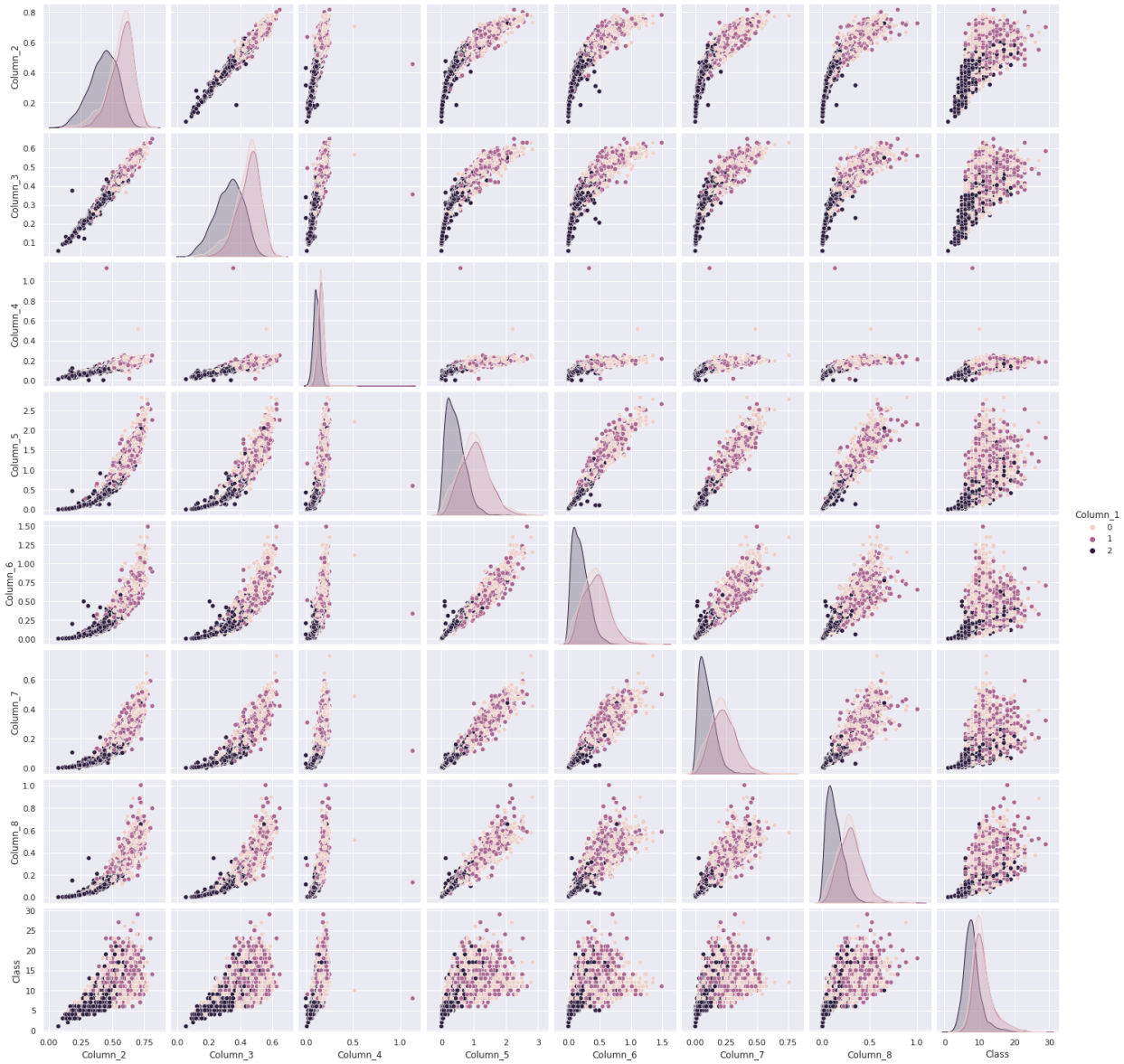
Pattern Recognition and Machine Learning
Lab 6

Question 1 - Implementing Network using Pytorch

1. Pre-processing and meaningful data analysis- Preprocessing was done on the dataset to make it fit for the task to be done. The dataset was normalized and a pair plot was plotted.
