# MATH 6373 DEEP LEARNING AND NEURAL NETWORKS
# Home Work 2

**Coauthors and Contributions:**

| S No | Name | Email | Contribution (%) |
|---|---|---|---|
| 1 | Thanh Hung Duong | duongthanhhung86@gmail.com | 33.33 |
| 2 | Kishore Tumarada | Kishore.t04@gmail.com | 33.33 |
| 3 | Dustin Vasquez | Dustin.vasquez2@gmail.com | 33.33 |

In this homework, we are using TensorFlow for deep analysis into a data set that describes different heart arrhythmias. We will use a Multi-Layer Perceptron (MLP) network to explore the dataset and properly classify the type of arrhythmia based on the features. After optimizing the parameters for the deep learning, we will look into the hidden layer and the neuron activity.

***Part 1: Describe your Data set and the associated classification task***

We are using the MIT-BIH Arrhythmia dataset, which contains 48 half-hour excerpts of two-channel ambulatory ECG recordings, obtained from 47 subjects studied by the BIH Arrhythmia Laboratory between 1975 and 1979.
This dataset has been used in exploring heartbeat classification using deep neural network architectures and observing some of the capabilities of transfer learning on it. The signals correspond to electrocardiogram (ECG) shapes of heartbeats for the normal case and the cases affected by different arrhythmias. These signals are preprocessed and segmented, with each segment corresponding to a heartbeat.

***1.1. indicate the number of cases N in the data set (after reduction of number of classes to 3)***
Table 1 shows the total cases and size of each class in original and reduced dataset.

| | **Original Dataset** | **Reduced dataset** |
|---|---|---|
| **No of cases** | 109,446 | 18,719 |
| **classes** | 5 | 3 |
| **Features** | 187 | 187 |
| **class names** | | |
| **N: Normal (0)** | 72741 | 6500 |
| **S: Supraventricular (1)** | 2223 | 0 |
| **V: Ventricular (2)** | 5788 | 5788 |
| **F: Fusion of ventricular and normal beat (3)** | 641 | 0 |
| **Q: Unclassifiable beat. (4)** | 6431 | 6431 |

*Table 1 Features of original and reduced dataset*

***1.2. Present in words the distinct classes C1 C2 C3 involved and their sizes***
***1.3. - Select a training set and a test set; give the size of each class within the training set and within the test set; give the proportions of cases in each class for the test set and for the training set; these two proportions should be similar for each class***
As shown in table 1 original dataset with 5 classes has been reduced to 3 classes dataset with balanced class sizes. The reduced dataset has been divided into train and test datasets in the ratio of **4: 1.** Class sizes of three classes C1, C2 and C3 are as shown in Table 2.

| **Classes** | **Train** | **Test** | **Total** |
|---|---|---|---|
| **C1 - N Normal (0)** | 5200 | 1300 | 6500 |
| **C2 - Q: Unclassifiable beat. (4)** | 5145 | 1286 | 6431 |
| **C3 - V: Ventricular (2)** | 4630 | 1158 | 5788 |

*Table 2 Class sizes for train and test datasets*

Proportion of classes in both train and test dataset is **1.12: 1.11: 1.**

Train data has following descriptive statistics for first 5 features:

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| **count** | 14975.000000 | 14975.000000 | 14975.000000 | 14975.000000 | 14975.000000 |
| **mean** | 0.799915 | 0.681370 | 0.507636 | 0.377554 | 0.337879 |
| **std** | 0.311802 | 0.283744 | 0.252459 | 0.258095 | 0.242414 |
| **min** | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| **25%** | 0.746667 | 0.523810 | 0.336364 | 0.159003 | 0.134047 |
| **50%** | 0.948310 | 0.768939 | 0.516854 | 0.360856 | 0.278135 |
| **75%** | 1.000000 | 0.909411 | 0.706069 | 0.553114 | 0.527923 |
| **max** | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |

8 rows × 188 columns

*Table 3 Train dataset descriptive statistics*

Test data has following descriptive statistics for first 5 features:

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| **count** | 3744.000000 | 3744.000000 | 3744.000000 | 3744.000000 | 3744.000000 |
| **mean** | 0.799026 | 0.680290 | 0.507333 | 0.376406 | 0.336908 |
| **std** | 0.307719 | 0.281333 | 0.253401 | 0.259073 | 0.244965 |
| **min** | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| **25%** | 0.739614 | 0.523387 | 0.329928 | 0.159760 | 0.134179 |
| **50%** | 0.944643 | 0.761905 | 0.517345 | 0.355177 | 0.274551 |
| **75%** | 1.000000 | 0.907244 | 0.706647 | 0.552963 | 0.529918 |
| **max** | 1.000000 | 1.000000 | 1.000000 | 0.984791 | 1.000000 |

8 rows × 188 columns

*Table 4 Test dataset descriptive statistics*

***Part 2: Define the MLP architecture of an Automatic Classifier with r =3 classes***

***2.1. Select an MLP architecture with three layers L1, H, L2, extended by a Softmax function and its final output layer OUT***
***L1 => H => L2=> softmax => OUT***
***of respective dimensions p = dim(L1) = # descriptors, h = dim(H), dim(L2) = 3; dim(OUT) =3***

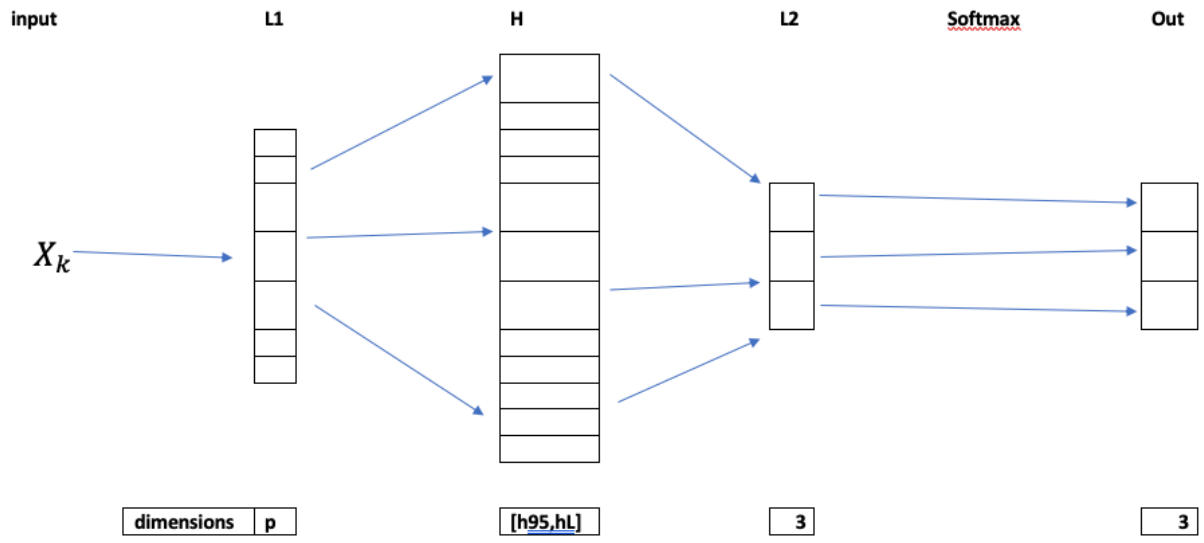



*Figure 1 MLP architecture*

Figure 1 shows the structure of neural network built for this homework. Here p = 187 features

**2.2. Select a response function (either the sigmoid function or the RELU function) to be set for all neurons**

The response function selected is ReLU, which is given by

$$f(x) = x^+ = \max(0, x),$$

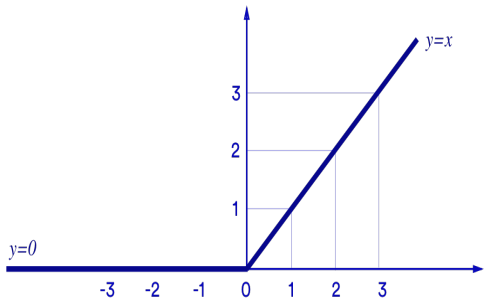



*Figure 2 ReLU function graph*

***2.3. The goal of the MLP classifier is to provide for each input vector Xk an output probability vector OUTk very close to the binary vector encoding the true class of the input Xk. Explain precisely what is the part is played by the softmax function, and how the final classification of Xk is computed from OUTk***

The numeric output of L2, the last layer in MLP neural network, is called logits. This output may not be positive or may be even greater than 1 in some cases. But we have to normalize this output to a probability distribution, so that we can identify the predicted class, which would have the highest probability.

In order to normalize the output of last layer, we have to use softmax function. The standard softmax function is defined by following formula:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \text{ for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K$$

We apply the standard exponential function to each element $z_i$ of the logits output of neural network and normalize these values by dividing by the sum of all these exponentials.

This normalization ensures that:

1. Each $\sigma$(zi) will be in the interval (0,1)
2. The sum of the components of output vector $\sigma(z)$ is 1.

**Final classification:**

As we have a classification problem with 3 classes, the 3 neurons in last layer L2 (and Out layer) indicate the class of the case $X_k$ .For instance, if neuron 1 has highest probability in out layer, then class 1 is the output. So, by reading the argument of maximum probability in OUT layer, which is neuron number, we can identify the predicted class of neural network.

***Part 3: Select 2 tentative sizes h for the hidden layer***

To estimate one small but plausible value for h, namely h = h95 < p , apply PCA analysis to your training set of input vectors to generate p eigenvalues ordered in decreasing order

***3.1. Plot this decreasing sequence of eigenvalues in yr report***

***H95***

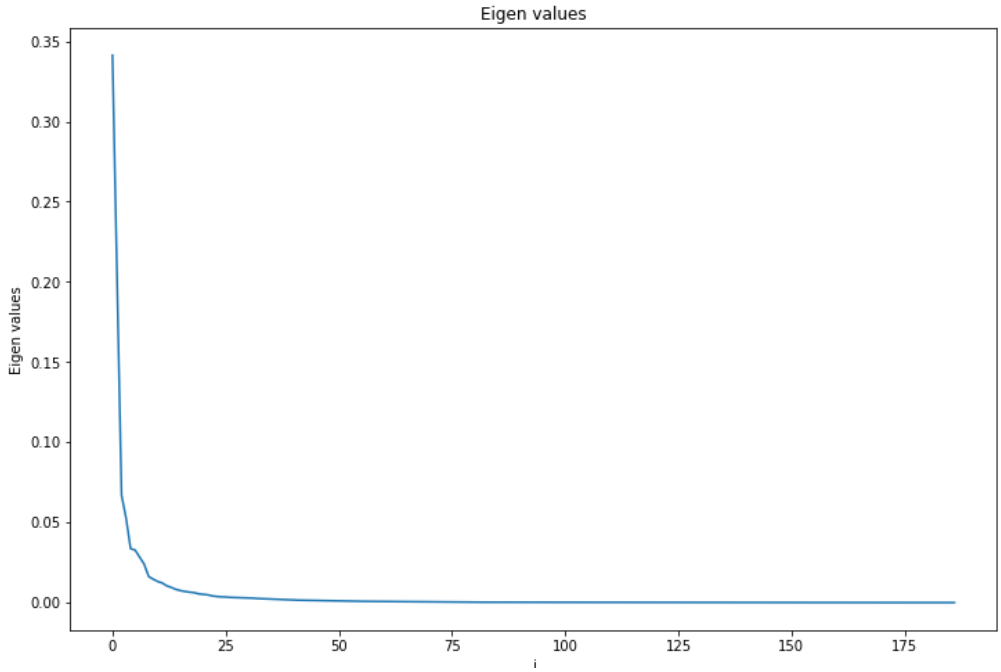



*Figure 3 Plot of Eigen values of train data*

***3.2. Compute the smallest number "h95" of eigenvalues preserving 95% of the total sum of eigenvalues***

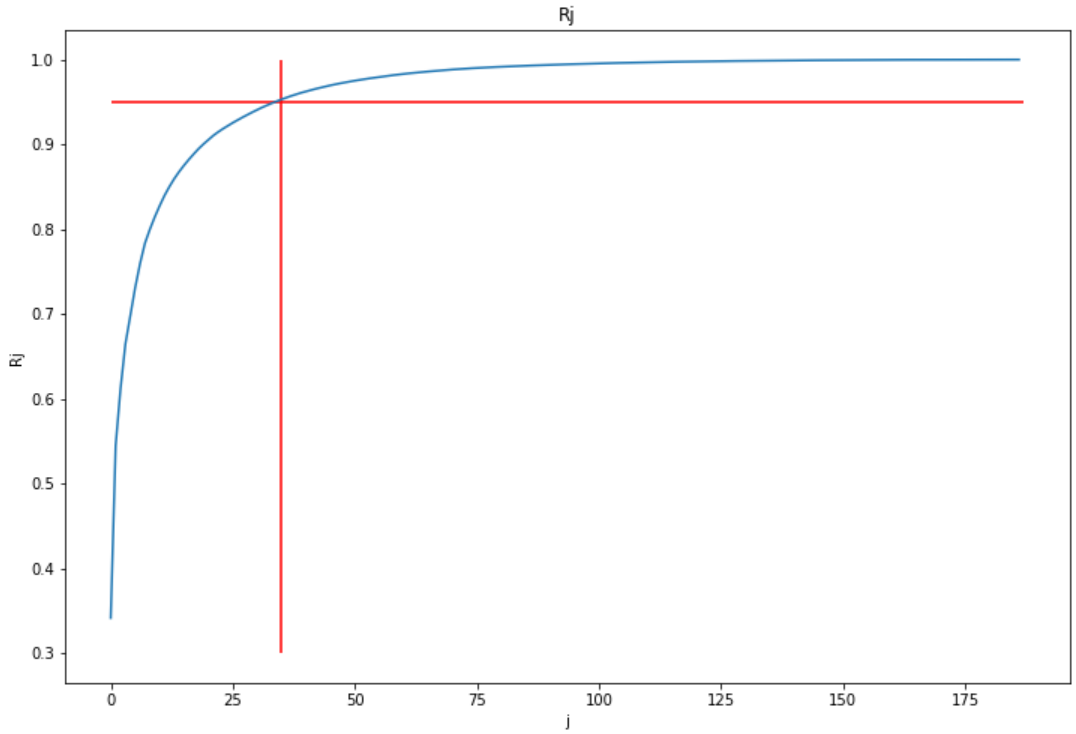



*Figure 4 Plot of Rj(explained variance ratio) vs j*

In figure 4, horizontal redline indicates the 95% line and the corresponding value on x axis is hL =35.

***3.3. To estimate one larger plausible value hL for the size h, proceed as follows***

***- apply PCA analysis to the set of Mj input vectors corresponding to the class Cj, with j=1,2 3 to generate Mj eigenvalues in decreasing order, and compute the smallest number "Uj" of eigenvalues preserving 95% of the total sum of these Mj eigenvalues;***

***- then define hL = U1 + U2 +U3.***

For class 1, At eigenvalue 42 we get a ratio of 95.01%. Hence U1 = 42.
For class 2, At eigenvalue 21 we get a ratio of 95.07%. Hence U2 = 21.
For class 3, At eigenvalue 30 we get a ratio of 95.13%. Hence U3 = 30.

***hL =93***

***Part 4: Implement automatic training on two sizes of h***
***4.1. Use gradient descent to minimize avCRE= average CROSS ENTROPY error between computed and true values of the probability OUTk***

In automatic training phase, after each batch new weights are calculated through following formula :

$$\boldsymbol{W(s+1) = W(s) - \epsilon_s * grad(avCRE)|_{W(s)}}$$

$\boldsymbol{\epsilon_s}$ is the learning rate calculated by exponential decay function.

