Artificial Intelligence II Exercise 2

Michael Mitsios (cs2200011)

Jan 2022

Part 1

Compute the gradient of the cross entropy loss function with respect to its logits.
The Cross entropy function is the following:

$$L = -\sum_{l=1}^{k} y_l log(p_l)$$

where:
K is the number of classes
m the number of inputs
p is the softmax function

p is described by the function below:

$$p_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

For each input our neural network will produce K different values. Each one of these values describes a propability of one input to belong to the specific class. We will have one **y for each one of the K classes.** In simple words:

$$output = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{bmatrix} =$$

use softmax function for each y->

$$\begin{bmatrix} \frac{e^{y_1}}{\sum_{j=1}^{k} e^{y_j}} \\ \frac{e^{y_2}}{\sum_{j=1}^{k} e^{y_j}} \\ \vdots \\ \frac{e^{y_k}}{\sum_{j=1}^{k} e^{y_j}} \end{bmatrix}$$

For each output neuron we take a propability using the softmax function.

First step we need to calculate the derivative of softmax function with respect to output **y** because we are gonna use it. For each output y_j ,jϵ[1-k]

$$\frac{\partial p_i}{\partial y_j} = \frac{\frac{\partial e^{y_1}}{\partial \sum_{l=1}^{k} e^{y_l}}}{\partial y_j} (1)$$

We will use this derivative rule: for a function $f(x) = \frac{g(x)}{h(x)}$

$$f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{h(x)^2}$$

Following the previous rule for the relation (1) we will have 2 cases:

CASE 1 i=j: The softmax function $p(y_i)$ will be derived with respect to the same y_i that gets as input.

$$\frac{\partial p_i}{\partial y_j} = \frac{\frac{\partial e^{y_1}}{\partial \sum_{l=1}^{k} e^{y_l}}}{\partial y_j} =^*$$

$*\frac{\partial \sum_{l=1}^{k} e^{y_l}}{\partial y_l} = \frac{\partial (e^{y_1}+e^{y_2}+\ldots+e^{y_k})}{\partial y_l} = e^{y_l}$

$$\frac{e^{y_j}\sum_{l=1}^{k} e^{y_l} - e^{y_j}e^{y_j}}{(\sum_{l=1}^{k} e^{y_l})^2} =$$

$$\frac{e^{y_j}(\sum_{l=1}^{k} e^{y_l} - e^{y_j})}{(\sum_{l=1}^{k} e^{y_l})^2} =$$

$$\frac{e^{y_j}}{\sum_{l=1}^{k} e^{y_l}} \cdot \frac{\sum_{l=1}^{k} e^{y_l} - e^{y_j}}{\sum_{l=1}^{k} e^{y_l}} =$$

$$\frac{e^{y_j}}{\sum_{l=1}^{k} e^{y_l}} \cdot (\frac{\sum_{l=1}^{k} e^{y_l}}{\sum_{l=1}^{k} e^{y_l}} - \frac{e^{y_j}}{\sum_{l=1}^{k} e^{y_l}}) = \frac{e^{y_j}}{\sum_{l=1}^{k} e^{y_l}} \cdot (1 - \frac{e^{y_j}}{\sum_{l=1}^{k} e^{y_l}}) =$$

$$p_i(1-p_j)$$

(2)

CASE 2 i≠j: The softmax function $p(y_i)$ will be derived with respect to a different y_j from the input.

$$\frac{\partial p_i}{\partial y_j} = \frac{\frac{\partial e^{y_1}}{\partial \sum_{l=1}^{k} e^{y_l}}}{\partial y_j} =$$

$$\frac{0\sum_{l=1}^{k} e^{y_l} - e^{y_i}e^{y_j}}{(\sum_{l=1}^{k} e^{y_l})^2} = \frac{0 - e^{y_i}e^{y_j}}{(\sum_{l=1}^{k} e^{y_l})^2}$$

$$\frac{-e^{y_i}}{(\sum_{l=1}^{k} e^{y_l})} \cdot \frac{e^{y_j}}{(\sum_{l=1}^{k} e^{y_l})} = -p_i \cdot p_j =$$

$$p_i(0-p_j)$$

(3)

From what we have seen until now the output of the gradient of softmax will have the form of a Jackobian matrix (K is the number of classes):

$$Jsoftmax = \begin{bmatrix} \nabla p_1 \\ \nabla p_2 \\ \vdots \\ \nabla p_k \end{bmatrix} = \begin{bmatrix} \frac{\partial p_1}{\partial y_1} & \frac{\partial p_1}{\partial y_2} & \cdots & \frac{\partial p_1}{\partial y_k} \\ \frac{\partial p_2}{\partial y_1} & \frac{\partial p_2}{\partial y_2} & & \vdots \\ \vdots & & \ddots & \vdots \\ \frac{\partial p_k}{\partial y_1} & \cdots & \cdots & \frac{\partial p_k}{\partial y_k} \end{bmatrix}$$

From the previous derivatives the matrix will have the pattern below.

$$\begin{bmatrix} p_1(1-p_1) & -p_1 \cdot p_2 & \cdots & -p_1 \cdot p_k \\ -p_2 \cdot p_1 & p_2(1-p_2) & \cdots & -p_2 \cdot p_k \\ \vdots & \vdots & \ddots & \vdots \\ -p_k \cdot p_1 & -p_k \cdot p_2 & \cdots & p_k(1-p_k) \end{bmatrix}$$

Now that we found the derivative of softmax its time to form the derivative of CE.

$$L = -\sum_{l=1}^{k} y_l log(p_l)$$

By using the (2) and (3) relations we have:

$$\frac{\partial L}{\partial z_j} = \frac{\partial - \sum_{l=1}^{k} y_l log(p_l)}{\partial z_j}$$

WHERE:
y is the true-target values
p is the softmax values- the predictions
z is the output layer values before softmax (K units in output layer equal to the classes)

$$-\sum_{l=1}^{k} y_l \frac{\partial log(p_l)}{\partial z_j} =^*$$

*using the chain rule

$$-\sum_{l=1}^{k} y_l \frac{\partial log(p_l)}{\partial p_l} \cdot \frac{\partial p_l}{\partial z_j} = -\sum_{l=1}^{k} y_l \frac{1}{p_l} \cdot \frac{\partial p_l}{\partial z_j} =$$

because the $\sum_{i=1}^{k} y_i$ is the total sum of $y_1 + y_2 + \ldots + y_j + \ldots + y_k$we can detouch the $y_j$from the total sum, because the derivative of softmax-p is different for i=j and $i \neq j$.

$$-\sum_{l=1, l\neq j}^{k} y_l \frac{1}{p_l} \cdot \frac{\partial p_l}{\partial z_j} + y_j \frac{1}{p_j} \cdot \frac{\partial p_j}{\partial z_j} =$$

$$-\sum_{l=1, l\neq j}^{k} y_l \frac{1}{p_l} \cdot (-p_l p_j) + y_j \frac{1}{p_j} \cdot p_j(1-p_j) =$$

$$\sum_{l=1, l\neq j}^{k} y_l p_j - y_j(1-p_j) =$$

$$\sum_{l=1, l\neq j}^{k} y_l p_j - y_j + y_j p_j =$$

$$p_j \sum_{l=1, l\neq j}^{k} y_l - y_j + y_j p_j =$$

$$(\sum_{l=1, l\neq j}^{k} y_l + y_j) p_j - y_j$$

Where in parenthesis we reconstructed the sum that we had at the beggining .

$$p_j \sum_{l=1}^{k} y_l - y_j$$

Because of y which is an one hot encoded vector [0,0, ...1,..0] a vector of 0s with only one 1 inside. The sum of all the $y_i$that consists this vector will be 1.

This is why the $\sum_{l=1}^{k} y_l = 1$

Because of that the (4) takes this final form:

$$p_j \cdot 1 - y_j = p_j - y_j$$

Now we can calculate the gradient of Cross Entropy Loss function.

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial z_1} \\ \frac{\partial L}{\partial z_2} \\ \vdots \\ \frac{\partial L}{\partial z_k} \end{bmatrix} = \begin{bmatrix} p_1 - y_1 \\ p_2 - y_2 \\ \vdots \\ p_k - y_k \end{bmatrix}$$

Part 2

We want to calculate all the partial derivatives of MSE error with respect to all the different parameters.

y is the activation function $ReLU = max(0, a)$

a is described by this equation $a = x \cdot m + b$

Mean Squared Error function is $MSE = (y - y^*)^2$

y is our prediction and $y^*$is the target value

We will start taking the parameters from the end to begining (to use the chain rule later).

calculate the derivative of MSE with respect to y and y^*

$$\frac{\partial MSE(y, y^*)}{\partial y} = \frac{\partial (y - y^*)^2}{\partial y} = 2 \cdot (y - y^*)$$

$$\frac{\partial MSE(y, y^*)}{\partial y^*} = \frac{\partial (y - y^*)^2}{\partial y^*} = 2 \cdot (y^* - y)$$

Now calculate with respect to the others parameters

$$\frac{\partial MSE(y, y^*)}{\partial b}$$

First the derivatives of the ReLU function is:

$$ReLU(x) = \left\{ \begin{array}{c} 0, x \leq 0 \\ x, x > 0 \end{array} \right\}$$

$$\frac{\partial ReLU(x)}{\partial x} = \left\{ \begin{array}{c} 0, x \leq 0 \\ 1, x > 0 \end{array} \right\}$$

Having said that the previous derivative will become $\frac{\partial MSE(y,y^*)}{\partial b}$

$$\frac{\partial MSE(y, y^*)}{\partial b} = \frac{\partial MSE(y, y^*)}{\partial y} \cdot \frac{\partial y}{\partial b} = \left\{ \begin{array}{c} 0, a \leq 0 \\ 2 \cdot (y - y^*) \cdot 1, a > 0 \end{array} \right\}$$

Let's calculate the derivatives with respect to m and y.

$$\frac{\partial MSE(y, y^*)}{\partial m} = \frac{\partial MSE(y, y^*)}{\partial y} \cdot \frac{\partial y}{\partial m} = \left\{ \begin{array}{c} 0, a \leq 0 \\ 2 \cdot (y - y^*) \cdot x, a > 0 \end{array} \right\}$$

$$\frac{\partial MSE(y, y^*)}{\partial x} = \frac{\partial MSE(y, y^*)}{\partial y} \cdot \frac{\partial y}{\partial x} = \left\{ \begin{array}{c} 0, a \leq 0 \\ 2 \cdot (y - y^*) \cdot m, a > 0 \end{array} \right\}$$

Part 3

Tweets Cleance

Now Let's talk about our Neural Network. The first study I did was about the input that our model will take. This input will be a combination of the the tweets of our dataset and the pre-trained embending of Glove. First, we are going to cleance the given tweets in order to make it easier for our model to be trained and to find relation between the meaning of each tweet and its label. Here we are not going to vectorize the given words, after we clear the tweets from unwanted words/characters or punctuation we are going to match them with their corresponding Glove embending.

The first thing we do, after we take all the tweets, its to clean them from words that will not help us in total, and will give our model much noise. Words that are often repeated, links, emojis or punctuation are some examples that will not identify the real preference of someone when it comes to covid-vaxination topic. For example, if a tweet has many times the word **"the"**, this information does not actually help us to say whether this tweet belongs to a pro-vax, an anti-vax or a neutral person. Another example, would be that if a tweet has many **question-marks** or **dots**, this does not give us a clue for our predictions. The words we want to keep, our target-words, are semantic words, words with negative/positive meaning or words that establish something. Also, links to different websites could be used from all the different sides (pro, anti, neutral) to support each opinion. These are the reasons why I preffered to let out these characteristics in combination with some experiments, that denote that these characteristics do not offer something essential.

***Note: Even if in the first exercise the stopwords provided better results, in this exercise using a Neural Network, having stopwords inside our input dataset makes us lose around 1%-2% accuracy. This happens propably because the Neural Network being a more complex model than softmax regression may be more sensitive to the given data.**

After, the cleance of each tweet, its time to tokenize the tweets into words. In the first exercise we used Vectorizer. In this exercise we have the pre-trained embending. These are pre-trained vectors for each individual word, in a sence that if two words are semantically close they will have their vectors close as well. We can imagine it as a dictionary which has as keys the words and as values the corresponding embending. We want to tokenize the tweets to words because we want to convert them into those given vectors. Afterwards, given a tweet, we will combine all the vectors of each word using the **mean** function. In this way, each tweet will eventually have its own vector which describes it.

Having said the big picture, the tokenization of each word is very important, because we split the tweets into their words. After that each one tokenized word has to be matched with an embending. If there is no embending for the given word then the vector of this word becomes 0s. This is the reason why we want to tokenise the words as good as possible in order to have the best possible matching ratio between words and embendings (and the best possible formed vectore for each tweet). To accomplish this I used the **TreebankWordTokenizer**, which tokenize the words in a way which fits best in the words of Glove embenging.

So to Sum up, until now we follow these steps:

- Remove the english stopwords
- Remove the emojis
- Remove the website-links
- Remove the punctuation
- Tokenize the tweets

Word embendings Set Choice

Let's examine now the best Glove embengings to use. I tested all the different tweets embending. The bigger the dimenssions of the embengings the better the results. This is a logical outcome because the bigger the dimmension of the embengings the better the descriptions of the tweets. For example if we use a 50dim embendings the information is compressed in comparison with the 100dim or even the 200dim. It will take some more time when it comes to train the model, or load the embendings but in the end we take better results that can reach **4% more accuracy**. So the bigger the dimensions the better the results and the better the description ability of each embending.

Having said that there are 2 Glove sets with big accuracy. The **glove.300d** which contains words from Wikipedia+Gigaword and the **glove.twitter.27B.200d** which contains words from the twitter. I made a study behind the words in terms of ratio and unique words that found. It is logical that the more unique words a glove embending-set contains the more of the twitters' words will have a matching embengings. What I got was:

The 6billion glove word tokens set with 300dim:

	TYPE	WORDS WITH EMBENDINGS	WORDS WITHOUT EMBENDINGS
TRAIN	TOTAL WORDS	181256	18750
	UNIQUE WORDS	14837	10369

	TYPE	WORDS WITH EMBENDINGS	WORDS WITHOUT EMBENDINGS
VALIDATION	TOTAL WORDS	25743	2596
	UNIQUE WORDS	5640	1848

The 27billion twitter glove word tokens set with 200dim:

	TYPE	WORDS WITH EMBENDINGS	WORDS WITHOUT EMBENDINGS
TRAIN	TOTAL WORDS	182642	17364
	UNIQUE WORDS	15623	9583
VALIDATION	TYPE	WORDS WITH EMBENDINGS	WORDS WITHOUT EMBENDINGS
	TOTAL WORDS	25953	2386
	UNIQUE WORDS	5640	1673

The 42 billion glove word tokens set with 300dim:

	TYPE	WORDS WITH EMBENDINGS	WORDS WITHOUT EMBENDINGS
TRAIN	TOTAL WORDS	184402	15604
	UNIQUE WORDS	16493	8713
VALIDATION	TYPE	WORDS WITH EMBENDINGS	WORDS WITHOUT EMBENDINGS
	TOTAL WORDS	26145	2194
	UNIQUE WORDS	5767	1546

As we can see from the above tables the best results-matching comes from the **42B_glove_300d** word tokens. This is a logical outcome because having more tokens available for matching will eventually have more words connected to their embendings. The experiments that I took was in the **27B twitter 200d words set**, due to the fact that the word embendings are trained based on a twitter dataset. This will propably be more suitable set to the problem we have to solve. Also, I preferred this embendings set for the experiments to combine both big dimentionality and less time spent on each experiment.

