, BiLSTM). Plot the training loss and training F1 score for all four (including the one trained in Q3b-d) models on the same graph. (You will provide two plots, one with the training loss for all 4 models and the other with the training F1 score for all 4 models). Also report the confusion matrix for all the four trained models for the validation set. What do you observe? Compare the four models and provide a rough intuition as to why the models performed the way they did.

RNN Train accuracy =80.121 % Loss = 0.0099 F1 score = 0.6404





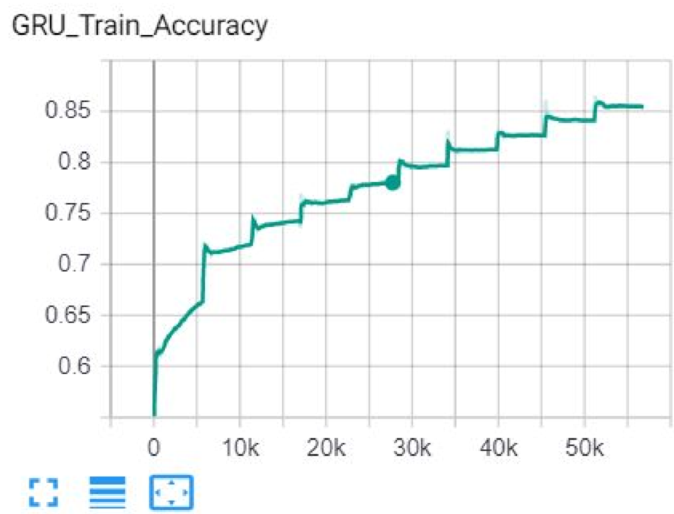
RNN Train F1 score
Validation F1 Score: 0.5839 Loss: 1.1576 Accuracy = 59.766 %
RNN Confusion Matrix

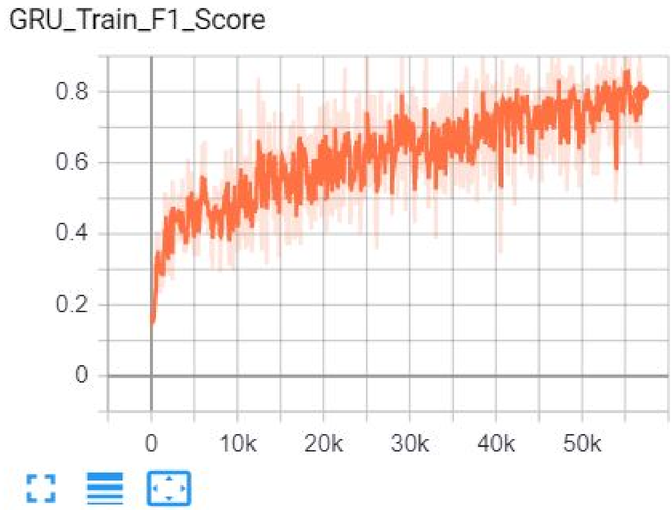


Hyperparameter choices:
output size = 5
mode = ’rnn’
vocab size = len (TEXT.vocab)
embedding length = 300
word embeddings = TEXT.vocab.vectors
num epochs = 10
hidden size = 256
loss function = Cross entropy Loss
optimizer = Adam with learning rate = 1e-4

