Homework 5

1.2b)

Since we have N = 178 features and a hidden layer with H = 16 neurons, there will be (N+1)\*H trainable weights for the first layer (the +1 is from the bias term). So there are 2864 trainable parameters for the first layer. For the second layer we have (H+1)\*5 weights (5 since that’s the number of output classes). This corresponds to 85 trainable weights. Thus, overall we have 2864+85=**2949 trainable weights**.

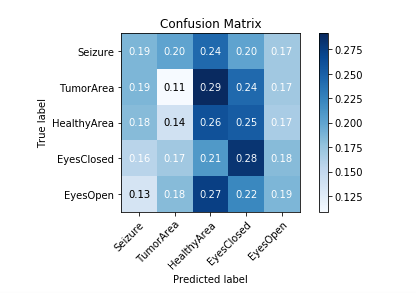
When a single data point comes in, in the first layer we are computing sigmoid(Wx + b) where W is of dimension (h x n), x is (n x 1) and b is (h x 1). This involves n multiplications and (n-1) additions h times + h for adding the bias term + h for the element-wise sigmoid. The overall result is (2n-1)h + 2h = (178\*2-1)16 + 2\*16 = 5712 floating point operations for the first layer. For the second layer we compute softmax(Wh + b) where h is the output of the previous layer and W is of dimension (5 \* h). This corresponds to (2h-1)5 + 2\*5 = 165 computations. The total is **roughly 6000 floating point operations** (assuming elementary multiplication, division, addition, and subtraction count as a single operation).

1.2c)



We see that our simple FNN does not perform very well. The accuracy is what you would get if you chose the classes for each data point at random pretty much.

1.2 d)



Once again, we see that the results are not very good. In a strong model, the diagonal would be most darkly colored and the rest would be close to white. However, our confusion matrix is shaded very sporadically. The lightest shade is actually along the diagonal - the model does a very bad job at classifying TumorArea.

1.2e)

After making some modifications, the results improved significantly. The changes I made include:

Standard scaling: This makes all the features comparable by subtracting by their mean and dividing by the standard deviation. Empirically, changing this alone helps slightly.

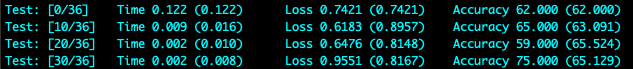
Adding more hidden units and layers: Now we have 2 hidden layers. The first one is of size 64 and the second one is of size 16. Before the model was under fitting since the training loss was high so by adding this additional complexity, we see that the results have improved significantly.

Activation function: Switching from sigmoid to relu helped. This is thanks to the non-saturation of the relu function for large values. With sigmoid, large values give values of 0 which results in the vanishing of gradient. This issue does not occur with relu. However, since this is a relatively shallow neural net, the results of swapping the activations was not significant but it did help a little.