Neural Network Model

After some experiments the Neural Network's base form I chose was this one below:

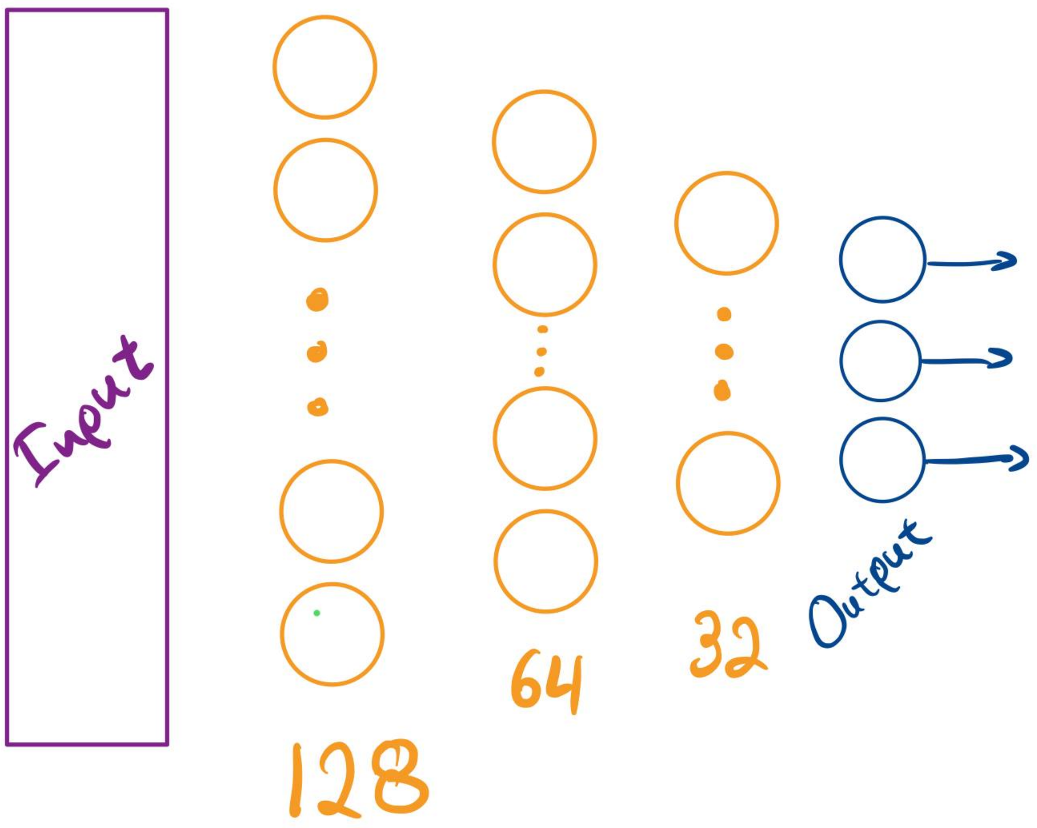


Figure 1: Neural Network Model

It consists of the **input layer**, which will be a vector with size equal to the dimensions of the embendings used (in our case it will be 200 because I used 200d twitter-set). The **output layer** is a layer which consists of 3 neurons. Each Neuron represents the three classes which each tweet can belong to (anti-vax, pro-vax and neutral). I tried a more complex NN with more neuron size (double neurons) in each layer but in those cases I faced a model which needed more time to train and more overfitting problem. This is why I used this one in the end. I used this model to train and run the experiments I wanted in order to choose activation-function, optimizer, batch-size and loss function.

Activation Function

The activation function is the function each neuron is going to use in order to form the final "constructed" function. **Using as default the Cross Entropy Loss function and the Adam optimizer the same lr=0.00001, batch size=32 and the previous Neural Network I tried all the different choices for activation function.** I got the results below:

FOR ALL THE EXPERIMENTS THE 50D TWITTER SET USED

I followed this tactic for at least the first experiments that I did, in order to save some time. Either way the low dimentionality will affect every activation function. In terms of comparisson all the activation functions will "lose information" and in the end they will be compared under the same circumstances.

Activation Function	Validation best Accuracy	Train best Accuracies
ReLU	**0.66**	**0.69**
Softmax	0.61	0.63
Elu	0.64	0.63
handshrinck	0.46	0.46
LeakyReLU	**0.678**	**0.69**
PReLU	0.664	0.68
ReLU6	**0.668**	**0.688**
RReLU	0.659	0.676
SeLU	**0.658**	**0.66**
CeLU	**0.648**	**0.646**
GeLU	0.65	0.66
Sigmoid	not very good	not very good
Silu	0.64	0.645
Mish	0.629	0.63

Table 1: Activation function results

From the above results the activation functions in **Bold** were the most possible one to be used in the final model.

Experiments

Starting the experiments.. The logic behind the experiments below is to take the activation functions with best results or potentials, test them and try to get the best possible results from them, having always standar model and standar loss function.

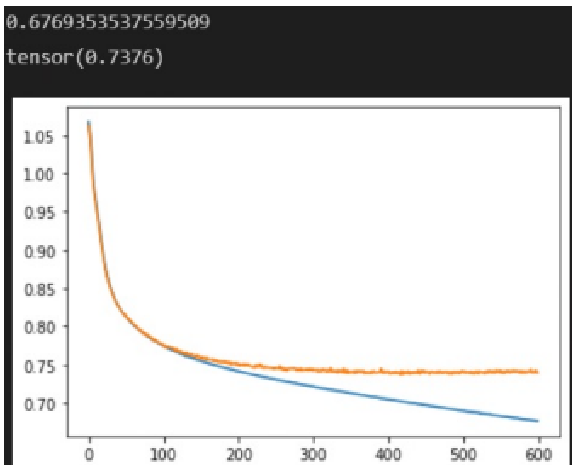
FOR ALL THE EXPERIMENTS THE 50D TWITTER SET USED

For each one of the experiments I kept the **Accuracy to Epochs chart and Loss to epochs chart,** as well as the **maximum values that the model achieved during the training,** in order to have a picture for each model.

BLUE LINE REFFERS TO TRAIN SET
ORANGE LINE REFFERS TO THE VALIDATION SET

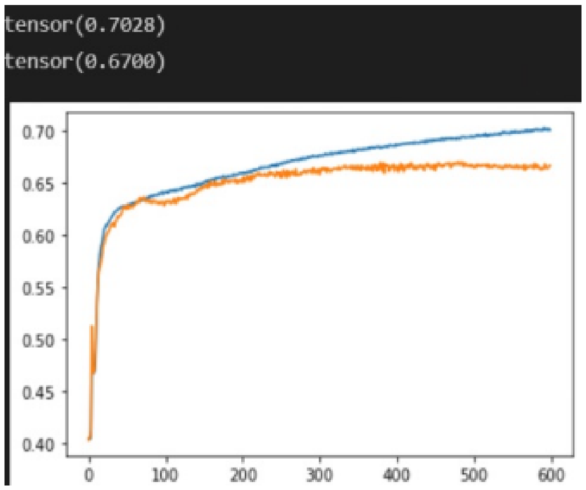
ReLU

First of all I tried the most basic version, having **lr=0.00001, Adam optimizer** and **600 epochs.**



```
0.6769353537559509
tensor(0.7376)
```

(a) Loss to Epochs



```
tensor(0.7028)
tensor(0.6700)
```

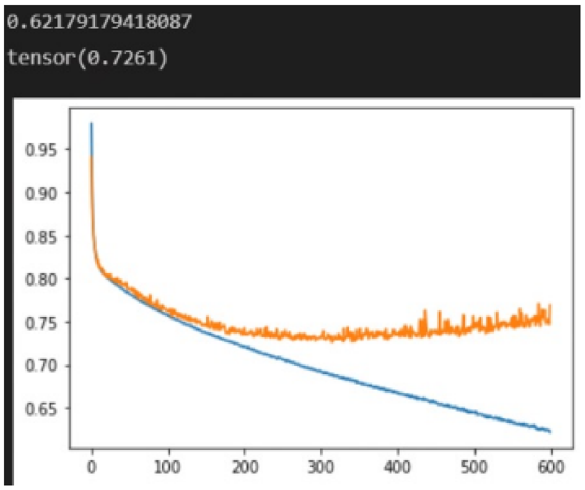
(b) Accuracy to Epochs

Figure 2: **ReLU** activation function

These results are good, so we will keep in mind that the ReLU activation function did well, even though there is a gap between the 2 lines which denotes that we are going to face overfitting. **Overfitting of a model is when the model itself is been trained very well for the data set given and cannot perform good generalizations.** There are ways to overcome this problem but for know we are investigating the activation function and the potentials of each one of them.

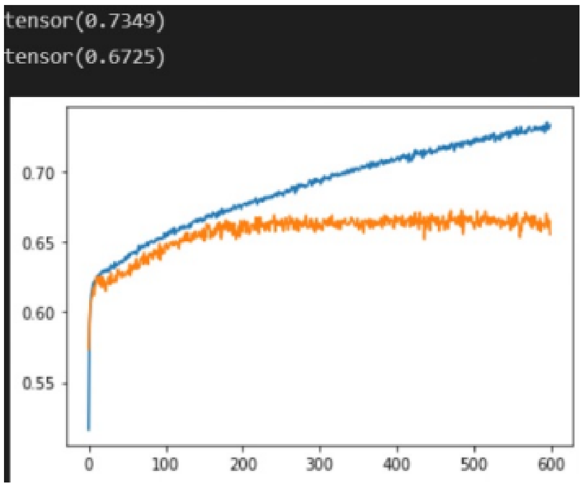
SeLU

Then we will see the **SeLU** activation function. We will use a bigger lr now in order to check if we can do better than ReLU. **lr=0.00005**



```
0.62179179418087
tensor(0.7261)
```

(a) Loss to Epochs



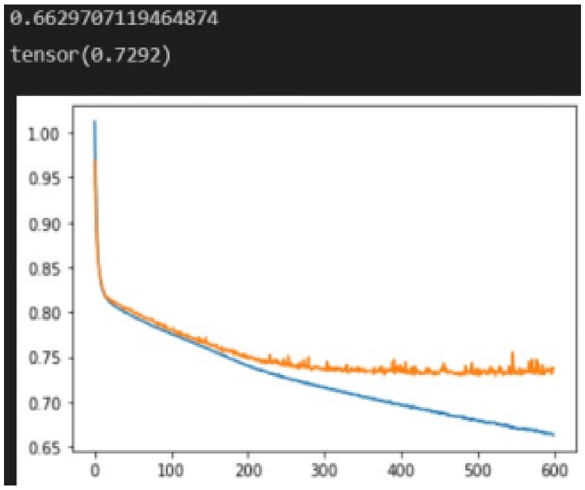
```
tensor(0.7349)
tensor(0.6725)
```

(b) Accuracy to Epochs

Figure 3: **SeLU** activation function

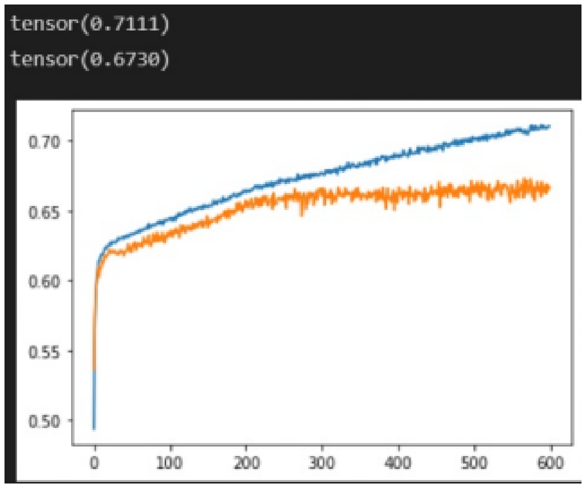
As we can see the big lr even if hit better results to our training set it has more obvious the effect of overfitting. We cannot hit the minimum value due to our large "steps". But in addition we want to hit better results, thats why we are risking by putting higher lr (The ReLU and SeLU function are pretty similar that's why I tried instantly to increase the lr).

Then we try to reduce the **lr=0.00003.** As a result we achieved to decrease the overfitting effect but again it is visible.



```
0.6629707119464874
tensor(0.7292)
```

(a) Loss to Epochs



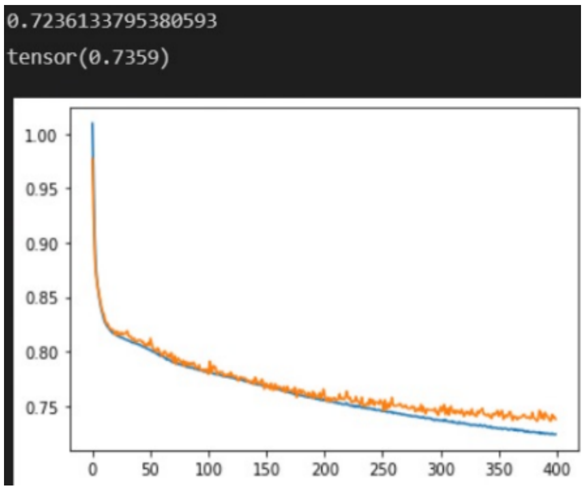
```
tensor(0.7111)
tensor(0.6730)
```

(b) Loss to Epochs

Figure 4: **SeLU** activation function

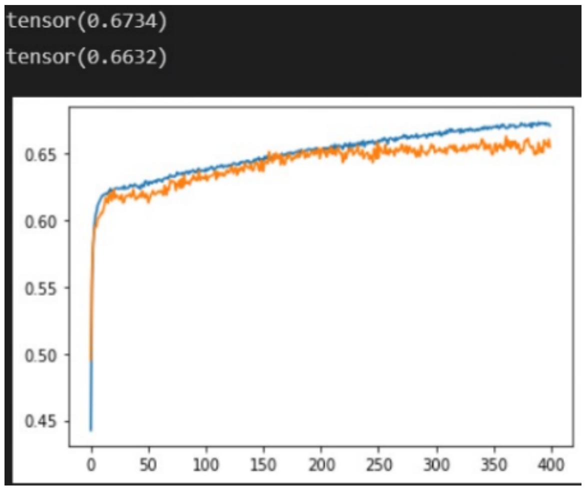
CeLU

After that its CeLU's turn. We again used a bigger lr in order to achieve better results from the ReLU (see Table 1). So by using **lr=0.00003** the activation function couldn't reach the results of the previous two.



```
0.7236133795380593
tensor(0.7359)
```

(a) Loss to Epochs

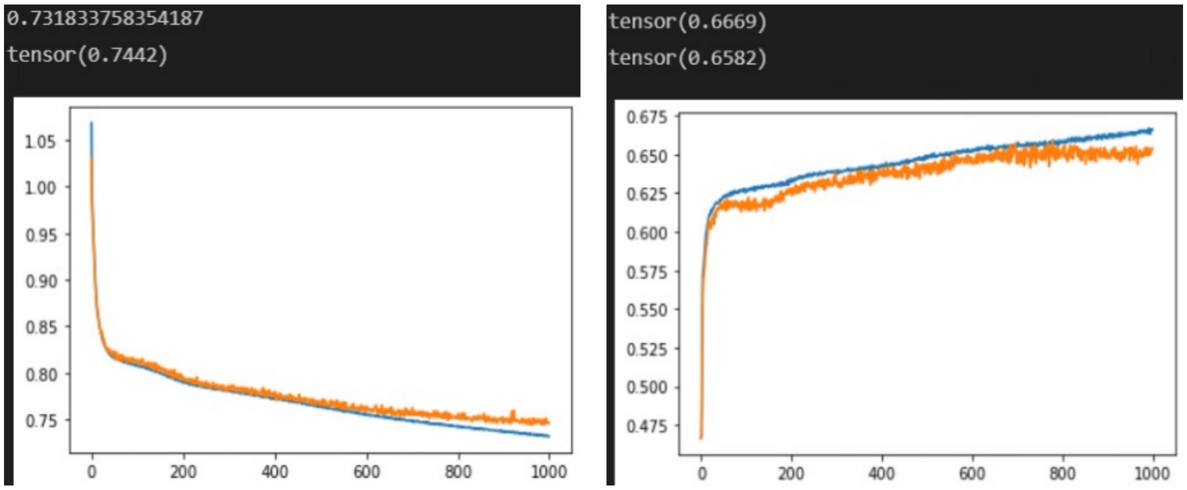


```
tensor(0.6734)
tensor(0.6632)
```

(b) Accuracy to Epochs

Figure 5: **CeLU** activation function

Unfortunately even if this activation function does better in terms of overfitting (blue and orange lines are close), when it comes to result, it cannot reach the previous even with **lr>=previous_lrs**. On the other hand, because the 2 lines are close, we could try more epochs and reduce a little more the lr in order to avoid overfitting. Using **lr=0.00002** we have:



```
0.731833758354187
tensor(0.7442)
```

```
tensor(0.6669)
tensor(0.6582)
```