$\boldsymbol{grad(avCRE)|_{W(s)}}$ - refers to gradient descent calculated by partial differentiation of average Cross Entropy calculated at W = W(s), weights of current batch.

Gradient descent is essentially a first-order iterative optimization algorithm for finding a local minimum of a differentiable function.

As shown in figure 1 neural network structure, after L2 layer , a softmax function is used to convert logits output of network to prediction probability. Cross Entropy can be calculated by

$$CE = \sum_1^k -q_i * \log p_i$$

Where $q_i$ $-$ $true\ class\ probability$,
$p_i$ $are\ the\ predicted\ class\ probabilities, obtained\ after\ softmax$
k = 3, number of classes.

***4.2. What is avCRE during training and after each epoch***

avCRE is calculated after each batch of training by calculating average of Cross entropies, which are computed as per above equation.

While avCRE after each epoch is the average CROSS ENTROPY errors of the whole training set. This value is named “loss”

***4.3. Which software environment you will use for HW2***

We used TensorFlow version 2.1 in Python with Keras library.

### *4.4. Which software functions you are choosing to implement MLP learning with aCRE loss function*

Model.compile(optimizer, loss, metrics) is a syntax for configuring the model. Inside it, Loss function is used to measures how accurate the model is during training. This function will be minimized to "steer" the model in the right direction. To define avCRE as a loss function we used tf.keras.losses.CategoricalCrossentropy() function.

### *4.5. What are the options offered for this task and which option did we select?*

- For initialization of the weights and thresholds, list of options are:
  - All values are zeros
  - All values are ones
  - All values are constants
  - All values are generated from a normal distribution.
  - All values are generated from a uniform distribution

- For batch learning, when fitting the model, we assigned batch_size as B. This value might vary from 1 to the total number of cases in the training set. Normally, it is set between 100-500.

- For the successive gradient descent steps sizes ε(n) in TensorFlow, we have 5 options:
  - class ExponentialDecay: A LearningRateSchedule that uses an exponential decay schedule.
  - class InverseTimeDecay: A LearningRateSchedule that uses an inverse time decay schedule.
  - class LearningRateSchedule: A serializable learning rate decay schedule.
  - class PiecewiseConstantDecay: A LearningRateSchedule that uses a piecewise constant decay schedule.
  - class PolynomialDecay: A LearningRateSchedule that uses a polynomial decay schedule.

- For stopping the learning, tf.keras.callbacks.EarlyStopping function will stop training when a monitored quantity has stopped improving was applied. Some of its arguments are:
  - monitor: quantity to be monitored.
  - min_delta: minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change of less than min_delta, will count as no improvement.
  - patience: number of epochs that produced the monitored quantity with no improvement after which training will be stopped. Validation quantities may not be produced for every epoch, if the validation frequency (model.fit(validation_freq=5)) is greater than one.

- For intermediary outputs to monitor learning quality, we defined a class named MyHistory(). Herein, the function “on_train_begin” was for setup the initial values of the intermediary outputs. Function “on_epoch_end” will define how the intermediary output should be computed at the end of each epoch.

### *4.6. Selections for batch size B, for time dependent gradient descent step size, for gradient descent algorithm, for stopping the learning, for random initialization of weights and thresholds.*

- As mentioned in class, mini size batch help avoid the local minimum, we set B =100
- Gradient descent step size learning_rate argument was set as optimizers.schedules.ExponentialDecay( ). It has the mathematical form $lr = lr_0 * e^{-kt}$. These hyperparameters was choose as follow:
  - Initial rate $lr_0 = 0.1$
  - $k = 0.4$

- For gradient descent algorithm, the optimizer as Stochastic Gradient Descent : keras.optimizers.SGD(learning_rate, momentum, nesterov)
- For stopping the learning, we chose the arguments as follow: patience=10, monitor='val_loss'. It means that the learning will stop if the no change in the loss of the test set 10 times consecutively.
- For initial values of weights and threshold, we generate them from a normal distribution that has mean = 0 and standard deviation =0.05

### *4.7. Compute and plot the curve baCREn*

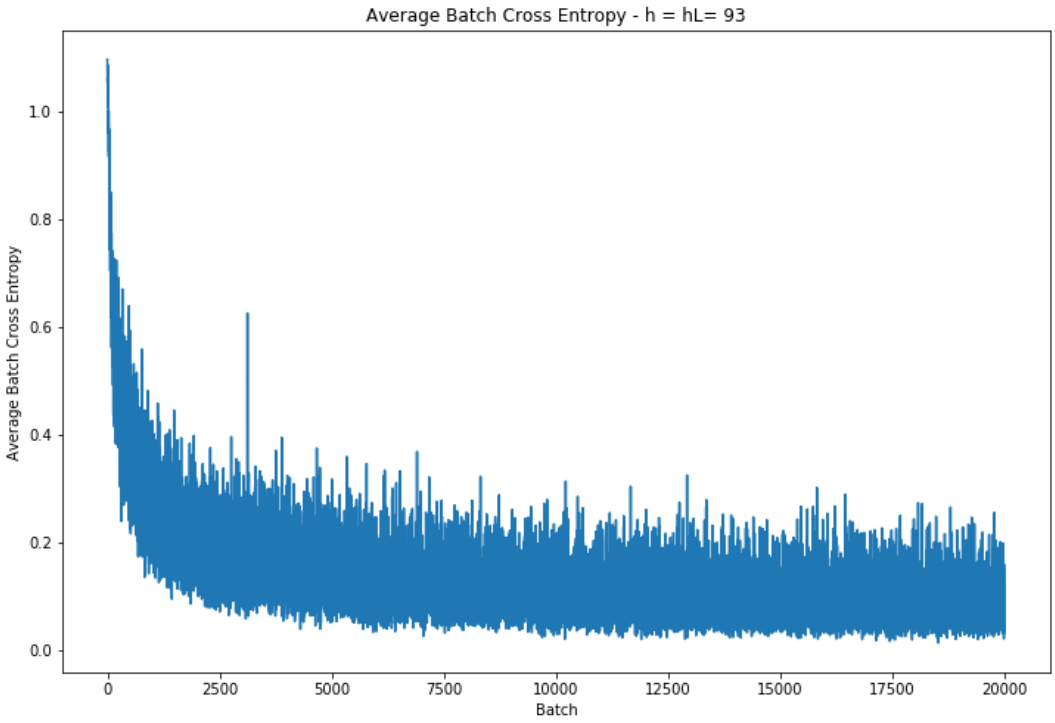



***Figure 5 Average batch cross entropy for h = hL =93***

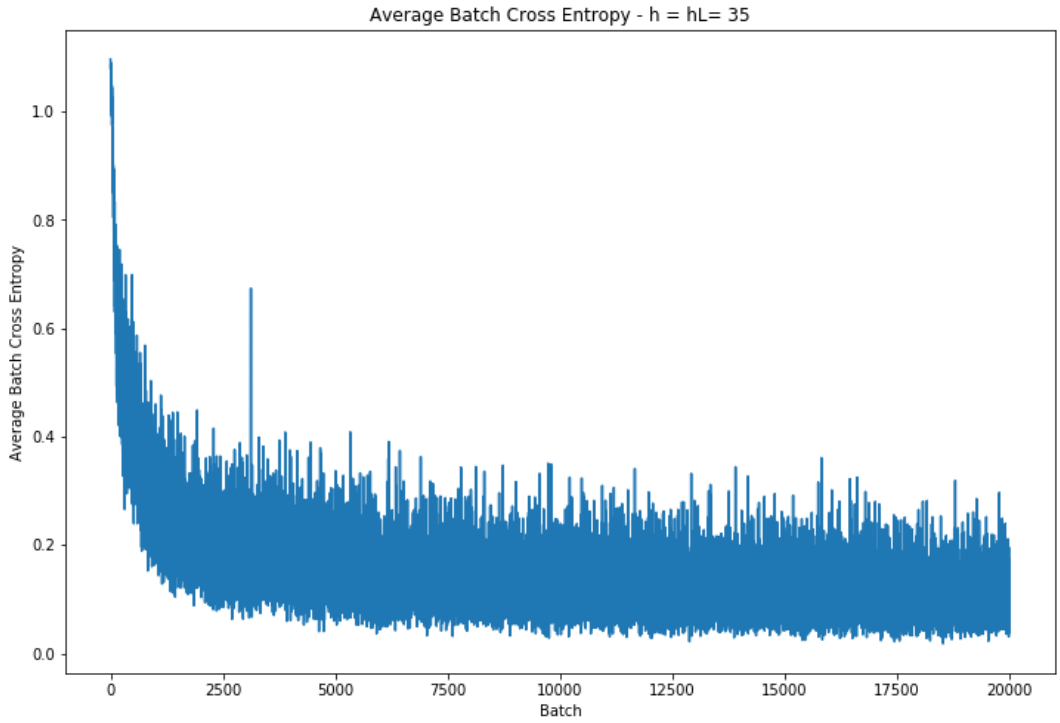



*Figure 6 Average batch cross entropy for h = h95 = 35*

**Comment:** The baCREn has a largest value at the beginning but decrease gradually but it oscillates frequently. From the around 10000$^{th}$ batch, the baCREn seem stables and very close to 0.

### *4.8. Compute and plot the curve*

|| W(n+1)- W(n)|| / ||Wn||

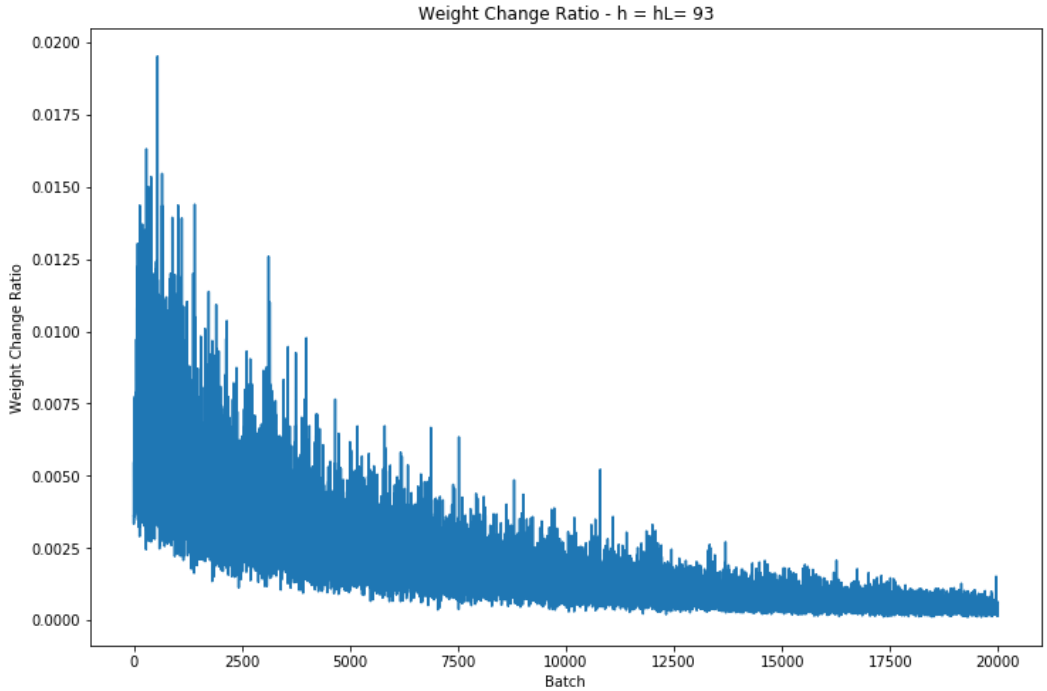



*Figure 7 Weight change ratio for each batch for h = 93*

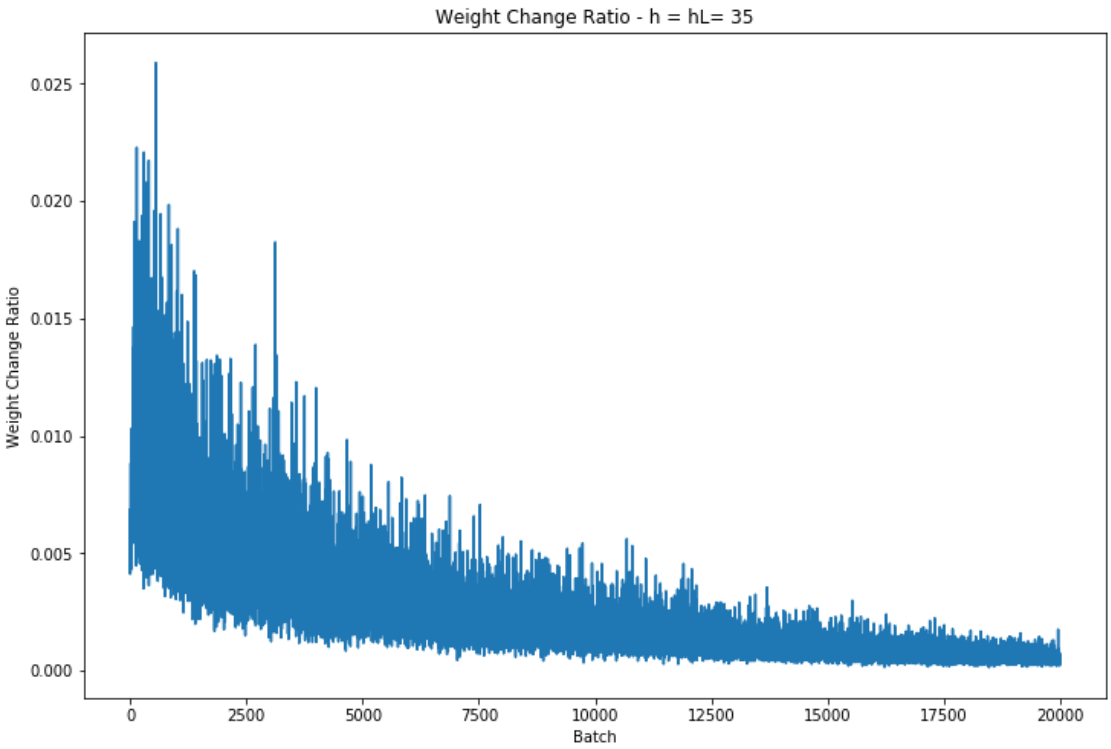



*Figure 8 Weight change ratio for each batch for h = 35*

**Comment:** At the beginning, the difference between matrices weights of 2 consecutive steps was large. As the batch learning increase, this difference was narrow down to below 0.001. It means that there is almost no change in weight matrix between 2 consecutive steps.