   The Plot can be seen below.



   Many permutations of hyperparameters were tested.
   HyperParameters that were chosen for the network can be seen below:

```
batch_size = 32 #sample batch
num_epochs = 500 #number of times dataset is seen by model
learning_rate = 0.01
size_hidden_1 = 8 #neurons
size_hidden_2 = 8 #neurons
num_classes = 30
batch_no = len(X_train) // batch_size  #batches
cols = X_train.shape[1] #Number of columns in input matrix
```

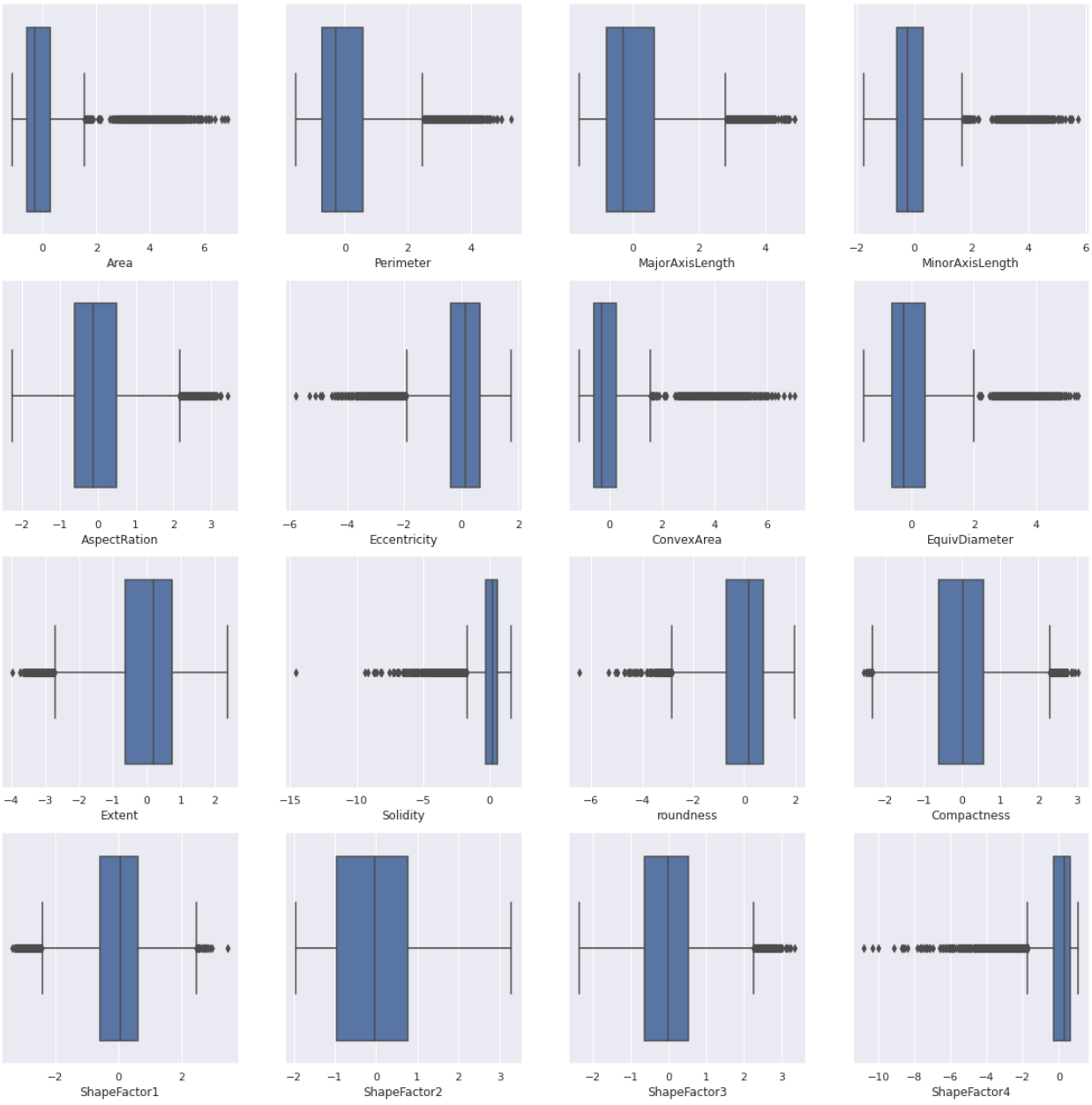
Summary of the model:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                [-1, 1, 8]              72
              Tanh-2                [-1, 1, 8]               0
            Linear-3                [-1, 1, 8]              72
              Tanh-4                [-1, 1, 8]               0
            Linear-5               [-1, 1, 30]             270