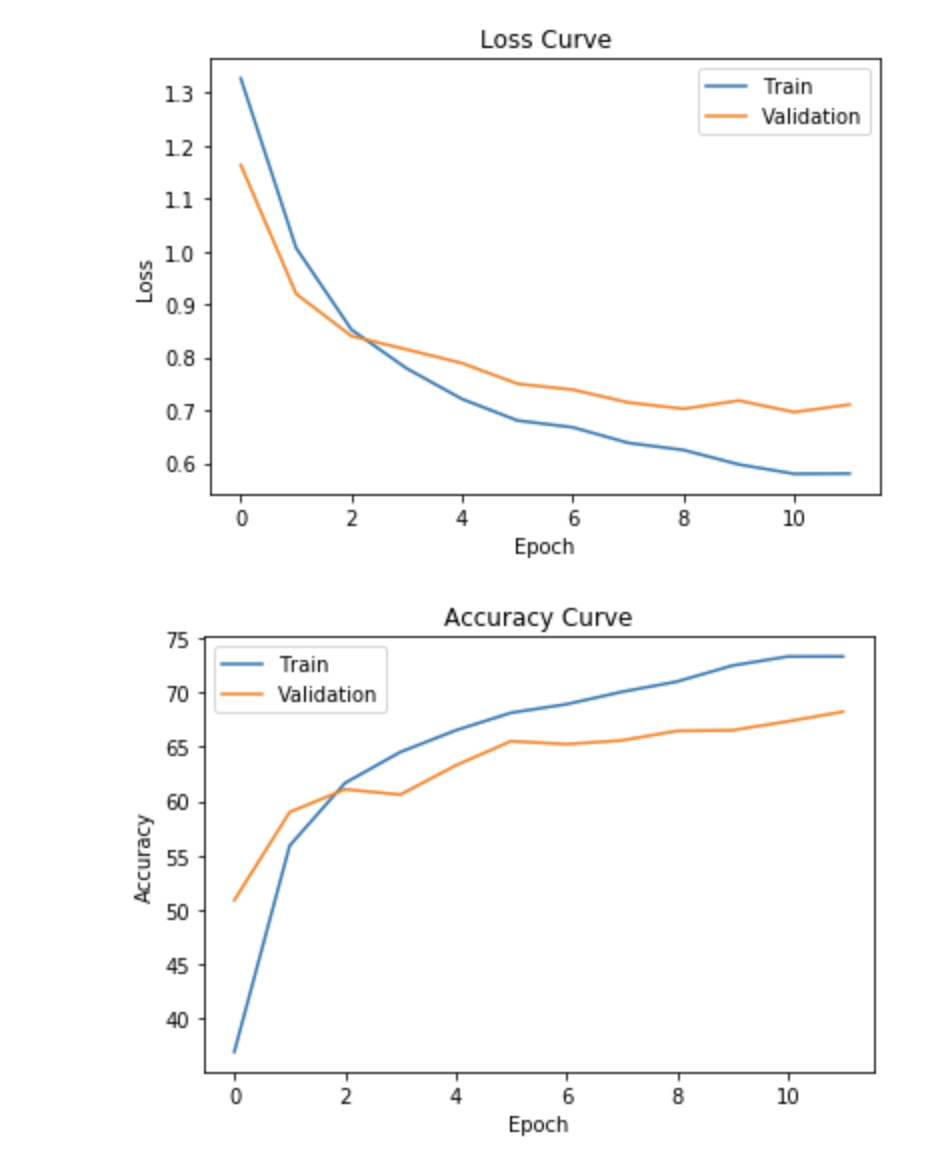
Epochs: The number of epochs was increased to 12. This was found to be a good value before overfitting occurred.

Here are the results after making these modifications:

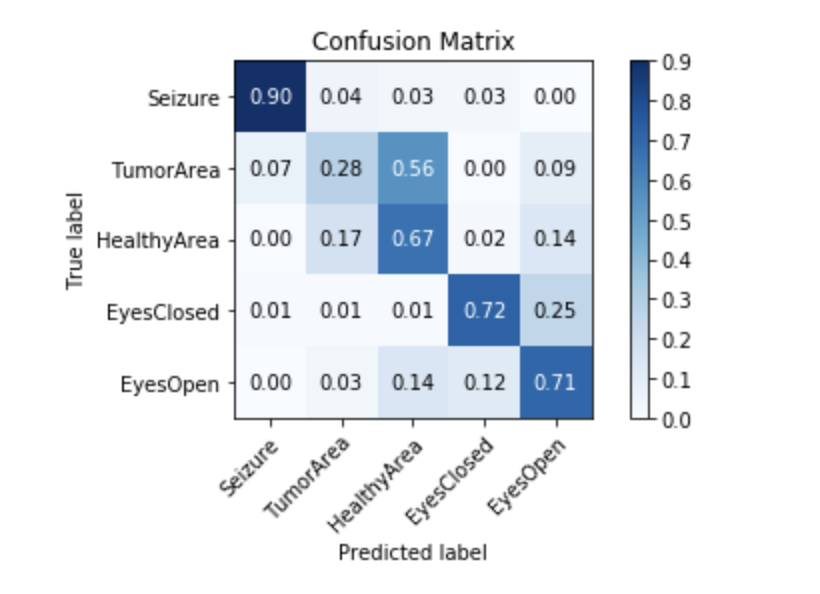
Test set results:



Learning curves:



Confusion matrix:



We notice that the model is still quite poor at classifying TumorArea and often thinks it’s a HealthyArea. This might be due to the imbalance of the examples: there are many more HealthyArea examples than TumorArea examples and thus we have a class imbalance. This is known to be a difficult setting for machine learning models but there are techniques that exist to combat this (I do not explore those here however).

1.3 b) First let us acknowledge the generalized calculation for parameters for each of the types of layers.

