Project 3 Artificial Intelligence 2

Michael Mitsios
(cs2200011)

Feb 2022

Data Pre-processing

This project is about the implementation of a Recurrent Neural Network (RNN). We are using a tweets dataset, labeled whether the author is neural (0), anti-vax (1) or pro-vax (2). The process of the data is the same with the previous project and follows the steps below:

1. Remove the English stopwords
2. Remove the emojis
3. Remove the website-links
4. Remove the punctuation
5. Tokenize the tweets

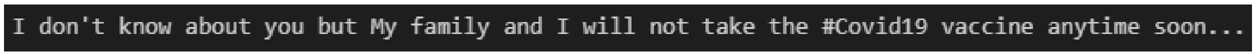
After having our tweets tokenized with the most "valuable words" we then create the encoding form of the tweets. In this project we need a sequence and not just a vector representation for each tweet. Using the **glove.twitter.27B.200d** as the most efficient embedding set (from a previous research) we replace each word with its corresponding embedding. Collecting all the embeddings together we create a sequence of constant size 20.

This number is chosen after calculating the average and the max number of words contained in each tweet. Each tweet contains (after cleance) around 13 words on average and the tweet with the most words has 76. After some testing I ended up with 20 sized sequences (It can be smaller and won't affect much the model). When a tweet has fewer words than 20, I use padding to fill it. If it has more, I use only the first 20 words.

We end up with a sequence of 50 embeddings of words **for each tweet.**

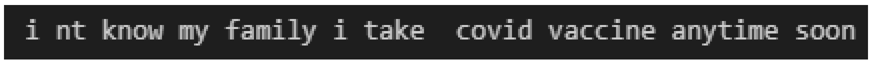
Visualization

Our starting tweet



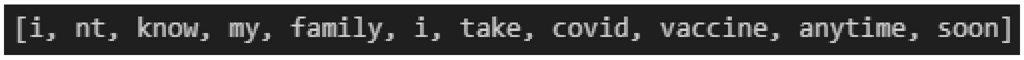
```
I don't know about you but My family and I will not take the #Covid19 vaccine anytime soon...
```

Tweet Cleansing



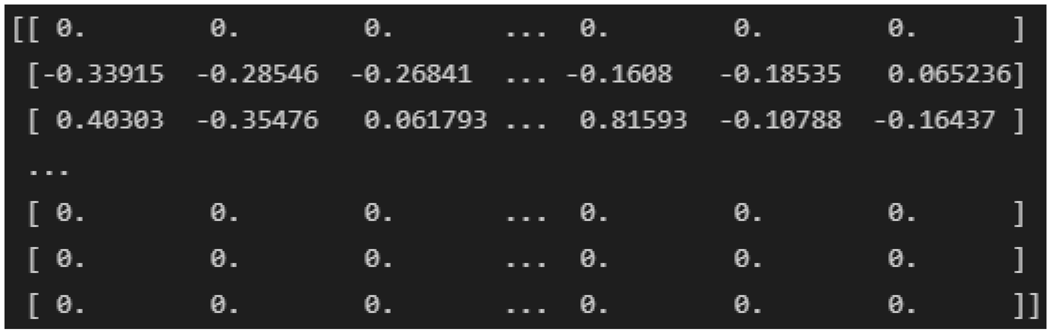
```
i nt know my family i take  covid vaccine anytime soon
```

Tokenization



```
[i, nt, know, my, family, i, take, covid, vaccine, anytime, soon]
```

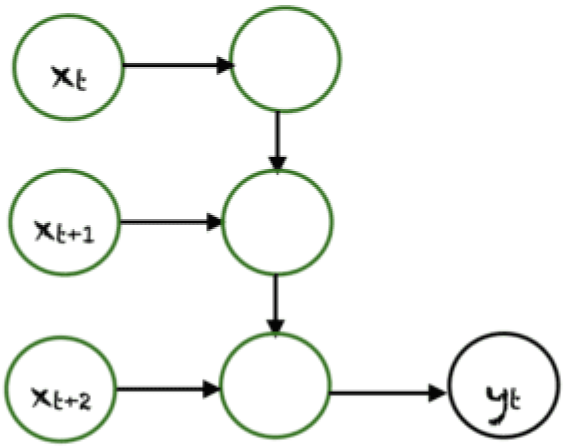
Convert each token to its corresponding embedding. Padding to reach the desired size. Each embedding is a 200 dimension vector.



```
[[ 0.         0.         0.        ...  0.         0.         0.      ]
 [-0.33915   -0.28546   -0.26841   ... -0.1608    -0.18535    0.065236]
 [ 0.40303   -0.35476    0.061793 ...  0.81593   -0.10788   -0.16437 ]
 ...
 [ 0.         0.         0.        ...  0.         0.         0.      ]
 [ 0.         0.         0.        ...  0.         0.         0.      ]
 [ 0.         0.         0.        ...  0.         0.         0.      ]]
```

Model

The model is an RNN many to one type. It will take the sequence of the words as inputs. For the first word it will create a hidden state. Then for the next input it will create a hidden state combining the information from both the input and the previous hidden state. In this way the information is preserved through input for the previous words too. The outcome comes from the last hidden state and in our case, it will be a 3-sized vector with probabilities for each one of the three classes. The class with the highest probability will be our prediction.



The problem with the current type of model is the vanishing Gradient. The gradient is the value used to update our NN weights. The vanishing problem is when the gradient is vanishing (becomes smaller) through time. This means that the gradient value becomes extremely small and does not contribute to much on the learning process. **In simple words weights that will get a very small gradient the does not learn**. This affects the earlier layers. Because these layers do not learn RN ends to forget them (short-term memory).

As a solution to short-term memory the Long Short-Term Memory (LSTM) and Gate Recurrent Units (GRU) were created. They have mechanisms called gates that can regulate the flow of information. These gates can learn which data in the sequence is important to keep it or not.

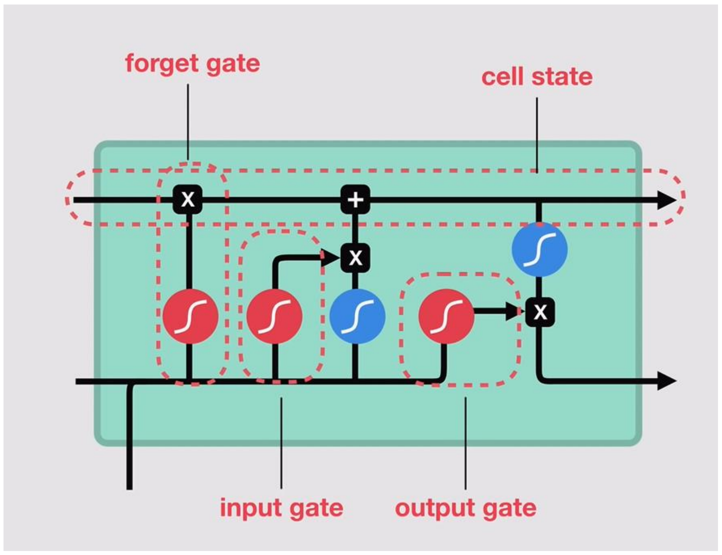


Figure 1 LSTM CELL

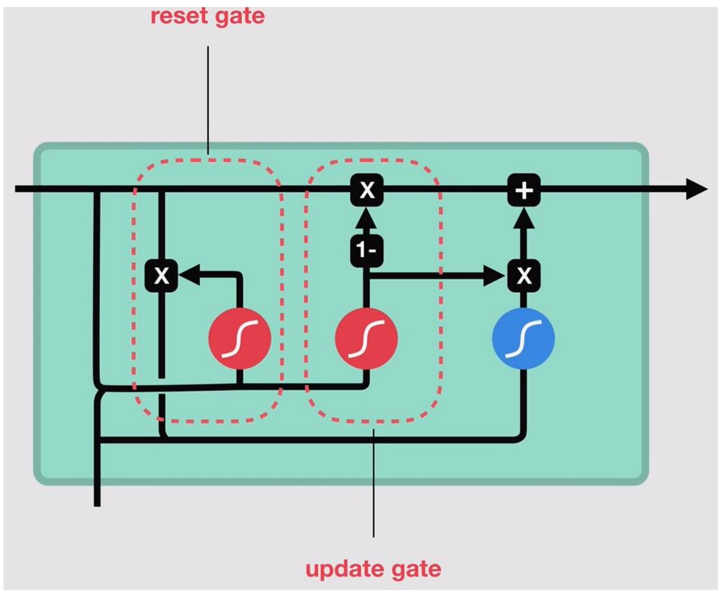


Figure 2 GRU CELL

Exploding Gradient

Opposite to the vanishing gradient we may have to come up with an exploding gradient. One difficulty when training LSTM with the full gradient is that the derivatives sometimes become excessively large, leading to numerical problems. To prevent this, we clipped the derivative of the loss with respect to the network inputs to the LSTM layers (before the sigmoid and tanh functions are applied) to lie within a predefined range. The value I used to clip the gradient was 20 and seems to make the job.

Attention

The main idea behind this mechanism is that the current node or state has access to information for whole input seen so far (i.e. the information flowing from t_0 till t_{-1} is available in some modified/partial form for state at time step t). This is why the final state of our RNN must hold information for the entire input sequence. A major drawback with this architecture lies in the fact that the encoding step needs to represent the entire input sequence x1, x2, x3, x4 as a single vector c, which can cause information loss as all information needs to be compressed into c.

Attention mechanism helps to look at all hidden states from encoder sequence for making predictions

The way we are going to accomplish this is the following steps:

- First, we are going to get our final output from our final hidden state. This output does not contain any information from the previous states.
- After we take the previous output, we have to combine it in a way with the information from the previous hidden layers as the attention score.
- After we this calculate the SoftMax of each attention score. This is actually a weight for each attention. Using SoftMax we are going to end up with a probability-weight that shows how much the result focuses on each hidden state.
- Then we weight the output and we concat it with each hidden state.
- Lastly, we give it to a tanh layer and the result is the outcome where it contains the information from all the passed hidden layers.

By doing that we make it easier in a way for our model to have a better representation for the current sequence, considering the previous hidden state more.

Bidirectional

In a sequence we create many hidden states. Each hidden state is produced using the current input and the previous hidden layer. In this way we are using only the information from the previous words ONLY. But we can introduce two models. One going in one way and another which scans the sequence from the opposite direction. In this way we can see see the whole sequence and produce a more complete image for each hidden state.

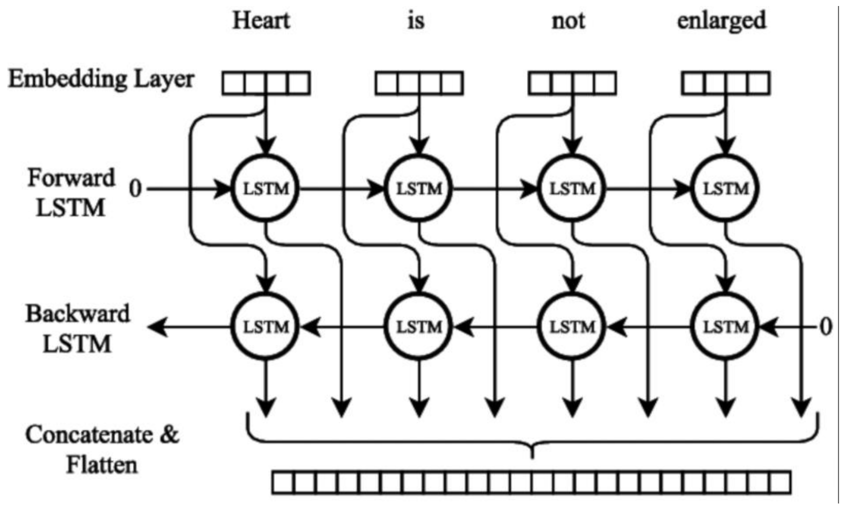


Figure 3 Bidirectional NN with LSTM CELLS

The problem here is that the linear layer that we are using at the end will need to have its length doubled, because now we are having 2 ways.