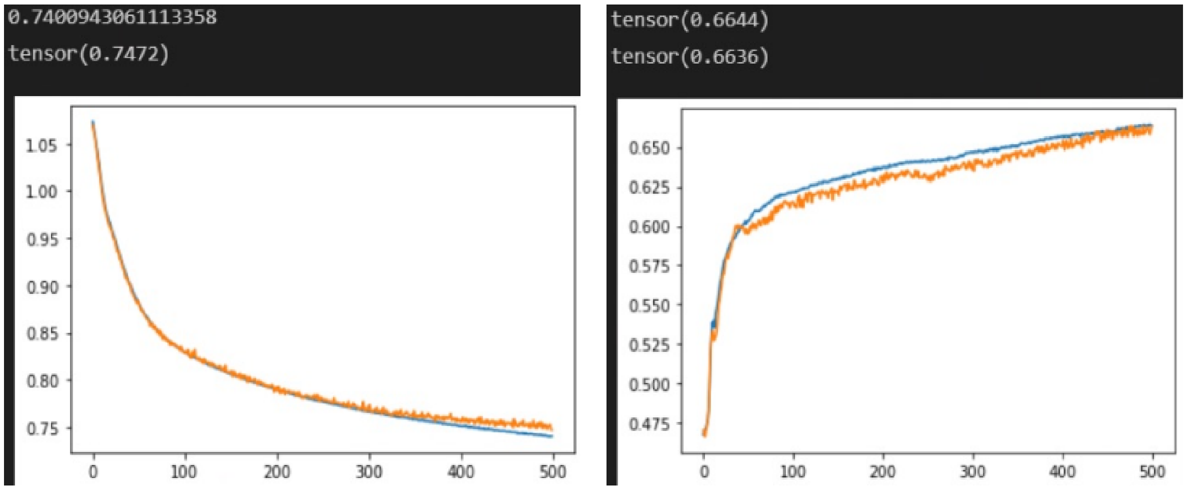
(a) Loss to Epochs (b) Accuracy to Epochs

Figure 6: **CeLU** activation function

From the results we can see that this model it's on his limits. It cannot provide better results. We can see that the gradient of 2 lines is really small, almost horizontal.

LeakyReLU

The next activation function runs with **lr=0.000003**. It needs a smaller lr from the previous ones. On the first run we got:



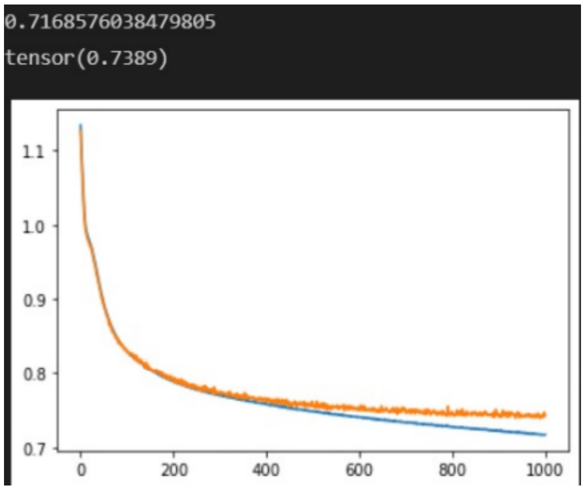
```
0.7400943061113358
tensor(0.7472)
```

```
tensor(0.6644)
tensor(0.6636)
```

(a) Loss to Epochs (b) Accuracy to Epochs

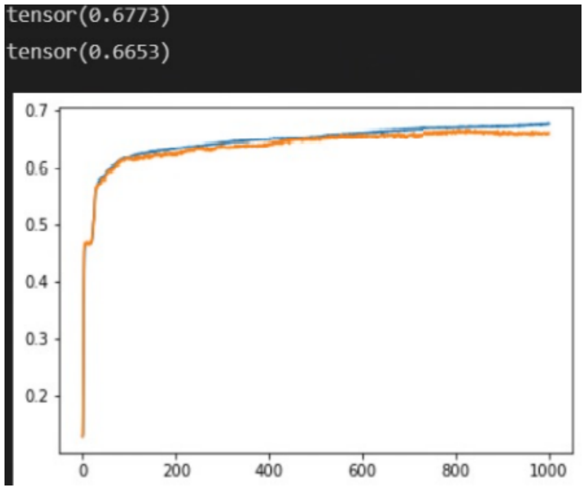
Figure 7: LeakyReLU activation function

The charts and the results are pretty good and promissing. This is why I rerun these results as it is, with more epochs.



```
0.7168576038479805
tensor(0.7389)
```

(a) Loss to Epochs



```
tensor(0.6773)
tensor(0.6653)
```

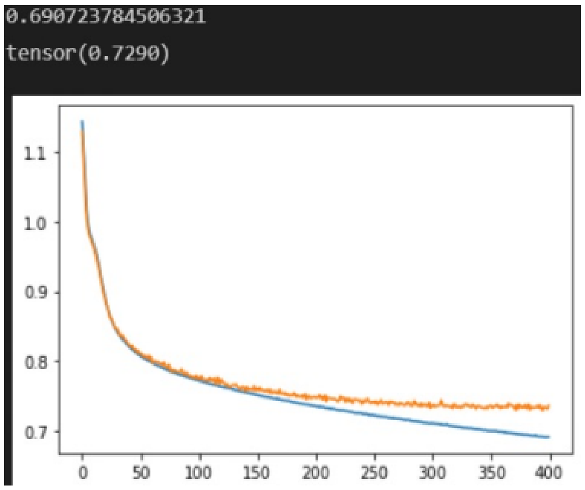
(b) Accuracy to Epochs

Figure 8: LeakyReLU activation function

these are good results and consistent.

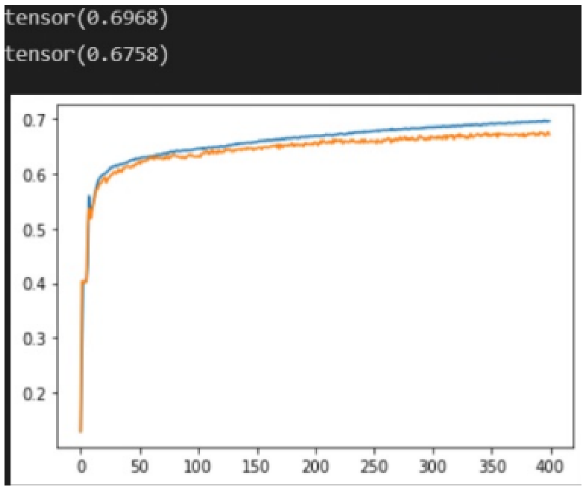
ReLU6

This is the final activation function we are going to exam from the Table1. First run with **lr=0.00001**.



```
0.690723784506321
tensor(0.7290)
```

(a) Loss to Epochs

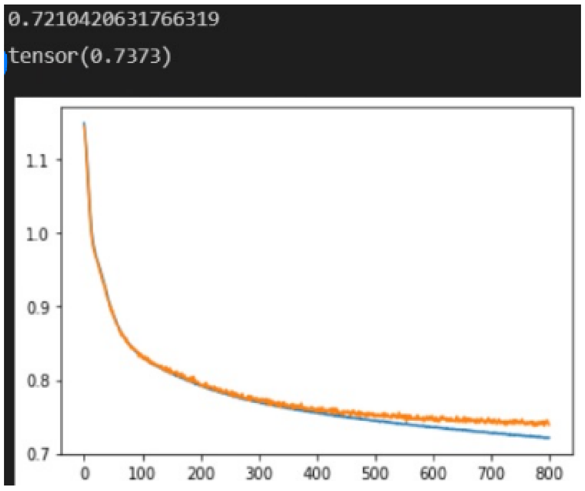


```
tensor(0.6968)
tensor(0.6758)
```

(b) Accuracy to Epochs

Figure 9: ReLU6 activation function

Very good results and also pretty good charts too. Let's investigate this activation function more by lowering the **lr=0.000003** and give more epochs to see his behaviour.


```
0.7210420631766319
tensor(0.7373)
```
(a) Loss to Epochs

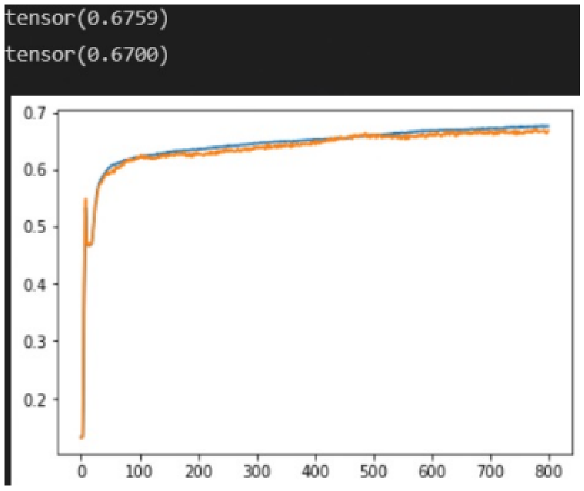

```
tensor(0.6759)
tensor(0.6700)
```
(b) Accuracy to Epochs

Figure 10: ReLU6 activation function

We can see here that ReLU6 gives as really consistent results with some good charts too. **The above experiments showed that the ReLU6 function is better than the others in terms of both consistency and results.**

Optimizer

The role of the optimizer in our model is how the weights and bias are going to be updated in each epochs. I followed the same logic for the optimizers as well. I tried each one and take their best scores (Under the same cicomstances). Then revisit the ones that had potentials.

Optimizer	Validation Accuracy	Train Accuracy
Adam	0.66	0.69
AdamW	equally good with Adam	
Adamax	0.64	0.65
NAdam	0.65	0.68
RAdam	0.65	~0.67
Rprop	0.61	~0.63
SGD	The most basic way for updating weights	Not good results even when momentum used

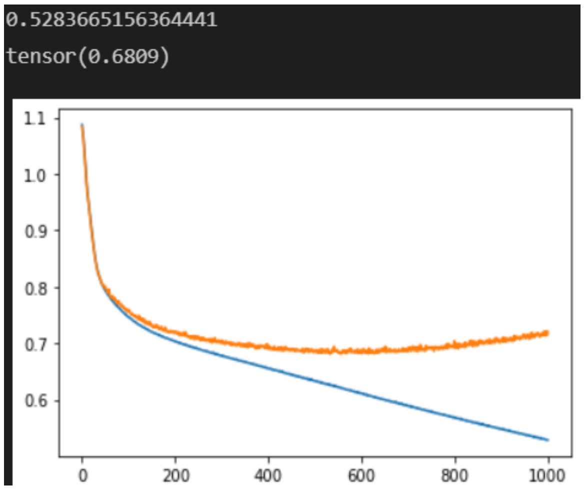
Table 2: Optimizers results

Adamax was the optimizer that had the more consistency and both validation and train lines are very close. And also provided good results. That's why I focused testing between **Adam, NAdam, AdamW and Adamax**.

FOR THE EXPERIMENTS BELOW I USED THE TWITTER 200D EMBENDINGS, in order to have better results.

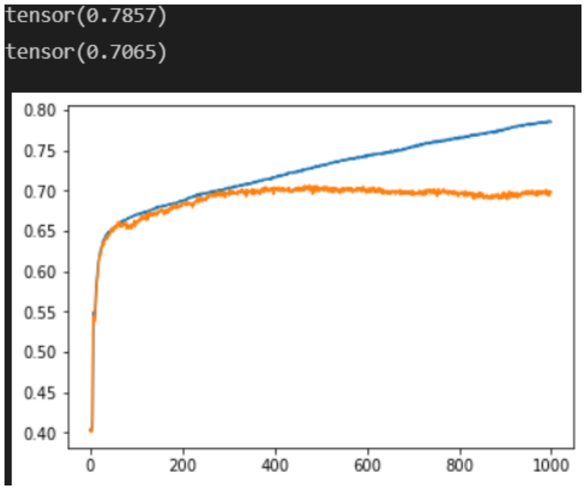
Adam

Adam is an optimizer which makes a really good results when it comes to the train set.



```
0.5283665156364441
tensor(0.6809)
```

(a) Loss to Epochs



```
tensor(0.7857)
tensor(0.7065)
```

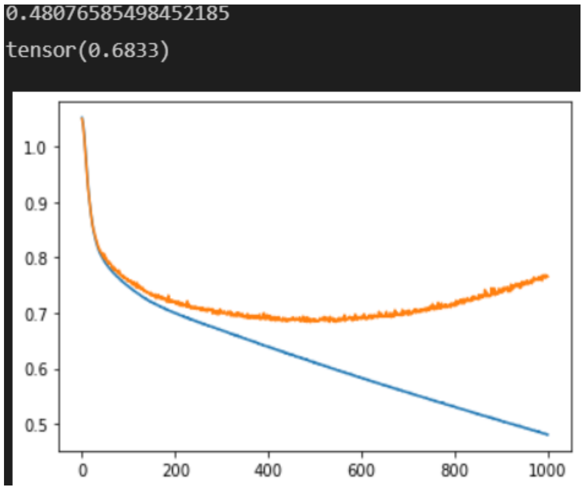
(b) Accuracy to Epochs

Figure 11: Adam optimizer

As we can se we have to face overfitting when it comes to Adam. Although we can see that in lower epochs ~500 before the 2 lines totally seperate we can achieve pretty good results.