Conv1d layer: This is simply (#filters \* #channels + 1)\*filter\_size (assuming stride = 1). The +1 corresponds to the parameter for the bias term.

MaxPool: One of the main benefits of pooling layers are that they don’t add any parameters to the model. Thus maxpool always introduces 0 trainable parameters.

Dense layer: We derived this one in the last part.

So for the given architecture, are calculation is as follows:

(5+1)\*6 + 0 + (5\*6+1)\*16 + 0 + (41\*16+1)\*128 + (128 + 1)\*5 = 36 + 496 + 84,096 + 645 = **85,273 trainable parameters**. Note how the vast majority of them come from the linear layers.

These are the # of calculations:

The lecture slides give them formula for the number of calculations. Namely:

Conv Layer: (W\*C2)\*F\*C1(where W is output length, C2 is #filters, F is filter\_size, C1 is input\_depth

Max Pool: (W\*C2) \* F

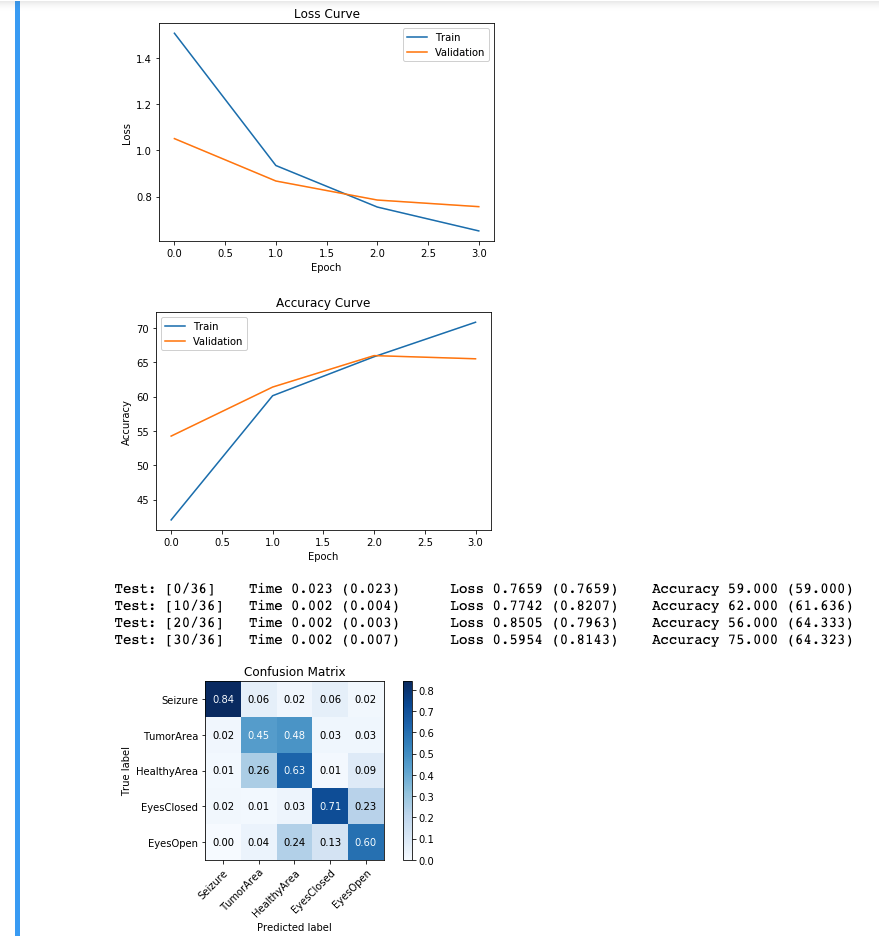
Dense: (W\*C1)\*D

Putting this together, we get:

(174\*6)\*5\*1 + (87\*6)\*5 + (83\*16)\*5\*6 + (41\*16)\*5 + (41\*16)\*128 + (small number for the last one) ~= 150,000 calculations.