Skip Connections

This mechanism is used to provide an alternative path for the gradient in backpropagation. Skip connections in deep architectures skip some layer in the neural network and feeds the output of one layer as the input to the next layers (instead of only the next one).

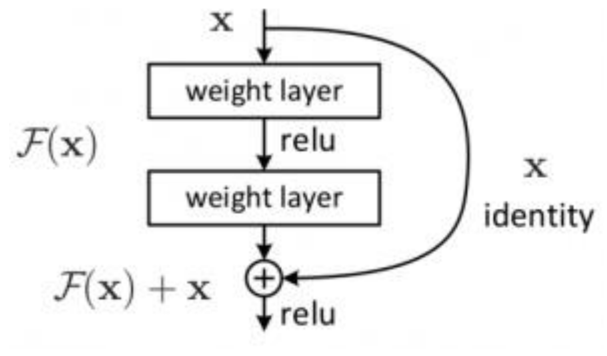

x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 4 Skip Connection Using Addition

The problem using the skip connection is that the hidden size should be the halve of the embeddings, in order to calculate the addition!! (In order to make the element-wise addition we need to have the same dimensions)

Drop-out

Drop-out probability is a percentage which shows a percentage of the input that will be kept. In this way we are adding some error to our model and help it to generalize easier.

Experiments

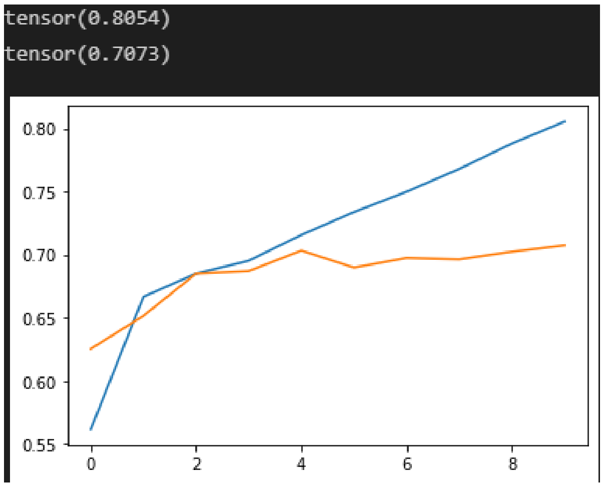
FOR ALL THE EXPERIMENTS WE USE ADAM OPTIMIZER AND CROSS ENTROPY AS LOSS FUNCTION. The Learning rate is stable at 0.001. These models are able to converge in way less epochs than the previous models that we had. The most models are running to 8-10 epochs even though the converge may be sooner.

The simplest version of our model gives really good results!

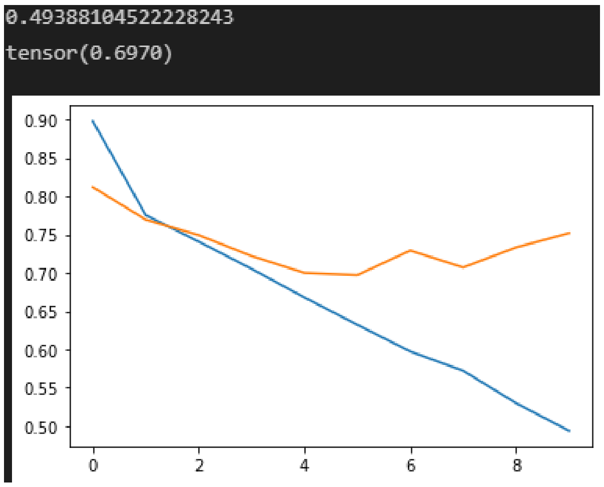
In the first run we are using LSTM cells with:

- Stacked layers=1
- Hidden size=128
- Drop-out=0
- Skip layers=No
- Bidirectional=No
- Attention=No

Results:



```
tensor(0.8054)
tensor(0.7073)
```



```
0.49388104522228243
tensor(0.6970)
```

Epoch 0: Loss = 0.89779 Accuracy = 0.56142 VAL_Loss = 0.81172 VAL_Accuracy = 0.62527

Epoch 1: Loss = 0.77531 Accuracy = 0.66661 VAL_Loss = 0.76937 VAL_Accuracy = 0.65150

Epoch 2: Loss = 0.74073 Accuracy = 0.68497 VAL_Loss = 0.74920 VAL_Accuracy = 0.68496

Epoch 3: Loss = 0.70513 Accuracy = 0.69517 VAL_Loss = 0.72188 VAL_Accuracy = 0.68698

Epoch 4: Loss = 0.66780 Accuracy = 0.71562 VAL_Loss = 0.69987 VAL_Accuracy = 0.70321

We can see the converge happens in the first 4 epochs. No 10 needed.

Metrics

Accuracy: 0.7072743207712533
F1_score: 0.649320855963457
Precision: 0.659240589882932
Recall: 0.6436293716756625

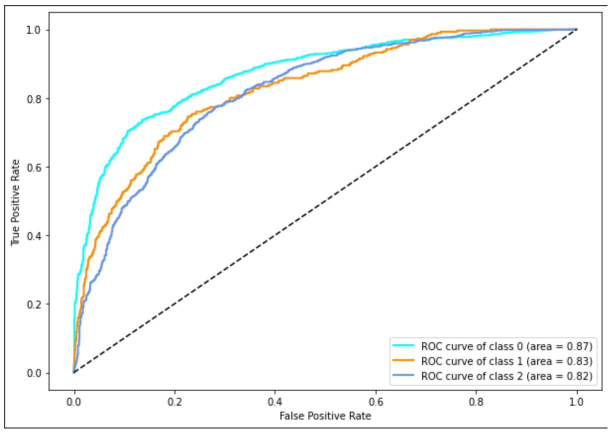


Figure 5 ROC

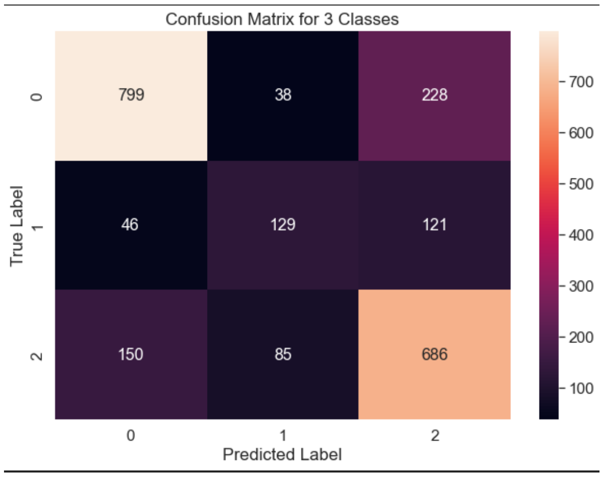
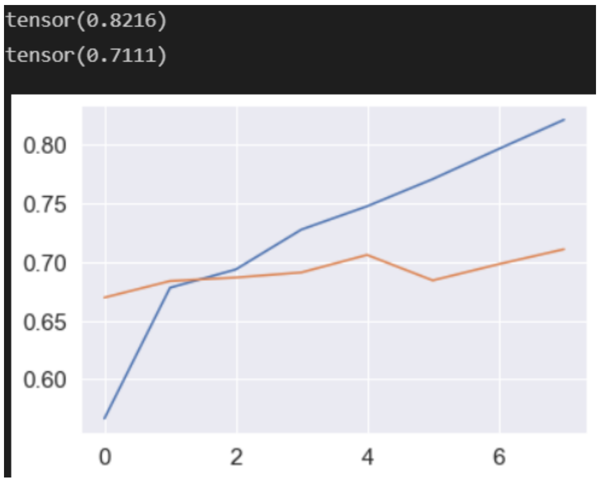


Figure 6 CONFUSSION MATRIX

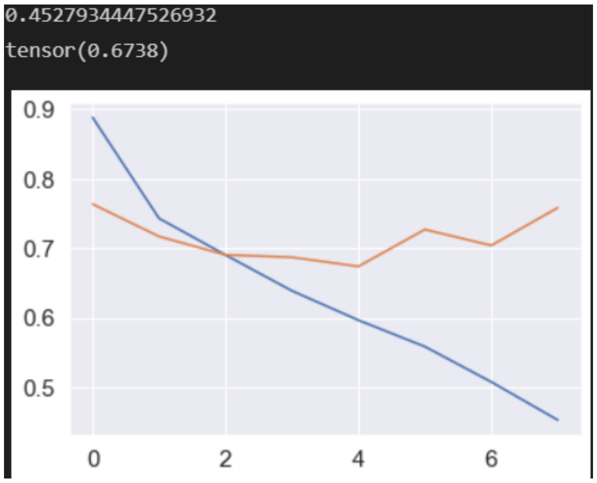
Very good results, that surpass the previous model!!

*We run the same model, but now we will use **GRUs** cells. The GRU follows the same logic with the LSTM, so we expect the same results. It is debatable and depends on the problem, which way we are going to follow.*

Results



```
tensor(0.8216)
tensor(0.7111)
```



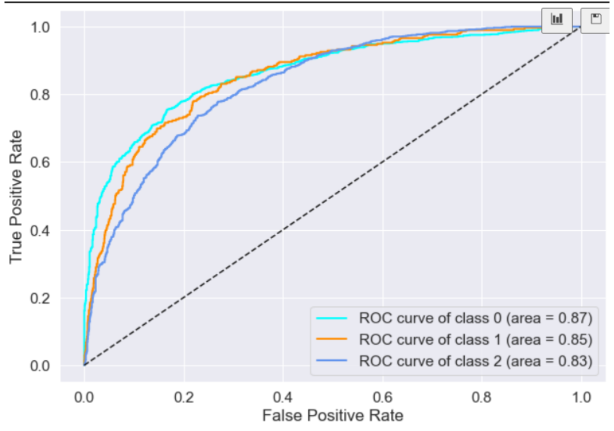
```
0.4527934447526932
tensor(0.6738)
```

Epoch 0: Loss = 0.88834 Accuracy = 0.56654 VAL_Loss = 0.76355 VAL_Accuracy = 0.66981
Epoch 1: Loss = 0.74306 Accuracy = 0.67822 VAL_Loss = 0.71700 VAL_Accuracy = 0.68392
Epoch 2: Loss = 0.69037 Accuracy = 0.69377 VAL_Loss = 0.69049 VAL_Accuracy = 0.68690
Epoch 3: Loss = 0.63880 Accuracy = 0.72784 VAL_Loss = 0.68704 VAL_Accuracy = 0.69124
Epoch 4: Loss = 0.59652 Accuracy = 0.74771 VAL_Loss = 0.67384 VAL_Accuracy = 0.70623