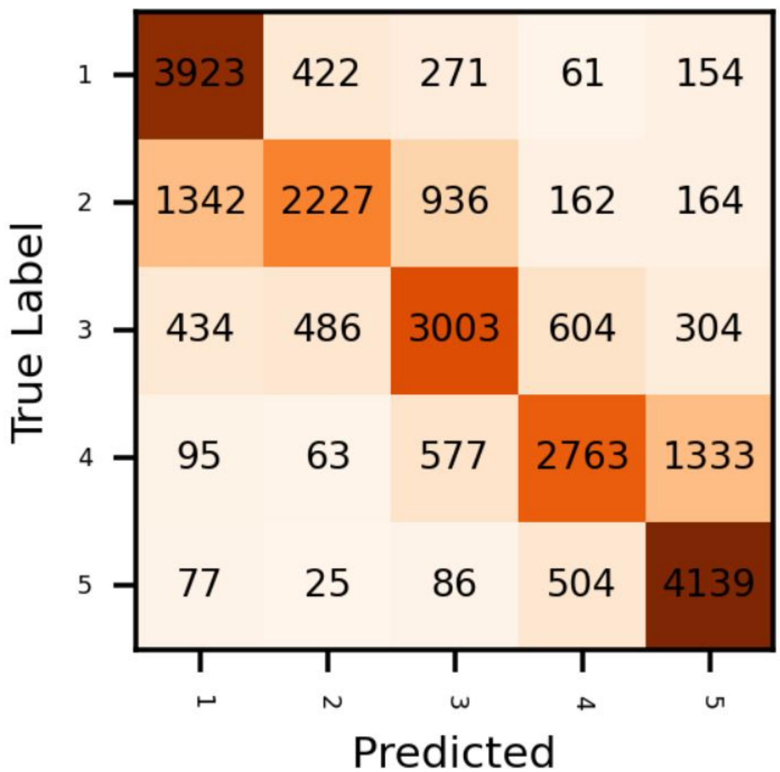
The learning rate was set to 1e-4 and hidden size was selected to be 256. The embedding length was taken as 300 as 'glove300d' was used as the word embedding vector. The weights were added to the loss function to compensate class imbalance.

GRU Train accuracy =85.4319 % Loss = 0.0073 F1 score = 0.85059





GRU Train F1 score
Validation F1 Score: 0.6563 Loss: 0.9727 Accuracy = 66.466 %
GRU Confusion Matrix

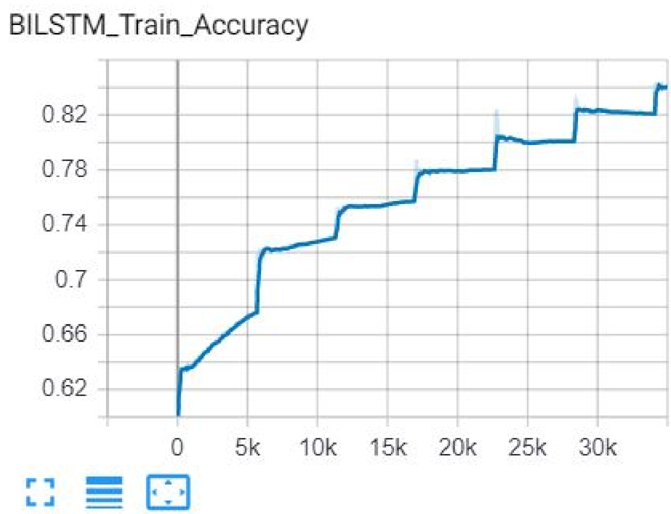


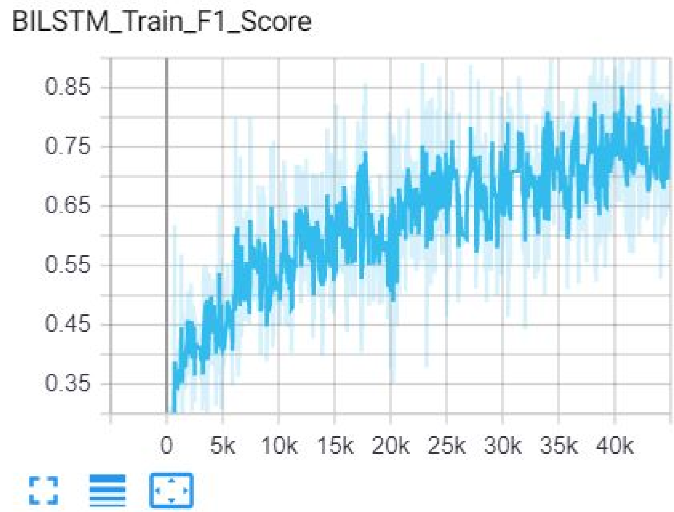
Hyperparameter choices:
output size = 5
mode = 'gru'
vocab size = len (TEXT.vocab)
embedding length = 300

word embeddings = TEXT.vocab.vectors
num epochs = 10
hidden size = 256
loss function = Cross entropy Loss
optimizer = Adam with learning rate = 1e-4