           Softmax-6               [-1, 1, 30]               0
================================================================
Total params: 414
Trainable params: 414
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.00
Estimated Total Size (MB): 0.00
----------------------------------------------------------------
```

After training for 500 epochs accuracy reported was:
Train: 100%
Test: 43.03%

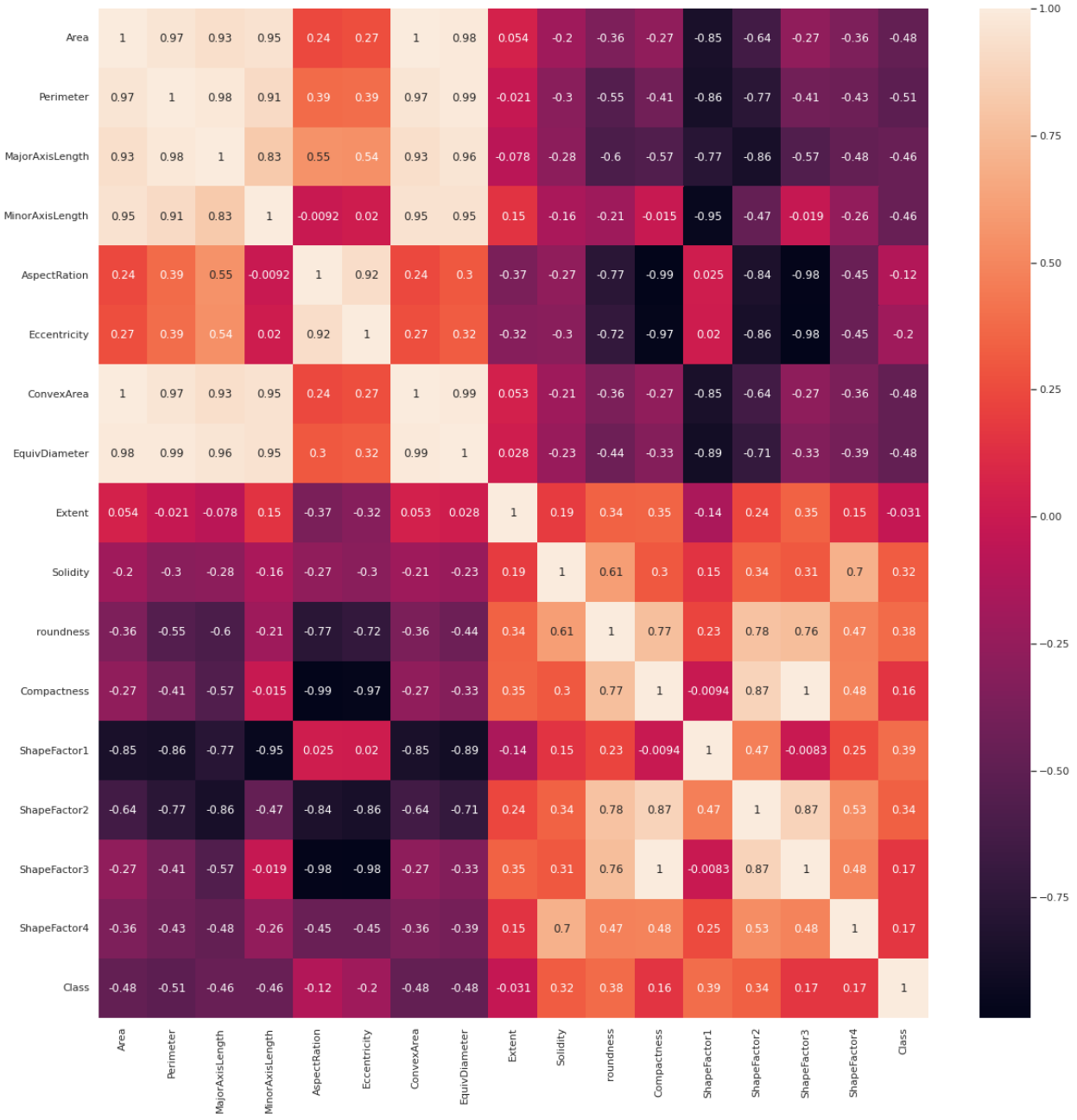
Question 2 - Implementing MLP from Scratch

1. The data was imported and then pre-processed to make it fit for the model to be implemented. Boxplots were plotted for different columns to find out the outliers. The plot can be seen below:



A heatmap was plotted to find the correlation between different columns and to find out which columns are more correlated with the class column.