### *4.9. Compute the gradient norm ||Gn|| and dimenstion D of Gn*

For h=93, D= 17766
For h=35, D= 6688

### *4.10. Compute and plot the curve ||Gn||/d where d is the square root of D*

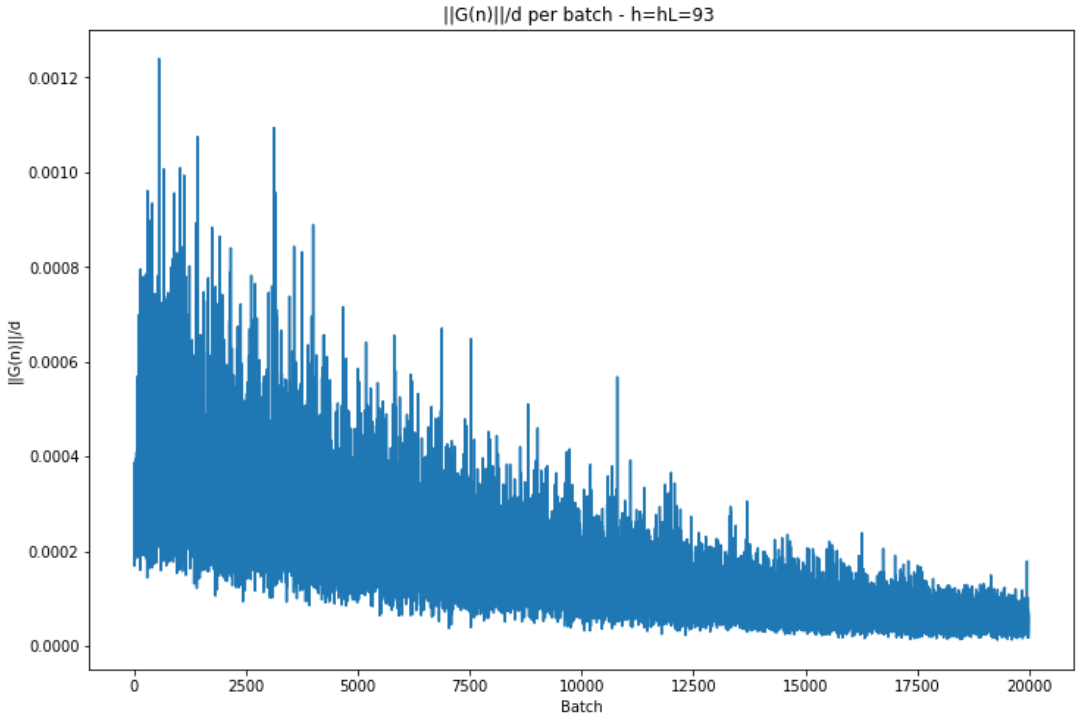



***Figure 9 ||Gn||/d for each batch for h = 93***

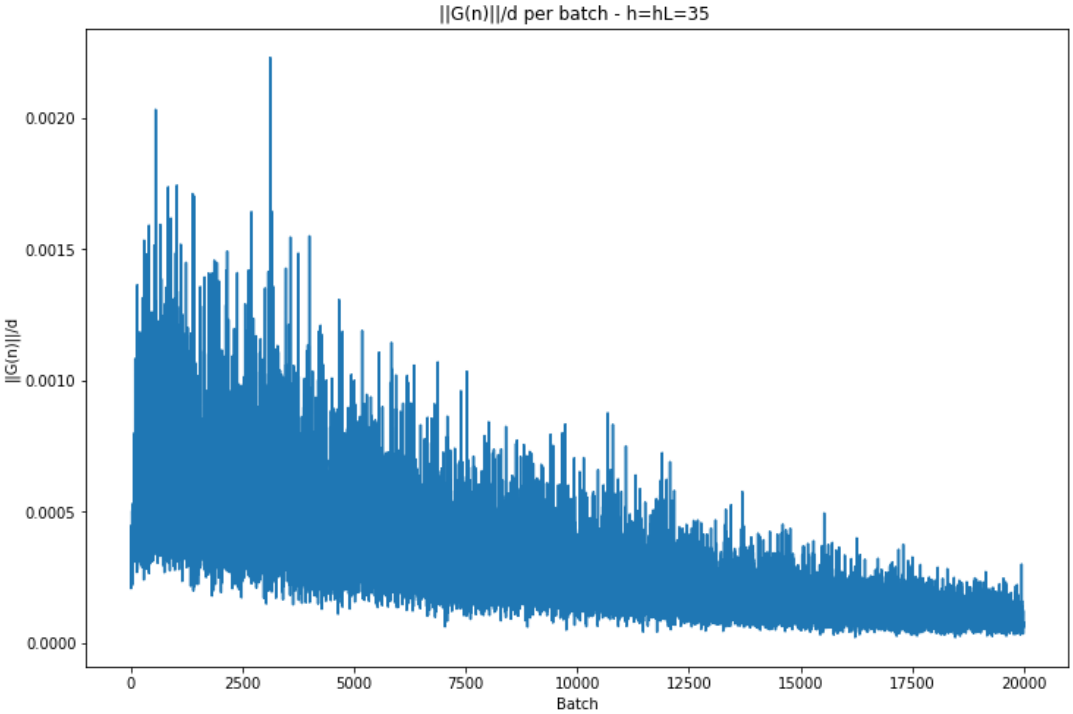



***Figure 10 ||Gn||/d for each batch for h = 35***

**Comment**: ||Gn||/d reduces gradually and oscillates continuously. The oscillating range also went down from 0.001 to 0.0001.

***Part 5: Performance Analysis***

In order to do the performance analysis, we optimized the parameters batch size, initialization method, and decay rate so we can look at the performance of two different hidden layer sizes, h95 and hL. These parameters were set as follows:

| | |
|---|---|
| Batch Size | 75 |
| Initialization Method | Random Normal (mean = 0, std. dev. = 0.5) |
| Decay Rate | Exp(-.02) |

*Table 5 Optimized parameters*

h95 and hL were found earlier by getting the number of features needed to explain 95% of variance in the entire dataset, h95, and by summation of the individual clusters, hL. Once we got these two values for our Hidden Layer size parameter, we ran the MLP on each to see the impact of the deep learning.

When doing the automatic learning, Keras keeps track of the training and validation set average Cross Entropy, or loss, and accuracy at the end of each Epoch, m. In our program, we ran a max number of Epochs at 100 in parallel to an early termination function that looked at the validation loss at each m and compared it to the previous 10. If there was very little change in the last ten epochs, it would kick out of the automatic learning and present our results. Doing it this way saved time and computing power. The code used for running this with Keras is as follows.

```
>stop_learn =  tf.keras.callbacks.EarlyStopping( patience=10, monitor='val_loss')
```

Note that patience is the parameter for saying how long you are willing to have no positive change go on for. Almost like a sliding window mean of the last 10 validation loss results and when that value becomes less than 0 it kicks out of the automatic learning. Also, keep in mind that an Epoch is made up of about 200 batches (N = 14,975, B = 75) which is saying that the MLP has learned 200 times with a random set of 75 cases, with replacement, from the training set.

### *5.1. For h = hL = 93*

Using the parameters displayed earlier and hidden layer size of 93, we ran the automatic learning and got the following line chart for training and validation, or test, accuracies at each epoch. The value for 0 was with the random initialization of weights and thresholds we designated at the start of the automatic learning. As you can see, the MLP learned the data well and fast. After the first 20 Epochs, you can see that the improvement in accuracy really starts to diminish and level out. Also, you can see that the automatic learning continued until it reached our max value of 100, meaning it was slowly still learning all the way to our max *m*, set at 100. So, in our case, $m^* = 100$. Keep in mind if we set max epochs to some arbitrarily large value, say 100,000 and we wait until the early termination function to kick in, then we may have a different value for $m^*$.

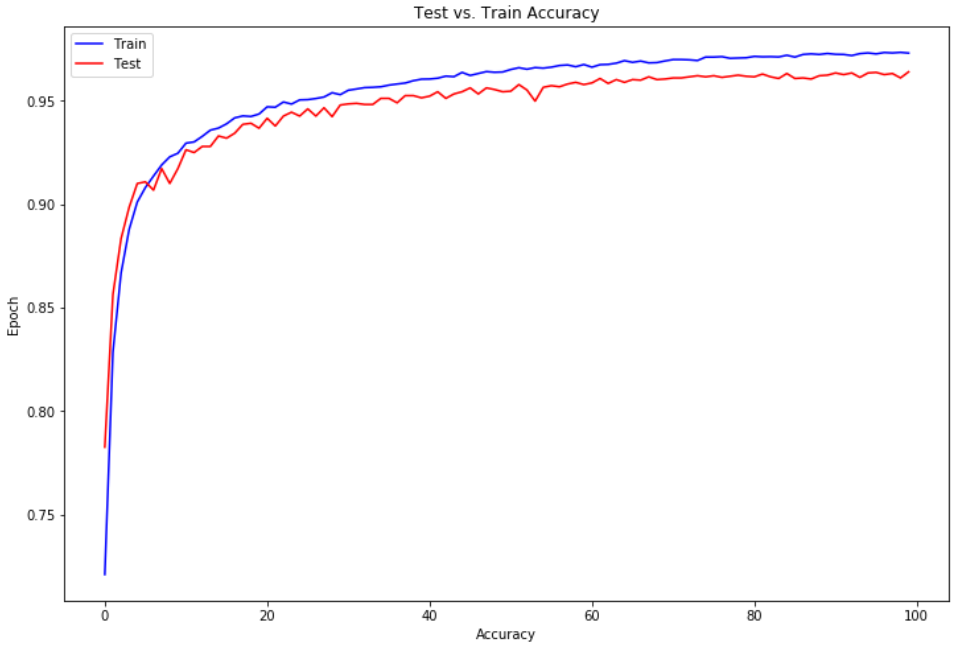



*Figure 11 Train vs Test accuracy for all epochs for h = 93*

Happy with our $m^* = 100$ and the relative accuracy, we continue to look at the percent confusion matrices for the training and test set. Shown below in the train set percent confusion matrix, you can see that it learned the data very well getting a total accuracy of about 97%. It has the hardest time classifying Cl2, or Ventricular arrhythmia, with a 96% accuracy which is still really good.

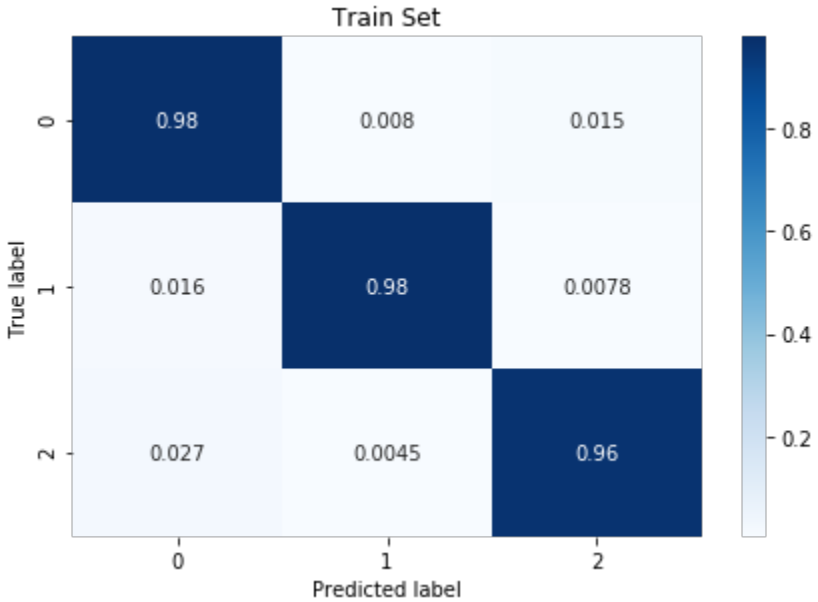



*Figure 12 Confusion matrix for train data for h =93*

Looking at the test set, we can see that our total accuracy dropped to about 96% which is really close to our training set. This makes me feel confident that we have a fairly robust and accurate MLP for classification.

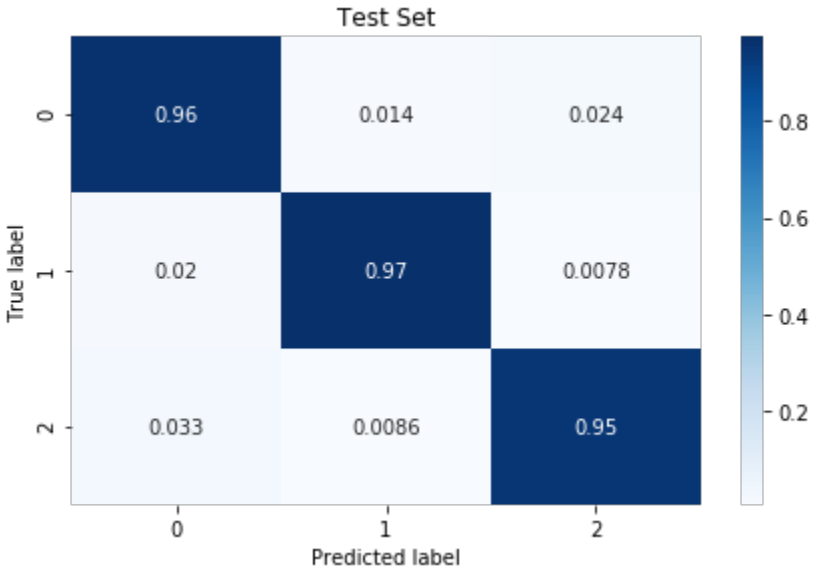



*Figure 13 Confusion matrix for test data for h =93*

### *5.2. For h = h95 = 35*

Doing the same automatic learning and retrieving the results with a Hidden Layer size of 35, h95, gets the line chart displayed below. Comparing this to the line chart for hL, you can see that there is larger oscillation in the test accuracy from start to finish. This could be due to the fact that there are too few neurons in the hidden layer and potentially the number of classes trained in an epoch. It also looks as though there is more separation between the training and test set as we move up in Epochs.

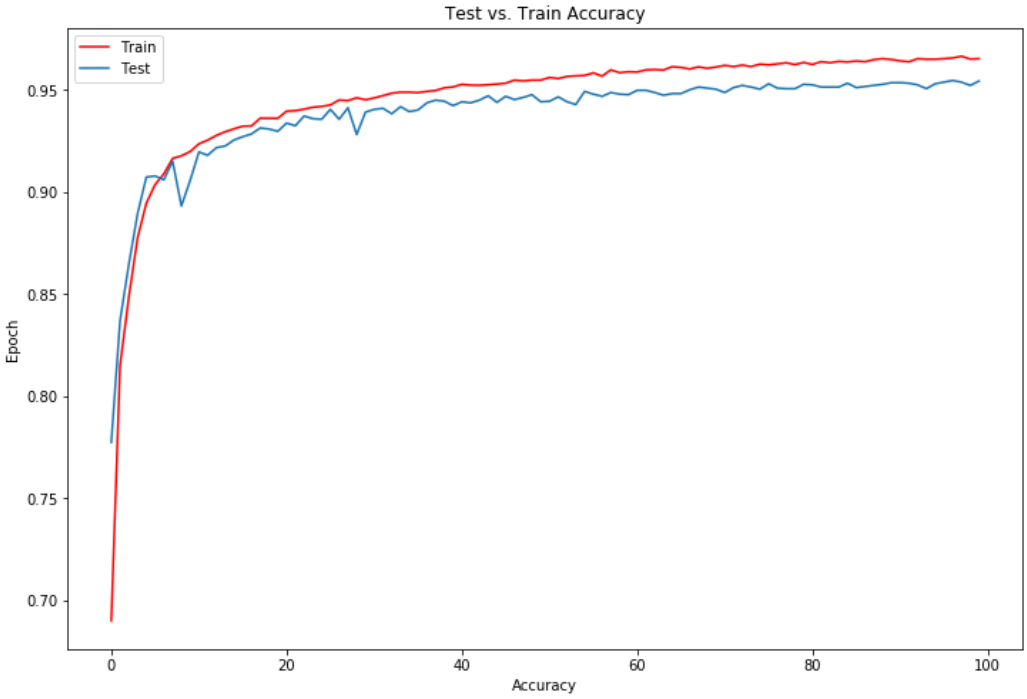



*Figure 14 Train and test accuracy for h = 35*

Looking at the training and test set percent confusion matrices displayed below, we can see that model is

still really accurate in classifying data it has not been trained on. The training set has about a 96% accuracy and the test set has about a 95% accuracy.