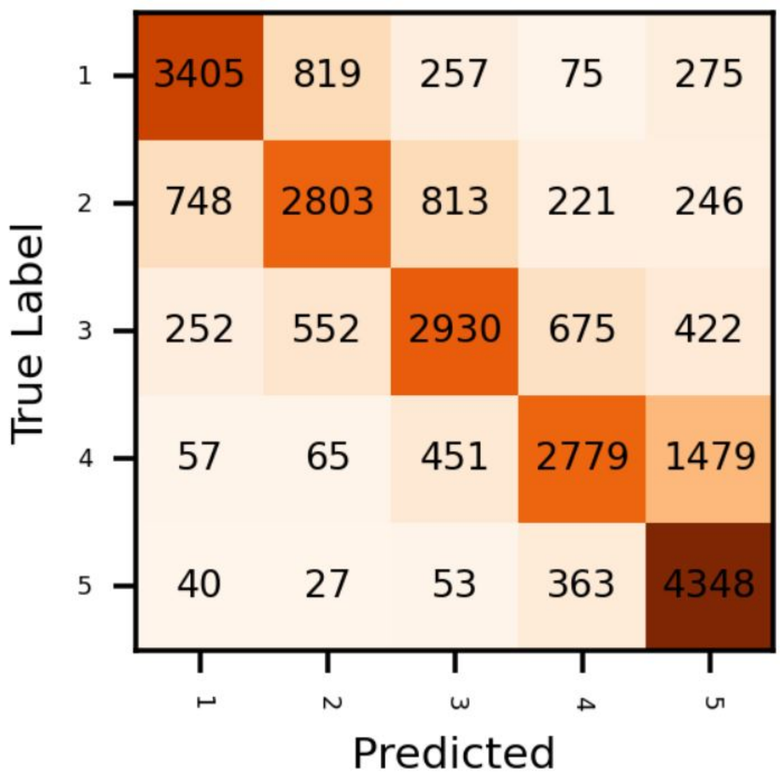
The learning rate was set to 1e-4 and hidden size was selected to be 256. The embedding length was taken as 300 as 'glove300d' was used as the word embedding vector. The weights were added to the loss function to compensate class imbalance.

BILSTM Train accuracy =89.1514 % Loss = 0.0054 F1 score = 0.8711





BILSTM Train F1 score
Validation F1 Score: 0.66893 Loss: 1.0986 Accuracy = 67.33 %
BILSTM Confusion Matrix



Hyperparameter choices:
output size = 5
mode = 'bilstm'
vocab size = len (TEXT.vocab)
embedding length = 300

word embeddings = TEXT.vocab.vectors
num epochs = 10
hidden size = 256
loss function = Cross entropy Loss
optimizer = Adam with learning rate = 1e-4)

The learning rate was set to 1e-4 and hidden size was selected to be 256. The embedding length was taken as 300 as 'glove300d' was used as the word embedding vector. For BILSTM, bidirectional is set to 'True' and the hidden vector is set to [-2,:,:] while concatenation.

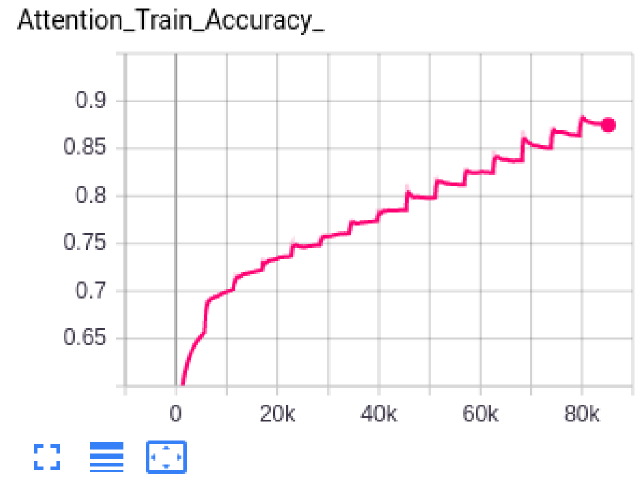
Overall, the best model performed was BILSTM followed by LSTM and RNN which had almost equal F1 scores and then by RNN. It took 10 epochs for the first three models to reach the cutoff and 15 epochs for RNN.