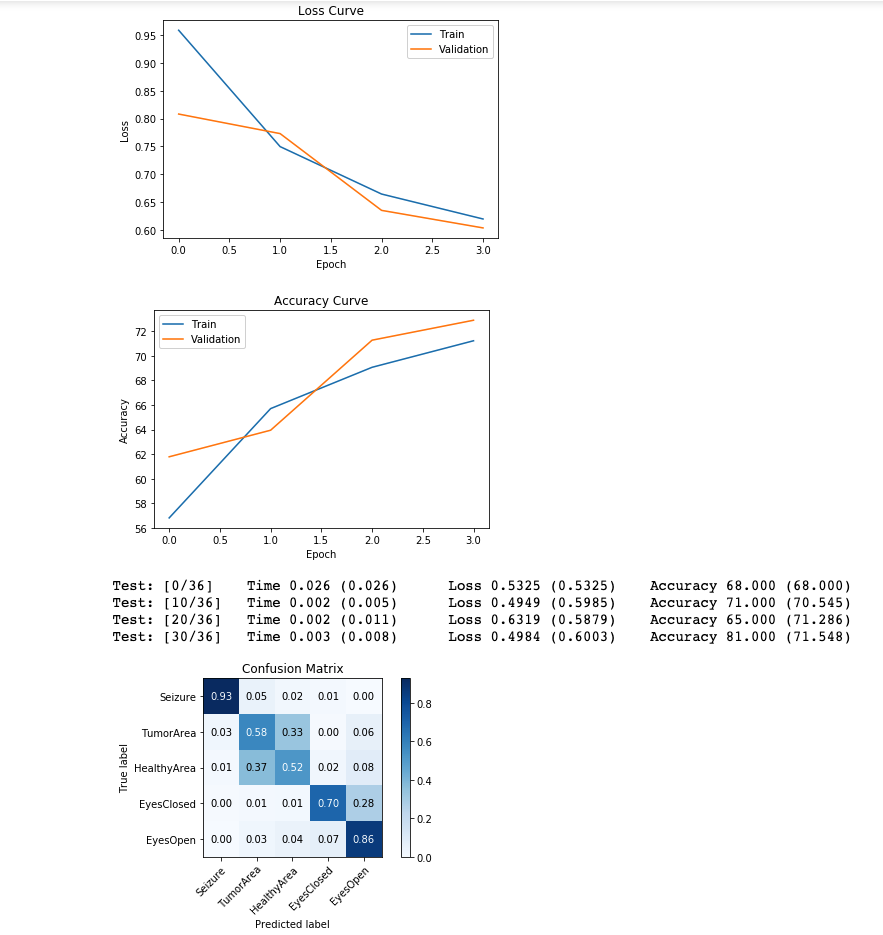
**So the CNN model involves roughly 150,000 calculations.** Note that this is much greater than for the densely connected neural network from the first part.

1.3 c)

These are the results of the baseline CNN:

1.3d)

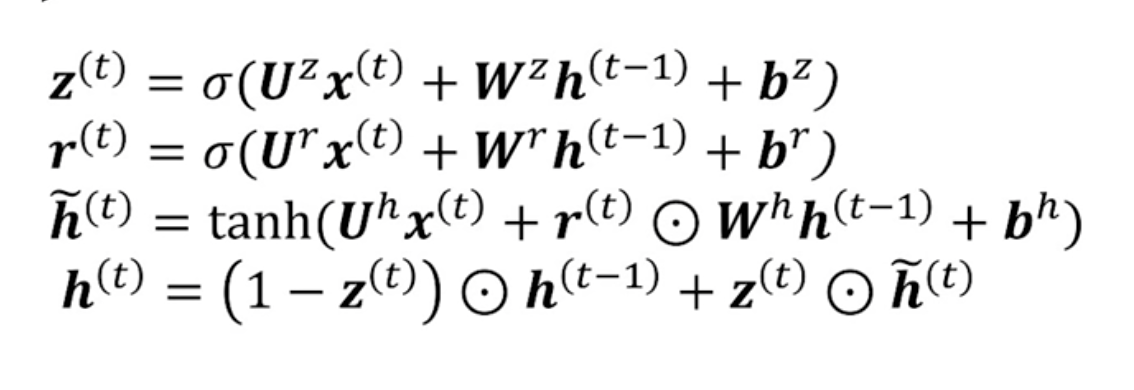
I employ two methods that improve the results noticeably from above: Batch normalization and standardization of the input (same as from the MLP model). Here are the results:



This model does tend to overfit if we increase the epochs above 4 or so, so there are still techniques we could employ to improve the model further. Namely regularization techniques such as dropout could help. Also different weight init strategies, different activation functions, different optimizers, etc. The possibilities are nearly endless but for the sake of time, I just experimented with the above.