*Figure 15 Confusion matrix for Train data for h = 35*

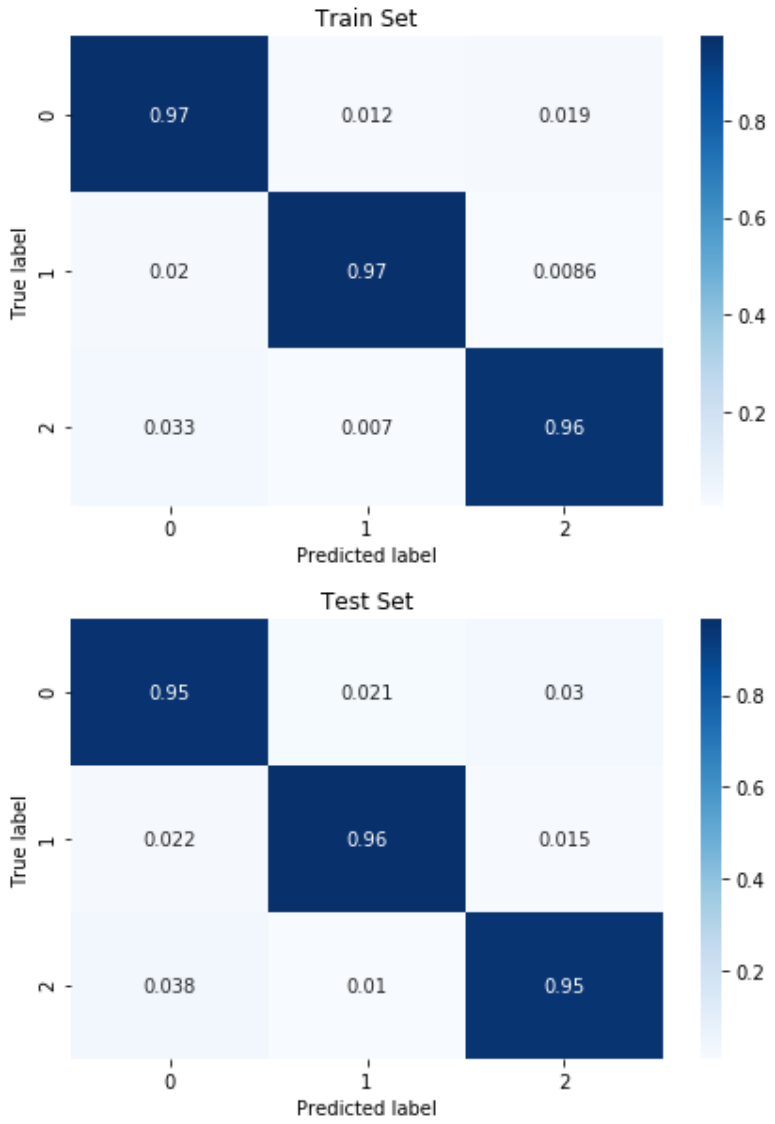



*Figure 16 Confusion matrix for Test data for h = 35*

### *5.3. Comparison of h95 and hL*

In comparing the two different models, Hidden Layers with h95 and hL, we really need to calculate confidence intervals based on the proportions. This is done by the following equation:

$$\hat{p} \pm 1.96 * \sqrt{\left(\frac{\hat{p} * (1 - \hat{p})}{n}\right)}$$

*Where n = number of cases and p = proportion of correct cases*

Looking at the upper bounds and lower bounds, UB and LB respectively, you can see that there is a significant difference, with alpha = 0.05, between the two different values of the Hidden Layer size parameter. hL lower bound is larger than h95 upper bound on both the training and test set. This means we will continue with the hidden layer investigation where the hidden layer is of size 93, hL.

**hL**

| | Class | Correct_prediction | Ncases | Accuracy | LB | UB |
|---|---|---|---|---|---|---|
| 0 | 0 | 5051 | 5200 | 0.971 | 0.966 | 0.976 |
| 1 | 1 | 5000 | 5145 | 0.972 | 0.967 | 0.977 |
| 2 | 2 | 4423 | 4630 | 0.955 | 0.949 | 0.961 |
| 3 | Total | 14474 | 14975 | 0.967 | 0.964 | 0.970 |

| | Class | Correct_prediction | Ncases | Accuracy | LB | UB |
|---|---|---|---|---|---|---|
| 0 | 0 | 5089 | 5200 | 0.979 | 0.975 | 0.983 |
| 1 | 1 | 5024 | 5145 | 0.976 | 0.972 | 0.980 |
| 2 | 2 | 4467 | 4630 | 0.965 | 0.960 | 0.970 |
| 3 | Total | 14580 | 14975 | 0.974 | 0.971 | 0.977 |

**h95**

| | Class | Correct_prediction | Ncases | Accuracy | LB | UB |
|---|---|---|---|---|---|---|
| 0 | 0 | 1238 | 1300 | 0.952 | 0.940 | 0.964 |
| 1 | 1 | 1240 | 1286 | 0.964 | 0.954 | 0.974 |
| 2 | 2 | 1095 | 1158 | 0.946 | 0.933 | 0.959 |
| 3 | Total | 3573 | 3744 | 0.954 | 0.947 | 0.961 |

| | Class | Correct_prediction | Ncases | Accuracy | LB | UB |
|---|---|---|---|---|---|---|
| 0 | 0 | 1254 | 1300 | 0.965 | 0.955 | 0.975 |
| 1 | 1 | 1251 | 1286 | 0.973 | 0.964 | 0.982 |
| 2 | 2 | 1104 | 1158 | 0.953 | 0.941 | 0.965 |
| 3 | Total | 3609 | 3744 | 0.964 | 0.958 | 0.970 |

*Table 6 Accuracy and confidence intervals for hL and h95 hidden layer neurons.*

***Part 6: Impact of various learning options***

In evaluating experimentally, the impact of the different parameters, or optimizing the model, we have run the model in a for loop with the different values I wanted to look at. The final performance we will be looking at is the loss value and the accuracy, which will be displayed in tables. We will also look at the loss value in a graph, to see the difference between the parameters. The reason we wanted to look at loss value is because you can see more of a difference in these values than in the accuracy values, which is possibly why TensorFlow uses the loss value for the validation set to determine whether to kick out of the automatic learning method. We went ahead and looked at the parameter h, or Hidden Layer Dimension as well to show the loss function, but did not include the accuracy since that is discussed thoroughly above.

The initial parameter values are displayed below. Note, we didn't include h because it is the first parameter we optimize. These values are replaced with the best values of the experimentation, or optimization. We chose the best value by picking the parameter that had the best validation set accuracy in its automatic learning.

| | |
|---|---|
| Batch Size | 100 |
| Initialization Method | Random Normal (mean = 0, std. dev. = 0.5) |
| Decay Rate | Exp(-0.4) |

*Table 7 Initial parameter values*

***6.1. Hidden Layer Dimension***

When I ran the optimization of this parameter, the early termination function kicked the automatic learning out after 48 epochs. The resulting best Hidden Layer Dimension was 93, which was also shown above. The tabular results are displayed below. The validation set accuracy and loss may be better in the h = 93, but the time is a little bit longer, this is because of there are more weights and biases for the extra neurons in the hidden layer which increases computation.

| | h= 35 | h= 93 |
|---|---|---|
| **Validation Set Accuracy** | 0.84 | 0.86 |
| **Training Set Loss (avCRE)** | 0.41 | 0.37 |
| **Time** | 14.62 | 15.30 |

*Table 8 Accuracy and loss for different hidden layer size*

The charts for loss that relate to the table above are displayed below. You can see that the loss value decreases the fastest with a hidden layer dimension of 93. Also, you can see that there are less epochs for h = 93, meaning it level out faster also.

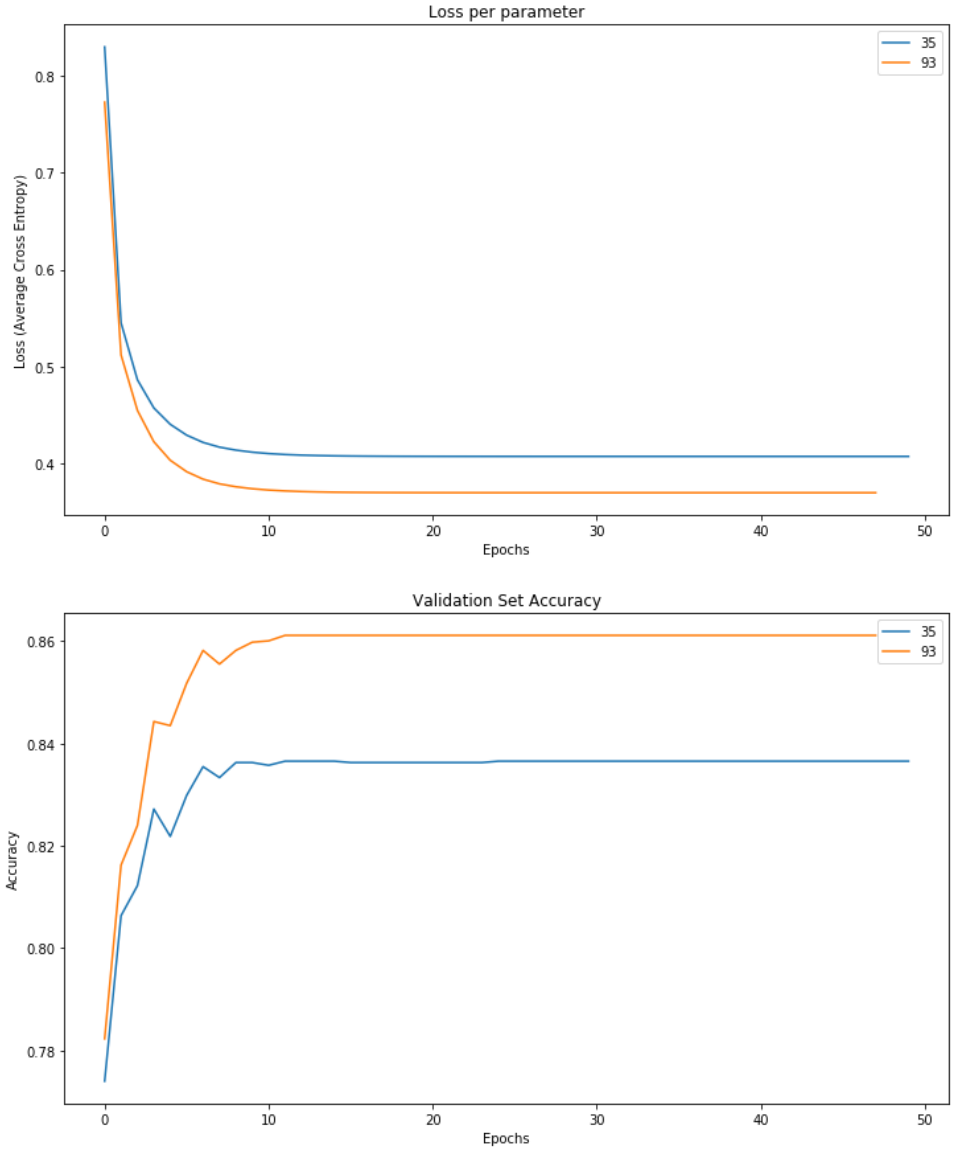



*Figure 17 Loss and Accuracy per epoch for test data for h = 35 and 93*

### *6.2. Batch Size*

Looking at the impact batch size has on the MLP, you can see from the table that the smaller the value, the more accurate the validation set and the smaller the loss, but the more time it takes. The conclusion we came to in this optimization is that a batch size of 75 is more accurate. It took 5 seconds longer to run a batch size of 75 as opposed to 100. If we had more cases and neurons in our hidden layer, this would increase much more. The batch size determines how many steps are needed in an epoch by saying the size

of a batch that the MLP will apply automatic learning to until it reaches an N equal to training set size. The smaller this number you have a higher potential for overfitting the training set. In this case we did not go to small, so we are okay.

| | B = 75 | B = 100 | B = 125 |
|---|---|---|---|
| **Validation Set Accuracy** | .89 | .86 | .84 |
| **Training Set Loss (avCRE)** | .32 | .37 | .41 |
| **Time** | 19.15 | 14.09 | 11.74 |

*Table 9 Accuracy and loss for different batch sizes*

You can see in the below graphs how much a difference batch size makes on the loss and validation accuracy. Also note the difference in epochs for batch size, it is interesting that 100 and 125 are about the same but 75 ran more epochs before kicking out.