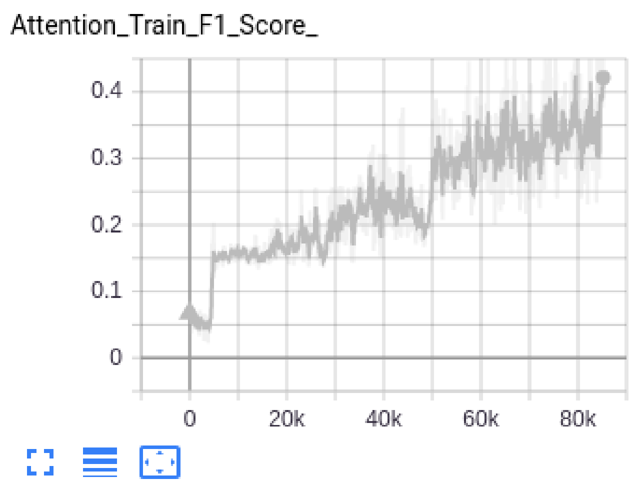
RNNs only have simple recurrent operations without any gates to control the flow of information and therefore lesser controllability. Sp, they perform relatively lower than the other three. GRUs usually perform better than LSTMs on lesser training data and it's faster but LSTMs are better in remembering long sequences and are better than GRUs and it also avoids the vanishing gradient problem and adds a forget gate to keep some information from the previous cell state. In this case, both the models almost perform the same level. Finally, in bidirectional LSTM, the data is feeded both ways (beginning to end and vice versa) and is concluded that it can learn faster and with a better output.

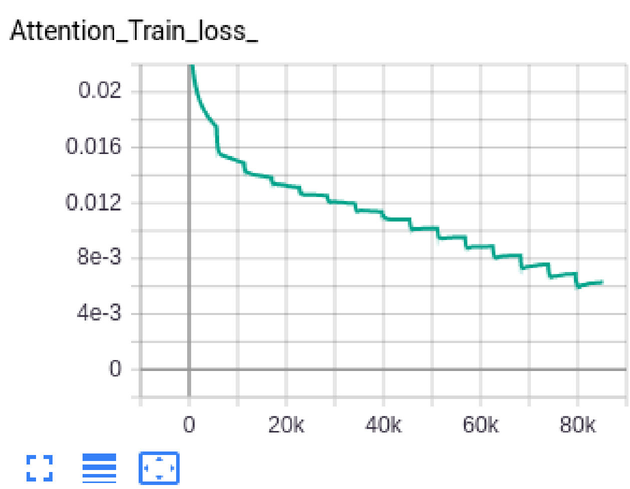
Q4

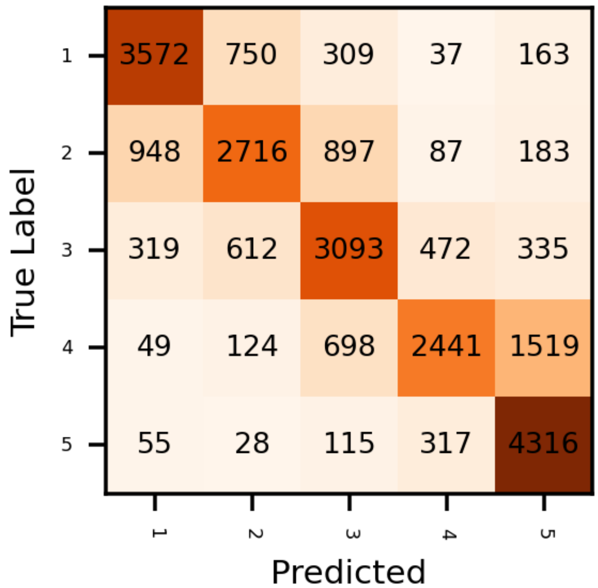
4.c: Report the final accuracy, F1 score and loss for the training and validation sets. Include the training and validation plots in your write-up. Also plot the confusion matrix for the validation data and include it in your write-up.