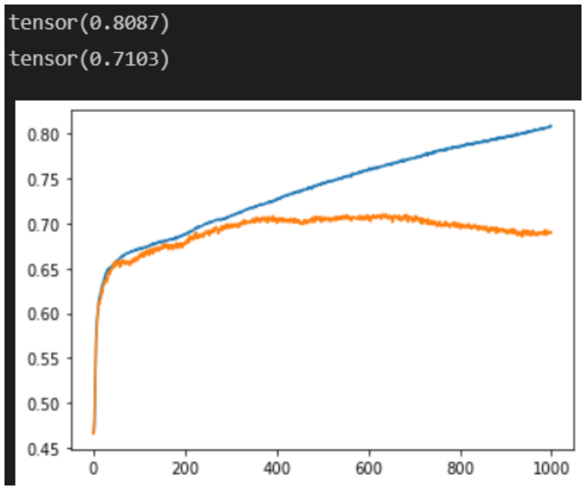
AdamW

AdamW Its better than the simple Adam. Because at least the accuracy of the validation set exceeds the 71% as we can see from the charts below:



```
0.48076585498452185
tensor(0.6833)
```

(a) Loss to Epochs

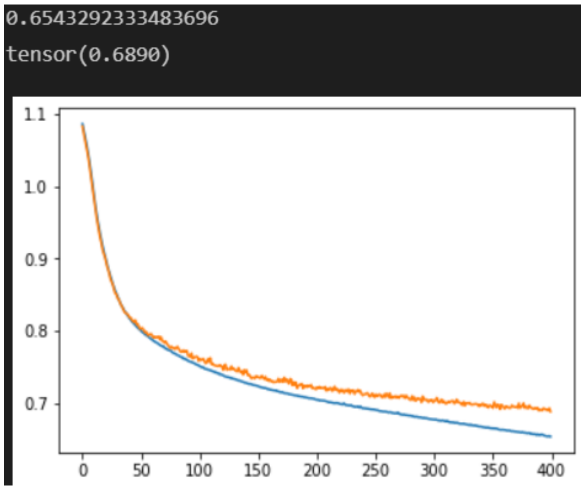


```
tensor(0.8087)
tensor(0.7103)
```

(b) Accuracy to Epochs

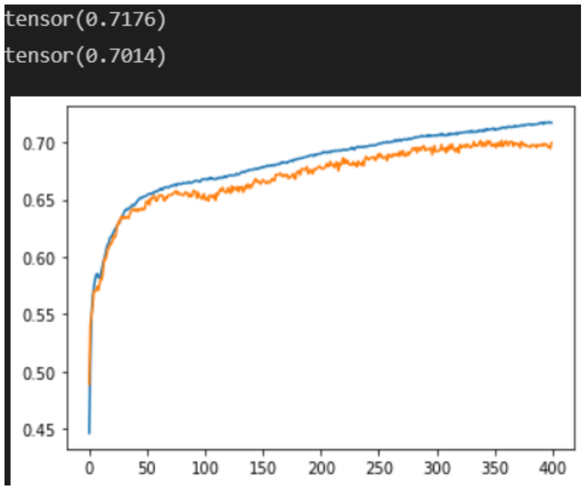
Figure 12: **AdamW** optimizer

Having more potentials than the Adam we try running it in smaller amount of epochs=400:



```
0.6543292333483696
tensor(0.6890)
```

(a) Loss to Epochs



```
tensor(0.7176)
tensor(0.7014)
```

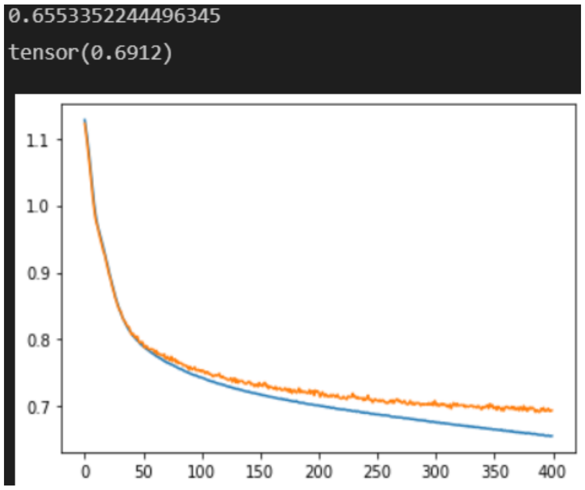
(b) Accuracy to Epochs

Figure 13: **AdamW** optimizer 400 epochs

That is a really good model!!!. Using lr=0.000004

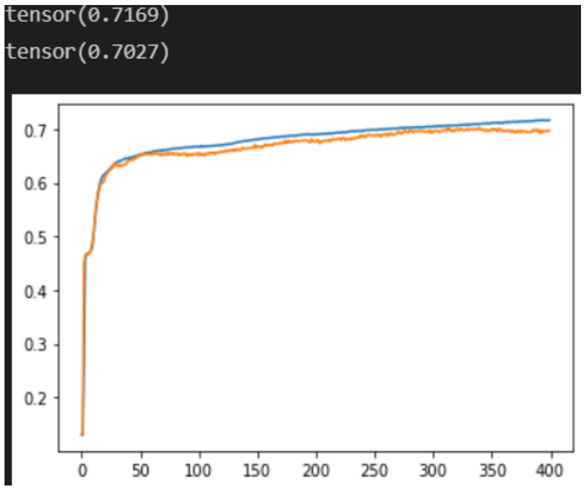
Keep the previous Lr but add 2 dropout layers (L1=0.4, L2=0.2) in order to help the lines to get closer.

***Dropout Layers are some layers that get between the first layers of our Neural Network and gives an error in the training process. This error helps our model to avoid overfitting and generalize easier. More specifically what they do is to randomly zeroes some of the elements of the input of each layer with probability p. But in exchange we are losing results in the train set.**



```
0.6553352244496345
tensor(0.6912)
```

(a) Loss to Epochs



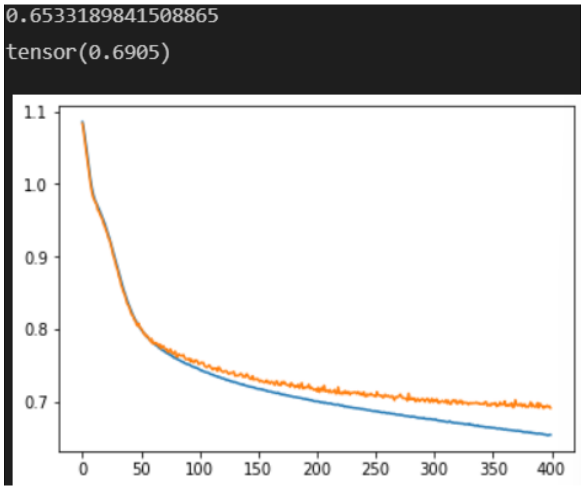
```
tensor(0.7169)
tensor(0.7027)
```

(b) Accuracy to Epochs

Figure 14: **AdamW** optimizer 400 epochs/with Dropout layers

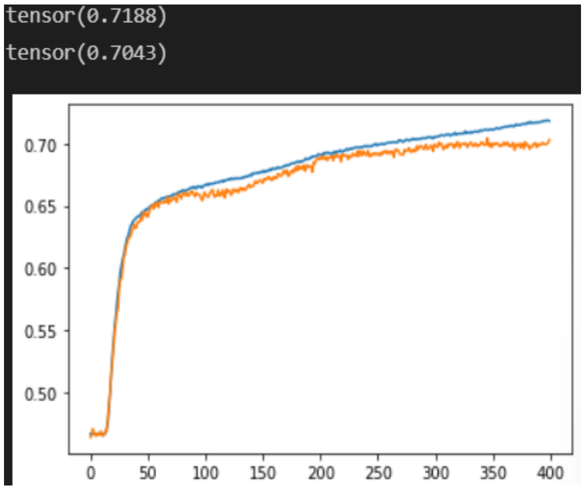
Compare with the previous results we can see that the error lines are closer and start to seperate in later epochs. In addition, the maximum error is higher that the previous example.

If we increase even more the Dropout (L1=0.5, L2=0.5) rate we get:



```
0.6533189841508865
tensor(0.6905)
```

(a) Loss to Epochs



```
tensor(0.7188)
tensor(0.7043)
```

(b) Accuracy to Epochs

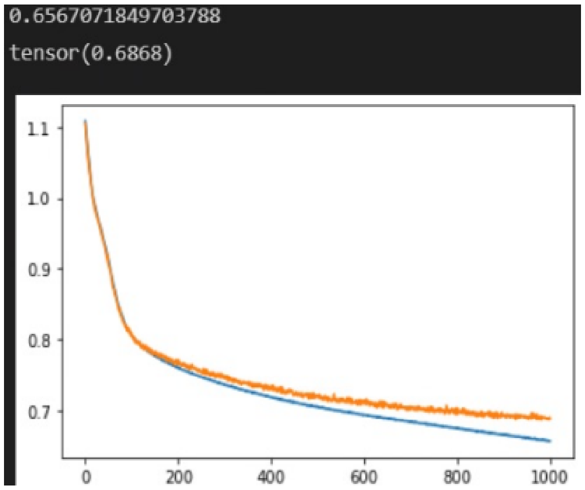
Figure 15: **AdamW** optimizer 400 epochs/ Dropout layers=0.5

The above effects of the dropout layers are more obvious.

Adamax

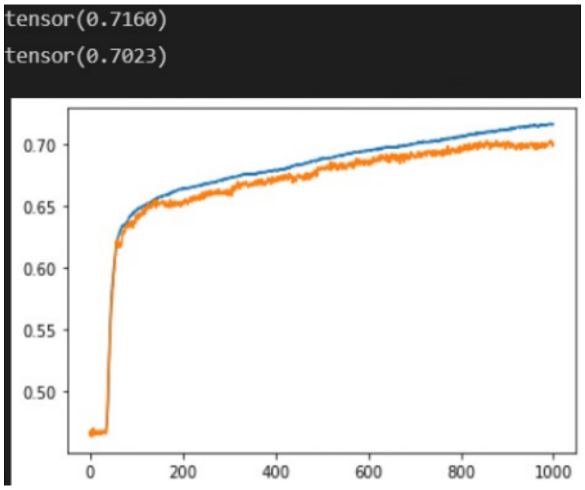
Next optimizer to be checked is **Adamax**. Adamax is an optimizer which needs more epoches to be trained than AdamW. But is more consistent when it comes to overfitting.

Adamax Experiments:



```
0.6567071849703788
tensor(0.6868)
```

(a) Loss to Epochs

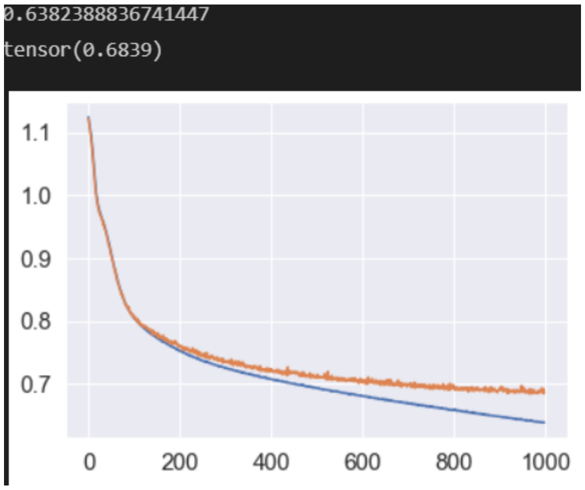


```
tensor(0.7160)
tensor(0.7023)
```

(b) Accuracy to Epochs

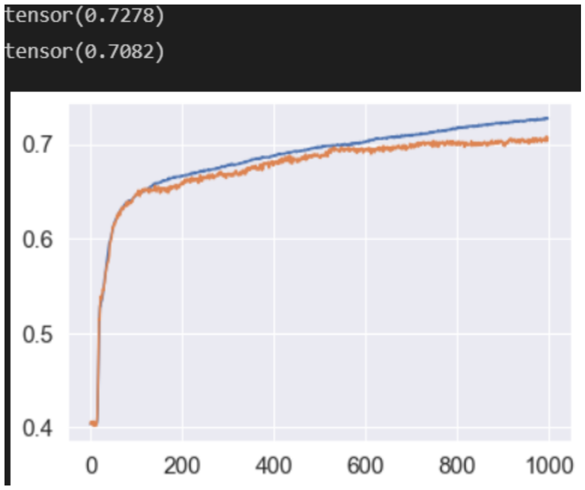
Figure 16: **Adamax** optimizer /with small Dropouts

We needed a lot more epochs to have a good results even with **lr=0.000005**.
Having a little bigger dropouts layers (L1=0.4, L2=0.2) and less epochs, we get these results:



```
0.6382388836741447
tensor(0.6839)
```

(a) Loss to Epochs

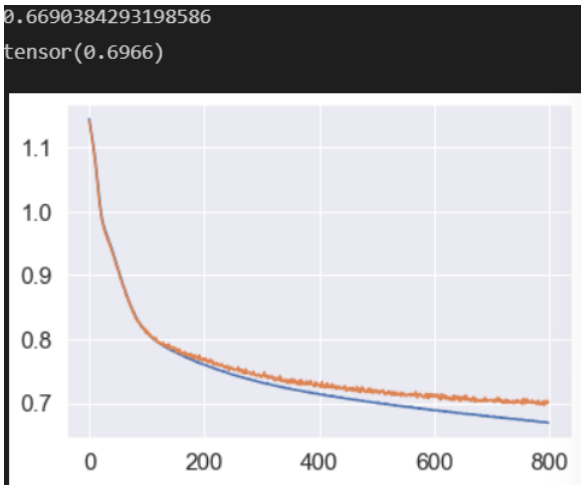


```
tensor(0.7278)
tensor(0.7082)
```

(b) Accuracy to Epochs

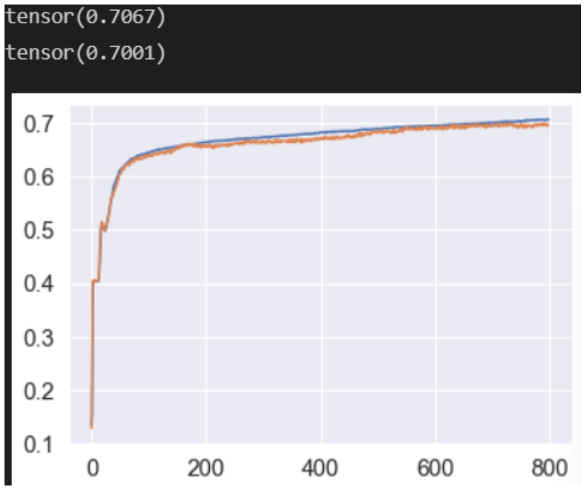
Figure 17: **Adamax** optimizer /with bigger Dropouts

Now let's reduce the epochs and keep the counter at 800 (about the point in the previous chart where the 2 lines go even further).



```
0.6690384293198586
tensor(0.6966)
```

(a) Loss to Epochs



```
tensor(0.7067)
tensor(0.7001)
```

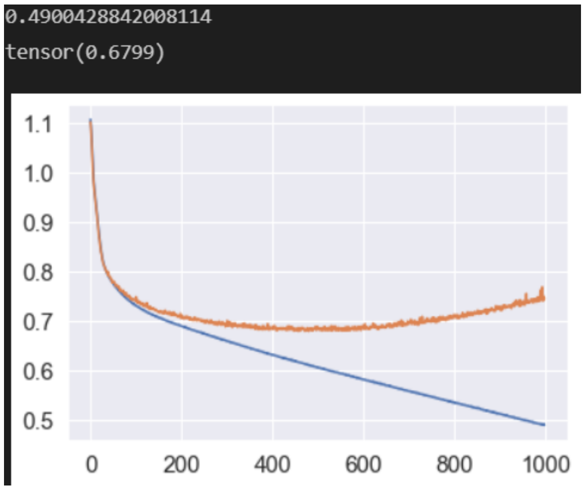
(b) Accuracy to Epochs

Figure 18: **Adamax** optimizer /800 epochs

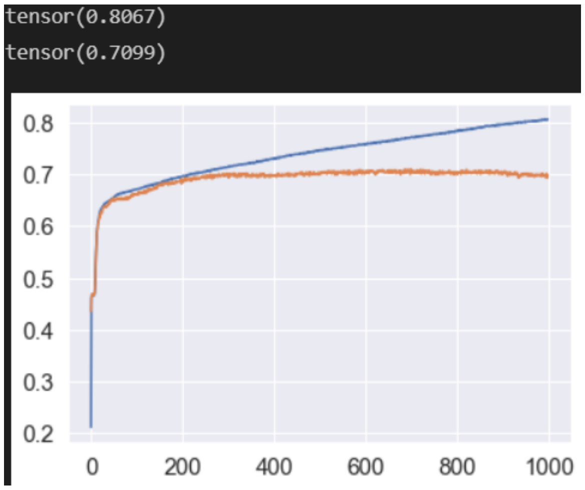
Very good results!! Very consistent, the train and the validation lines are very close.