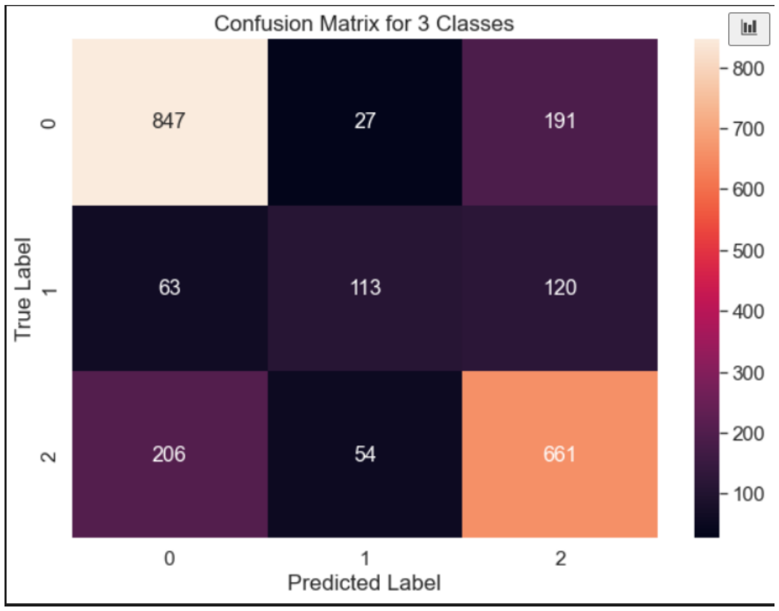
Metrics

Accuracy: 0.71034180543383
F1_score: 0.6454316032271531
Precision: 0.6738253175147335
Recall: 0.6315866917520815

ROC



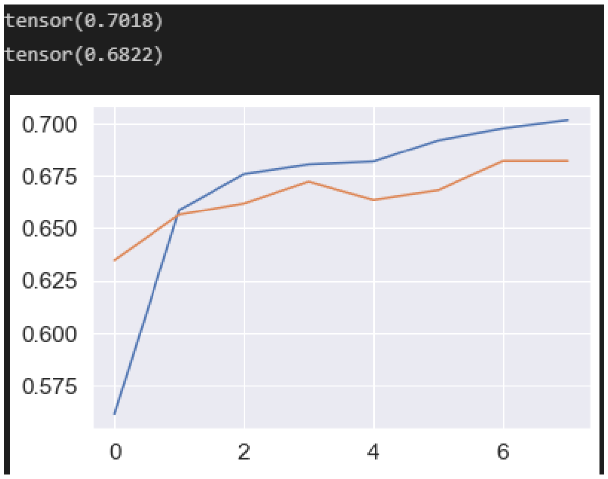
CONFUSSION MATRIX



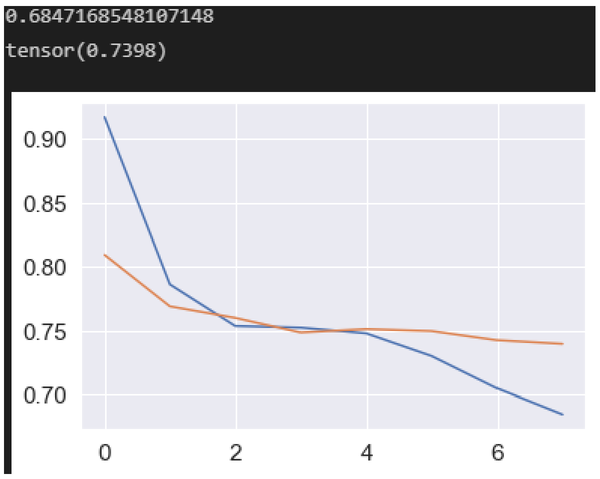
Because the differences are very small between the 2 different types of cells, I will continue the experiments using the LSTM model.

The next modification I will try is to have a stacked LSMT model (**stacked layers=2**). Which means a more complex model. What we expect is a better performance on the train set but probably a problem in generalization. Also, we expect to meet quicker converge, because our model will be more complex.

Results



```
tensor(0.7018)
tensor(0.6822)
```



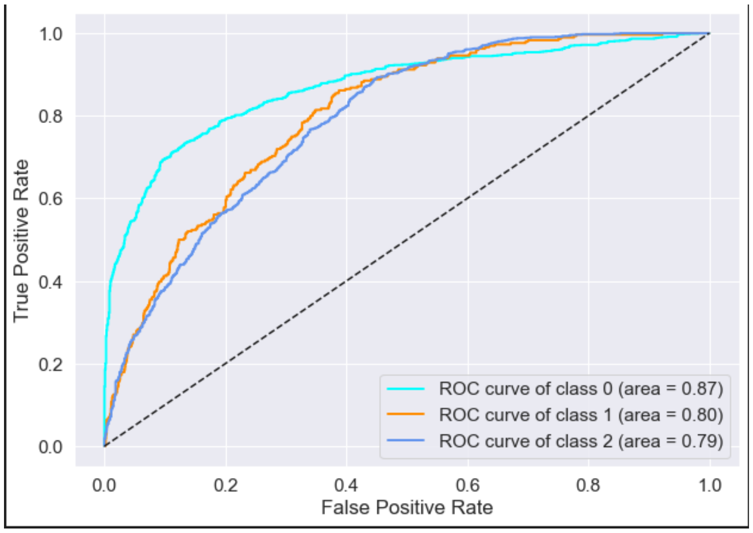
```
0.6847168548107148
tensor(0.7398)
```

Epoch 0: Loss = 0.91714 Accuracy = 0.56104 VAL_Loss = 0.80946 VAL_Accuracy = 0.63476
Epoch 1: Loss = 0.78644 Accuracy = 0.65869 VAL_Loss = 0.76922 VAL_Accuracy = 0.65652
Epoch 2: Loss = 0.75384 Accuracy = 0.67593 VAL_Loss = 0.76018 VAL_Accuracy = 0.66191
Epoch 3: Loss = 0.75253 Accuracy = 0.68054 VAL_Loss = 0.74873 VAL_Accuracy = 0.67233
As we expected the converge happened sooner. Due to the potential overfitting the results are not good enough.

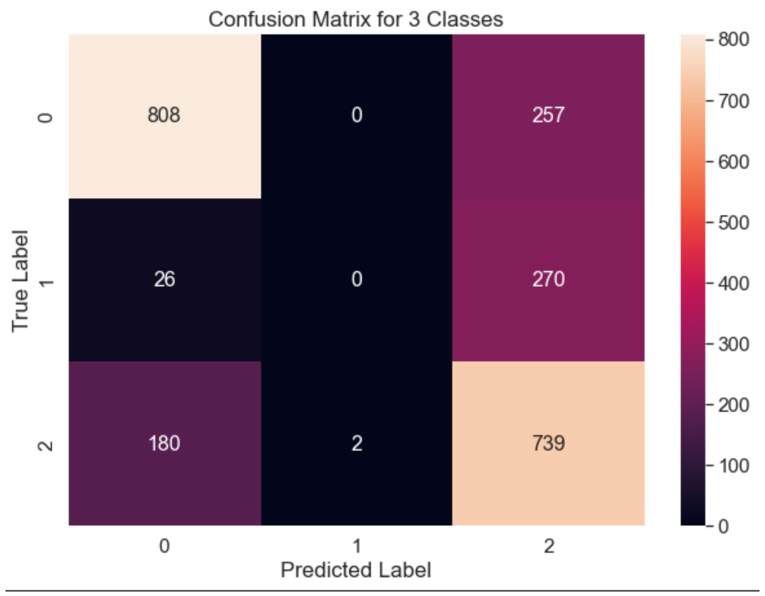
Metrics

Accuracy: 0.6779141104294478
F1_score: 0.48436946379333207
Precision: 0.46019081983354676
Recall: 0.5203580513118523

ROC

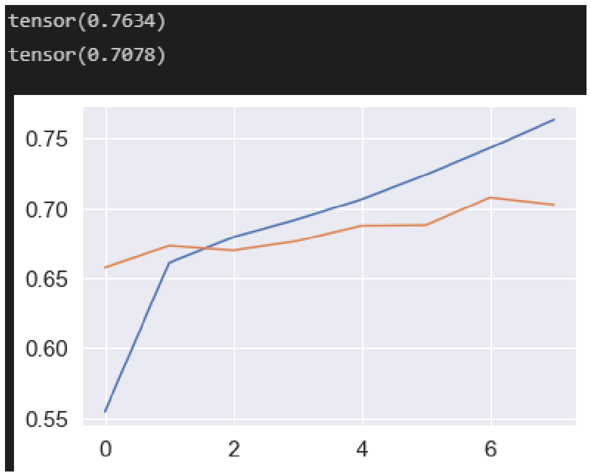


CONFUSSION MATRIX

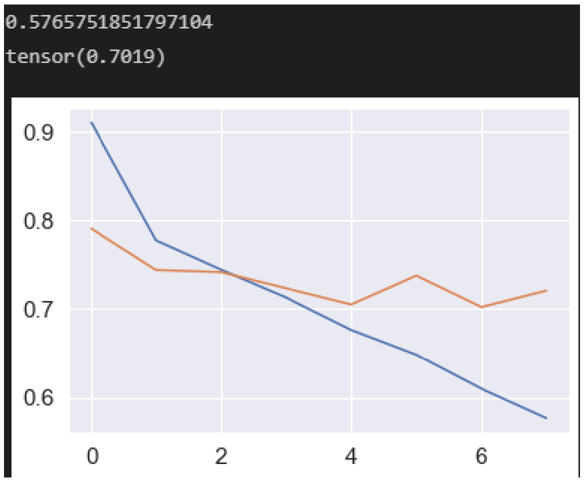


Because we probably suffering from overfitting let's try to make our model generalize easier. This will be achieved using **drop-out layers (0.3).**
As a result, we can see a model that has overcome overfitting (even from the confusion matrix) but performs a little bit worse.

Results



```
tensor(0.7634)
tensor(0.7078)
```



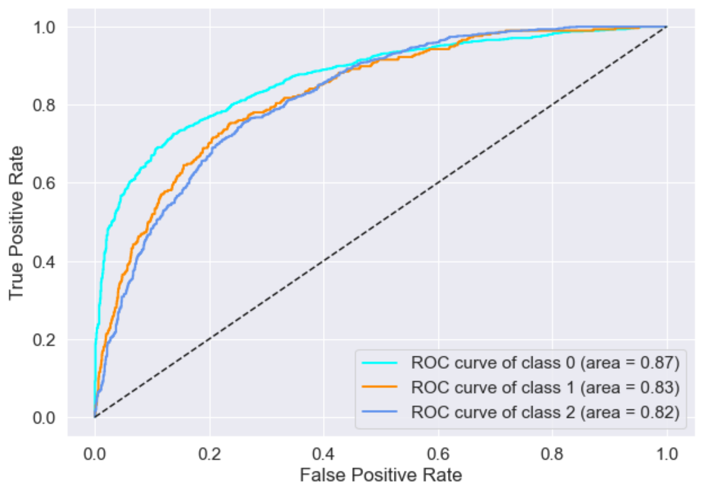
```
0.5765751851797104
tensor(0.7019)
```

Epoch 0: Loss = 0.90998 Accuracy = 0.55466 VAL_Loss = 0.79037 VAL_Accuracy = 0.65786
Epoch 1: Loss = 0.77699 Accuracy = 0.66118 VAL_Loss = 0.74368 VAL_Accuracy = 0.67328
Epoch 2: Loss = 0.74400 Accuracy = 0.67944 VAL_Loss = 0.74129 VAL_Accuracy = 0.66999
Epoch 3: Loss = 0.71260 Accuracy = 0.69211 VAL_Loss = 0.72311 VAL_Accuracy = 0.67671
Epoch 4: Loss = 0.67614 Accuracy = 0.70645 VAL_Loss = 0.70488 VAL_Accuracy = 0.68760