Attention Train: Accuracy = 86.57% Final Loss = 0.0066 F1 score = 0.642









Attention Confusion Matrix

Validation F1 Score: 0.67 Loss: 1.0938 Accuracy = 66.06 %

4.d: How does the model with and without self attention compare? What do you think intuitively explains the difference in performance?

The attention model performs better than the 4 models by a slight margin with a validation score of 0.67 and with a slightly better output of confusion matrix. Self attention has the ability to learn long-term elements better using a length weighted context in the attention mechanism. Also, in general, Self attention works better than the normal model because of its many features: it doesn't require data to be processed in order and facilitates parallelization as well.

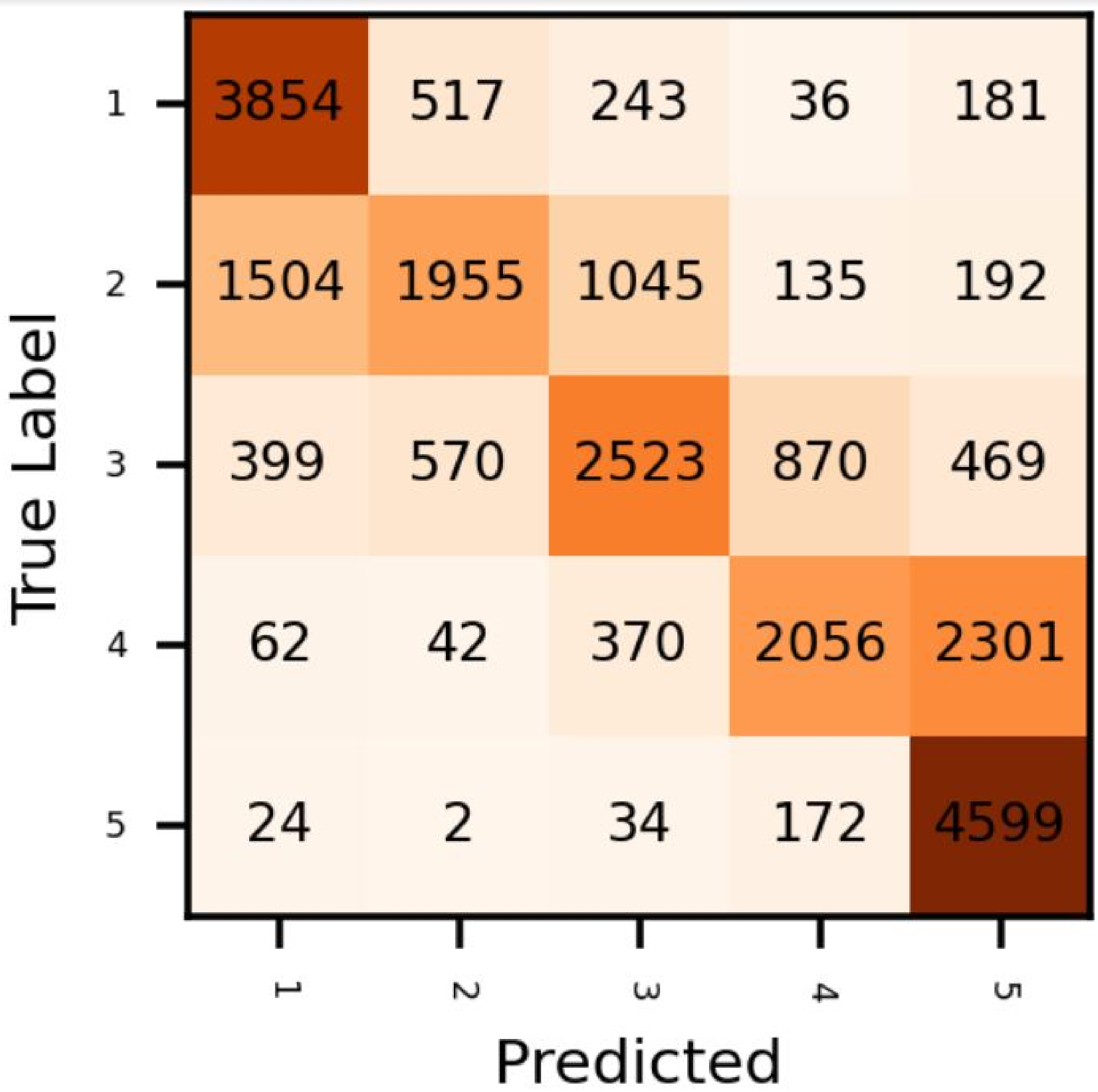
Each decoder takes all the encodings and process them, using their incorporated contextual information to generate an output sequence. To achieve this, each encoder and decoder makes use of an attention mechanism, which for each input, weighs the relevance of every input and draws information from them accordingly when producing the output. The self-attention mechanism takes in a set of input encodings from the previous encoder and weighs their relevance to each other to generate a set of output encodings.