1.4 b)

Trainable parameters: The dense layer has 85 parameters using the same reasoning as from earlier parts. The GRU layer has the parameters as seen in this picture (namely the weight matrices):



There are 6 different weight matrices and 3 corresponding bias terms. Our hidden state is of size 16, our input x is of size 1 (just the scalar value at that dimension in the original data matrix), and the sequence length is 178 but this doesn’t matter since the parameters are the same at every time step.

Therefore all the Us are vectors of size 16x1, all of the W’s are of size 16x16 and are bias terms are of size 16. Therefore, the GRU has 3 \* (16^2 + 16 + 16) = 864 parameters.

This the total number of parameters are: 85 + 864 = **949 trainable parameters**

For the number of floating point calculations, I will do a rough calculation. Precise calculation is straightforward but tedious. Anyways, in the GRU layer, the number of calculations required can be seen from the above equations:

Z(t): 16 + 16\*16 + 16 + 16 = 304

R(t): exactly the same for z = 304

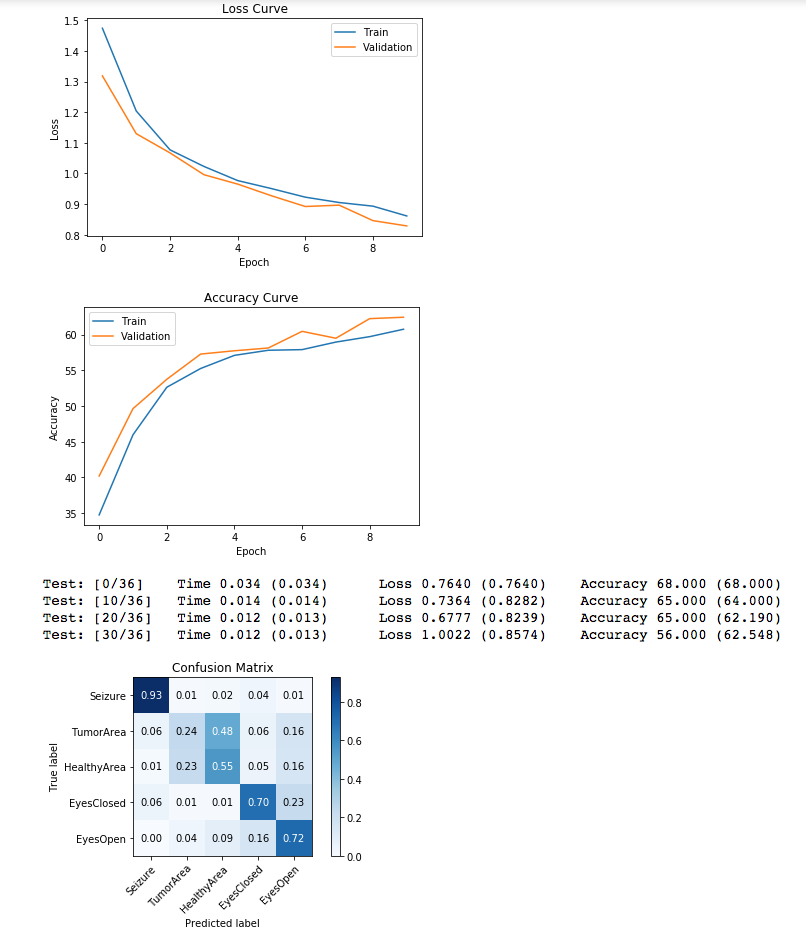
~H(t): exactly the same as above: 304

H(t): 16 + 16 + 16 = 48

The total is 304\*3 + 48 = 960. We do this 178 times for a single data point so after the final time step of the GRU, we’ve performed 960\*178 ~= 170,000 calculations.