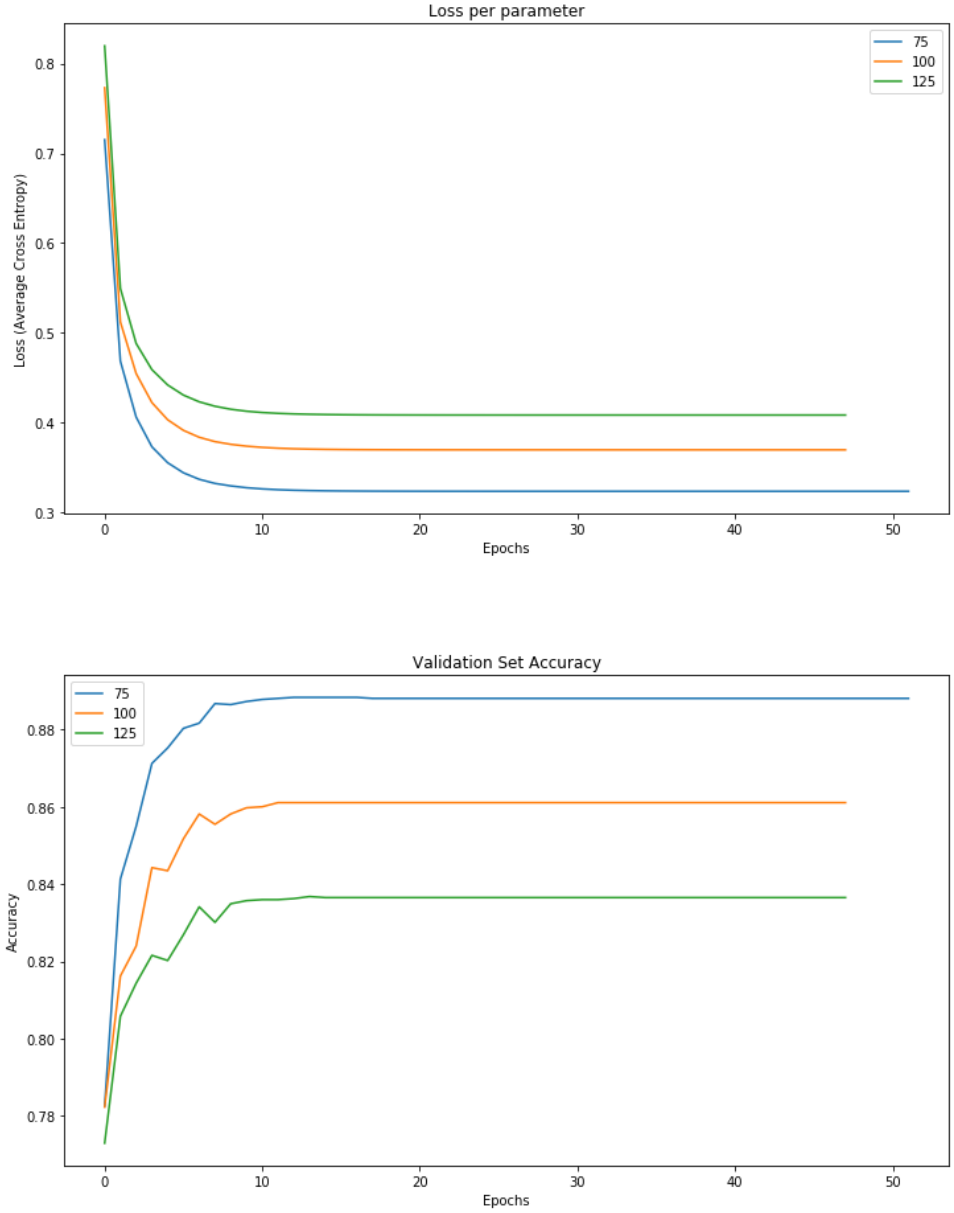



***Figure 18 Loss and Accuracy per epoch for test data for batch size = 75,100,125***

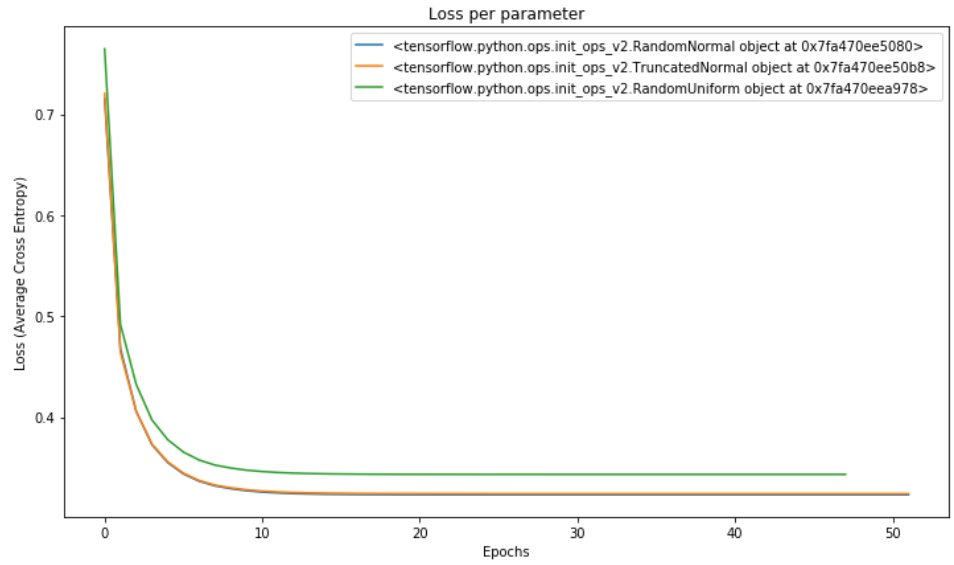
### *6.3. Initialization Method*

Looking at three different initialization methods for randomly selecting weights and biases, we get the following table. The difference between Random Normal and Truncated Random Normal is that Truncated Random Normal chops off everything greater than or less than 2*sigma from mean 0. If you look at the loss between the two Random Normal Methods, there is no difference, but there is a difference in the validation set accuracy. This is interesting and I am not 100% sure why this is.

| | **Init = Random Normal** | **Init = Truncated Random Normal** | **Init = Random Uniform** |
|---|---|---|---|
| **Validation Set Accuracy** | .89 | .88 | .88 |
| **Training Set Loss (avCRE)** | .32 | .32 | .34 |
| **Time** | 18.9 | 18.9 | 17.5 |

*Table* **10** *Accuracy and loss for different initialization values*

The line charts below iterate what was stated above about the table of results.



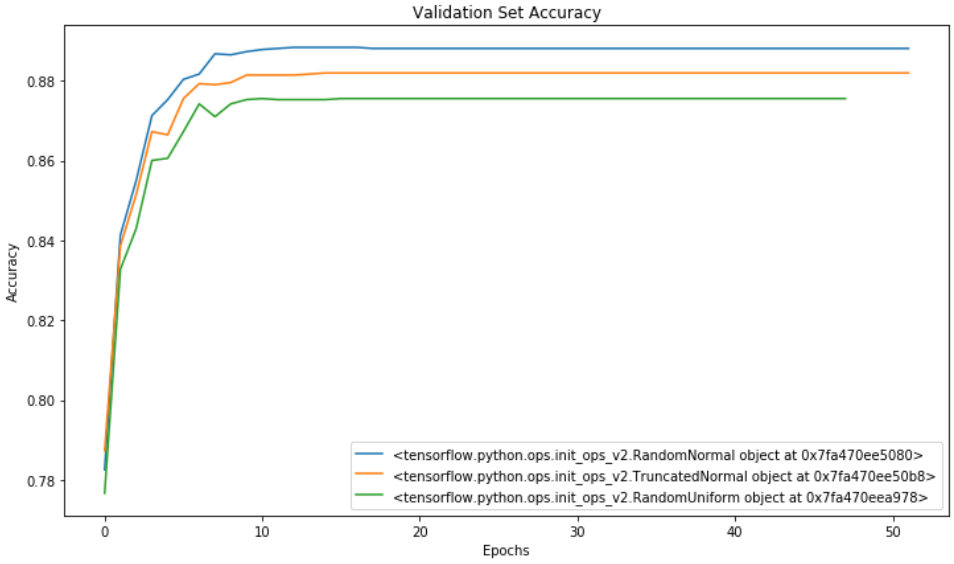



*Figure 19 Loss and Accuracy per epoch for test data for different initialization methods*

### *6.4. Decay Rate*

Looking at the table below for decay rates, you can see that as the decay rate approaches 1, the longer its rate of learning. The reason I say closer to one, is because exp(0) = 1, and the closer we get to exp(0) the better our results.

| | **decay = exp(-.02)** | **decay = exp(-.04)** | **decay = exp(-.06)** |
|---|---|---|---|
| **Validation Set Accuracy** | .96 | .95 | .94 |
| **Training Set Loss (avCRE)** | .09 | .12 | .14 |
| **Time** | 31.50 | 30.88 | 30.96 |

*Table 11 Accuracy and loss for different decay rates*

What is displayed in the table above really stands out in these graphs. You can see that the loss value really decreases quickly together until it starts to level out, with which they all level out at different rates. The closer to one the decay rate, the longer it takes to level out its Cross Entropy reduction, resulting in better accuracy.

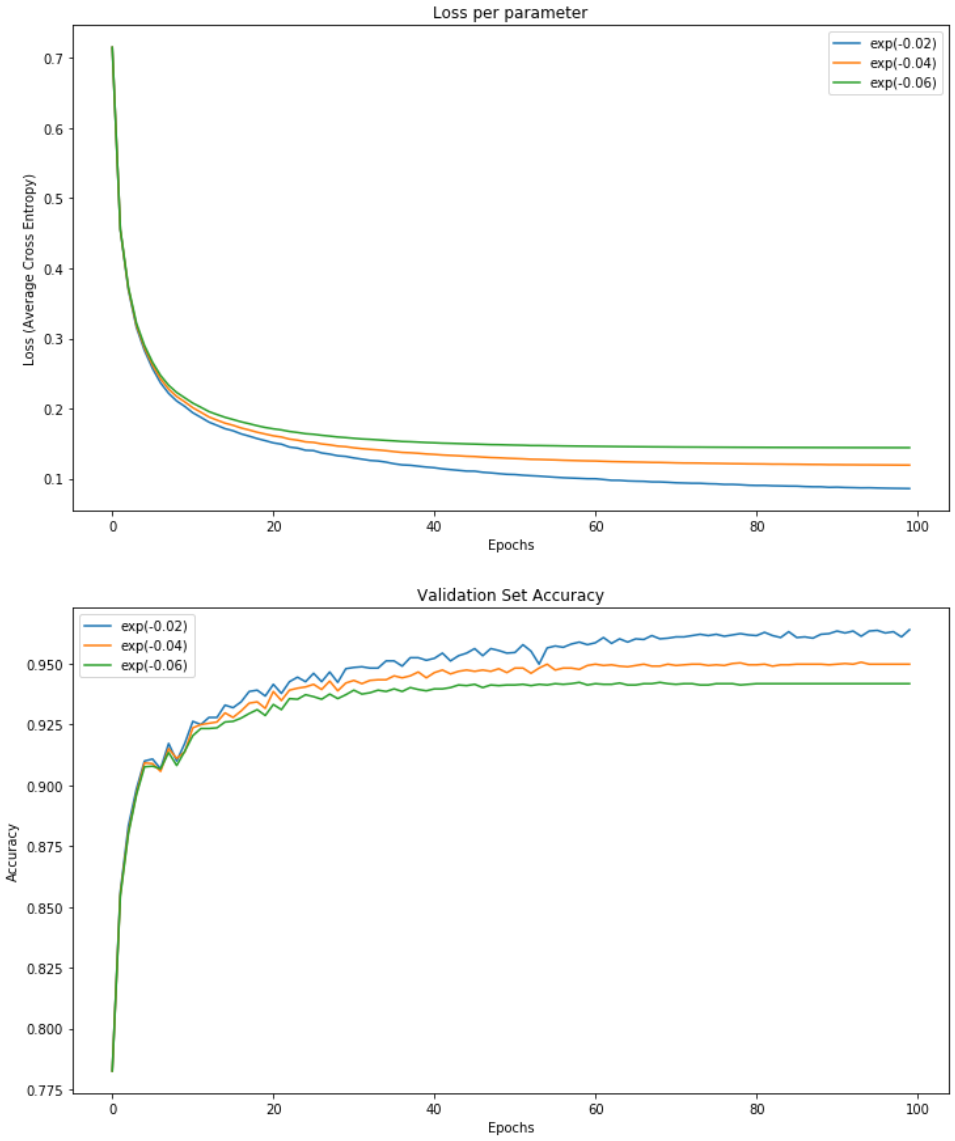



*Figure 20 Loss and Accuracy per epoch for test data for different decay rates*

***Part 7: Analysis of the hidden layer behavior after completion of automatic learning***

From our previous implementation, we learned that the best hidden layer dimension, h* = 93, and our best number of epochs, m* was 100. With the following parameters we ran the training set through the MLP and captured the hidden layers neurons activity for each case in N.

| | |
|---|---|
| m* | 100 |
| h* | 93 |
| Batch Size | 75 |
| Initialization Method | Random Normal (mean = 0, std. dev. = 0.5) |
| Decay Rate | Exp(-0.02) |

*Table 12 Optimized parameters for training MLP*

With this hidden layer state, we are going to examine the effect of the different cases on the neurons activity levels and hopefully find some neurons we can get rid of to improve robustness without losing much accuracy. We may also find some pattern to the neuron activity leading to a viable reason for a second hidden layer for increased learning and accuracy.

### *7.1. PCA on Hidden Layer*

The first step will be to apply PCA to the hidden layer and see how much variance is explained in the first three eigenvalues and then project the cases, split by cases, on those three principal components. In order to do this, we looked at a correlation matrix on the states of all the neurons for all the cases in the training set.

In looking data for the hidden layer activity to all of the cases, I noticed that there were 3 neurons that were 0 throughout all of the cases. This immediately tells me that I can get rid of those three neurons and it won't effect the model accuracy very much. It also tells me that I will have Null values in my correlation matrix, so I dropped those columns before creating the correlation matrix.

With the resulting 90 neurons I got the correlation matrix and looked at the neurons that had more than 0.90 rho, correlation, value. The table of these values are displayed below. There are 13 different pairs of neurons that had a high enough relationship to each other that we might consider getting rid of them and then creating another model and looking at the loss and accuracy to see if there was much of a difference.

From correlation matrix (correlations > .90)

| Correlation | Neurons |
|---|---|
| 0.918 | (18, 45) |
| 0.919 | (2, 33) |
| 0.925 | (50, 78) |
| 0.927 | (50, 69) |
| 0.930 | (41, 72) |
| 0.931 | (72, 80) |
| 0.932 | (18, 85) |
| 0.953 | (45, 85) |
| 0.959 | (73, 74) |
| 0.959 | (1, 60) |
| 0.984 | (11, 38) |
| 0.985 | (69, 78) |
| 0.986 | (41, 80) |

*Table 13 Hidden layer neuron pairs with correlation more than 0.90*

From here we continue on to getting the eigenvalues and looking at the ratio of variance explained for the first 3 principal components. As you can see in the following graph, 43% of the variance explained in the first 3 principal components. This makes me think we may be to get some good visualization in a 3d plot.

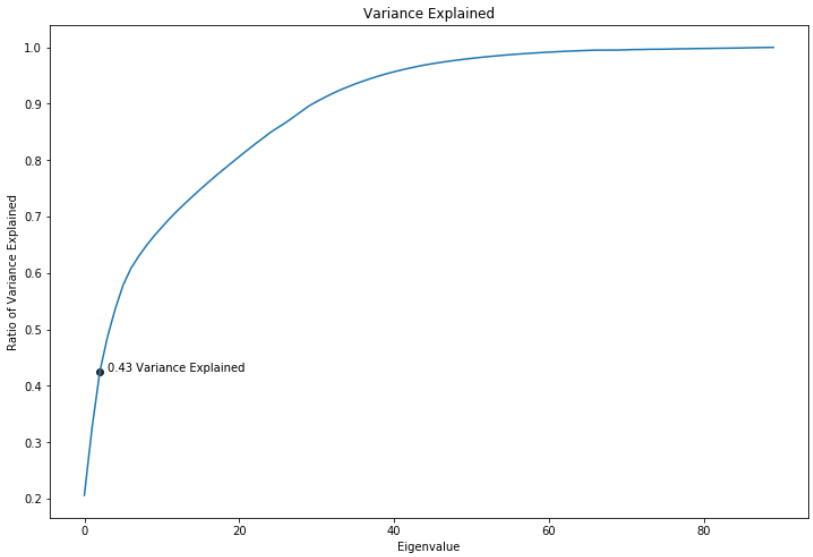



***Figure 21 Variance explained plot for PCA of hidden neurons***

Separating the cases by the different type of heart arrhythmias, we plotted all of the N cases on the first 3 principal components. You can see that the Normal Cluster is pretty tight and mostly grouped together. It also looks like that Ventricular has a descent clustering on the right, but a portion of the cases are also spread out between the other cases. Finally, it looks like the unclassified class has 3 possibly 4 separate clusters, but still seem separate from the other two clusters. This visualization matches the story the percent confusion matrix tells us with training set accuracy in decreasing size (Unclassified, Normal, and Ventricular).