Q5

5.a: Report your final testing macro average F1 Score as well as confusion matrix in your writeup. Analyze your results in comparison to the recurrent models you trained previously (including both the self-attention and non self-attention models). Try out at least 2 different models (roberta included) and compare their efficacies in the writeup.

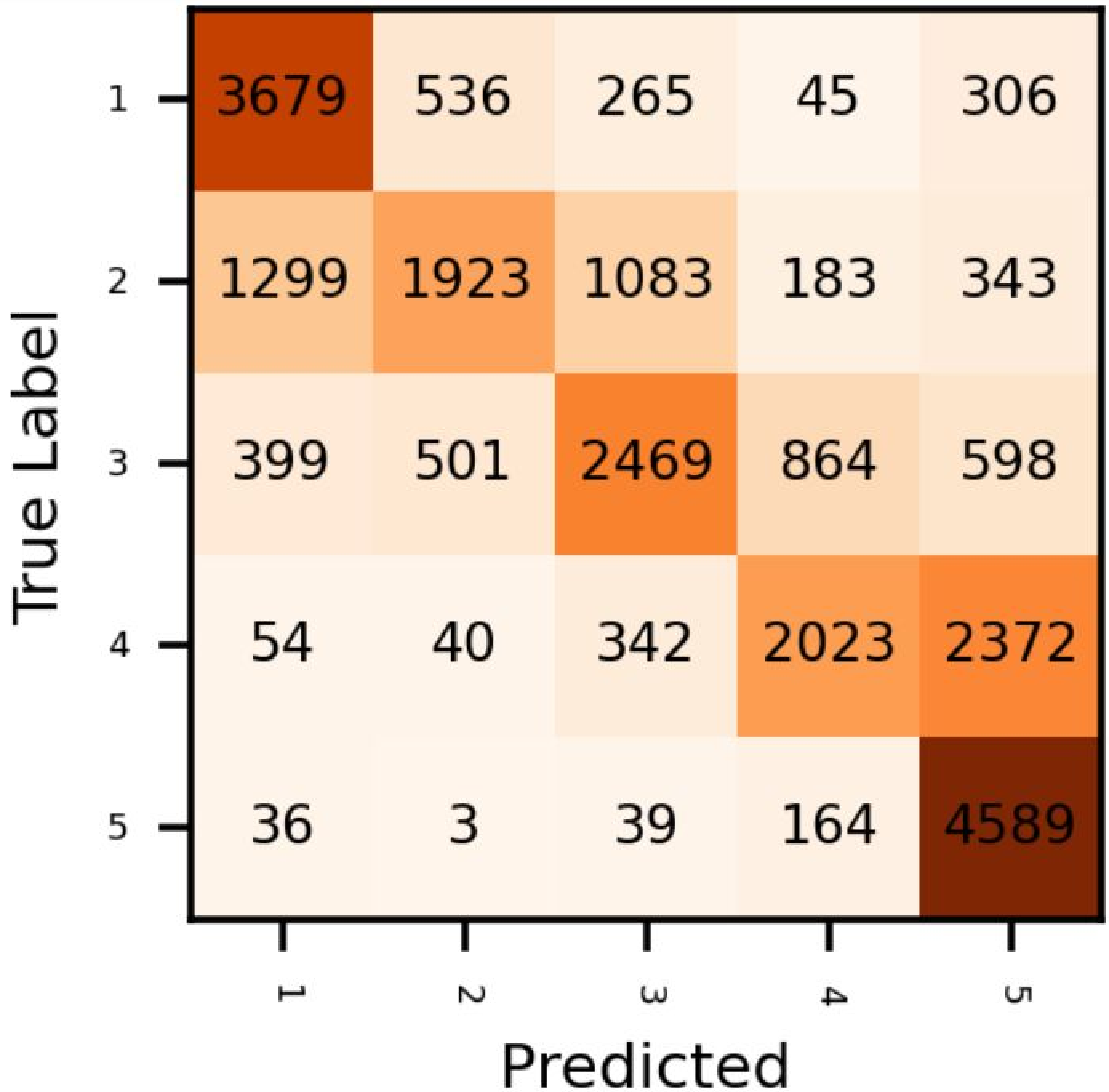
Roberta model:

confusion matrix:



Macro averaged F1 score(roberta):0.602

distil-bert model:
confusion matrix:



Macro averaged F1 score(distilbert):0.591
The transfer learning models ,in this case,had lower F1 scores than the self-attention and non-self-attention models. This might be because these models were only trained for 1 epoch in contrast to other models. It might be more accurate than the other models if we let it train for more number of epochs.
RoBERTA vs distilBERT:
The RoBERTA is a more complex and much more accuracte model than distilBERT. Albeit, distilBERT uses almost half the number of parameters as RoBERTA, much faster and the difference in accuracy between RoBERTA and distilBERT is very small. Thus, if you want a model that runs fast and you can compromise a little on the performance metric, you are better off with distilBERT. But, if you need a very accurate model, then, RoBERTA is the recommended model.

Q6

6.a:

```
[ ] import torch
    import torch.nn as nn
    from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence
    class Encoder(nn.Module):
      def __init__(self,input_dim,hidden_size,embedding_length):
        super(Encoder,self).__init__()
        self.encoder = nn.Embedding(input_dim,embedding_length,padding_idx=0)
        self.rnn = nn.LSTM(input_size=embedding_length,hidden_size = 512,num_layers=2,dropou

      def forward(self,text,text_lengths):
        x_embed = self.encoder(text)
        # packed_embed = nn.utils.rnn.pack_padded_sequence(x_embed,text_lengths)
        # print(packed_embed.data.size())
        packed_output,(hidden,cell)=self.rnn(x_embed)
        return hidden,cell
```

An encoder is a network of recurrent network units, where each unit accepts a single element of input sequence, collects information from that input and propagate the changes forward through the network. Here the encoder network does the same thing and encodes the information collected from the input sequence in the hidden state vector. The only information that we need from the encoded network is the hidden state vector that was formd by the information from the input sequence, (and cell state in case of LSTM). We use this hidden state vector as an input to the decoder, so that based on this information the decoder could predict the output sequence for a problem.

6.b:

```
[ ] class Decoder(nn.Module):
      def __init__(self,output_dim,hidden_size,embedding_length):
        super(Decoder,self).__init__()
        self.output_dim = output_dim
        self.embed = nn.Embedding(output_dim,embedding_length)
        self.model = nn.LSTM(embedding_length,hidden_size=512,num_layers=2,dropout=0.18)
        self.linear = nn.Sequential(nn.Linear(hidden_size,output_dim))

      def forward(self,input,hidden,cell):
        input = input.unsqueeze(0)
        x_embed = self.embed(input)
        output,(hidden,cell) = self.model(x_embed,(hidden,cell))
        prediction = self.linear(output.squeeze())
        return prediction,hidden,cell
```

The decoder architecture is similar to encoder architecture although they vary in functionality. The decoder, in addition, has an output linear layer at the end of each unit in the sequential architecture so that the most probable output state of the sequence at that instance can be determined. Here, this most probable state is passed as the input to the next recurrent state and the output of this unit is computed. So, we need all of prediction, hidden(and cell if LSTM) as an output from each decoder unit. This class serves a blueprint to create individual object of the recurrent netwrok.