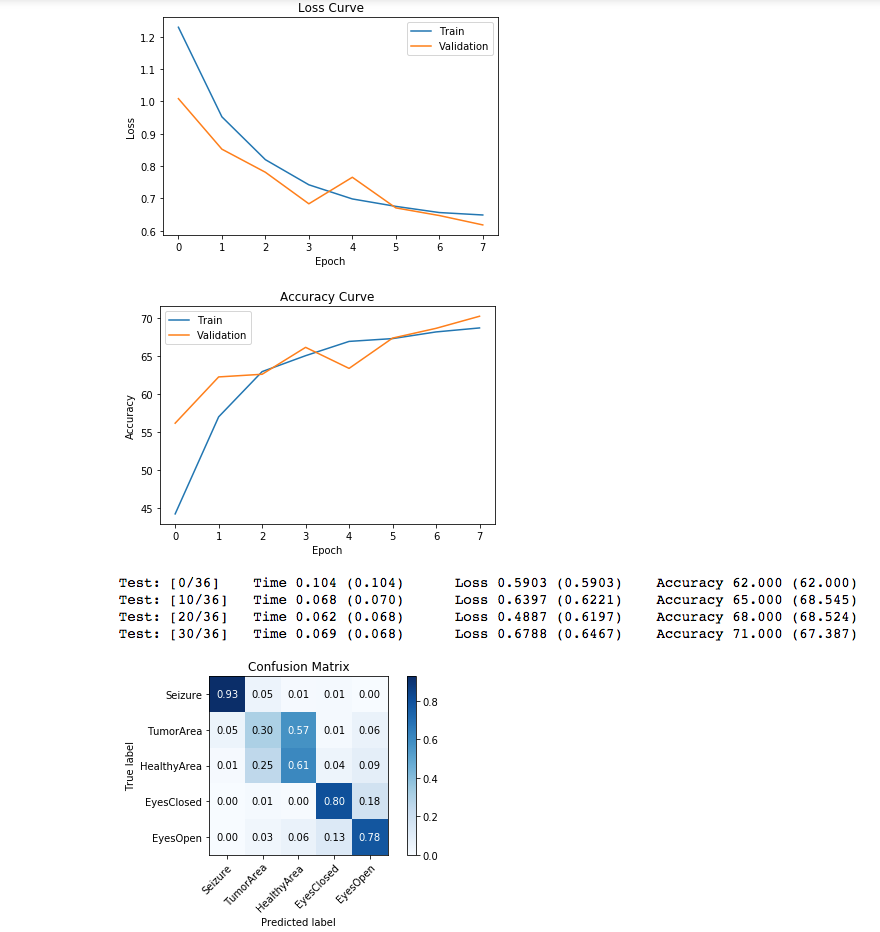
For the fully connected layer, we’re doing the following calculation: softmax(Wh + b) where W is of size (5x16) and b is of size 5. So this is about 5\*16 + 5 + 5 = 90 calculations. But we only pass the last hidden output to the classification layer so this is only performed once (as opposed to if we had a many-to-many RNN in which we would perform it 178 times). So this is a negligible amount of calculations and **the total number of calculations is roughly 200,000**.

1.4c) Here are the results of the baseline model:



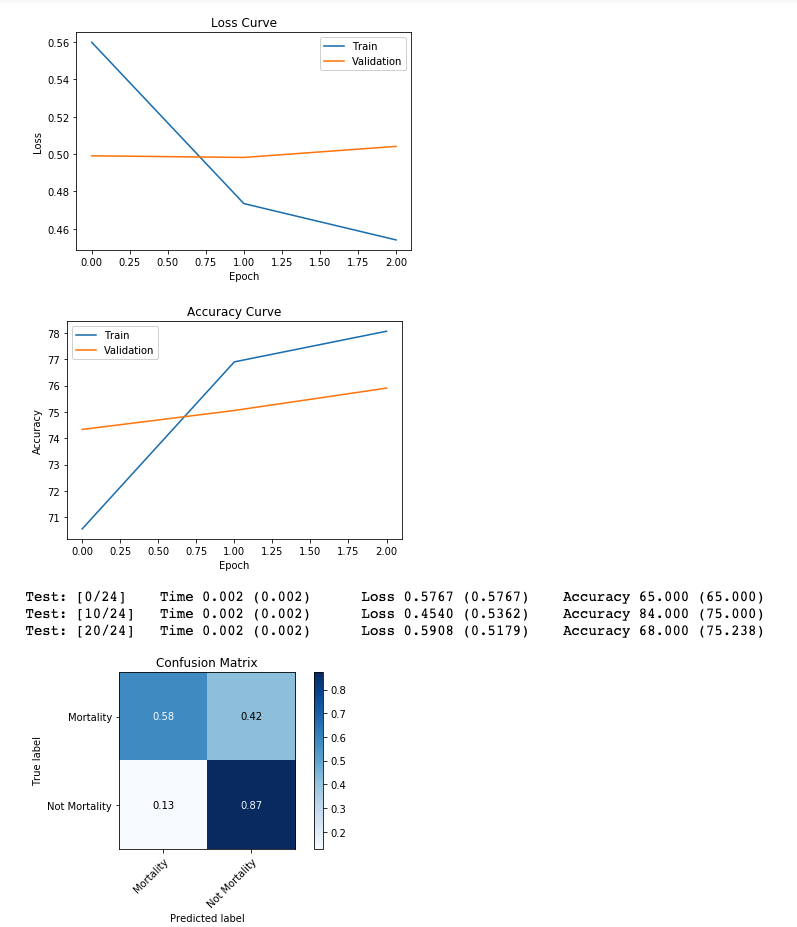
1.4d)