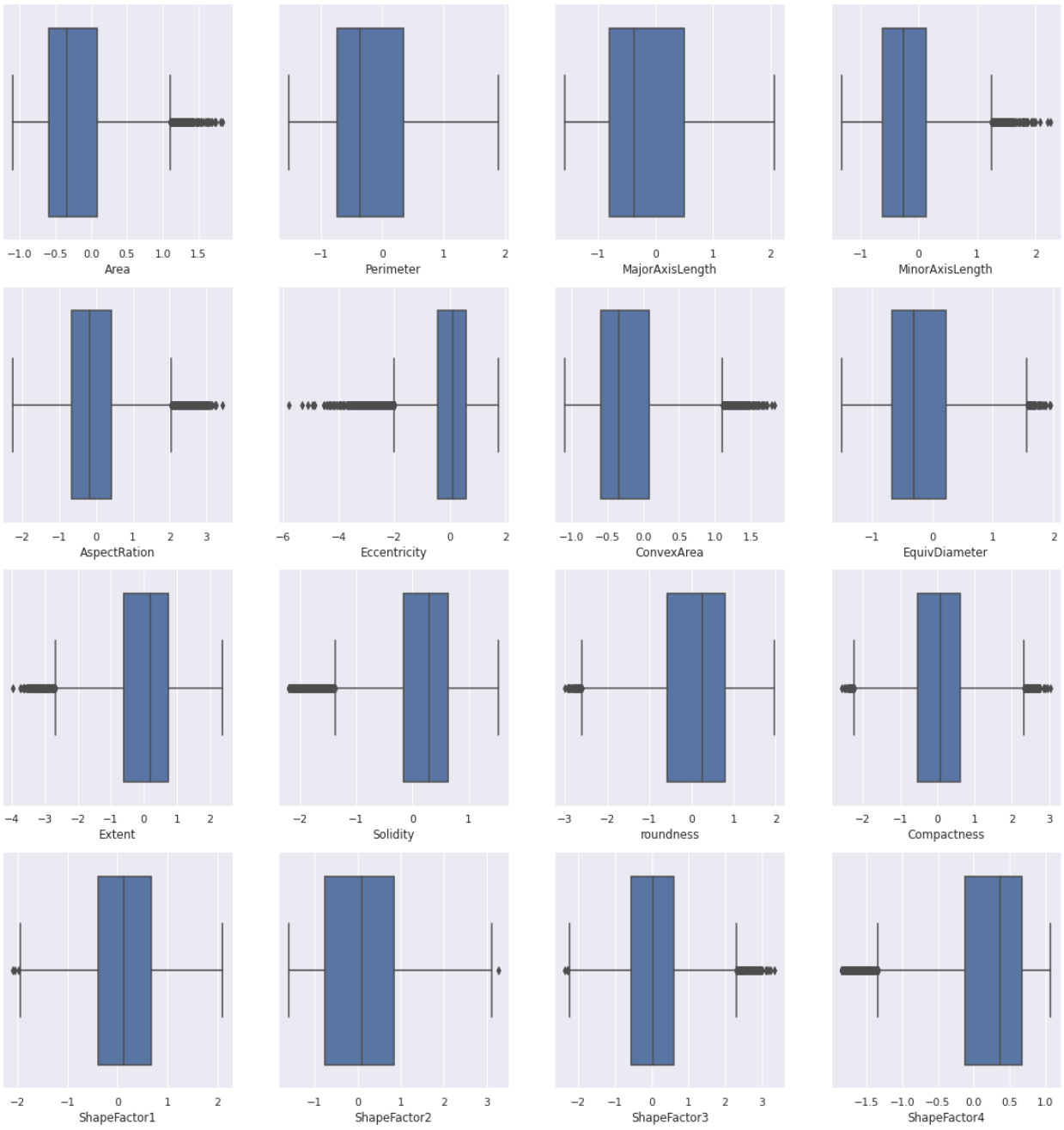
The heatmap can be seen below:



The columns which had more correlation were selected and their outliers were removed. The code of which can be found in the colab notebook.