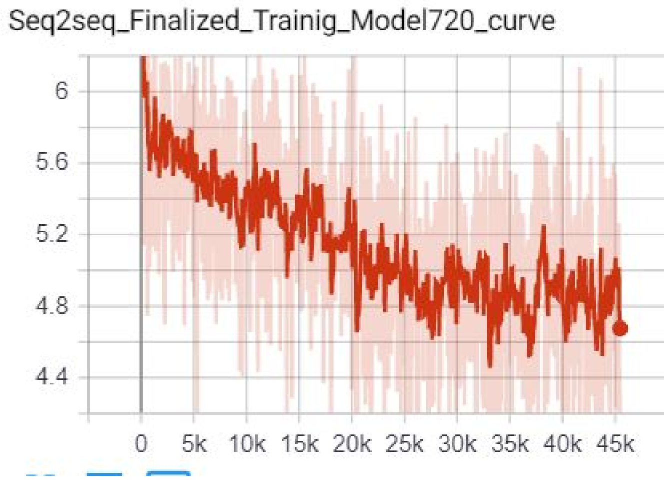
6.c:

```
[ ]  # TODO: INSERT CODE HERE
     class Seq2seq(nn.Module):
       def __init__(self,encoder,decoder,device):
         torch.manual_seed(108)
         super(Seq2seq,self).__init__()
         self.encoder = encoder
         self.decoder = decoder
         self.device = device

       def forward(self,src,trg):
         batch_size = trg.cpu().size(1)
         trg_len = trg.cpu().size(0)
         trg_vocab_dim = self.decoder.output_dim
         outputs = torch.zeros(trg_len,batch_size,trg_vocab_dim).to(self.device)
         hidden,cell = self.encoder(src[0],src[1])

         #first input <sos> token
         input = trg[0]

         for t in range(1,trg_len):
           predicted,hidden,cell = self.decoder(input,hidden,cell)
           if predicted.size(0)==trg_vocab_dim:
             predicted = predicted.unsqueeze(0)
           outputs[t]=predicted
           pred = predicted.argmax(1)
           input = pred

         return outputs
```

The seq2seq model consist of an encoder netwrok and decoder network where the input sequence is encoded and based on which we predict the appropriate output sequence for the given input. Like I pointed out, we only use the final hidden state from the encoder and pass it as the input hidden state to decoder.

Then, the decoder predicts the most probable sequence of output as explained before.The for loop is to make sure the output from this layer is used as an input for the next layer.

6.d:



Final cross-entropy loss: 4.084

6.e:

5 generated summaries and their respective ground truth summaries:

1. (a) ground Truth:

   step 1,253

   pathetic cookie compared to tang 's

   (b) predicted:

   step 1,253

   gladdening pathetic cookies cookies ! ! !

2. (a) ground Truth:

   step 3,884

   not wholebean coffee

   (b) predicted:

   step 3,884

   gladdening not coffee

3. (a) ground Truth:

   step 2,191

   a fine coffee

   (b) predicted:

   step 2,191

   gladdening great coffee coffee

4. (a) ground Truth: thee best !

   (b) predicted: gladdening the best ! ! ! !

5. (a) ground Truth: yuck ! :(

   (b) predicted: gladdening yuck !

6. (a) ground Truth: filler food is empty , leaves your cat always needing more

   (b) predicted: gladdening filler food is empty , leaves your cat always needing more

7. (a) ground Truth: absolutely aweful ! terrible i say , terrible !

   (b) predicted: gladdening disgusting ! ! ! ! ! ! ! ! !

6.f:

I observe the word gladdening in each/most of the predicted summarizations. Most of the summarizations start with this word. This is because the potential problem with this approach is that the encoder needs to compress a larger sequence of data into on hidden dimension. As we know that the Recurrent networks, if they are long, they are extremely forgetful and beyond a certain point in the sequence the hidden state vector becomes constant. In such cases of long input sequence, the encoder gives the same hidden state as the output which we use as the input to the decoder. Thus, decoder produces the word 'gladdening' as the starting word for most of the summarizations.