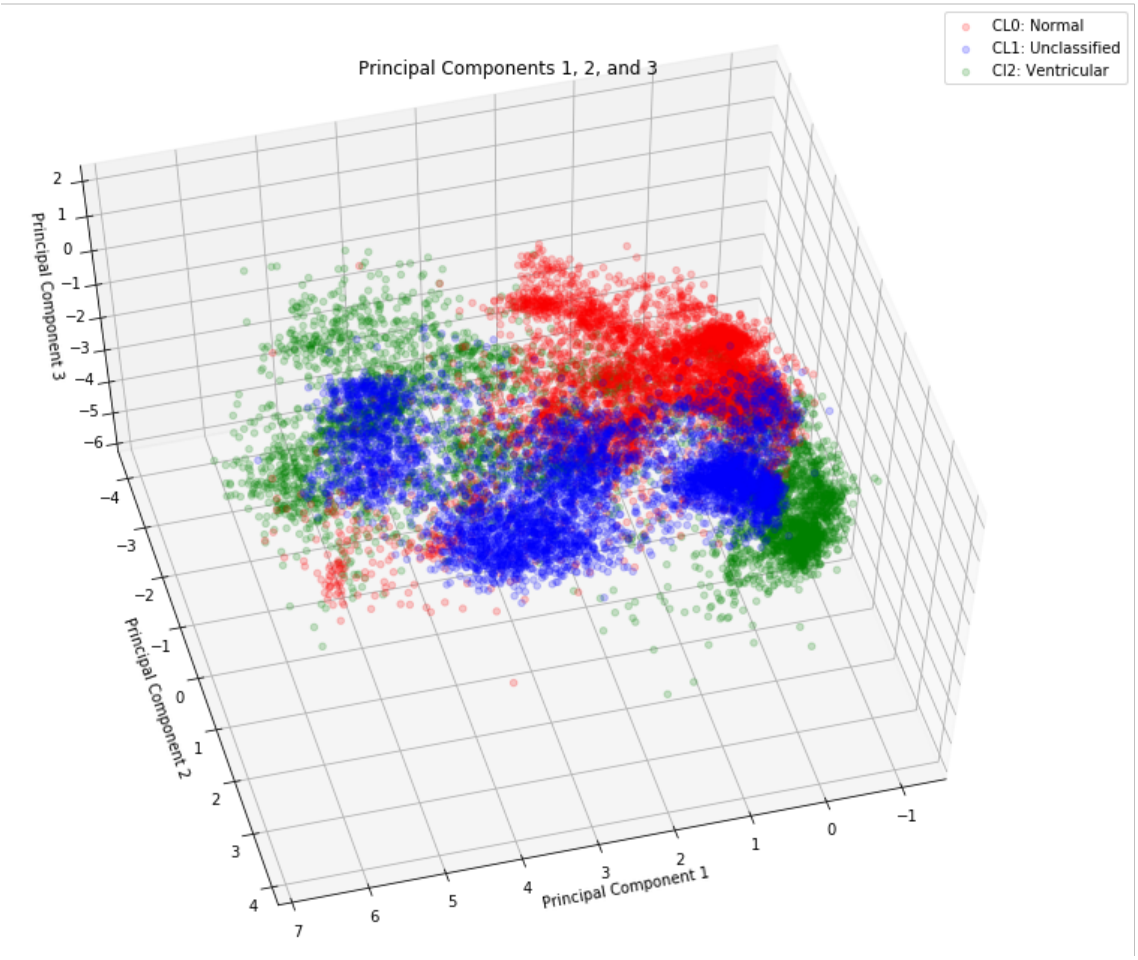



*Figure 22 3D visualization of PCA of hidden neurons*

***7.2. display and compare the average hidden neurons activity profiles PROF1, PROF2, PROF3 where PROFj is the average of the H(k) over all cases case(k) belonging to class j***

In this analysis, we have created profiles of hidden layer activity (h* = 93) for each class by calculating mean of all states of each hidden neuron for all cases in that particular class. The profiles are plotted in the following scatterplot:

1. profile 1 refers to CL0, which is class of normal cases
    a. If we consider threshold as 0.5 for average activity for good neurons, the plot shows that good neurons are concentrated in the range of 30 – 45 and 70 – 93
    b. Lazy neurons with very low activity (near zero) are concentrated in 0 – 20.
2. profile 2 refers to CL1, which is class of unclassifiable cases
    a. Good neurons have very high level of activity even above 1 in some neurons. They are concentrated in the range of 10 – 20 and 40 – 50.
    b. Lazy neurons are concentrated in the range of 0 – 10 and 30 – 40.
3. profile 3 refers to CL2, which is class of ventricular cases
    a. Good neurons in the range of 30 - 50 and 70 – 80.
    b. Many lazy neurons with near zero activity are found across the whole range, mainly clustered from 0 – 30, 55 – 70 and 80 – 90.

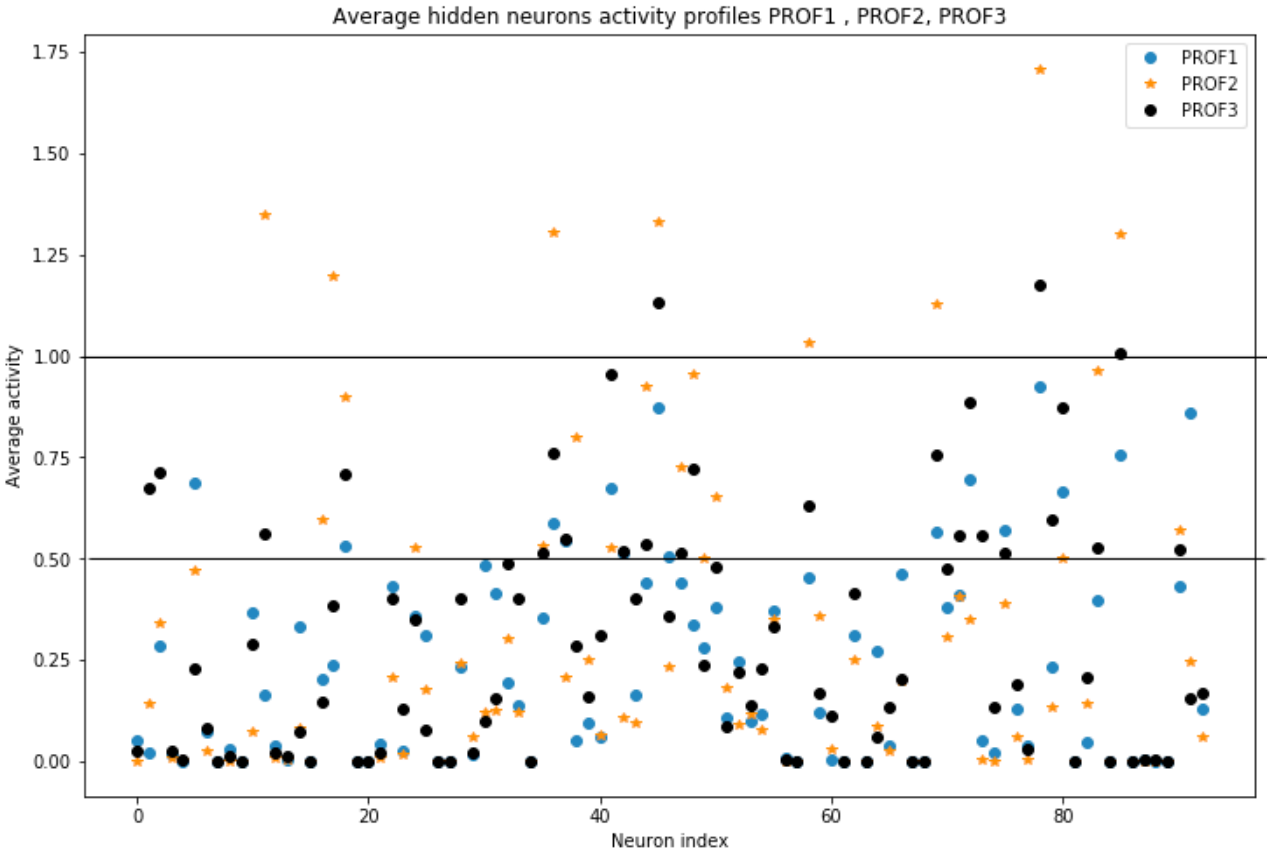



*Figure 23 Average hidden neurons activity profiles*

**Histogram Analysis:**

The histograms of three profiles indicate that majority of neuron activity is less than 0.2 in all three profiles. So we can see that high proportion of neurons are lazy ones in all three profiles.

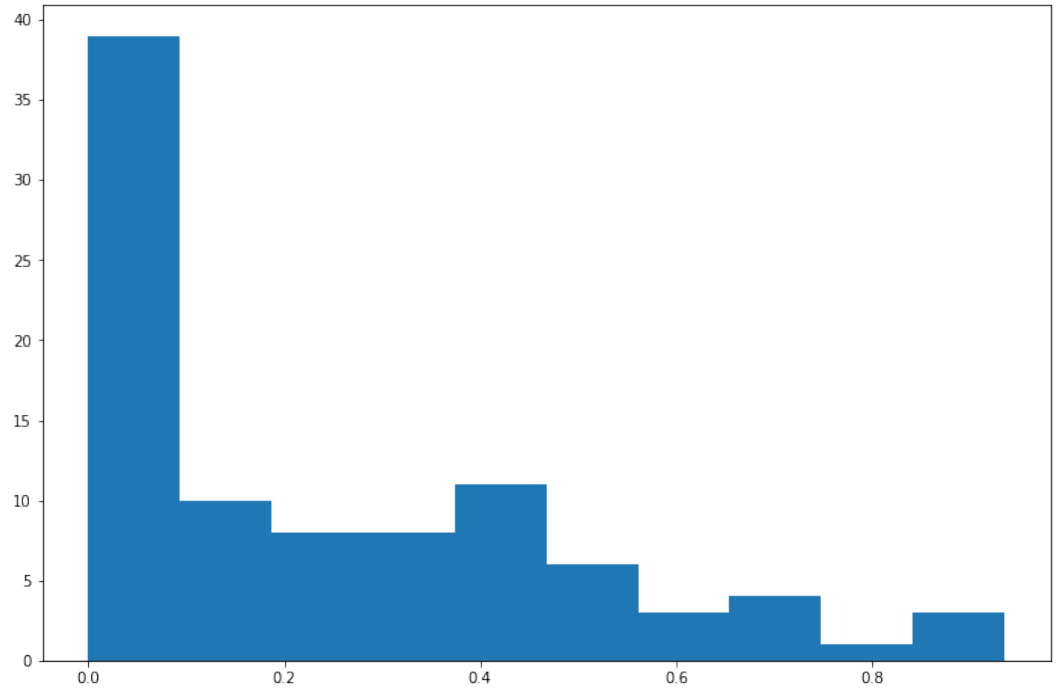



*Figure 24 Histogram for activity Profile 1 of hidden neurons*

In profile 1 histogram, maximum value is around 0.9 and as explained before 0.5 value was considered as threshold to identify good neurons.

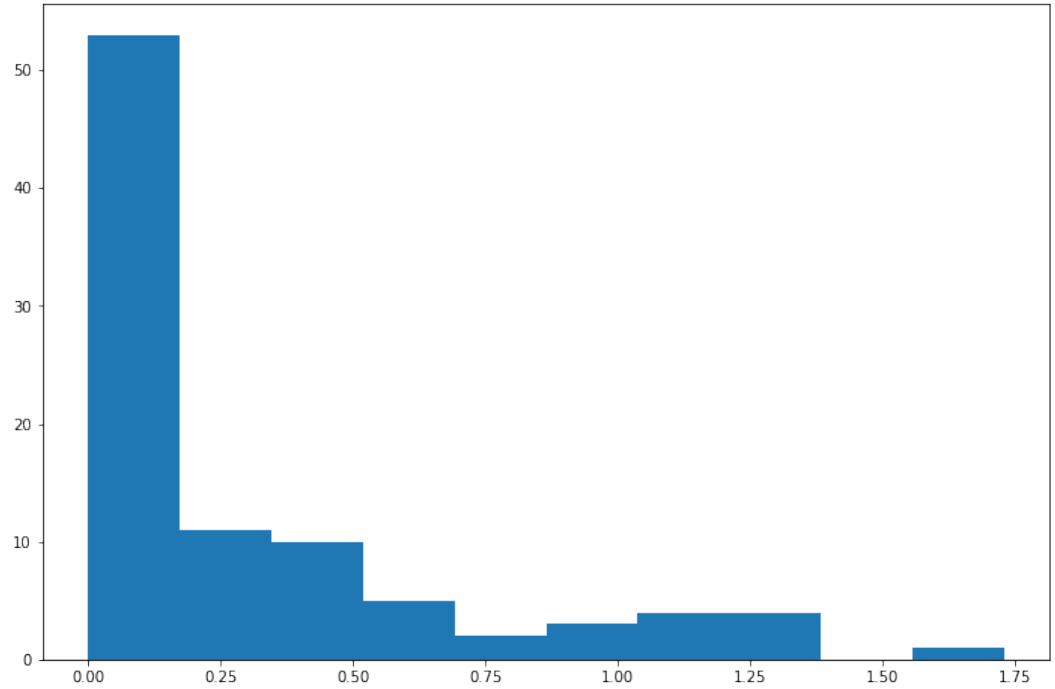



*Figure 25 Histogram for activity Profile 2 of hidden neurons*

In profile 2 histogram, range is larger than profile 1 and profile 3. Hence threshold for good neurons is considered as 1.0.

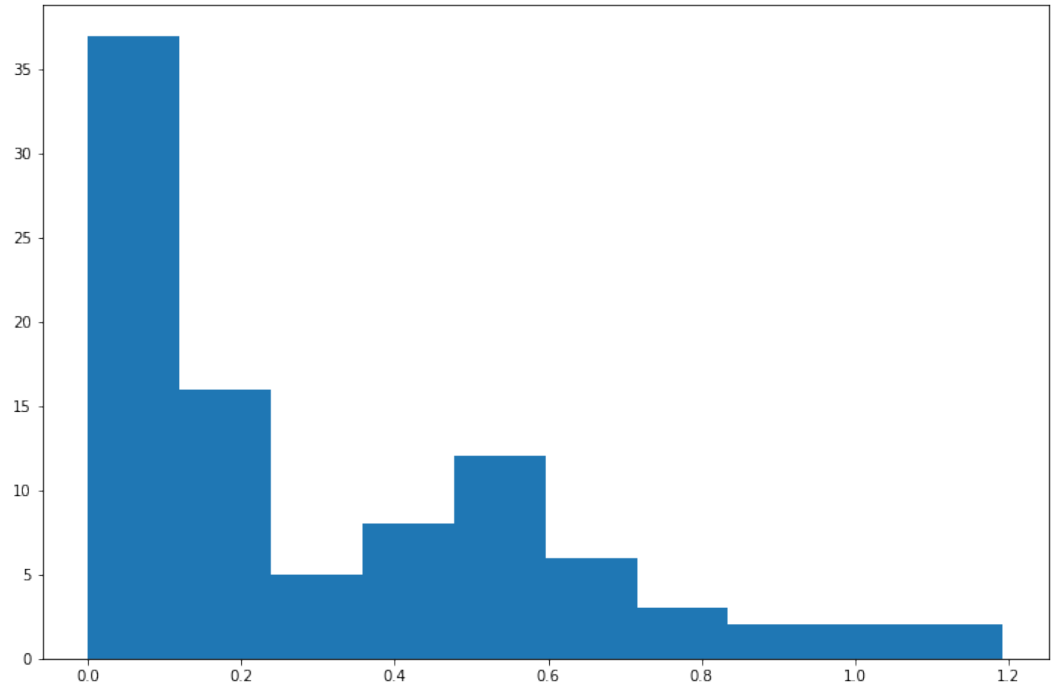



*Figure 26 Histogram for activity Profile 3 of hidden neurons*

Profile 3 histogram , unlike previous histograms, have two regions of high frequencies – one near 0.0 to 0.1 and second in 0.5-0.6.

Profiles of all 3 classes show that neuron activity for normal class (C0 – prof1) is medium level of activity across all neurons. But for unclassifiable class (C1 – prof2) neuron activity is much higher than remaining two classes. Further, for ventricular class(C2 – prof3) average activity of many neurons is near zero. The activity levels can be further explored by looking at Differentiation plots in next question.