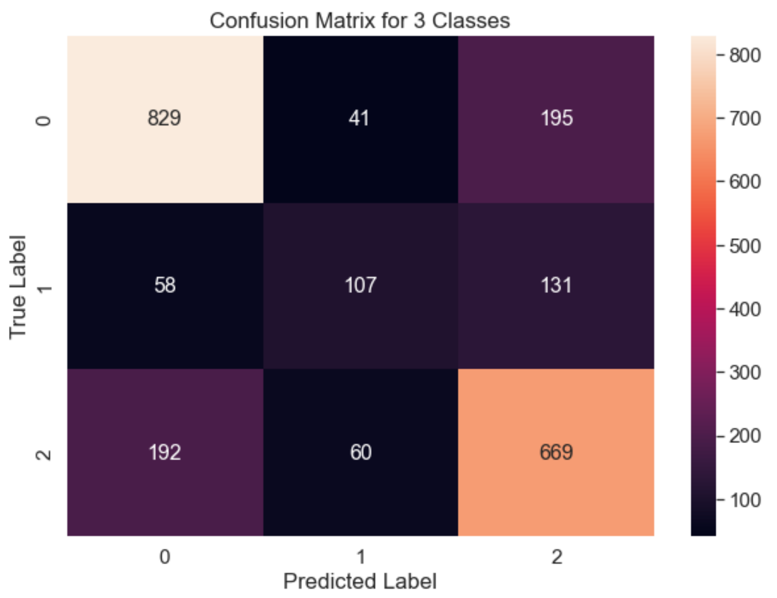
Metrics

Accuracy: 0.703330411919369
F1_score: 0.6320846413292585
Precision: 0.6516962903799194
Recall: 0.6220915357252926

ROC



CONFUSSION MATRIX

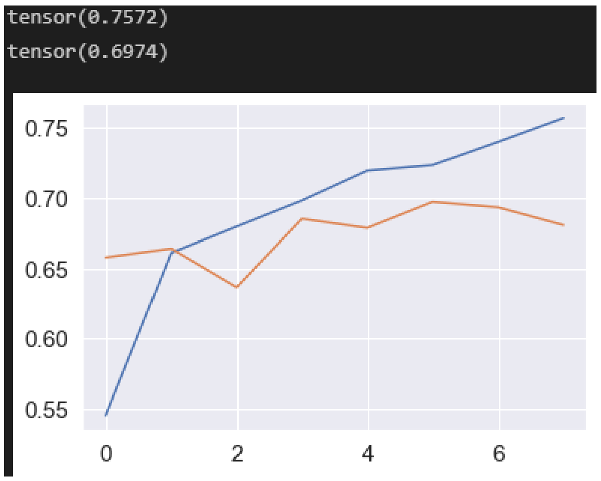


We can see clearly a better model.

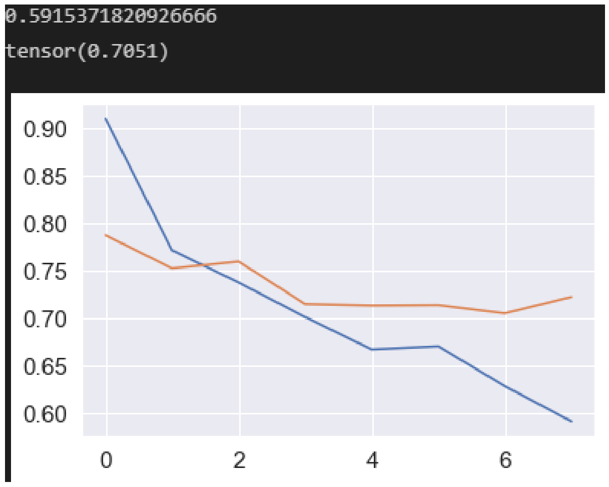
For more than 2 stacked layers the model becomes very complex for the specific problem. Therefore, there is no experiments on this path.

Then we are going to try the model in its bidirectional mode. This is done by doing the **bidir argument to True**. This will end up into a more complex model.

Results



```
tensor(0.7572)
tensor(0.6974)
```



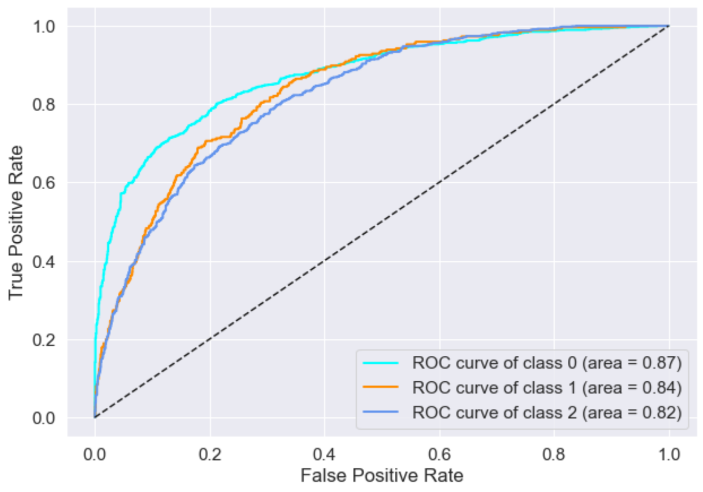
```
0.5915371820926666
tensor(0.7051)
```

Epoch   0: Loss = 0.90959 Accuracy = 0.54537 VAL_Loss = 0.78701 VAL_Accuracy = 0.65786
Epoch   1: Loss = 0.77119 Accuracy = 0.66114 VAL_Loss = 0.75228 VAL_Accuracy = 0.66412
Epoch   2: Loss = 0.73741 Accuracy = 0.67989 VAL_Loss = 0.75943 VAL_Accuracy = 0.63662
Epoch   3: Loss = 0.70107 Accuracy = 0.69840 VAL_Loss = 0.71471 VAL_Accuracy = 0.68543

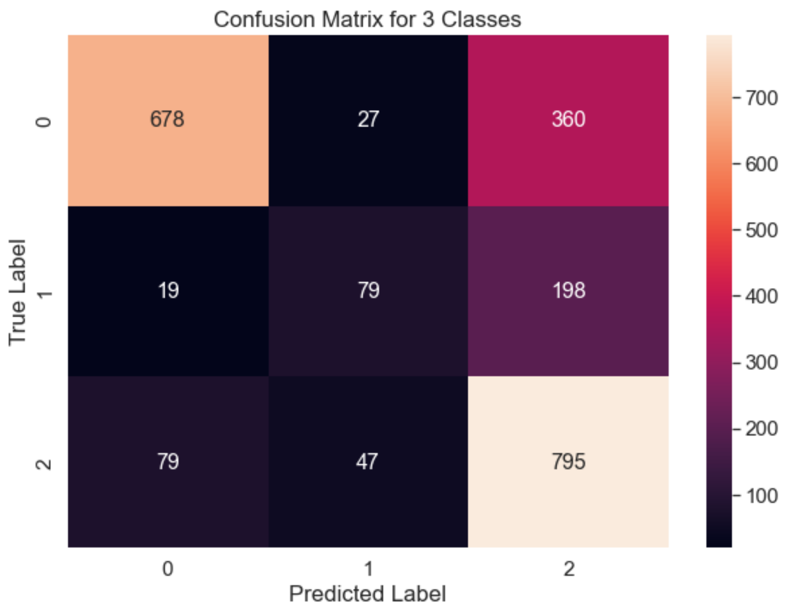
Metrics

Accuracy: 0.6801051709027169
F1_score: 0.5958859194953651
Precision: 0.6592114526819964
Recall: 0.5889012642040582

ROC

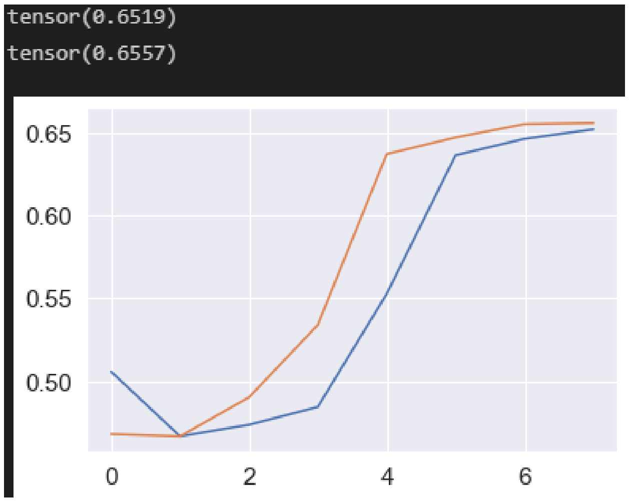


CONFUSSION MATRIX

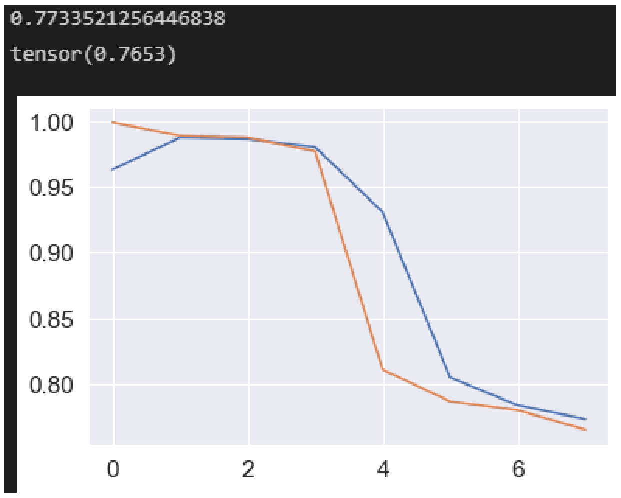


The model does not perform well. In theory this mechanic should help us, but it does not improve the model in this specific problem. Let's try applying the skip connections and see whether it can help. Because we are going to need the same dimensionality for the addition, we have to put the hidden size to 100 (Input size/2)
The new model seems more balanced.

Results



```
tensor(0.6519)
tensor(0.6557)
```

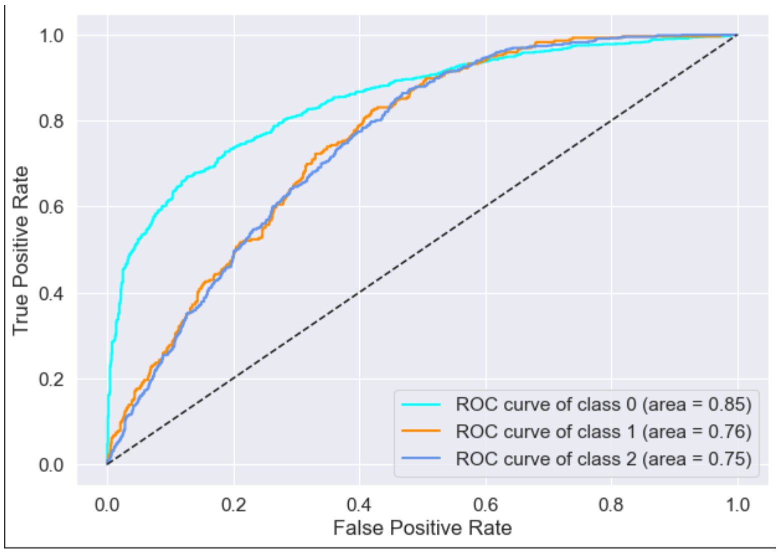


```
0.7733521256446838
tensor(0.7653)
```