NAdam

Last one to be tested is **NAdam** optimizer. It's like Adam optimizer, really good results and training for the train set and not so good when it comes the validation set (**lr=0.000005, Dropout<0.4,0.2>, epochs=1000**).


```
0.4900428842008114
tensor(0.6799)
```

(a) Loss to Epochs


```
tensor(0.8067)
tensor(0.7099)
```

(b) Accuracy to Epochs

Figure 19: **NAdam** optimizer

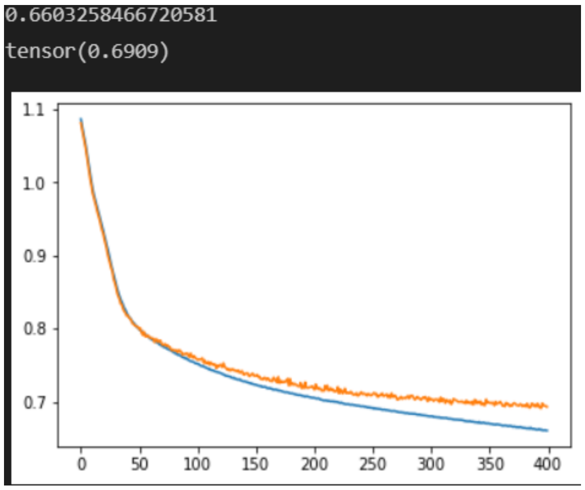
The overfitting effect is pretty clear here. Even if we make the model more consistent it will be difficult to decrease further the Loss of validation set.

Conclusion

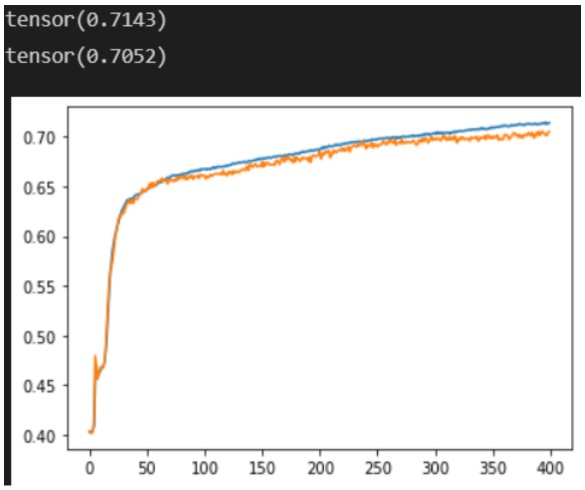
From the optimizer above the best choices would be **AdamW** and **Adamax**. The Adamax is very consistent algorithm. The reason why AdamW does better than the other version of Adam is because it has non-zero the weight_decay parameter. This parameter **apply the L2 penalty to the model trained**. The L2 penalty is an error that we want to affect our model in a good way. In simple words this error stops the model from overfitting. In this way the final model will have the ability to generalize and have good results on the validation set. (Actually AdamW is Adam with the weight_decay=0.001).

The optimizer that I will choose will be AdamW. It gives us very good and consistent results in a smaller number of epochs compaired with Adamax.

To prove my previous point I run the model using this weight decay on NAdam optimizer and the results were very close to AdamW.


```
0.6603258466720581
tensor(0.6909)
```

(a) Loss to Epochs


```
tensor(0.7143)
tensor(0.7052)
```

(b) Accuracy to Epochs

Figure 20: **NAdam** optimizer with weight_decay=0.001

All in all the optimizer that I will use after all this experiments, is AdamW.

Loss Function

The loss function is the function the “error” which we are trying to decrease in each step of training (error). The Loss function that I use, after research, is the **Cross Entropy** Loss Function. This function is the main Loss function which is responsible for multi-class classification problems. The step this function follows are: First it takes the results of the Neural Network. The results will have the form of a vector size of 3. This happens because the output layer of our Neural Network has 3 Neurons-Units. This model is architectured this way due to the fact that we have 3 different classes (neutral, pro-vax, anti-vax) where our inputs can belong to. This loss function what it does is to take all of those 3-sized vectors and pass them through a softmax function (**LogSoftmax**). This is done to convert the 3 values into 3 propabilities. **This is the reason why I do not use LogSoftmax into the last layer of NN**, **because the Cross Entropy includes it in its process**. Afterwards these propabilities are passed to the negative log likelihood loss (**NLLLoss**) function. The target that this loss expects should be a class index in the range [0, C-1].

“Obtaining log-probabilities in a neural network is easily achieved by adding a LogSoftmax layer in the last layer of your network. You may use CrossEntropyLoss instead, if you prefer not to add an extra layer.”

I tried to use other methods of loss function converting the Neural Network into a single-Neuron-output, but there is no reason extend into this path (The overall results were not good). This is why I am using this Loss Function. Into the second model I am using the more extended form, but in general this is the loss function I stuck with.

Neural Network Model Architecture

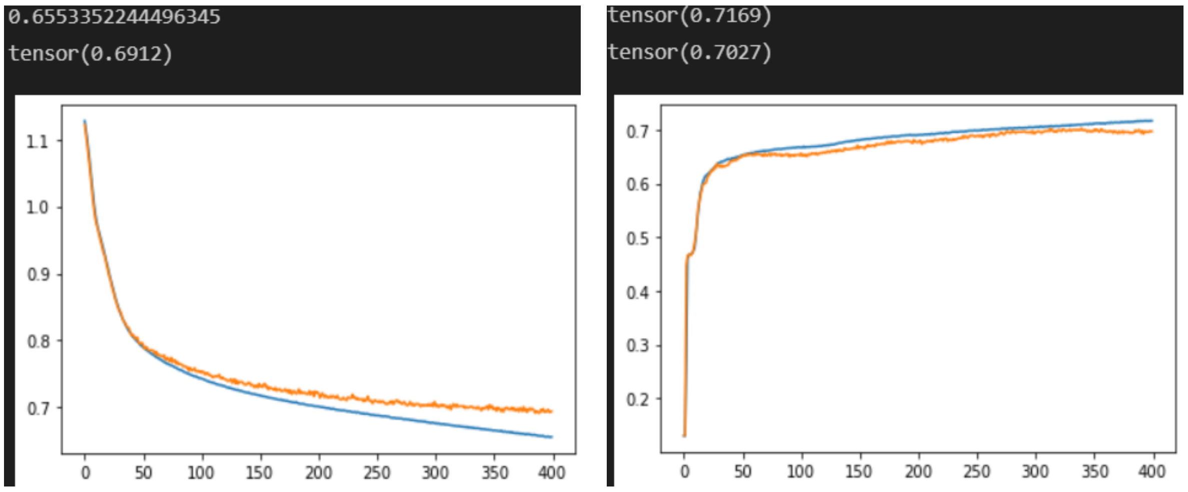
128-64-32

The main model that I have conclude from the experiments has 3 hidden layers:

128-Dropout(0.4)-64-Dropout(0.2)-32 neurons in each layer. As loss function I am using the Cross Entropy Loss, as optimizer the AdamW, learning rate=0.000004, epochs=400 and as activation the ReLU6.

Validation Scores:

- Accuracy: 0.6998247151621385
- F1_score: 0.6079992517911358
- Precision: 0.6562109645859671
- Recall: 0.597102030460979



```
0.6553352244496345
tensor(0.6912)
```

```
tensor(0.7169)
tensor(0.7027)
```

(a) Loss to Epochs

(b) Accuracy to Epochs

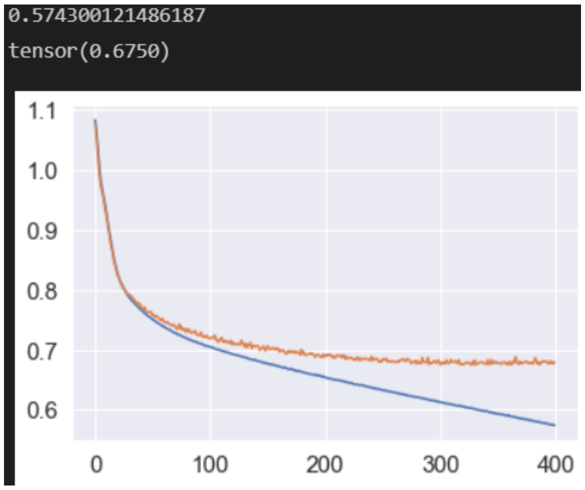
Figure 21: **128-64-32 version I**

256-128-64

The same situation with above with the difference that I doubled the Neurons in each layer (256-Dropout(0.4)-128-Dropout(0.2)-64).

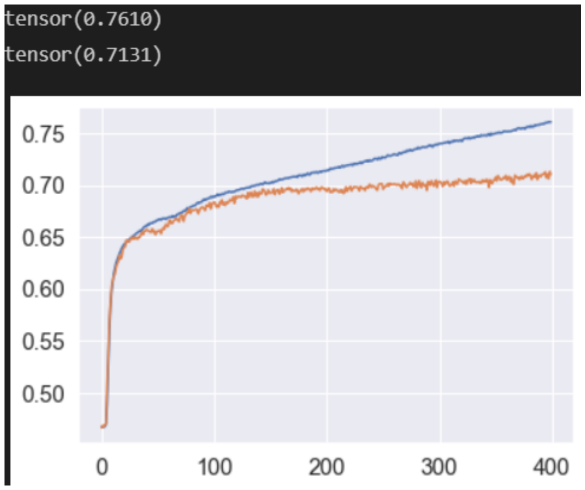
Validation Scores:

- Accuracy: 0.71034180543383
- F1_score: 0.6375037020302047
- Precision: 0.6578583674289965
- Recall: 0.627441915237254



```
0.574300121486187
tensor(0.6750)
```

(a) Loss to Epochs



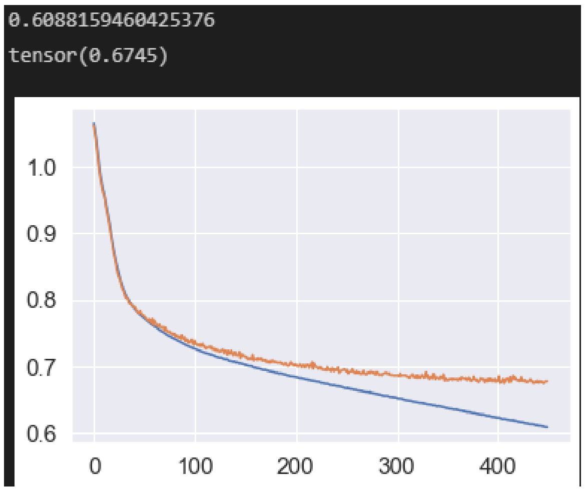
```
tensor(0.7610)
tensor(0.7131)
```

(b) Accuracy to Epochs

Figure 22: **256-128-64 version I**

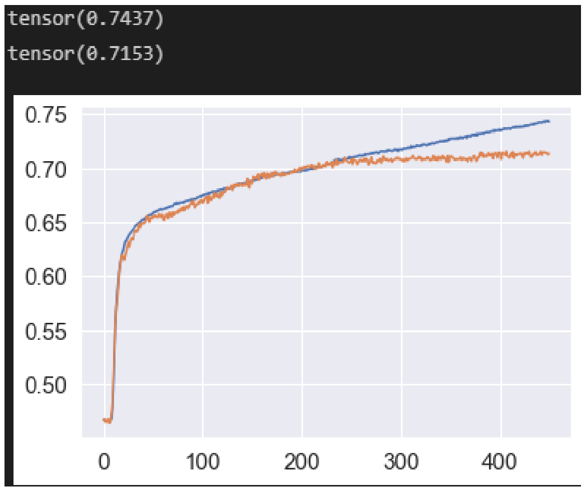
As we can see from the above results the Model that has more Neurons is more complex overall. This means that will have higher variance, because the model being more complex, it will represent more accurately the data set. But it also gives better results as a more complex model.

Trying some experiments to improve this model, I concluded to the model below. **Lr=0.000003, weight_decay=0.5, epochs=450, AdamW, (256-Dropout(0.5)-128-Dropout(0.5)-64)**



```
0.6088159460425376
tensor(0.6745)
```

(a) Loss to Epochs



```
tensor(0.7437)
tensor(0.7153)
```

(b) Accuracy to Epochs

Figure 23: **256-128-64 version II**

- Accuracy: 0.7134092900964066
- F1_score: 0.6304972719812193
- Precision: 0.6620496273274051
- Recall: 0.619227647384045

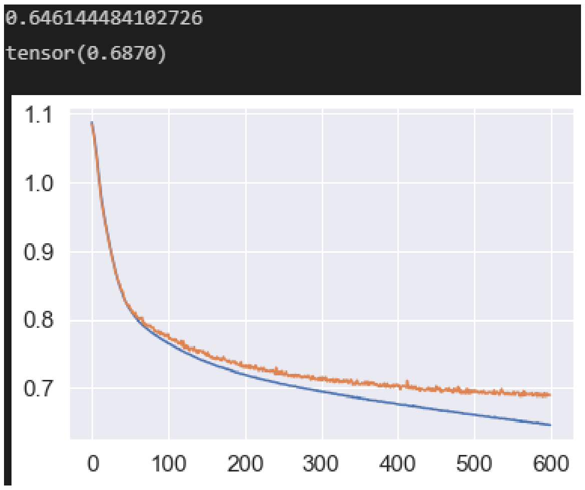
We could have interrupt the epochs earlier, around the 400 epoch, where the values are closer than later epoch 350.

Epoch 350:
Loss = 0.63762
Accuracy = 0.72725
VAL_Loss = 0.68043
VAL_Accuracy = 0.71085

Epoch 351:
Loss = 0.63701
Accuracy = 0.72737
VAL_Loss = 0.67913
VAL_Accuracy = 0.71094

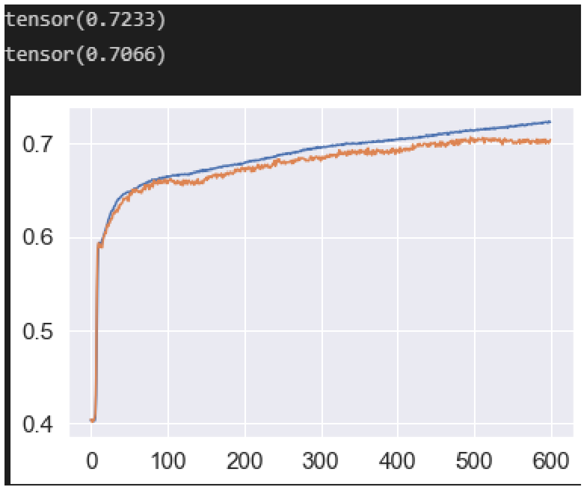
As a more complex model by increasing the L2 penalty, we achieved (taking advantage the bigger training results), some good results on the validation set, still having close error and accuracy results. Of course the train will take some more time.

Quick_check: Trying the same parameters with the 128-64-32 model will have results similar to the previous model but with highers Loss-error and smaller accuracy.



```
0.646144484102726
tensor(0.6870)
```

(a) Loss to Epochs



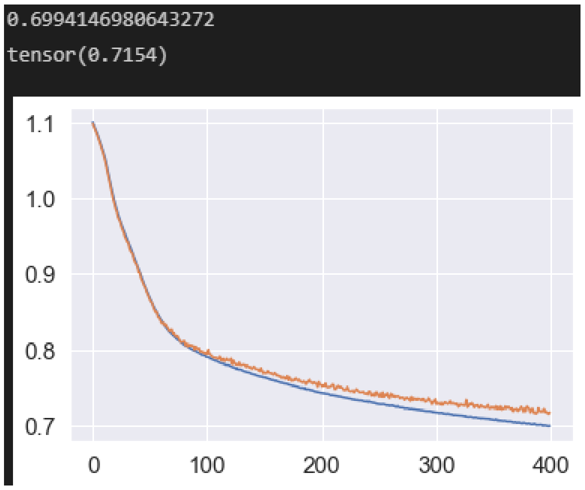
```
tensor(0.7233)
tensor(0.7066)
```

(b) Accuracy to Epochs

Figure 24: **128-64-32 version II**

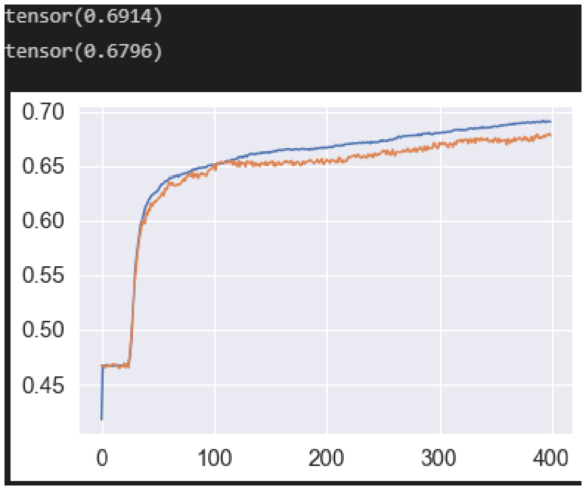
64-32-16

Last but not least let's see what happens when we have less Neurons (or less layers), testing a **64-32-16** NN. Try the model using the parameters set to **128-64-32 version I**.



```
0.6994146980643272
tensor(0.7154)
```

(a) Loss to Epochs



```
tensor(0.6914)
tensor(0.6796)
```

(b) Accuracy to Epochs

Figure 25: **64-32-16 version I**

These results were expected. In the model with fewer Neurons we have less complexity which means less variance (the 2 lines are closer) and higher bias (worse overall results).