***7.3. list the hidden neurons which achieve best DIFFERENTIATION between class C0 versus C1; - do the same for C1 versus C2, and then for C0 versus C3; interpret these results***

For evaluating hidden neurons for their discriminating capacity, normalized difference is calculated as follows:
For neuron "i" , differentiation between class 1 and 2,

$$dif_{12}(i) = \frac{\|MA_1 - MA_2\|}{AV_{12}}$$

Where, $MA_j - mean\ activity\ of\ neuron\ for\ cases\ in\ class\ j,$
$AV_{12} = \frac{MA_1 + MA_2}{2}$

The following table shows the top 5 neurons, that show best differentiation for three classes.

| S No | C0 vs C1 | | C1 vs C2 | | C0 vs C2 | |
|---|---|---|---|---|---|---|
| | neurons | $dif_{01}(i)$ | neurons | $dif_{12}(i)$ | neurons | $dif_{02}(i)$ |
| **1** | 9 | 2 | 57 | 2 | 67 | 2 |
| **2** | 7 | 2 | 9 | 2 | 84 | 2 |
| **3** | 20 | 2 | 63 | 2 | 19 | 2 |
| **4** | 26 | 2 | 26 | 2 | 9 | 1.94 |
| **5** | 63 | 2 | 68 | 2 | 1 | 1.88 |

*Table 14 Top 5 neurons with highest differentiation capacity for each pair of classes*

3 neurons (9,26,68) are common in C0 vs C1 and C1 vs C2. This indicates that these neurons are able to differentiate 3 classes effectively. Further some of these neurons are active neurons in two classes and lazy in other class. For example, 9 is lazy neuron in C0 and C2 but very active in C1 class.

Hence this analysis helps us to identify unique clustering characteristics of some neurons in the hidden layer, which can be useful for reducing hidden layer size.