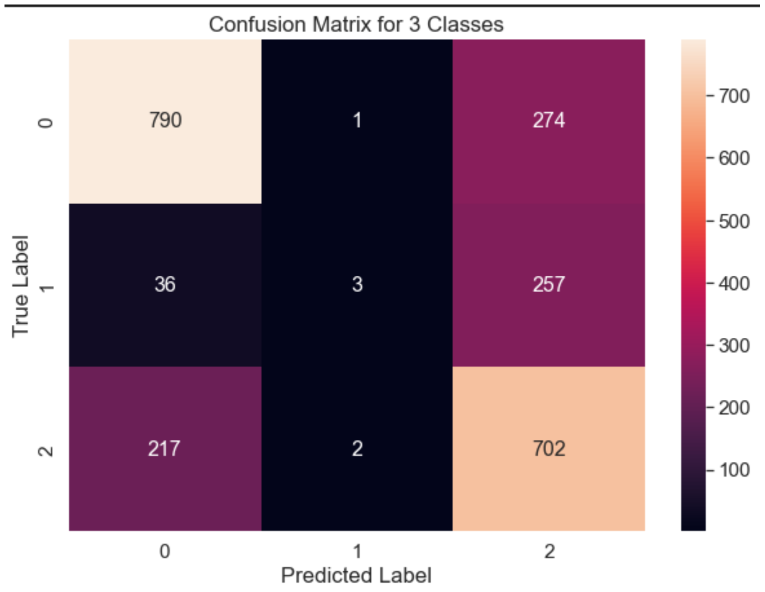
Metrics

Accuracy: 0.6551270815074496
F1_score: 0.47373458377512795
Precision: 0.6089245182225146
Recall: 0.5047113854690585

ROC

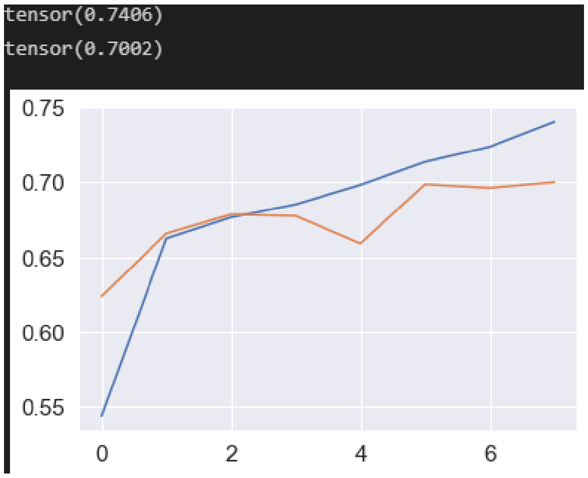


CONFUSSION MATRIX

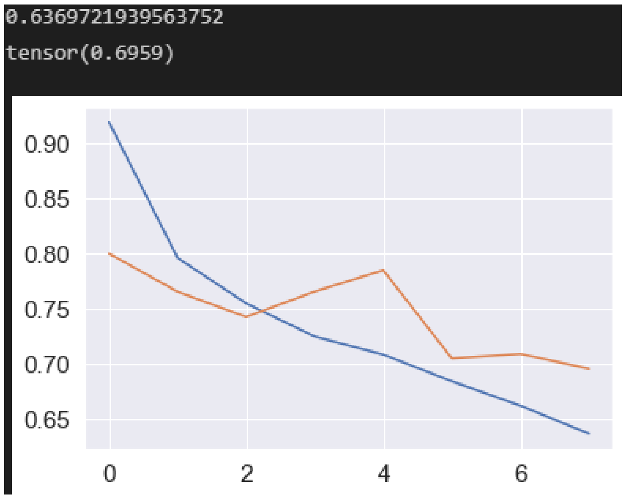


The previous model had the worst performance. BUT let's try to make it a little more complex, by adding layers in each LSTM (layers_num=3).

Results



```
tensor(0.7406)
tensor(0.7002)
```



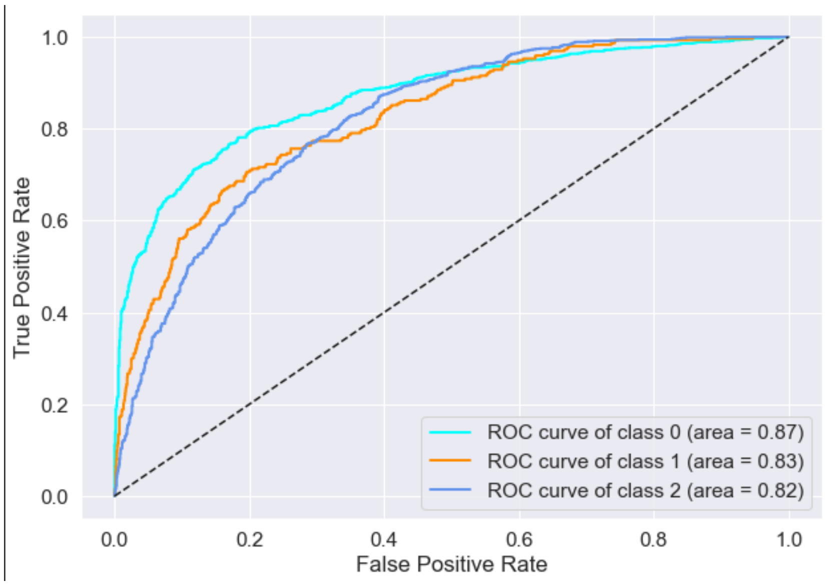
```
0.6369721939563752
tensor(0.6959)
```

```
Epoch   0: Loss = 0.91933 Accuracy = 0.54370 VAL_Loss = 0.80026 VAL_Accuracy = 0.62378
Epoch   1: Loss = 0.79628 Accuracy = 0.66247 VAL_Loss = 0.76566 VAL_Accuracy = 0.66582
Epoch   2: Loss = 0.75519 Accuracy = 0.67666 VAL_Loss = 0.74291 VAL_Accuracy = 0.67865
Epoch   3: Loss = 0.72507 Accuracy = 0.68522 VAL_Loss = 0.76586 VAL_Accuracy = 0.67762
Epoch   4: Loss = 0.70869 Accuracy = 0.69829 VAL_Loss = 0.78513 VAL_Accuracy = 0.65910
Epoch   5: Loss = 0.68446 Accuracy = 0.71376 VAL_Loss = 0.70521 VAL_Accuracy = 0.69868
Epoch   6: Loss = 0.66243 Accuracy = 0.72380 VAL_Loss = 0.70928 VAL_Accuracy = 0.69626
Epoch   7: Loss = 0.63697 Accuracy = 0.74058 VAL_Loss = 0.69586 VAL_Accuracy = 0.70019
```

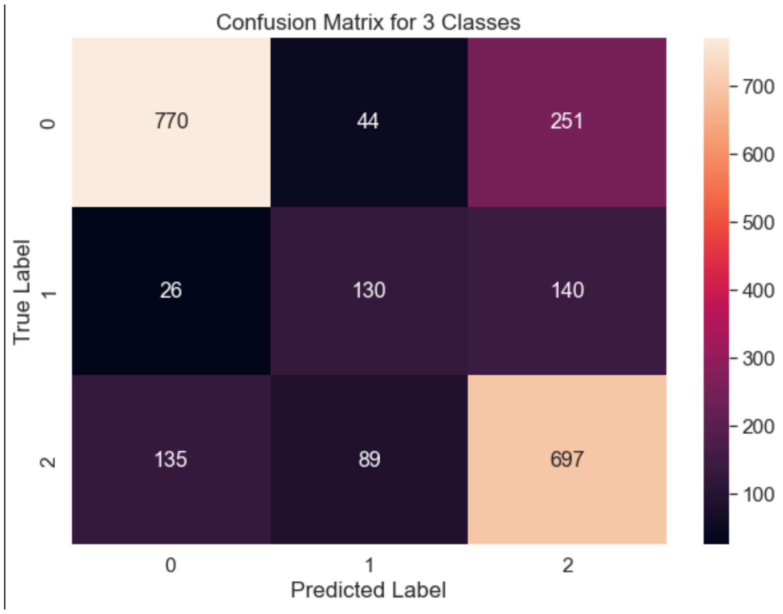
Metrics

Accuracy: 0.6998247151621385
F1_score: 0.64351230542084
Precision: 0.6539964157065667
Recall: 0.639659995362615

ROC



CONFUSSION MATRIX



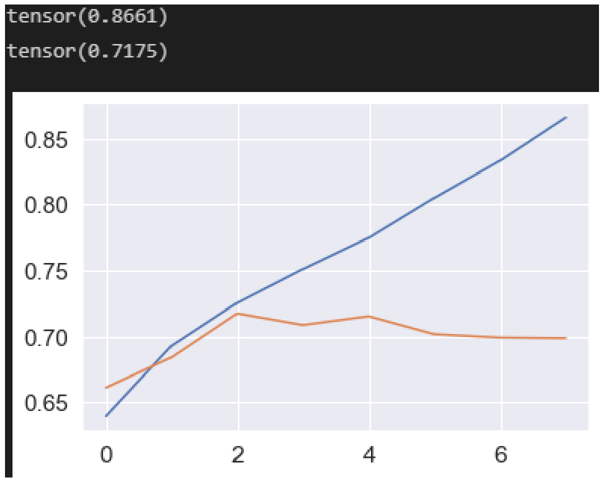
	0	1	2
0	770	44	251
1	26	130	140
2	135	89	697

It worked!! Adding more complexity makes the model works just fine with very good results.

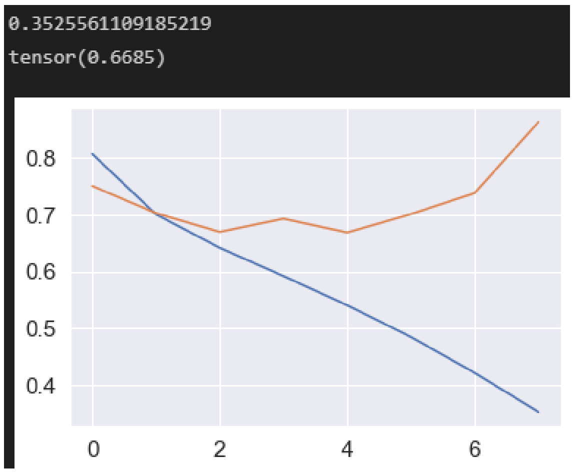
Our last attempt to improve our model will be to add attention to it. We can accomplish this by setting **att=True**

Using attention need about 3 epochs to converge.