Conclusion

In my opinion the best model to use is the 256-128-64 version II. This is because It hits good results when it comes to the validation set and at the same time it has not the effect of overfitting. I mean it's around 3%, and we compare a standar validation set. I thing that it worths the risk to pick this one as my final model.

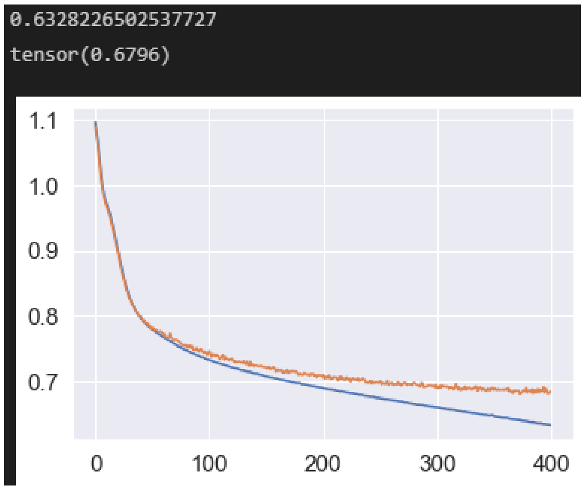
****Note: I chose to keep the layers on the same number because the effects of reducing/increasing the layers are the same with the reduction/increase of the Units in each layer. In both paths we are making our model less complex but in a different scale (The inpact is bigger when we change the layer architecture than the number of Neurons in a layer (of course there are always exceptions)).**

The 2 models that I choose to use are the **256-128-64 version II** and the **128-Dropout(0.4)-64-Dropout(0.2)-32** using different optimizer and activation function(**Adamax, LeakyReLU, epochs=800**), and try getting as close results as in my first model.

RECAP

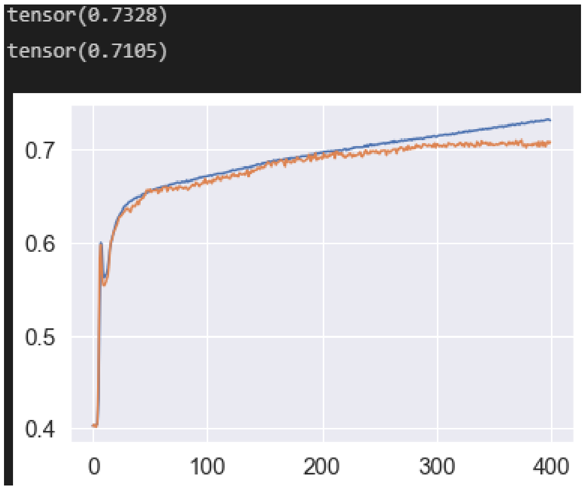
MODEL 1

The first model I choose has 3 hidden layers and 2 dropout layers. The Architecture of the model is **256-Dropout(0.4)-128-Dropout(0.2)-64**. We use **Cross Entropy** as the loss function, **AdamW** as optimizer, **lr=0.000003** and **ReLU6** as activation function. Final results:



```
0.6328226502537727
tensor(0.6796)
```

(a) Loss to Epochs



```
tensor(0.7328)
tensor(0.7105)
```

(b) Accuracy to Epochs

Figure 26: **256-128-64version II**

Validation Scores:

- Accuracy: 0.70683610867659955
- F1_score: 0.6149695409037227
- Precision: 0.6564953354172479
- Recall: 0.6042556480414901

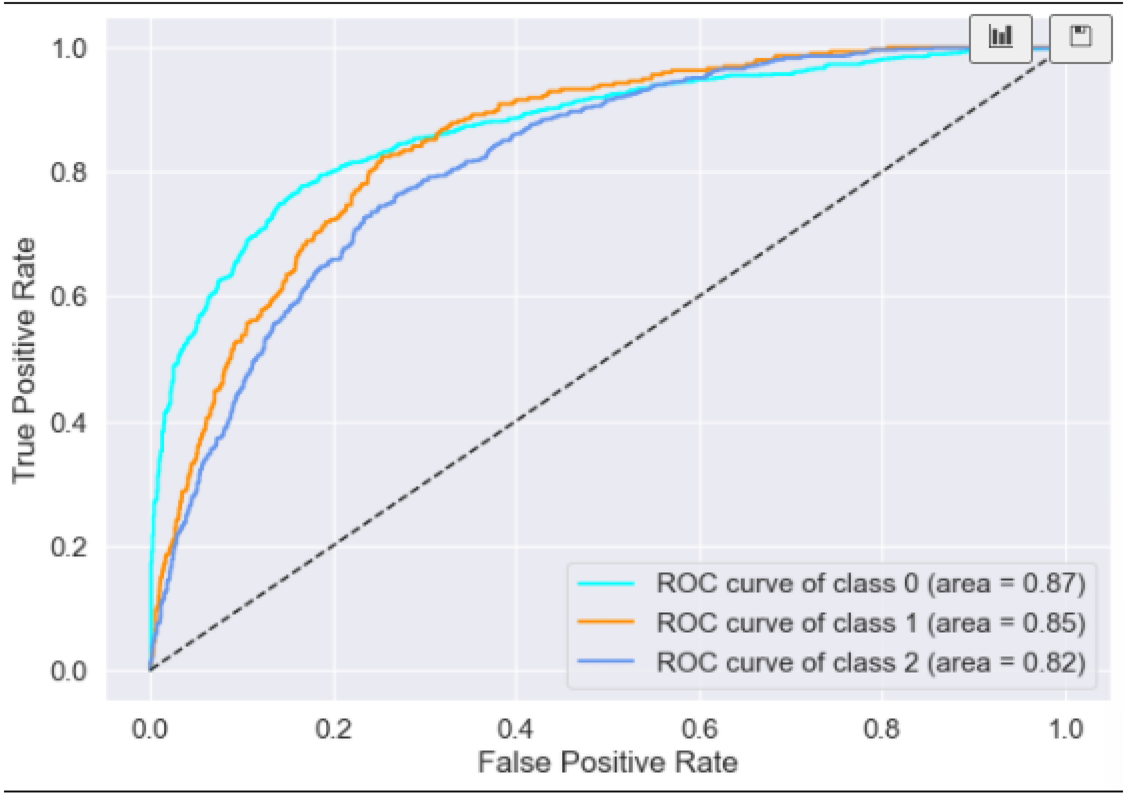


Figure 27: ROC CURVE

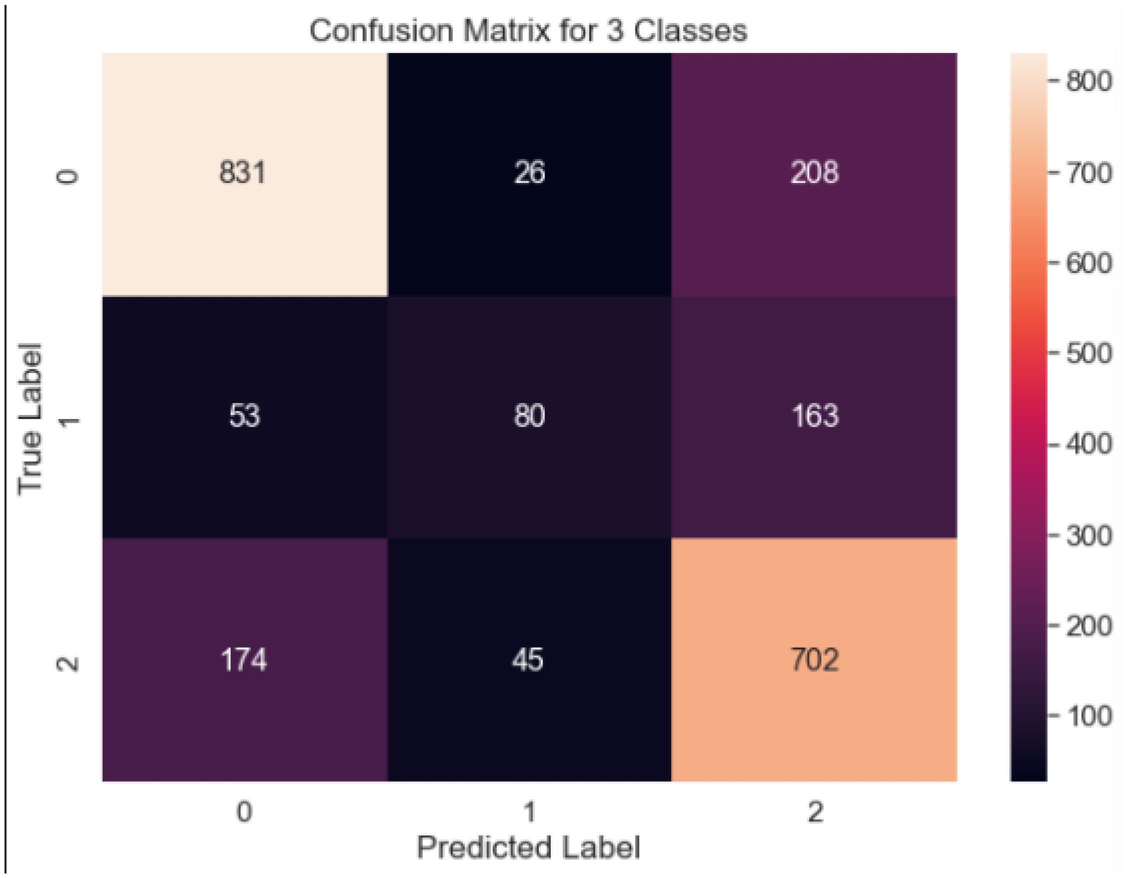
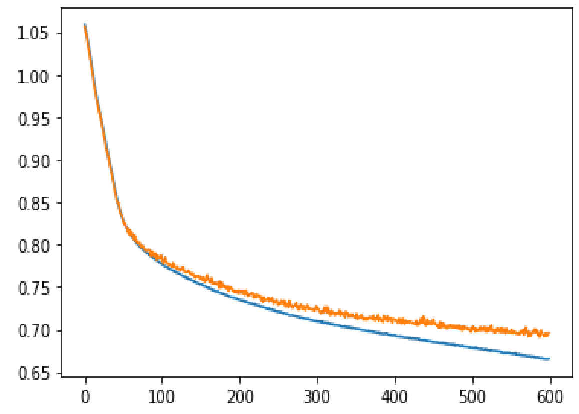


Figure 28: Confussion Matrix

MODEL 2

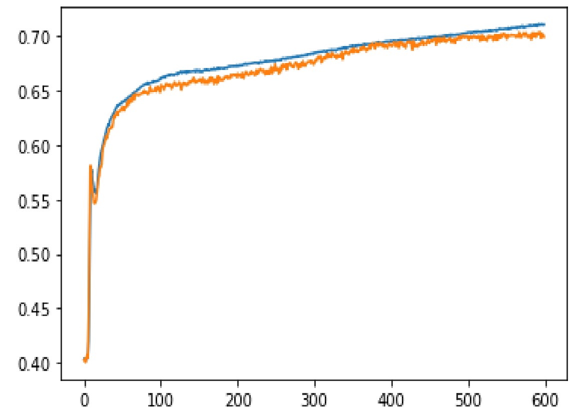
The second model follows this architecture: **128-Dropout(0.4)-64-Dropout(0.2)-32.** We use **Cross Entropy** as the loss function, **Adamax** as optimizer, **lr=0.000007** and **LeakyReLU** as activation function. Final results:

```
0.6653652125000954
tensor(0.6915)
```



(a) Loss to Epochs

```
tensor(0.7106)
tensor(0.7036)
```



(b) Accuracy to Epochs

Figure 29: **128-64-32**

- Accuracy: 0.6971954425942156

- F1_score: 0.6016250782477625
- Precision: 0.6431916378650656
- Recall: 0.5922953209666977