A pair plot was plotted which can be seen in the notebook.

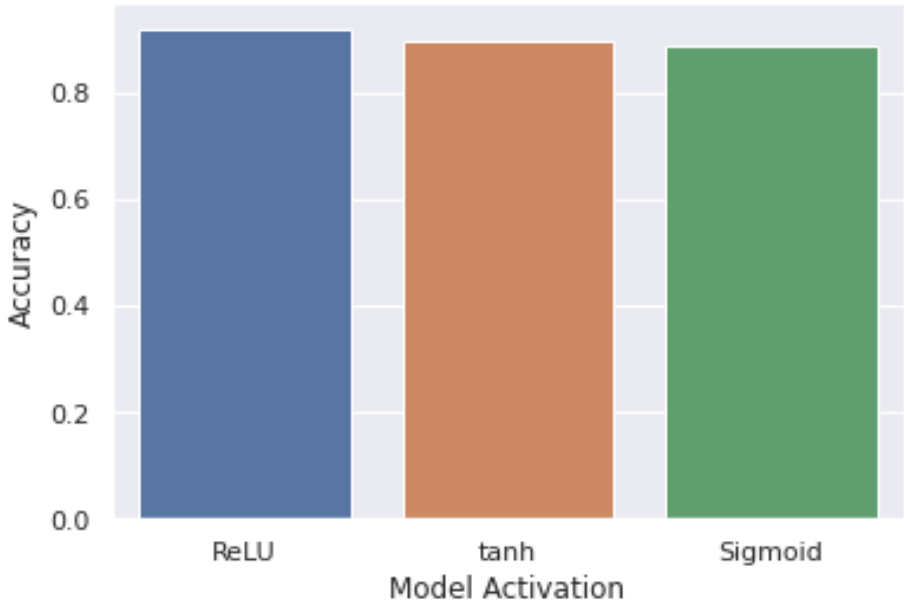
Boxplots after the removal of outliers can be seen below:



The class distribution was taken into consideration and the dataset was split accordingly which can be seen in the colab notebook. Once the dataset was ready it was fed into the coded model which had only one hidden layer.

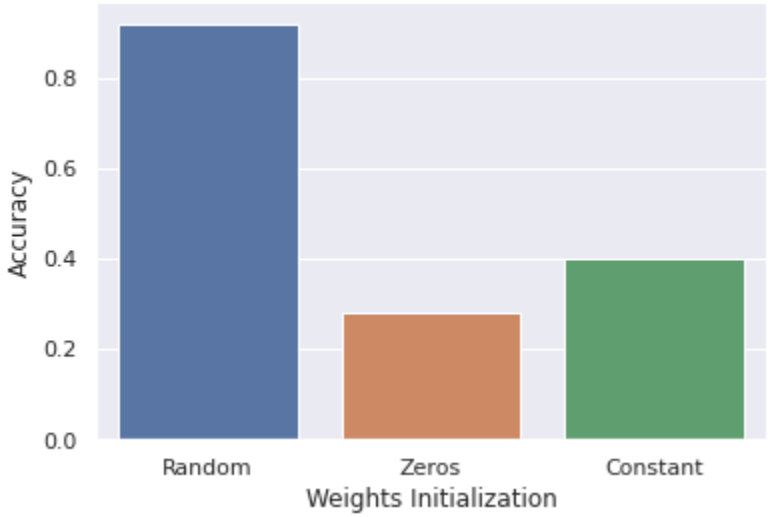
2. A multilayer perceptron was coded from scratch and the subtasks were performed. All the codes are self-explanatory and can be found in the colab notebook itself.