Results



```
tensor(0.8661)
tensor(0.7175)
```

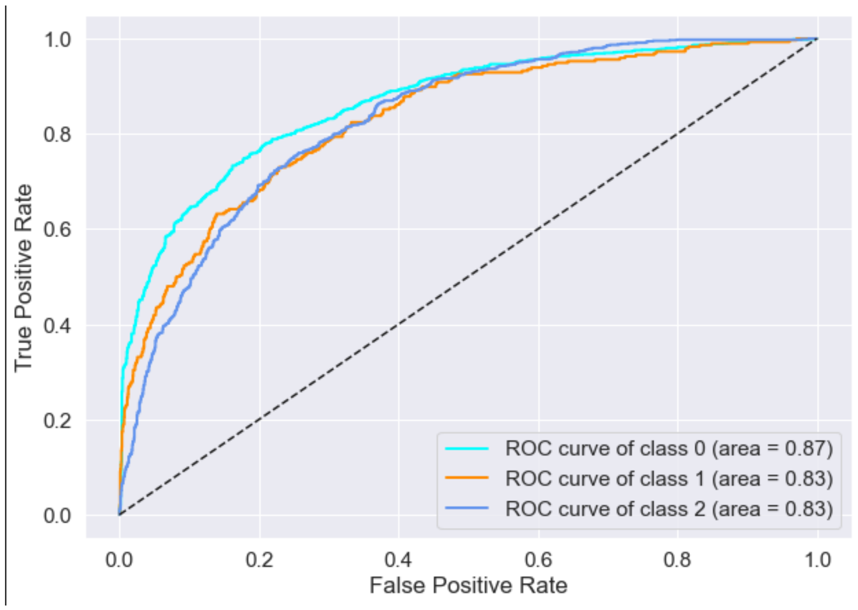


```
0.3525561109185219
tensor(0.6685)
```

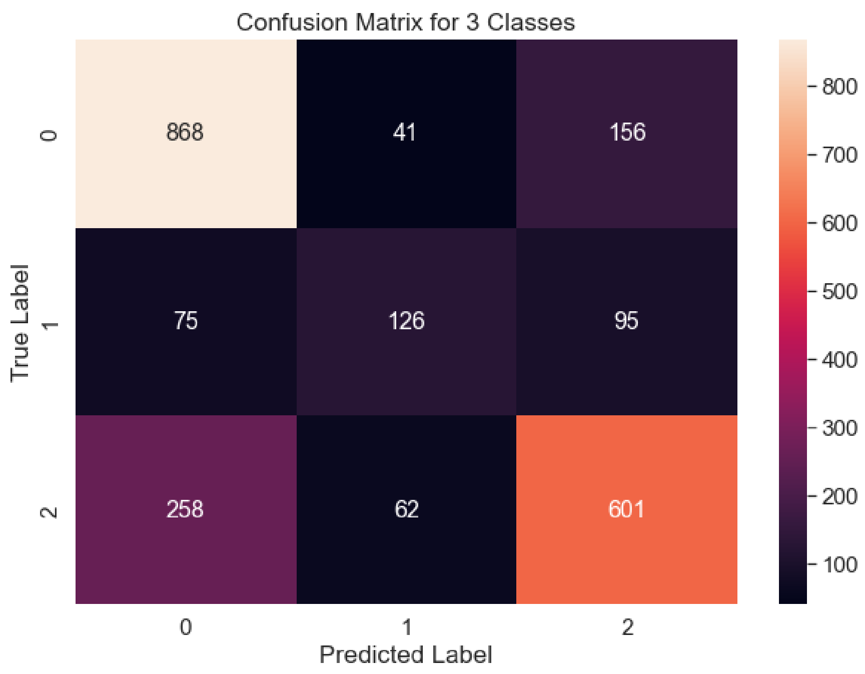
Epoch 0: Loss = 0.80694 Accuracy = 0.63978 VAL_Loss = 0.75043 VAL_Accuracy = 0.66136
Epoch 1: Loss = 0.70053 Accuracy = 0.69309 VAL_Loss = 0.70266 VAL_Accuracy = 0.68436
Epoch 2: Loss = 0.64156 Accuracy = 0.72570 VAL_Loss = 0.66955 VAL_Accuracy = 0.71751

Metrics

Accuracy: 0.6989482909728308
F1_score: 0.6413515537481828
Precision: 0.659449486365447
Recall: 0.6310835747432527

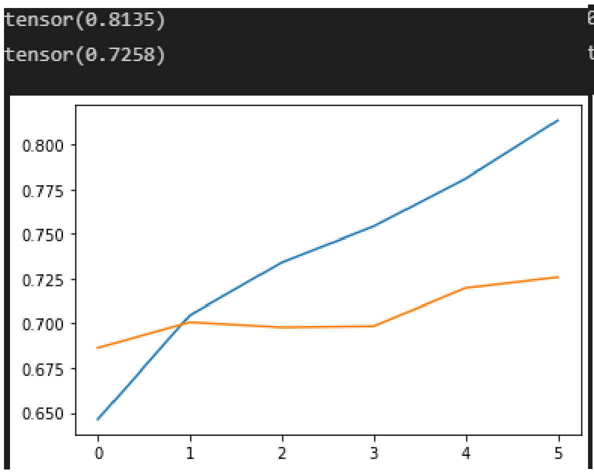


CONFUSSION MATRIX

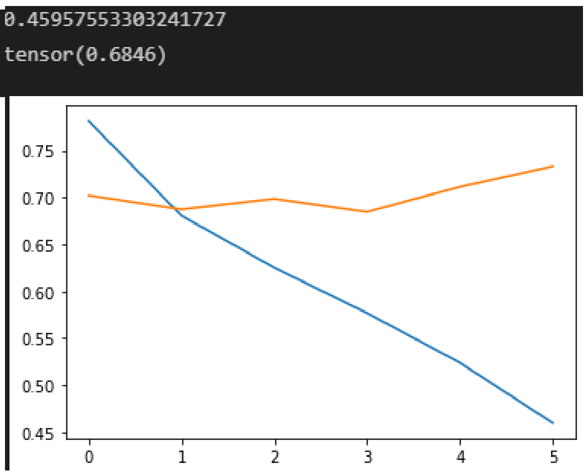


LASTLY I tried with the addition of the attention to make my model simpler. This is why I removed a layer from the stacked layers inside the LSTMs (layers_num=2)

Results



```
tensor(0.8135)
tensor(0.7258)
```



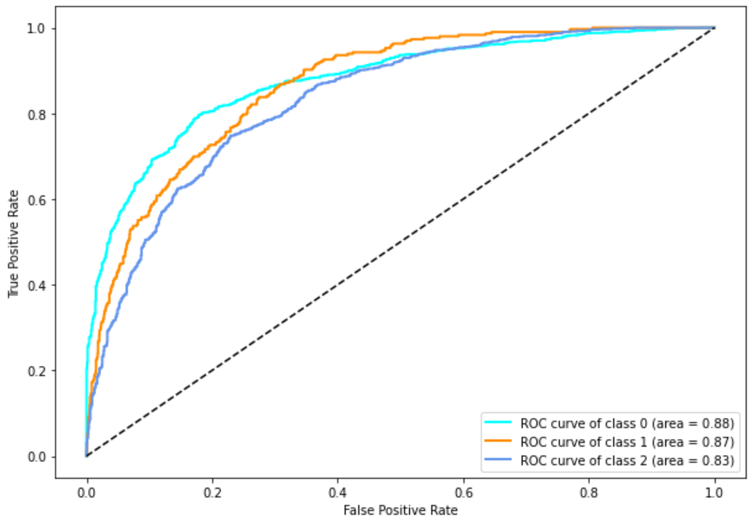
```
0.45957553303241727
tensor(0.6846)
```

Epoch 0: Loss = 0.78143 Accuracy = 0.64628 VAL_Loss = 0.70195 VAL_Accuracy = 0.68624
Epoch 1: Loss = 0.68061 Accuracy = 0.70454 VAL_Loss = 0.68738 VAL_Accuracy = 0.70062
Epoch 2: Loss = 0.62516 Accuracy = 0.73405 VAL_Loss = 0.69840 VAL_Accuracy = 0.69777
Converge around epoch 3

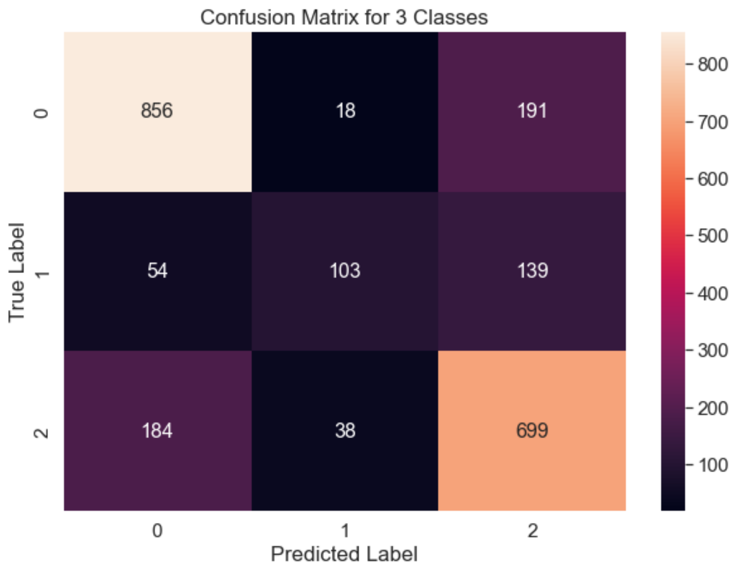
Metrics

Accuracy: 0.6989482909728308
F1_score: 0.6413515537481828
Precision: 0.659449486365447
Recall: 0.6310835747432527

ROC

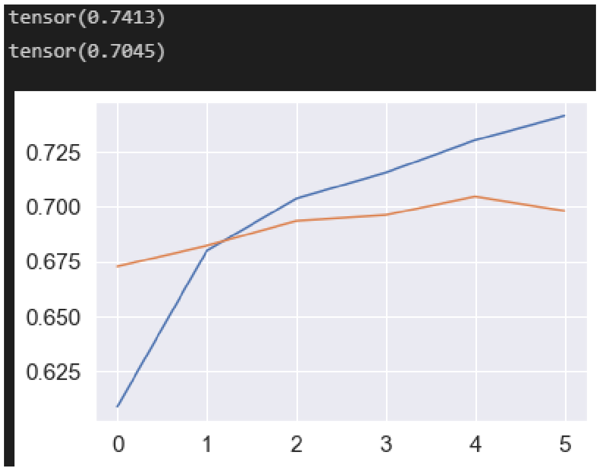


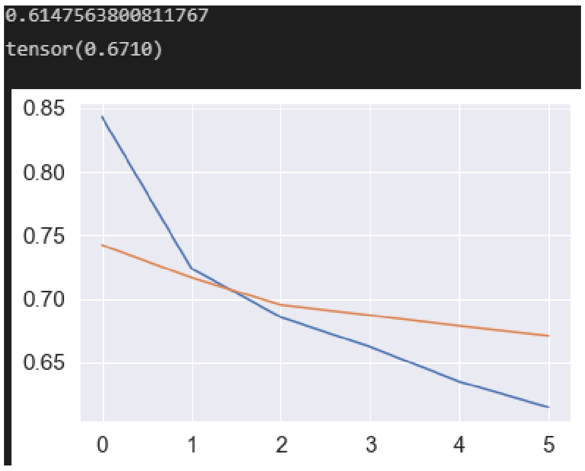
CONFUSSION MATRIX



The last change I performed to make my model is that the charts of loss and accuracy are not very “beautiful”. And shows overfitting try to make the learning rate less: 0.0003 and add to adam the L2 weight in order to grneralize easier. The results are below.