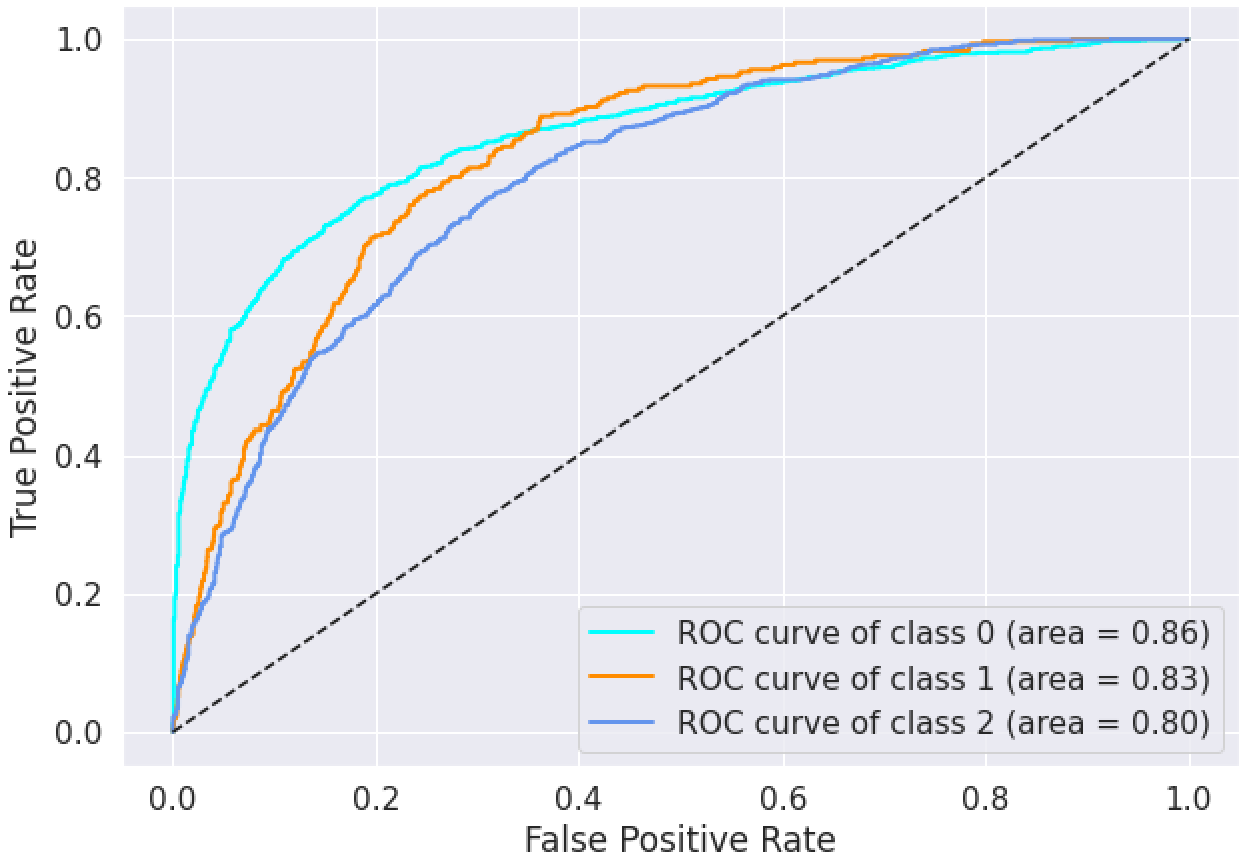


Figure 30: ROC CURVE

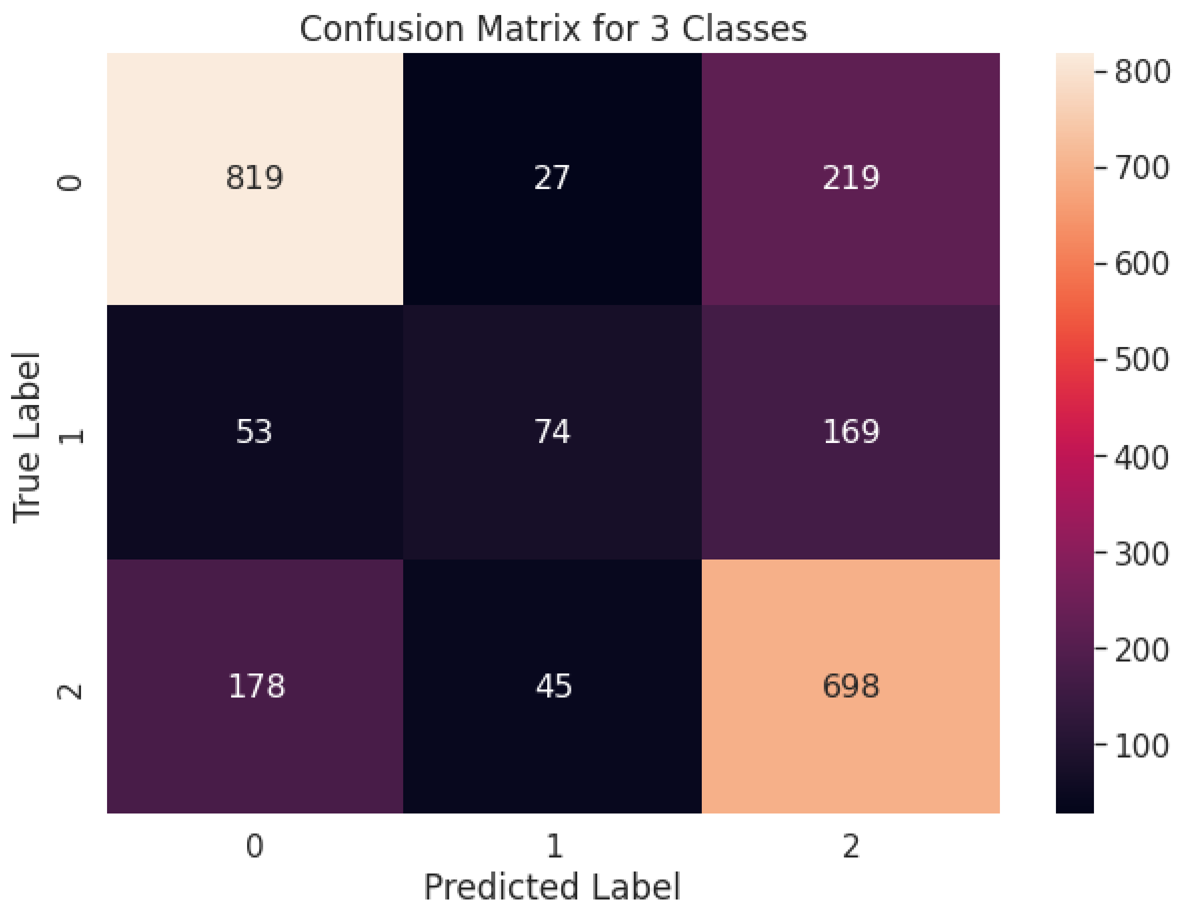


Figure 31: Confussion Matrix

The second model is provided in order to have an alternative with still some good results. It takes more epochs to be trained but as we saw from above Adamax is more consistent and achieve good F1 scores.

AUC-ROC

AUC - ROC curve is a performance measurement for the classification problems at various threshold settings. ROC is a probability curve and AUC (Area Under Curve) represents the degree or measure of separability. **It tells how much the model is capable of distinguishing between classes**. Higher the AUC, the better the model is at predicting the classes of each element. The ROC curve is plotted with **TRUE POSITIVE RATE** ($TPR = \frac{TP}{P} = \frac{TP}{TP+FN} = 1 - FNR$) against the **FALSE POSITIVE RATE** ($1 - SPECIFICITY = 1 - TNR$) where TPR is on the y-axis and FPR is on the x-axis. **TRUE NEGATIVE RATE** is described by this equation $TNR = \frac{TN}{N} = \frac{TN}{TN+FP}$.

An excellent model has AUC near to the 1 which means it has a good measure of separability. A poor model has an AUC near 0 which means it has the worst measure of separability. It is predicting 0s as 1s and 1s as 0s. And when AUC is 0.5, it means the model is making random guesses. The diagonal divides the ROC space. Points above the diagonal represent good classification results (better than random); points below the line represent bad results (worse than random).

Having said that let's examine the ROC we took from out 2 models.

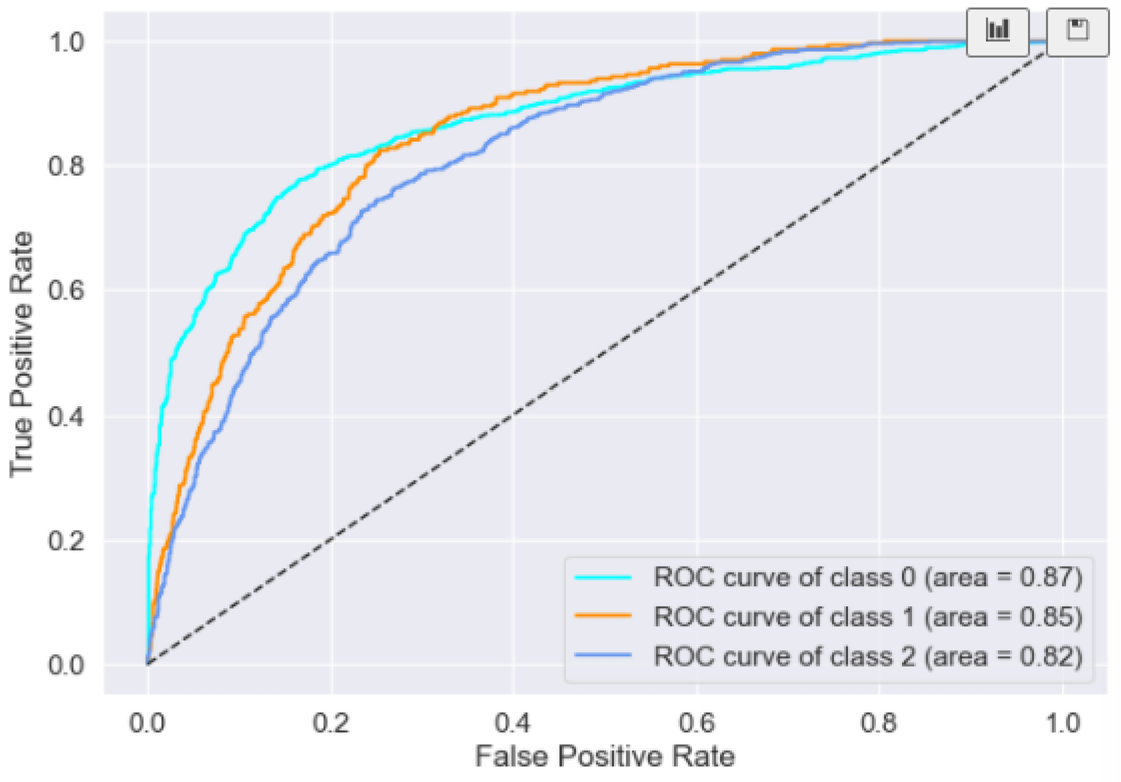


Figure 32: 256-128-64 ROC CURVE

The ROC curves we have here shows how good or bad our model is capable of distinguishing between classes. Measuring the AUC of each class we can see that our model can easier recognize the class 0 then the class 1 and last the class 2. **The ROC curve is actually the plot of the fraction of the correct predictions for the positive class versus the fraction of errors for negative classes**. Each one of these curves treats its own class as the positive one and the other two as the negatives. We want to have the correct positive class to be 1 and the fraction of incorrect negatives to be 0. The problem using ROC/AUC curve is that it does not recognize the imbalance that exists in our dataset. The class 1 has a small amount of realted data. But the ROC curve seems fine. The reason why this is happening is because having too many TN (from the other classes) examples which makes the **Denominator of FPR smaller**. Even if we have a bad classification for class 1.

classe	precision	recall	f1-score	support
0	0.82	0.74	0.78	1065
1	0.38	0.68	0.49	296
2	0.70	0.60	0.64	921

Table 3: Classification report

For example, consider the case of a dataset which has 10 positives and 100,000 negatives. We have 2 models:

- Model A: predicts 900 positives, in which 9 of them are true positives
- Model B: predicts 90 positives, in which 9 of them are true positives

Obviously, Model B has better performance. Although they both predict the same number of positives, Model B does not output as many false positives. In other words, Model B is more "precise". However, consider the ROC analysis of the two models, which measures true positive rate (TPR) against false positive rate (FPR):

- Model A: TPR = 9/10 = 0.9 and FPR = (900–9)/100,000 = 0.00891
- Model B: TPR = 9/10 = 0.9 and FPR = (90–9)/100,000 = 0.00081

TPR is, as expected, exactly the same between both models. On the other hand, since the number of negatives largely dominates that of positives, the difference of FPR between both models (0.00891–0.00081 = 0.0081) is lost in the sense that it can be rounded to almost 0. **In other words, a large change in the number of false positives resulted in a tiny change in the FPR and thereby, ROC is unable to reflect the superior performance of Model B in the context that true negatives are not relevant to the problem.**

The ROC curves of **MODEL 2** are a little bit **lower** than the ROC curves of **MODEL 1**, that happens because on MODEL 1 we have a better classification overall. But the same thing for the curves is happening here too.

classe	precision	recall	f1-score	support
0	0.78	0.77	0.78	1065
1	0.52	0.24	0.33	296
2	0.65	0.76	0.70	921

Table 4: Classification report

Last changes

As we saw earlier from the results our models are not trained in a balanced dataset.

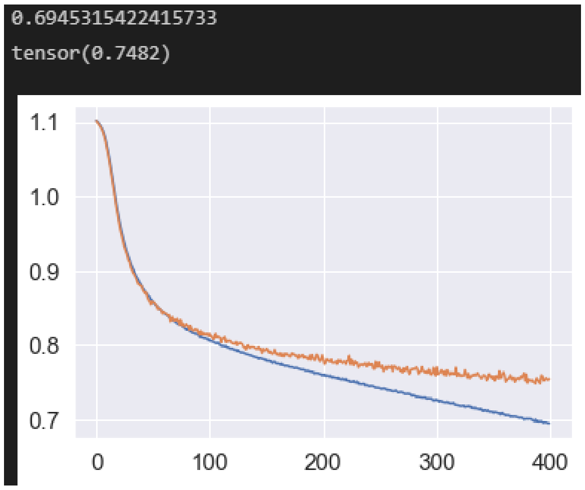
Class-Label	Number of tweets
0	7458
1	2073
2	6445

Table 5: Train set distribution

Class-Label	Number of tweets
0	1065
1	296
2	921

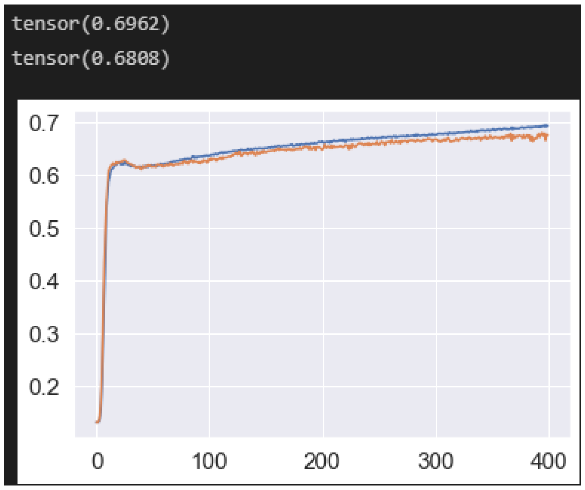
Table 6: Validation set distribution

From the above tables we can see more clearly the problem. The Class 1 does not have many examples and this does it's training not efficient.
This is why we are going to add class weights to our Cross Entropy Loss function in order to deal with this problem and probpably hit better F1-scores.
Let's see the results for MODEL 1.



```
0.6945315422415733
tensor(0.7482)
```

(a) Loss to Epochs



```
tensor(0.6962)
tensor(0.6808)
```

(b) Accuracy to Epochs

Figure 33: **256-128-64 weighted class version**

The results lost Accuracy and their error increased but now the model is more consistent and achieve better F1-score (which is the most important).

classe	precision	recall	f1-score	support
0	0.82	0.74	0.78	1065
1	0.38	0.68	0.49	296
2	0.70	0.60	0.64	921

Table 7: MODEL 1 Classification report/**weighted class version**

Overall scores:

- Accuracy: 0.6770376862401403
- F1_score: 0.6386230920102832
- Precision: 0.6338177687388923
- Recall: 0.6748346571097409

In simple words what we did was to make our model take into account the class 1 which is the smallest class. Previously the most possible scenario was that our model ignores totally the class 1, and focuses only on classifying correctly the 2 bigger classes.