Modifications made are: Using a bidirectional GRU, adding an additional layer, increasing the hidden size, and adding dropout for regularization for this increased complexity. The results are shown below, we see a modest improvement over the baseline:



2.3a)

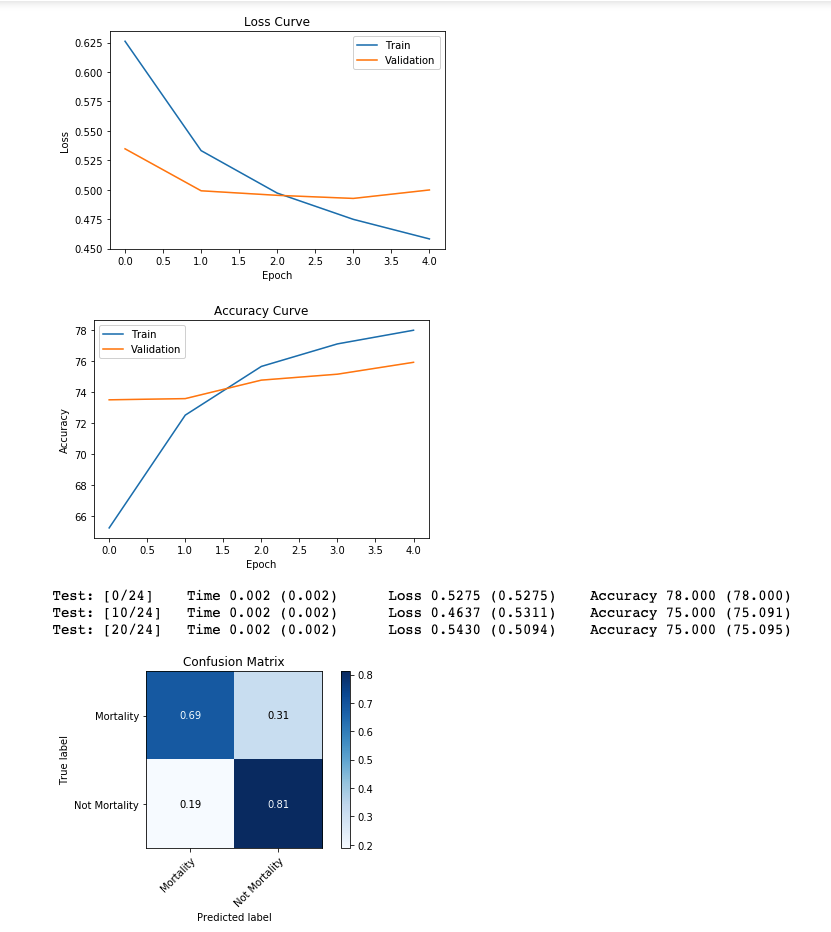
Here are the results for the baseline model:



2.3b)

Changes I made include adding dropout to both the first fully connected layer and the GRU (p=0.5). Increasing the number of layers in the GRU to 2 seemed to help as well. Also, initializing the weights of the fully connected layers using the xavier\_uniform init made a small improvement as well. Other things tried include batch normalization, adding more layers, changing # of neurons in the various layers, and more but none of these made a positive impact on the performance. Also I tried switching the RNN type to LSTM which resulted in roughly equivalent results, and also used the Adadelta optimizer instead of Adagrad which gave a noticeable improvement but I switched back to Adam as I was unsure if I’m allowed to change the optimizer.

The results are shown below:



We gained a large boost to correct classification of mortalities but seemed to have suffered a bit in predicting non-mortalities. There seems to be a tradeoff between less complex models doing better on non-mortalities and worse on mortalities vs higher complexity models doing more equally on both classes.