Results

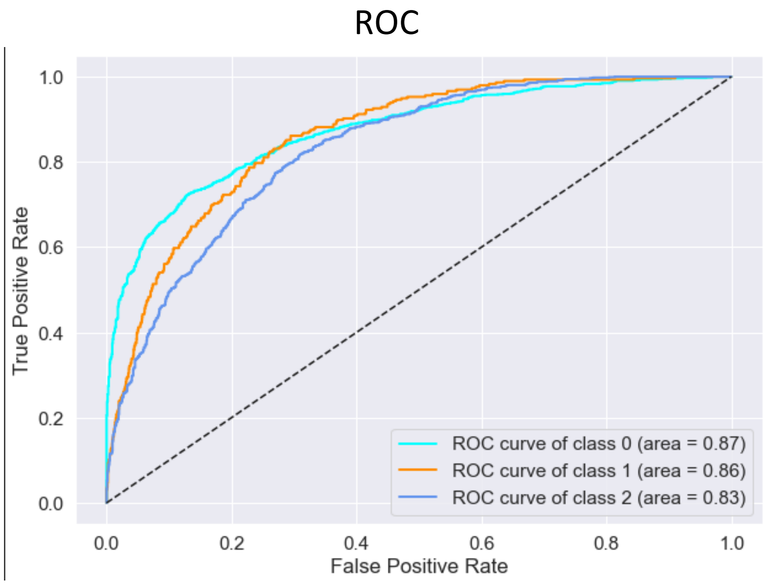

```
tensor(0.7413)
tensor(0.7045)
```


```
0.6147563800811767
tensor(0.6710)
```

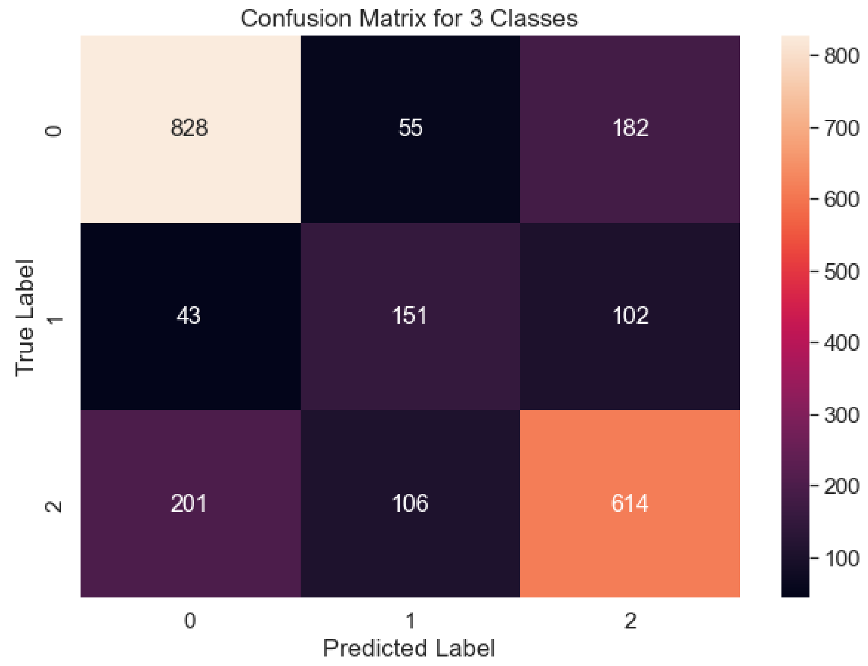
Epoch 0: Loss = 0.84295 Accuracy = 0.60870 VAL_Loss = 0.74225 VAL_Accuracy = 0.67264
Epoch 1: Loss = 0.72394 Accuracy = 0.67984 VAL_Loss = 0.71668 VAL_Accuracy = 0.68213
Epoch 2: Loss = 0.68563 Accuracy = 0.70359 VAL_Loss = 0.69522 VAL_Accuracy = 0.69341
Epoch 3: Loss = 0.66228 Accuracy = 0.71541 VAL_Loss = 0.68709 VAL_Accuracy = 0.69610
Epoch 4: Loss = 0.63495 Accuracy = 0.73019 VAL_Loss = 0.67876 VAL_Accuracy = 0.70451
Epoch 5: Loss = 0.61476 Accuracy = 0.74126 VAL_Loss = 0.67099 VAL_Accuracy = 0.69800

Metrics

Accuracy: 0.6980718667835232
F1_score: 0.6489082808406899
Precision: 0.6467013555942519
Recall: 0.6514221968447319

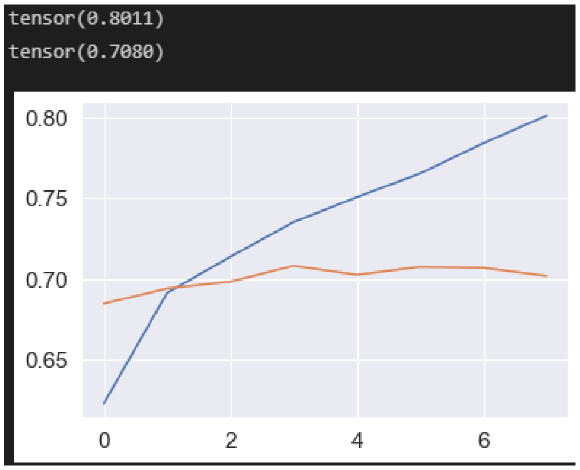


CONFUSSION MATRIX

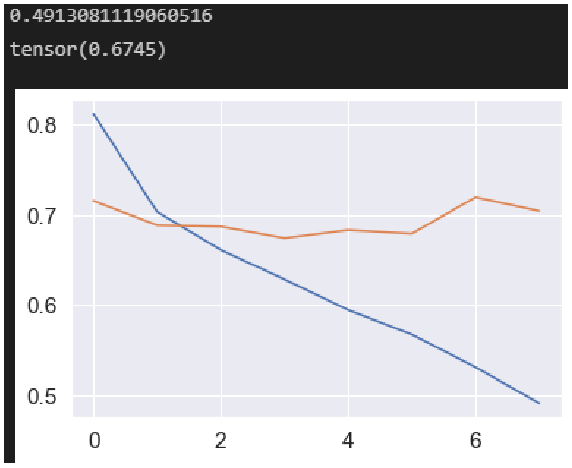


This results are very good and the errors are pretty balanced.
Try a simpler version of the previous model, having only 1 stacked layer. Also make the learning rate bigger, in order to help our less complex model to learn better.

Results



```
tensor(0.8011)
tensor(0.7080)
```



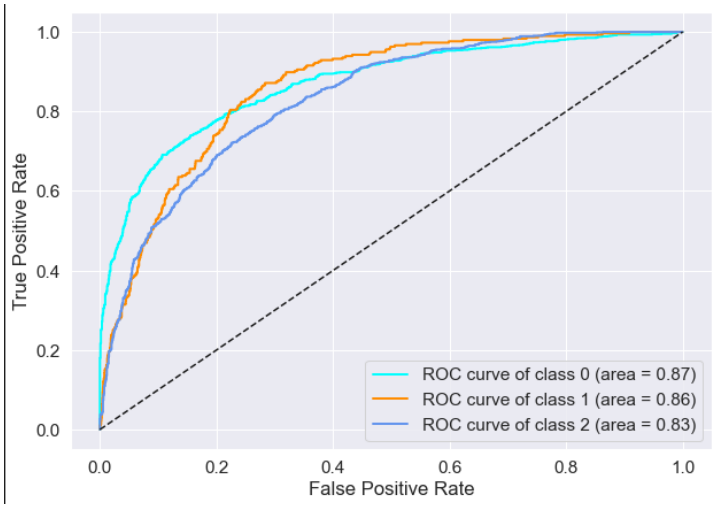
```
0.4913081119060516
tensor(0.6745)
```

Epoch 0: Loss = 0.81238 Accuracy = 0.62259 VAL_Loss = 0.71585 VAL_Accuracy = 0.68454
Epoch 1: Loss = 0.70377 Accuracy = 0.69119 VAL_Loss = 0.68883 VAL_Accuracy = 0.69391
Epoch 2: Loss = 0.66140 Accuracy = 0.71358 VAL_Loss = 0.68736 VAL_Accuracy = 0.69816
Epoch 3: Loss = 0.62880 Accuracy = 0.73507 VAL_Loss = 0.67448 VAL_Accuracy = 0.70798
Epoch 4: Loss = 0.59487 Accuracy = 0.75055 VAL_Loss = 0.68341 VAL_Accuracy = 0.70240
Epoch 5: Loss = 0.56763 Accuracy = 0.76522 VAL_Loss = 0.67943 VAL_Accuracy = 0.70730

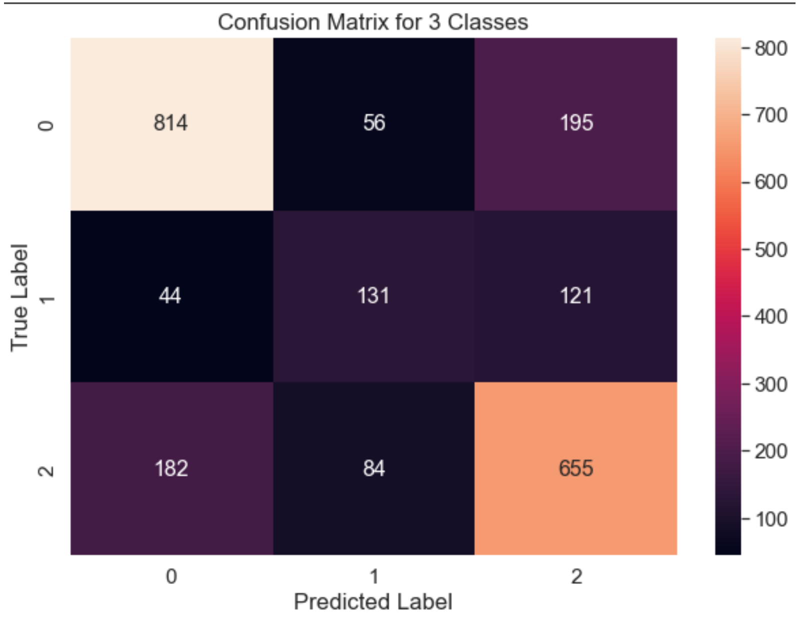
Metrics

Accuracy: 0.7011393514461
F1_score: 0.6426222698915378
Precision: 0.6468831495135834
Recall: 0.6393567708645471

ROC

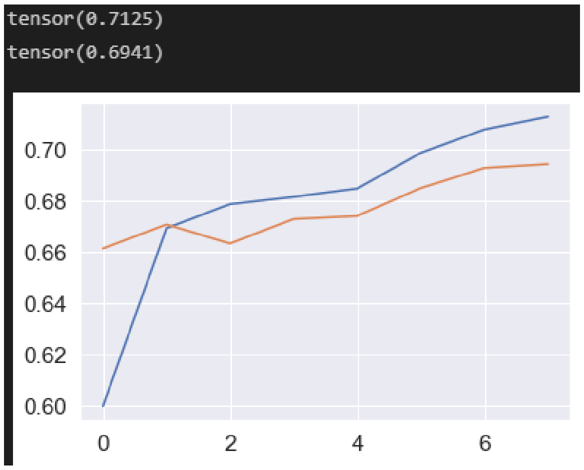


CONFUSSION MATRIX

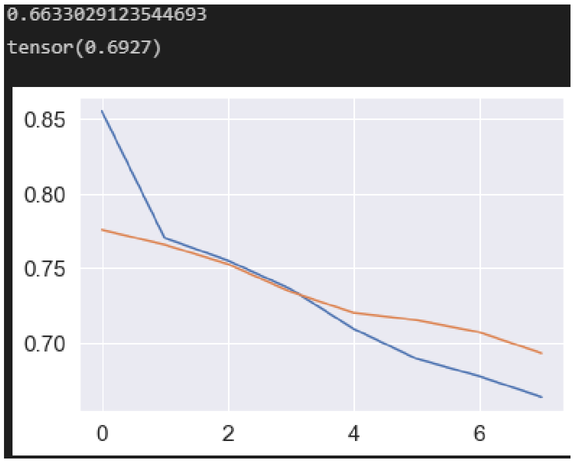


Try to reduce the error (Loss) which I face, because it stays up.