## APPENDIX

### A.1.Tensorflow Keras code

**https://colab.research.google.com/drive/1EBqoI5ekf5FoPX2WmhRFeYptVppH564U**

```
# -*- coding: utf-8 -*-
"""Copy of Copy of MATH6373_HW2_Duong.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1EBqoI5ekf5FoPX2WmhRFeYptVppH564U
"""

# Commented out IPython magic to ensure Python compatibility.
# %tensorflow_version 2.x
#import packages
import pandas as pd
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

from numpy.random import seed
seed(6373)
tf.random.set_seed(6373)
from sklearn.decomposition import PCA

"""# 1. Data Exploration"""

#Read input file
df_train=np.loadtxt('https://raw.githubusercontent.com/duonghung86/ECG-
heartbeat/master/train.csv',
                    delimiter=",", dtype='float32',skiprows=1)
df_train[:5,:10]

df_test=np.loadtxt('https://raw.githubusercontent.com/duonghung86/ECG-
heartbeat/master/test.csv',
                   delimiter=",", dtype='float32',skiprows=1)
df_test[:5,:10]

# change the label to  0, 1, and 2
import seaborn as sns
y_train=df_train[:,187].copy()
y_train[y_train==4]=1
y_train=y_train.astype('int8')
#sns.distplot(y_train)
#plt.show()
#plt.hist(y_train)
#plt.show()
y_train[:10]

# apply the same thing for test set
y_test=df_test[:,187].copy()
y_test[y_test==4]=1
y_test=y_test.astype('int8')
#sns.distplot(y_test)
#plt.show()
#plt.hist(y_test)
#plt.show()
y_test[:10]
```

```
X_train=df_train[:,:187].copy()
print(df_train.shape)
print(X_train.shape)
X_test=df_test[:,:187].copy()
print(df_test.shape)
print(X_test.shape)

"""# 2. Define MLP structure

# 3. Select 2 tentative sizes h for the hidden layer

## 3.1. h95
"""

plt.rcParams['figure.figsize'] = (12,8)

from sklearn.preprocessing import StandardScaler as SS
standardize=SS()
scaled_X_train = standardize.fit_transform(X_train)
pca = PCA(0.95) # apply pca for 95%
pca.fit(scaled_X_train)
plt.plot(pca.explained_variance_ratio_)
plt.xlabel('j')
plt.ylabel('Eigen values')
plt.title('Eigen values')
print(sum(pca.explained_variance_ratio_))
h95 = pca.n_components_
print(h95)

#Verify
pca = PCA(187)
pca.fit(scaled_X_train)
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.hlines(0.95,0,187,'r')
plt.vlines(35,0.3,1,'r')
plt.xlabel('j')
plt.ylabel('Rj')
plt.title('Rj')
plt.show()

"""## 3.2 hL"""

hL=0
for i in range(3):
  Mj=scaled_X_train[y_train==i,:].copy()
  standardize=SS()
  Mj = standardize.fit_transform(Mj)

  pca_Mij=PCA(0.95)
  pca_Mij.fit(Mj)
  hL+=pca_Mij.n_components_
  print(hL)

hL= 93
#for i in range(3):
#  pca_Mij=PCA(0.95)
#  pca_Mij.fit(X_train[y_train==i,:])
#  hL+=pca_Mij.n_components_
#  print(hL)
hL

"""# 5. Impact of various learning options
```

```
-dimension h  of H
-batch size
-initialization
-gradient descent step size
"""

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.initializers import RandomNormal, RandomUniform, TruncatedNormal
from tensorflow.keras import optimizers, losses
from tensorflow.keras import callbacks
from time import time

#Just for running the def Model_Search() since I set these values equal to values
h = 35
Initializer = RandomNormal(mean=0.0, stddev=0.05, seed=6373)
B = 100
decay_rate = 0.4

### Use your own layer size and initializers. Refer to https://keras.io/initializers/.
####
def Model_Search(X_train, y_train, X_test, y_test, layerH_dim = h, Initializer =
Initializer, B = B, decay_rate = decay_rate):

  #Build the Model for Keras
  layer1_dim=X_train.shape[1]
  layerH_dim=layerH_dim
  layer2_dim=3

  #D = layer1_dim*layerH_dim + layerH_dim + layerH_dim*layer2_dim + layer2_dim

  # Setup the layer
  model = Sequential()

  # From layer 1 to layer H, we use RELU
  # the initial Weights and Thresholds are kernel and bias
  # Random normal = random from normal distribution (mean=0.0, stddev=0.05, seed=None)
  model.add(Dense(layerH_dim, activation='relu', input_dim=layer1_dim,
                kernel_initializer=Initializer,
                bias_initializer=Initializer))

  # From layer H to 2, then softmax
  model.add(Dense(layer2_dim, activation='softmax',
                kernel_initializer=Initializer,
                bias_initializer=Initializer))

  ###### Configure your own learning rate schedule and mini-batch size. #########

  # Choose batch size
  N=X_train.shape[0] # number of cases in train set
  # Our task is classification and we use cross entropy so
  loss_func = losses.CategoricalCrossentropy()

  # Define learning rate
  lr_schedule = optimizers.schedules.ExponentialDecay(initial_learning_rate=0.1,
                                                      decay_steps=N/B,
                                                      decay_rate=np.exp(-decay_rate),
                                                      staircase=True)

  # Because of using batch, Stochastic Gradient Descent optimizer is a must
  model.compile(optimizer=optimizers.SGD(learning_rate=lr_schedule),
              loss=loss_func,
```

```
              metrics=['accuracy'])

  # For saving the best model during the whole training process.
  checkpointer = callbacks.ModelCheckpoint(filepath='BestModel.h5', monitor='val_loss',
save_best_only=True)

  #### Interrupt training if `val_loss` stops improving for over 10 epochs #######
  stop_learn= tf.keras.callbacks.EarlyStopping(patience=10, monitor='val_loss')
  # Save the weight at the end of each epoch
  #weight_record = tf.keras.callbacks.ModelCheckpoint('weights{epoch:08d}.h5',
                                    # save_weights_only=True)
  epo=100

  # Fit the model
  Monitor = model.fit(X_train, y_train,
                      epochs=epo,
                      batch_size=B,
                      callbacks=[checkpointer,stop_learn],
                      validation_data=(X_test, y_test),
                      verbose = 0
                      )

  return Monitor.history['loss'], Monitor.history['val_loss'],
Monitor.history['accuracy'], Monitor.history['val_accuracy']

#plot
#plot
def plot(results):
  keys = results.keys()

  #Plot the plots
  count = 0
  max_acc = 0
  fig, ax = plt.subplots(2,1, figsize = (12,15))
  for i in keys:
    ax[0].plot(results[i]['loss'])
    ax[1].plot(results[i]['val_accuracy'])
  ax[0].set_xlabel('Epochs')
  ax[0].set_ylabel('Loss (Average Cross Entropy)')
  ax[0].set_title('Loss per parameter')
  ax[0].legend(['{}'.format(i) for i in results.keys()])
  ax[1].set_xlabel('Epochs')
  ax[1].set_ylabel('Accuracy')
  ax[1].set_title('Validation Set Accuracy')
  ax[1].legend(['{}'.format(i) for i in results.keys()])

  for i in results.keys():
    #Get the best key (using the highest accuracy)
    key_max = max(results[i]['val_accuracy'])
    if key_max > max_acc:
      max_acc = key_max
      best_key = i

  for i in keys:
    print(min(results[i]['loss']), max(results[i]['val_accuracy']))

  plt.show()
  return best_key

"""#### Get the Y values in proper format"""

# Convert labels to one hot encodings
y_train=tf.keras.utils.to_categorical(y_train)
```

```
y_test=tf.keras.utils.to_categorical(y_test)

"""### Hidden Layer Dimensions:"""

h=[35, 93]

h_results = {}
delta_times = []
for i in h:
  a = time()
  loss, val_loss, accuracy, val_accuracy = Model_Search(X_train, y_train, X_test,
y_test, layerH_dim= i)
  h_results[i] = {'loss' : loss,
                  'val_loss': val_loss,
                  'accuracy': accuracy,
                  'val_accuracy': val_accuracy}
  delta_times += [time()-a]

best_h = plot(h_results)
print('Number of Epochs: {}\nBest Hidden Layer Dimension:
{}'.format(len(h_results[i]['loss']), best_h))
print(delta_times)

##Run 1 for comparison of seed
#h_results[93]['loss']

##Run 2 for comparison of see
#h_results[93]['loss']

"""### Batch Size
-best Hidden Layer Dimension = 93 (50 Epochs before kicking out)
"""

#Play with Batch Values
B=[75, 100, 125]

B_results = {}
delta_times = []
for i in B:
  a = time()
  loss, val_loss, accuracy, val_accuracy = Model_Search(X_train, y_train, X_test,
y_test, layerH_dim=best_h, B = i)
  B_results[i] = {'loss' : loss,
                  'val_loss': val_loss,
                  'accuracy': accuracy,
                  'val_accuracy': val_accuracy}
  delta_times += [time()-a]

best_B = plot(B_results)
print('Number of Epochs: {}\nBest Number of Batches Dimension:
{}'.format(len(B_results[i]['loss']), best_B))
print(delta_times)

print('Number of Epochs: {}\nBest Number of Batches Dimension:
{}'.format(len(B_results[i]['loss']), best_B))
print(delta_times)

"""### Best Initilization Results
-best Hidden Layer Dimension = 93  (49 Epochs before kicking out)
-best batch values = 75 (with 51 Epochs before kicking out)
"""

#Play with Initialization Methods
```

```
Initializer = [RandomNormal(mean=0.0, stddev=0.05, seed=6373),
               TruncatedNormal(mean=0.0, stddev=0.05, seed=6373),
               RandomUniform(minval=-0.05, maxval=0.05, seed=6373)]

Init_results = {}
delta_times = []
for i in Initializer:
  a = time()
  loss, val_loss, accuracy, val_accuracy = Model_Search(X_train, y_train, X_test,
y_test, layerH_dim=best_h, B = best_B, Initializer=i)
  Init_results[i] = {'loss' : loss,
                     'val_loss': val_loss,
                     'accuracy': accuracy,
                     'val_accuracy': val_accuracy}
  delta_times += [time() - a]

best_Init = plot(Init_results)
print('Number of Epochs: {}\nBest Initialization Method Dimension:
{}'.format(len(Init_results[i]['loss']), best_Init))
print(delta_times)

"""### Best Decay Rate Results
-best Hidden Layer Dimension = 93  (49 Epochs before kicking out)
-best batch values = 75 (with 51 Epochs before kicking out)
-best initialization method for params = RandomNormal (53 Epochs before kicking out)
"""

#Play with Decay Rates
decay_rate = [.02, .04, .06]

decay_results = {}
delta_times = []
for i in decay_rate:
  a = time()
  loss, val_loss, accuracy, val_accuracy = Model_Search(X_train, y_train, X_test,
y_test, layerH_dim=best_h, B = best_B, Initializer=best_Init, decay_rate = i)
  decay_results[i] = {'loss' : loss,
                     'val_loss': val_loss,
                     'accuracy': accuracy,
                     'val_accuracy': val_accuracy}
  delta_times += [time() - a]

best_decay = plot(decay_results)
print('Number of Epochs: {}\nBest Decay Rate Dimension:
{}'.format(len(decay_results[i]['loss']), best_decay))
print(delta_times)

"""### Best Parameters
-best Hidden Layer Dimension = 93  (49 Epochs before kicking out)
-best batch values = 75 (with 51 Epochs before kicking out)
-best initialization method for params = RandomNormal (53 Epochs before kicking out)
-best decay rate = 0.02 (Using all 100 Epochs)

# 4. Implementation

### Using the best results showed later in the search pattern
"""

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.initializers import RandomNormal, RandomUniform, TruncatedNormal

# Use your own layer size and initializers. Refer to https://keras.io/initializers/.
```

```
layer1_dim=X_train.shape[1]
layerH_dim=best_h
#layerH_dim=35

layer2_dim=3

D = layer1_dim*layerH_dim + layerH_dim + layerH_dim*layer2_dim + layer2_dim

# Setup the layer
model = Sequential()

#Edit the initializers
Initializer = best_Init

# From layer 1 to layer H, we use RELU
# the initial Weights and Thresholds are kernel and bias
# Random normal = random from normal distribution (mean=0.0, stddev=0.05, seed=None)
model.add(Dense(layerH_dim, activation='relu', input_dim=layer1_dim,
                kernel_initializer=Initializer,
                bias_initializer=Initializer))

# From layer H to 2, then softmax
model.add(Dense(layer2_dim, activation='softmax',
                kernel_initializer=Initializer,
                bias_initializer=Initializer))

"""### Compile the model

Before the model is ready for training, it needs a few more settings. These are added
during the model's *compile* step:

- *Loss function* —This measures how accurate the model is during training. You want to
minimize this function to "steer" the model in the right direction.
- *Optimizer* —This is how the model is updated based on the data it sees and its loss
function.
- *Metrics* —Used to monitor the training and testing steps. The following example uses
*accuracy*, the fraction of the images that are correctly classified.

#### learning rate with exponetial decay

- initial_learning_rate: A scalar float32 or float64 Tensor or a Python number. The
initial learning rate.
- decay_steps: A scalar int32 or int64 Tensor or a Python number. Must be positive. See
the decay computation above.
- decay_rate: A scalar float32 or float64 Tensor or a Python number. The decay rate.
- staircase: Boolean. If True decay the learning rate at discrete intervals
name: String. Optional name of the operation. Defaults to 'ExponentialDecay'.
"""

from tensorflow.keras import optimizers, losses

# Configure your own learning rate schedule and mini-batch size.

# Choose batch size
B = best_B
N=X_train.shape[0] # number of cases in train set
decay_rate = np.exp(-best_decay)
# Our task is classification and we use cross entropy so
loss_func = losses.CategoricalCrossentropy()

# Define learning rate
lr_schedule = optimizers.schedules.ExponentialDecay(initial_learning_rate=0.1,
                                                    decay_steps=N/B,
```

```
                                                    decay_rate=decay_rate,
                                                    staircase=True)

# Because of using batch, Stochastic Gradient Descent optimizer is a must
model.compile(optimizer=optimizers.SGD(learning_rate=lr_schedule),
              loss=loss_func,
              metrics=['accuracy'])

"""We will choose metrics later"""

from tensorflow.keras import callbacks

# For recording gradients and training loss over the whole training set
# at the end of each epoch.

class MyHistory(callbacks.Callback):
    # Set empty list for values we want to retrieve at the begining
    def on_train_begin(self, logs={}):
        self.grads = []
        self.gradsNorm = []
        self.train_loss = []
        self.test_loss = []
        self.weight=[]
        self.WC = []
        #Track some of the batch values
        self.bW = []
        self.b_WC = [] #change in weight size for every batch (page 3)
        self.bavCRE = [] #loss for each batch (page 3)
        self.bG = [] # batch Gradiend Descent (end of page 2)
        self.bGD = [] # Wants Size of batch gradient plotted each batch (page 3)

    # Define what values to keep at the end of each batch
    def on_batch_end(self, batch, logs):

        #Get the Batch Average CRE (Loss)
        self.bavCRE.append(logs.get('loss')) #plot

        # Weight
        modelWeights = [] # Vector W at the batch n
        for layer in model.layers:
          layerWeights = []
          # Retrieve the weight and threshold from each layer
          for weight in layer.get_weights():
            layerWeights=np.concatenate([layerWeights,weight.flatten()])
          modelWeights=np.concatenate([modelWeights,layerWeights])
        # Store it
        self.bW.append(modelWeights)

        if len(self.bW)>2:
          diff = (self.bW[-1]- self.bW[-2])
          self.bG.append( diff / decay_rate )
          self.b_WC.append( np.linalg.norm(diff)/np.linalg.norm(self.bW[-1]) ) #Plot
          self.bGD.append( np.linalg.norm(self.bG[-1]) / np.sqrt(D) ) #plot

    # Define what values we want to retrieve at the ending of epoch
    def on_epoch_end(self, epoch, logs={}):

        # Define gradient or Gi
        with tf.GradientTape() as tape:
          #train set
          y_pred = self.model(X_train)
          loss_value = loss_func(y_train,y_pred) # avCRE
          #test set
```

```
            test_y_pred = self.model(X_test)
            test_loss_value = loss_func(y_test, test_y_pred)

        #WHAT IS THIS DOING???
        grad = tape.gradient(loss_value, model.trainable_weights) # gradient(avCRE,W)
        self.grads.append(grad)

        self.gradsNorm.append(np.sqrt(sum([np.sum(np.square(g.numpy())) for g in
grad])))

        # Weight
        modelWeights = [] # Vector W at the Epoch n
        for layer in model.layers:
          layerWeights = []
          # Retrieve the weight and threshold from each layer
          for weight in layer.get_weights():
            layerWeights=np.concatenate([layerWeights,weight.flatten()])
          modelWeights=np.concatenate([modelWeights,layerWeights])
        # Store it
        self.weight.append(modelWeights)

        #Get the value of loss and store it
        self.train_loss.append(loss_value.numpy())
        self.test_loss.append(test_loss_value.numpy())

GradsReal_loss = MyHistory()

# For saving the best model during the whole training process.

checkpointer = callbacks.ModelCheckpoint(filepath='BestModel.h5', monitor='val_loss',
save_best_only=True)

"""- Train the model and store validation loss history in 'Monitor'.
- Be carefual that in Keras, the "loss" of each epoch (stored in
'Monitor.history['loss']') is calculated as the average of the losses of mini-batches in
that epoch - bavCREn.
- Vector Gn is stored in GradsReal_loss.grads
- For the real epoch loss over the whole training set, use 'GradsReal_loss.real_loss'
instead. This is avCRE
- But it is safe to use 'Monitor.history['val_loss']' for validation losses.
"""

# Interrupt training if `val_loss` stops improving for over 10 epochs
stop_learn= tf.keras.callbacks.EarlyStopping(patience=10, monitor='val_loss')
# Save the weight at the end of each epoch
#weight_record = tf.keras.callbacks.ModelCheckpoint('weights{epoch:08d}.h5',
                                    # save_weights_only=True)
epo=100

# Fit the model
Monitor = model.fit(X_train, y_train,
                    epochs=epo,
                    batch_size=B,
                    callbacks=[GradsReal_loss, checkpointer,stop_learn],
                    validation_data=(X_test, y_test),
                    )

"""### Plotting the Loss at the end of each Batch (n)"""

#At the end of each Batch (n)
fig, ax = plt.subplots(figsize = (12,8))
ax.plot(GradsReal_loss.bavCRE)
ax.set_xlabel('Batch')
```

```
ax.set_ylabel('Average Batch Cross Entropy')
ax.set_title('Average Batch Cross Entropy - h = hL= 35')

#At the end of each Epoch (m)
plt.plot(Monitor.history['loss']) #Loss of Training Set
#plt.plot(Monitor.history['val_loss']) #Validation Loss

"""### Plotting the size of weight change at the end of each Batch (n)"""

#At the end of each batch (n)
fig, ax = plt.subplots(figsize = (12,8))
ax.plot(GradsReal_loss.b_WC)
ax.set_xlabel('Batch')
ax.set_ylabel('Weight Change Ratio')
ax.set_title('Weight Change Ratio - h = hL= 35')

#At the end of Each Epoch(m)
from numpy import linalg as LA
def ratio(arr1,arr2):
  return(LA.norm(arr2-arr1)/LA.norm(arr1))
weight_ratio=[]
D=len(GradsReal_loss.weight[0])
epo_real=len(GradsReal_loss.weight) # the number of ran epoch
# it is <= epo because of interrupt learning
for i in range(epo_real-1):
  weight_ratio.append(ratio(GradsReal_loss.weight[i],GradsReal_loss.weight[i+1]))

plt.plot(weight_ratio)

"""### Plotting the ||G(n)|| at the end of each Batch (n)"""

GNn=[LA.norm(x) for x in GradsReal_loss.bG]
GNn[:10]

len(GradsReal_loss.bG[0])

fig, ax = plt.subplots(figsize = (12,8))
ax.plot(GNn)
ax.set_xlabel('Batch')
ax.set_ylabel('||G(n)||')
ax.set_title('||G(n)|| per batch - h = hL= 93')

"""### Plotting the ||G(n)||/d at the end of each Batch (n)"""

#At the end of each batch (n)
fig, ax = plt.subplots(figsize = (12,8))
ax.plot(GradsReal_loss.bGD)
ax.set_xlabel('Batch')
ax.set_ylabel('||G(n)||/d')
ax.set_title('||G(n)||/d per batch - h=hL=35 ')

#At the end of each Epoch (m)
plt.plot(GradsReal_loss.gradsNorm/np.sqrt(D))

"""## Epoch Evaluations

val_loss is the value of cost function for your cross-validation data and loss is the
value of cost function for your training data
"""

#plt.plot(Monitor.history['accuracy']) #Accuracy of Training Set
#plt.plot(Monitor.history['val_accuracy']) #Accuracy of Validation Set
```

```
#plt.plot(GradsReal_loss.gradsNorm)

"""# 5. Performance analysis"""

plt.figure(figsize = (12,8))
plt.plot(Monitor.history['accuracy'],color='blue',label='Train')
plt.plot(Monitor.history['val_accuracy'],color = 'red', label='Test')
plt.title('Test vs. Train Accuracy')
plt.xlabel('Accuracy')
plt.ylabel('Epoch')
plt.legend()
plt.show()

from tensorflow.keras.models import load_model

# Restore the best model and calculate confusion matrices.

model = load_model('BestModel.h5')
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)

cfm_train=tf.math.confusion_matrix(np.argmax(y_train, axis=1),
                                   np.argmax(y_train_pred, axis=1),
                                   num_classes=3).numpy()

cfm_test=tf.math.confusion_matrix(np.argmax(y_test, axis=1),
                                   np.argmax(y_test_pred, axis=1),
                                   num_classes=3).numpy()
print(cfm_train)
print(cfm_test)
# Access gradients history by 'GradsReal_loss.grads' and 'GradsReal_loss.gradsNorm'.

def acc_report(arr):
  nclass=arr.shape[0]
  report=pd.DataFrame(['0','1','2'],columns=['Class'])
  report['Correct_prediction']=([arr[i,i] for i in range(nclass)])
  report["Ncases"]=arr.sum(axis=1)
  report=report.append(report.sum(axis=0),ignore_index=True)
  report['Class'][3]="Total"
  report['Accuracy']=np.round(report.Correct_prediction/report.Ncases,3)
  report['LB']=np.round(report.Accuracy-1.96*np.sqrt(report.Accuracy*(1-
report.Accuracy)/report.Ncases),3)
  report['UB']=np.round(report.Accuracy+1.96*np.sqrt(report.Accuracy*(1-
report.Accuracy)/report.Ncases),3)
  report.iloc[:,1:3]=report.iloc[:,1:3].astype(int)
  return report
acc_report(cfm_train)

acc_report(cfm_test)

# a function to convert confusion matrix to percentage
def cfm2per(array):
  array=array/array.sum(axis=1)
  return(array)

# Turn to percentage and plot as heatmap
cfm_train=cfm2per(cfm_train)

sns.heatmap(cfm_train, annot=True,cmap=plt.cm.Blues)
plt.tight_layout()
plt.title('Train Set')
plt.ylabel('True label')
plt.xlabel('Predicted label')
```

```
plt.show()

cfm_test=cfm2per(cfm_test)

sns.heatmap(cfm_test, annot=True,cmap=plt.cm.Blues)
plt.title('Test Set')
plt.tight_layout()
plt.ylabel('True label')
plt.xlabel('Predicted label')
plt.show()

"""# 6. Analysis of hidden layer"""

Houtput = model.layers[0](X_train).numpy()

Houtput.shape

"""### PCA"""

Houtput = pd.DataFrame(model.layers[0](X_train).numpy())
abs(Houtput).sum()[abs(Houtput).sum() == 0]
#looks like 3 columns are completely 0
#Will have to fill the Nan values in correlation matrix

#Drop the columns with all 0 values
Houtput2 = Houtput.drop(columns = [15, 27,61])

H_corr = Houtput2.corr()
print(H_corr.shape)

#look at neurons with a correlation value greater than .90
top_tracker = {}
for i in range(len(H_corr+3)):
  if i in [15, 27, 61]:
    continue
  for j in range(len(H_corr+3)):
    if j in [15, 27, 61]:
      continue
    else:
      val = H_corr.loc[i,j]
    if val > .90:
      if val not in top_tracker.keys():
        top_tracker[val] = (i,j)

top_tracker.pop(1, None)
top_tracker

#Eigen Values and Vectors
H_eig, H_eig_v = np.linalg.eig(H_corr)
#ratio of variance explained
H_ratio = np.cumsum(H_eig)/np.sum(H_eig)

fig, ax = plt.subplots(figsize = (12,8))
ax.plot(H_ratio)
ax.scatter(2, H_ratio[2], c = 'black', alpha = .8)
ax.set_xlabel('Eigenvalue')
ax.set_ylabel('Ratio of Variance Explained')
ax.text(x = 3, y = H_ratio[2], s = '{} Variance Explained'.format(round(H_ratio[2],2)))
ax.set_title('Variance Explained')

print('Variance Explained by Projecting onto first 3 is {}'.format(round(H_ratio[2],2)))

#Project
```

```
proj = np.matmul(Houtput2, H_eig_v).iloc[:, 0:3]

#Get Index of different groups
#print(y_test[1:10,:])
#print(y_test2[1:10])
y_train2 = pd.Series(y_train2)

#class 0 = 0 (Normal)
cl0_idx = y_train2[y_train2 == 0].index
#Class 1 = 4 (unclassified)
cl1_idx = y_train2[y_train2 == 1].index
#Class 2 = 2 (ventricular)
cl2_idx = y_train2[y_train2 == 2].index

#Plot the 3d cluster by group
from mpl_toolkits import mplot3d

fig=plt.figure(figsize=(15,12))
ax = fig.add_subplot(111, projection='3d')
ax.view_init(60, 75)
ax.set_title('Principal Components 1, 2, and 3')
ax.set_xlabel('Principal Component 1')
ax.set_ylabel('Principal Component 2')
ax.set_zlabel('Principal Component 3')
ax.scatter(proj.loc[cl0_idx,0], proj.loc[cl0_idx,1], proj.loc[cl0_idx,2], c='r' , alpha
= .2)
ax.scatter(proj.loc[cl1_idx,0], proj.loc[cl1_idx,1], proj.loc[cl1_idx,2], c='b', alpha =
.2)
ax.scatter(proj.loc[cl2_idx,0], proj.loc[cl2_idx,1], proj.loc[cl2_idx,2], c='g', alpha =
.2)
ax.legend(['CL0: Normal','CL1: Unclassified' ,'Cl2: Ventricular'])

pca = PCA(35)
pca.fit(Houtput)
plt.plot(np.cumsum(pca.explained_variance_ratio_))
#plt.hlines(0.95,0,187,'r')
#plt.vlines(29,0.4,1,'r')
plt.show()

# Average activities of H
label_train=np.argmax(y_train, axis=1)
PROF1=np.mean(Houtput[label_train==0],axis=0)
plt.plot(PROF1,'o',label='PROF1')

PROF2=np.mean(Houtput[label_train==1],axis=0)
plt.plot(PROF2,'*',label='PROF2')

PROF3=np.mean(Houtput[label_train==2],axis=0)
plt.plot(PROF3,'ko',label='PROF3')
plt.xlabel('Neuron index')
plt.ylabel('Average activity')
plt.title('Average hidden neurons activity profiles PROF1 , PROF2, PROF3')
plt.legend()
plt.show()

plt.hist(PROF1)
plt.show()
plt.hist(PROF2)
plt.show()
plt.hist(PROF3)
plt.show()

plt.plot(PROF1,'red',label="CL1")
```

```
plt.plot(PROF2,'green',label="CL2")
plt.legend()
plt.show()
plt.plot(np.abs(PROF1-PROF2),label="CL1-CL2")
plt.legend()
plt.show()

#new code
count_neurons = 5#number of neurons
avg21 = (PROF1+PROF2)/2
norm_dif21 = np.abs(PROF2-PROF1) / avg21
norm_dif21.fillna(0,inplace=True)
#Top 5 neurons
print(norm_dif21.argsort()[-count_neurons:][::-1])
#Top 5 largest differences
print(np.sort(norm_dif21)[-count_neurons:][::-1])

#new code
avg23 = (PROF2 + PROF3)/2
norm_dif23 = np.abs(PROF3 - PROF2) / avg23
norm_dif23.fillna(0,inplace=True)
#Top 5 neurons
print(norm_dif23.argsort()[-count_neurons:][::-1])
#Top 5 largest differences
print(np.sort(norm_dif23)[-count_neurons:][::-1])

#new code
avg31 = (PROF1+PROF3)/2
norm_dif31 = np.abs(PROF3-PROF1) / avg31
norm_dif31.fillna(0,inplace=True)
#Top 5 neurons
print(norm_dif31.argsort()[-count_neurons:][::-1])
#Top 5 largest differences
print(np.sort(norm_dif31)[-count_neurons:][::-1])
```