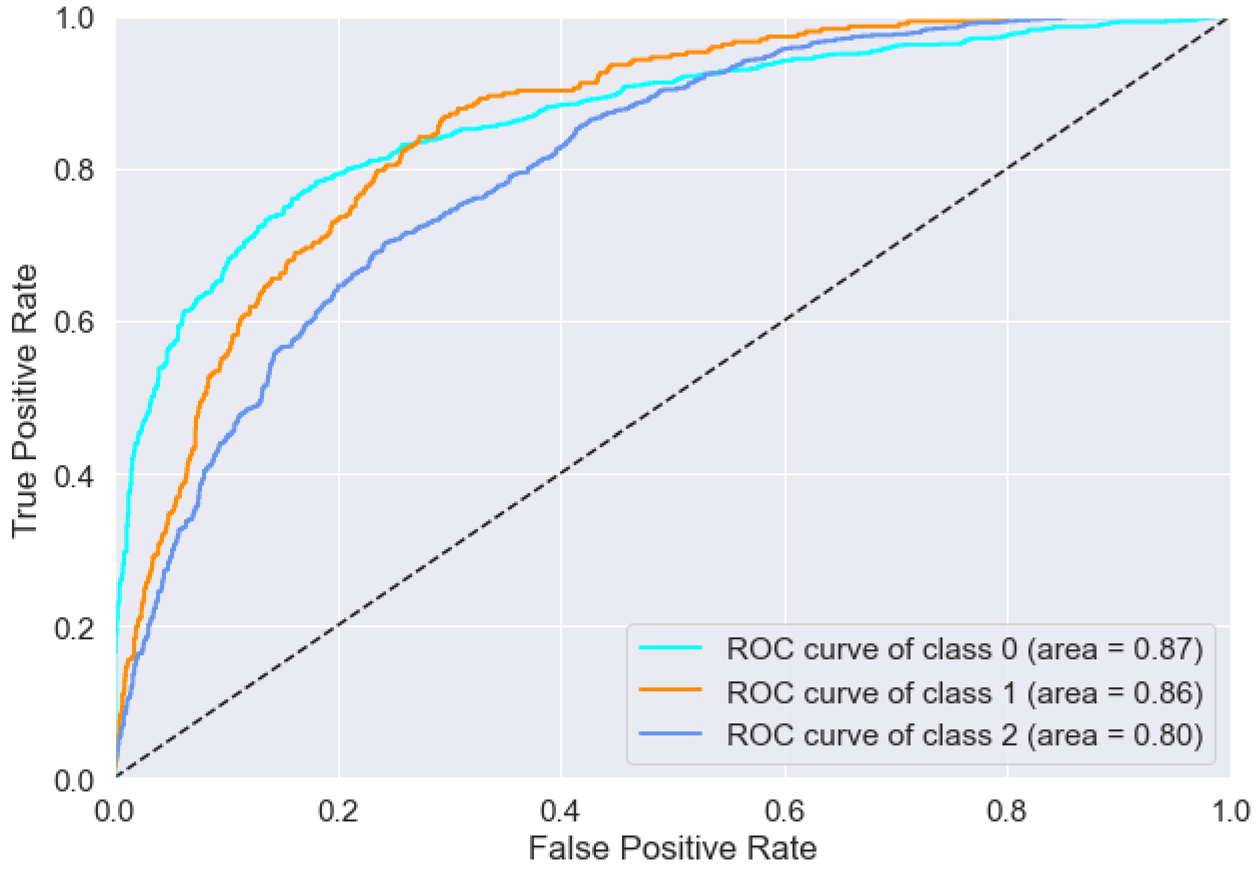


Figure 34: ROC CURVE

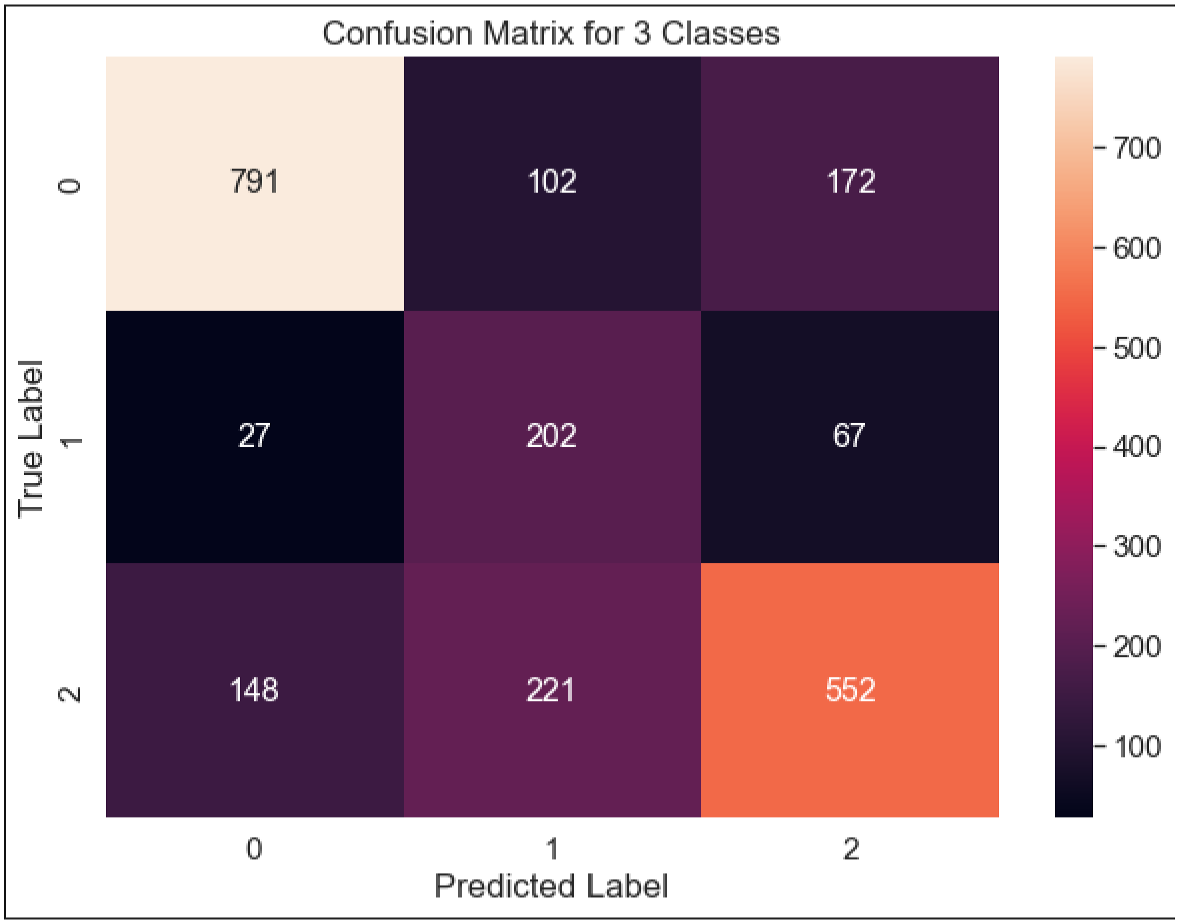
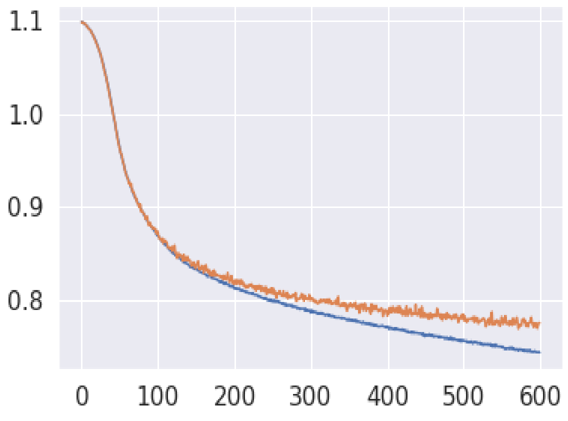


Figure 35: Confussion Matrix

The conclusion that we made earlier is more obvious on the confusion matrix. Where in a way we “tributed” some TP from the other classes in order to increase the TP in the class 1.

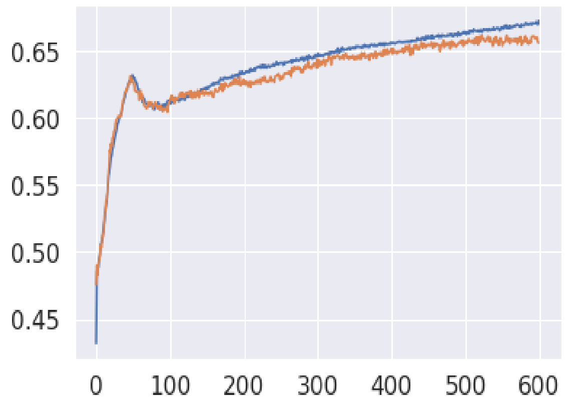
In order to verify the conclusions we made above we are going to test our theory in MODEL 2 too.

```
0.7426046977043151
tensor(0.7686)
```



(a) Loss to Epochs

```
tensor(0.6726)
tensor(0.6618)
```



(b) Accuracy to Epochs

Figure 36: **128-64-32 weighted class version**

These results, also, lost accuracy and their error increased but now the model is more consistent and achieve better F1-score (which is the most important).

classe	precision	recall	f1-score	support
0	0.82	0.71	0.76	1065
1	0.36	0.68	0.47	296
2	0.67	0.59	0.63	921

Table 8: MODEL 1 Classification report/**weighted class version**

Overall scores:

- Accuracy: 0.6577563540753725
- F1_score: 0.6215319487440252
- Precision: 0.6190936903768773
- Recall: 0.6599074472463012

ROC curves of MODEL 2 also lost AUC.

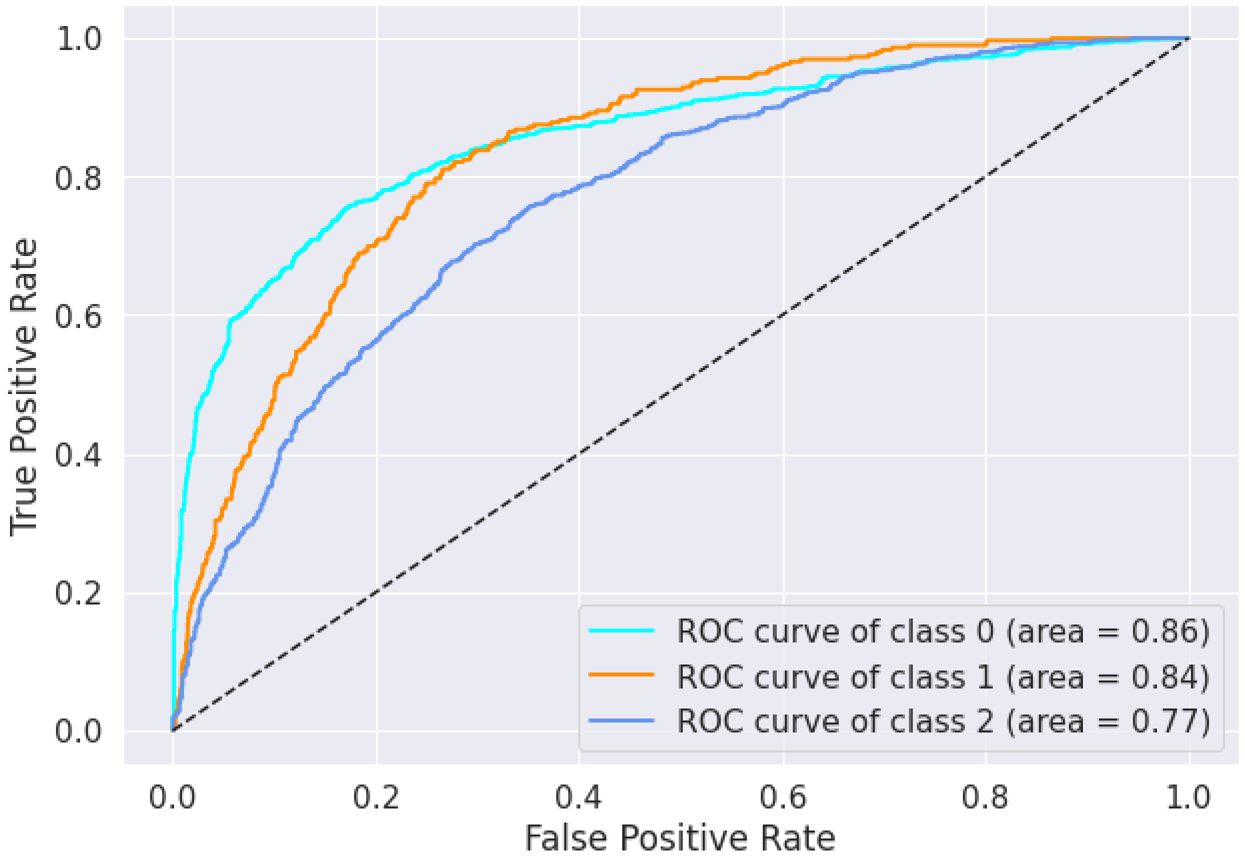


Figure 37: ROC CURVE

Furthermore, the Confusion matrix has less TP (True Positives) which shows how the 2 other classes loose correct prediction in order to get more the class 1.

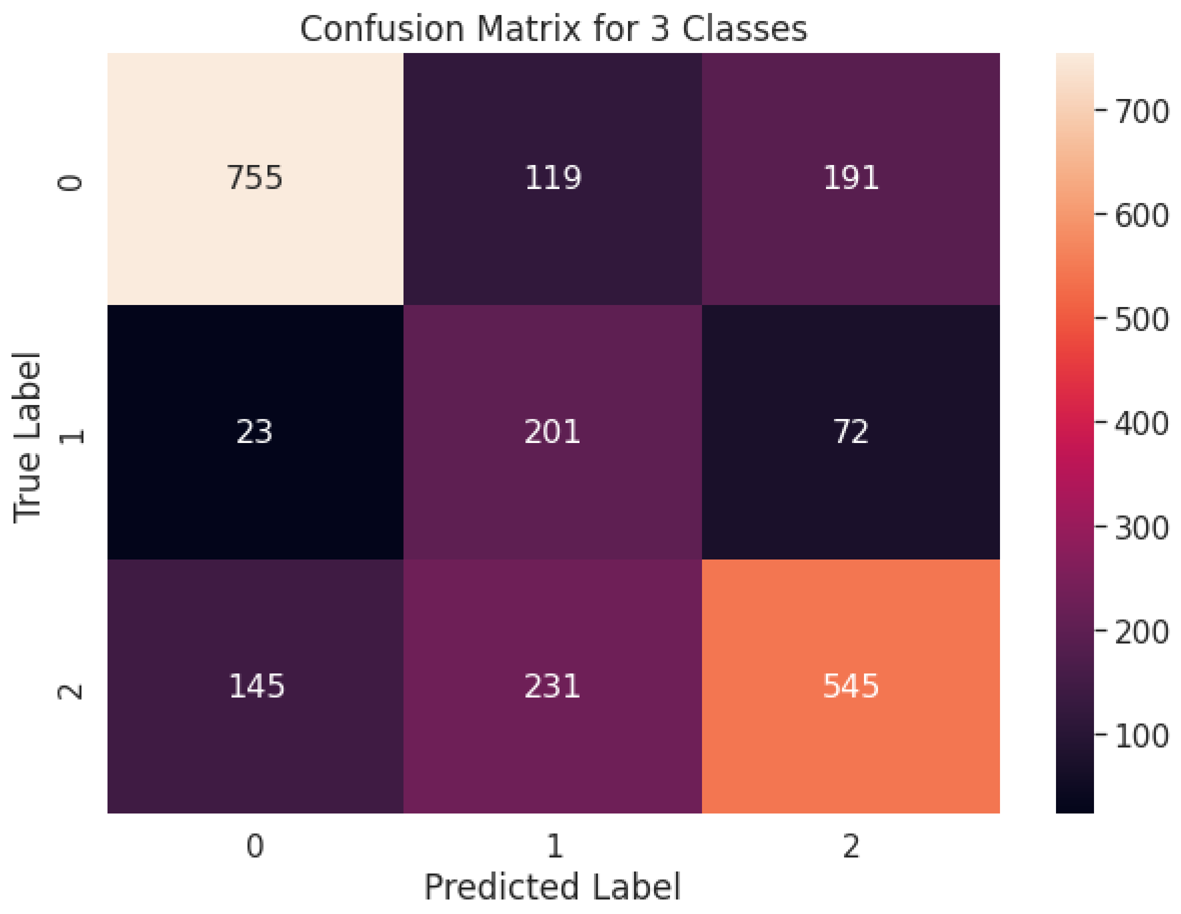


Figure 38: Confussion Matrix

Also, if we compare all the presented results of MODEL 1 and MODEL 2 we can see clearly that the MODEL 2 is always “behind” our best MODEL 1.

Comparison with previous excercise LogisticRegression.

The previous model was the **Softmax Regression**. Unfortunately I couldn’t reach the results of my previous model. (in comparison with model 2)

- Accuracy: 0.7261174408413672
- F1_score: 0.6735708797339278
- Precision: 0.7079955881508152
- Recall: 0.6564215727403321

BUT MODEL 2 is more consistent. In the previous excersice there were more the overfitting effect (even if I had better accuracy and F1-score). The current Neural Network Model duilt under this idea “Better generalization ability”. This is why I didn’t focused on accuracy. I tried to keep the values as close as possible between accuracy and error in the validation set. Adding the weights gave my model a more balanced way to face each example Dealing with each class as equally as possible. The runs with the test sets will show.