Results



```
tensor(0.7125)
tensor(0.6941)
```

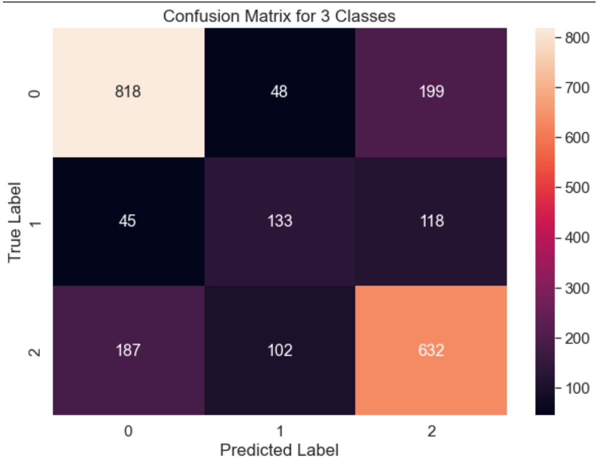


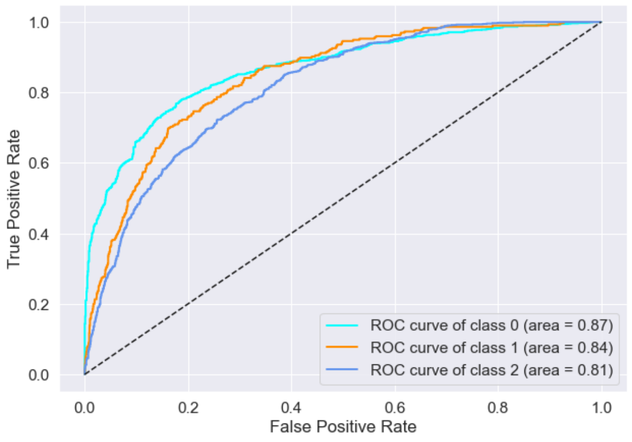
```
0.6633029123544693
tensor(0.6927)
```

Epoch 0: Loss = 0.85545 Accuracy = 0.60001 VAL_Loss = 0.77564 VAL_Accuracy = 0.66133
Epoch 1: Loss = 0.77018 Accuracy = 0.66916 VAL_Loss = 0.76564 VAL_Accuracy = 0.67061
Epoch 2: Loss = 0.75514 Accuracy = 0.67858 VAL_Loss = 0.75276 VAL_Accuracy = 0.66328
Epoch 3: Loss = 0.73600 Accuracy = 0.68140 VAL_Loss = 0.73407 VAL_Accuracy = 0.67278
Epoch 4: Loss = 0.70927 Accuracy = 0.68460 VAL_Loss = 0.71997 VAL_Accuracy = 0.67396
Epoch 5: Loss = 0.68913 Accuracy = 0.69849 VAL_Loss = 0.71500 VAL_Accuracy = 0.68479
Epoch 6: Loss = 0.67729 Accuracy = 0.70755 VAL_Loss = 0.70691 VAL_Accuracy = 0.69261
Epoch 7: Loss = 0.66330 Accuracy = 0.71251 VAL_Loss = 0.69272 VAL_Accuracy = 0.69411

Metrics

Accuracy: 0.6936897458369851
F1_score: 0.6362903560540504
Precision: 0.6383254853906869
Recall: 0.6345366941010836



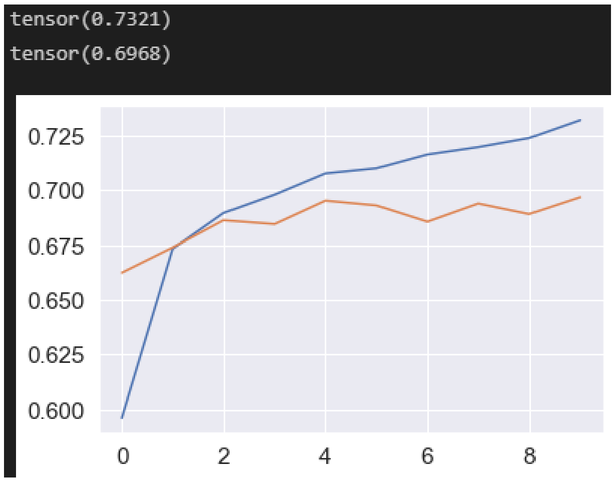


MY BEST MODEL

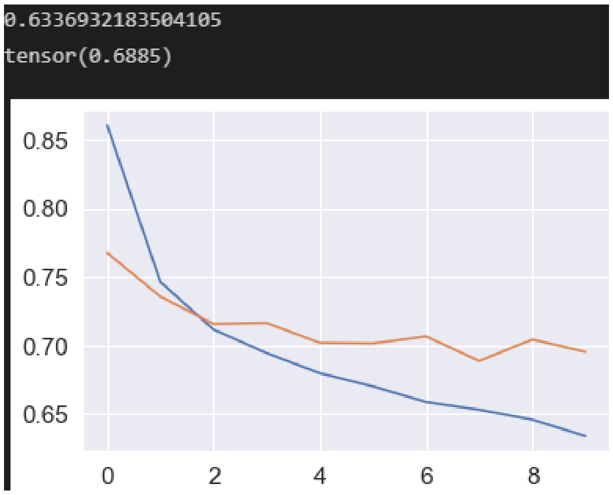
The final model that I present has arguments:

D_in=200,
hidden_s=100,
D_out=3,
cell_t="LSTM",
stacked_n=2,
layers_n=2,
sk_layers=True,
dr_out=0.3,
bidir=True,
b_s=64,
att=True

RESULTS



```
tensor(0.7321)
tensor(0.6968)
```



```
0.6336932183504105
tensor(0.6885)
```

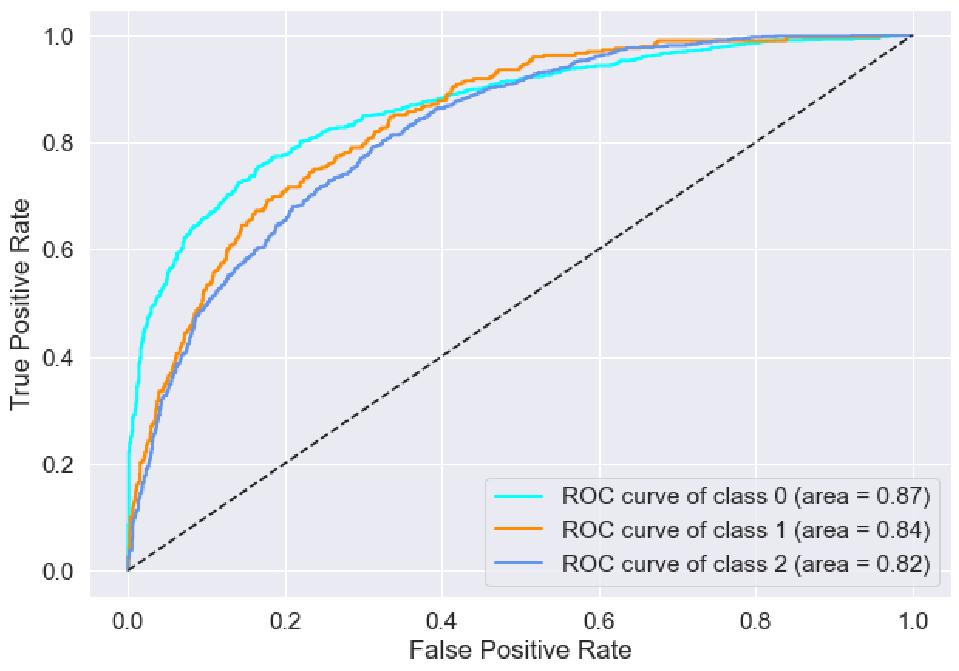
The converge happens around the 7th epoch, based on the loss chart.

```
Epoch   0: Loss = 0.86065 Accuracy = 0.59606 VAL_Loss = 0.76744 VAL_Accuracy = 0.66239
Epoch   1: Loss = 0.74629 Accuracy = 0.67326 VAL_Loss = 0.73556 VAL_Accuracy = 0.67388
Epoch   2: Loss = 0.71147 Accuracy = 0.68959 VAL_Loss = 0.71554 VAL_Accuracy = 0.68628
Epoch   3: Loss = 0.69429 Accuracy = 0.69795 VAL_Loss = 0.71622 VAL_Accuracy = 0.68454
Epoch   4: Loss = 0.67963 Accuracy = 0.70769 VAL_Loss = 0.70176 VAL_Accuracy = 0.69519
Epoch   5: Loss = 0.67001 Accuracy = 0.71004 VAL_Loss = 0.70137 VAL_Accuracy = 0.69302
Epoch   6: Loss = 0.65857 Accuracy = 0.71635 VAL_Loss = 0.70647 VAL_Accuracy = 0.68562
Epoch   7: Loss = 0.65289 Accuracy = 0.71974 VAL_Loss = 0.68851 VAL_Accuracy = 0.69387
```

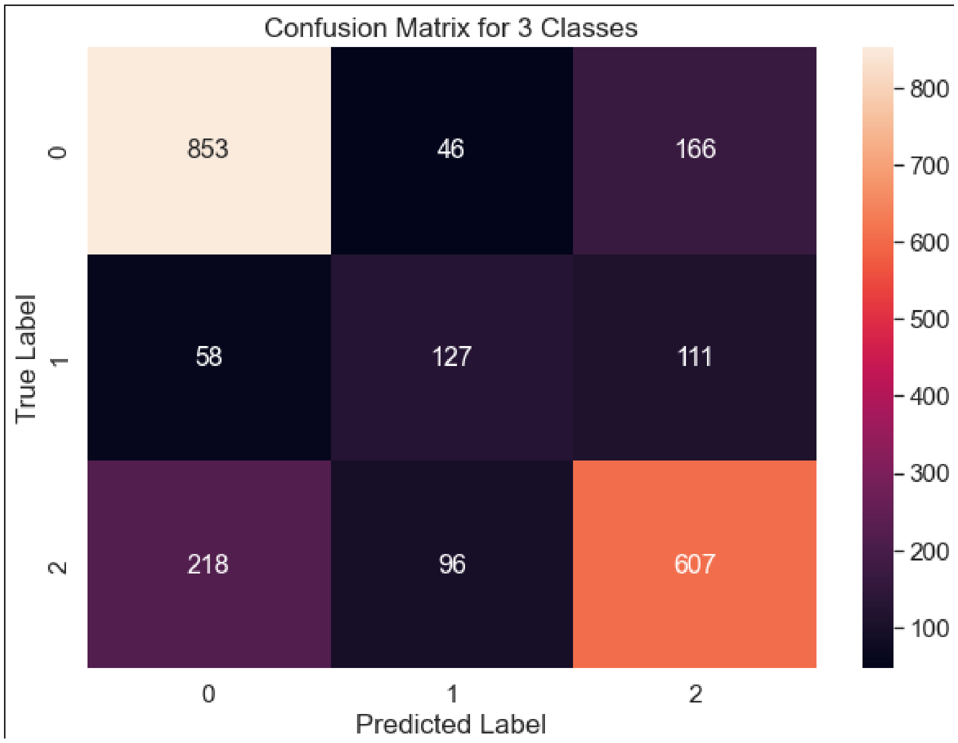
METRICS

```
           precision    recall  f1-score   support

       0.0       0.76      0.80      0.78      1065
       1.0       0.47      0.43      0.45       296
       2.0       0.69      0.66      0.67       921

    accuracy                           0.70      2282
   macro avg       0.64      0.63      0.63      2282
weighted avg       0.69      0.70      0.69      2282

Accuracy: 0.6954425942156004
F1_score: 0.6332363015046812
Precision: 0.6381021384239042
Recall: 0.6296864178487795
```

ROC



CONFUSION MATRIX



CONCLUSION

This model performs in terms of metrics as good as my previous model that uses NN. But in terms of loss this model does better job than the previous once and in a more balanced way. Overfitting does not exist in a very high grade. And also the previous models needed more epochs to be trained. The LSTM model needs only a few epochs to converge. Which is a logical outome because it uses batches of sequences and learn in a deeper way the semantic information of each sentence. Probably there is a format that can perform better, but for now this is how far I can go.