3. Three different activation functions were chosen namely ‘sigmoid’, ‘ReLU’, and ‘tanh’. Accuracies were reported and the plots were plotted. Results can be seen below:
   Accuracy (Sigmoid) ===> 0.8867872807017544
   Accuracy (ReLU) ===> 0.918859649122807
   Accuracy (tanh) ===> 0.8980263157894737

   The plot can be seen below:



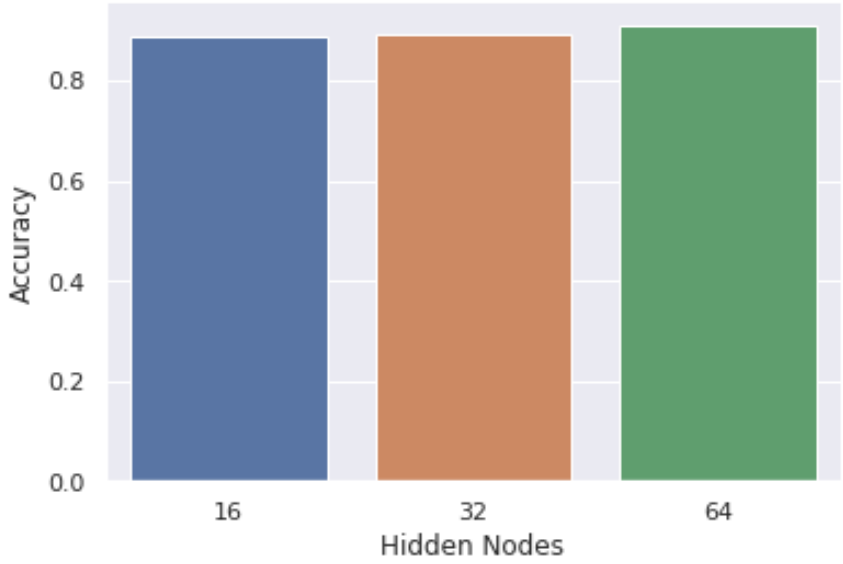
   ReLU performed better than tanh and sigmoid activation functions. ReLU generally performs better than sigmoid and another same kind of functions because of majorly two reasons. The first being it's easy to compute reason and its biological resemblance with brain neurons. The second is the sparsity it introduces because of the dead neurons that get generated whenever its value becomes less than zero. Generally, Sparse networks perform better than dense networks.
4. Different weight categories were taken into consideration namely, ‘Random’, ‘Zeros’, and ‘Constant’. The model was trained for each different weight category. The Random category performed much better than the zeros and constant category for obvious reasons.

The results were plotted which can be seen in the plot below:



If we set all the weights to be zero, then all the neurons of all the layers perform the same calculation, giving the same output and thereby making the whole deep net useless. If the weights are zero, the complexity of the whole deep net would be the same as that of a single neuron and the predictions would be nothing better than random. In the case of constant weights, no matter what was the input - if all weights are the same, all units in a hidden layer will be the same too. So, random initialization works best.

5. The hidden nodes were changed and the results were plotted for different nodes. 16, 32, and 64 were chosen as the varying hidden nodes.
   The plot can be found below:



Accuracies reported were:

16 nodes - 0.8889802631578947
32 nodes - 0.8933662280701754
64 nodes - 0.9095394736842105

Increasing nodes, increased accuracy in our case as more nodes provide more flexibility to the network. But after a certain amount of nodes, the model saturates and stops learning.

6. For saving and loading weights Pickle library was used. Code of which can be found in the colab notebook itself.

- Piyush Arora (B20CS086)