Pattern Recognition and Machine Learning

(Winter 2022)

Assignment 7: Neural Networks

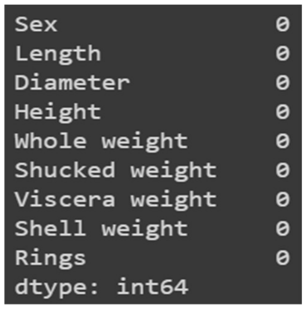
Report

Question - 1:

- **Data Pre-Processing.**

I first took the input of the given abalone dataset. Then, I performed the data pre-processing on the given dataset.

I checked for any nan/missing values present in the dataset. It appears that there are no missing values in the dataset.



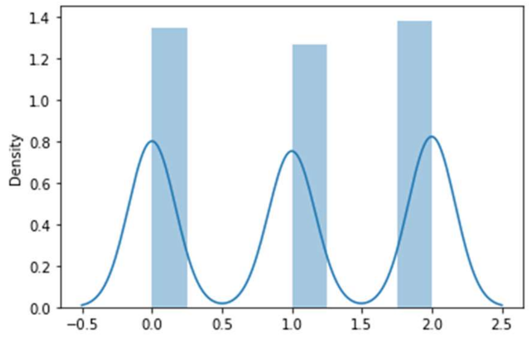
```
Sex               0
Length            0
Diameter          0
Height            0
Whole weight      0
Shucked weight    0
Viscera weight    0
Shell weight      0
Rings             0
dtype: int64
```

Then, I checked for the datatypes of the columns present in the dataset. Since, the column 'Sex' was categorical I applied the categorical encoding on that column with the help of Label Encoder.

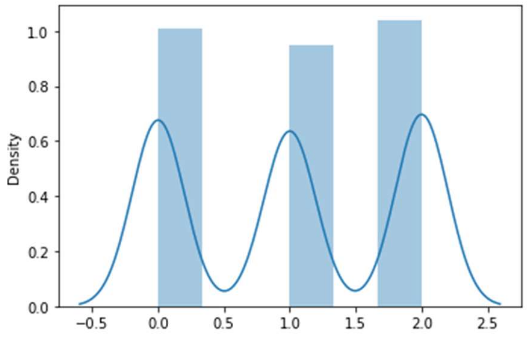
For better classification of the dataset, I divided the rings into 3 classes (i.e., class 0 for rings 0 to 8, class 1 for rings 9 to 10, else rings as class 2).

Then, I split the dataset into training and testing dataset.

I also plotted the distribution of target column in training and testing dataset. The plots clearly shows that the classes distribution in training and testing sets is uniform.



Training



Testing

- **Neural Network Implementation**

I wrote the code to create a neural network. For implementing the neural network, I first defined the training hyperparameters. For that, I took the parameters as batch size, number of epochs, learning rate, size of hidden layer 1, size of hidden layer 2 and number of classes.

Then, I created the neural network model for that I first initialized the hidden layer 1 as torch.nn.Linear(num_inputs, size_hidden_1) and also initialized the Hidden layer 2 as torch.nn.Linear(num_inputs, size_hidden_2). I also initialized the activation layer as torch.nn.Tanh() (activation function = Tanh) for hidden layers.

Then, I performed the addition of the two linear layers and I simultaneously defined the activation function of output activation layer as torch.nn.Sigmoid(). Then, I defined a function "forward" that return the output layer.

Then, I used the optimizer as optim.SGD (stochastic gradient descent) from torch library and passed the learning rate and net parameters. I also calculate the Cross Entropy Loss.

Then, I defined the function "get_accuracy" which calculates the accuracies for the training round.

Then, I computed the loss and accuracies for each epoch and also in this step, I tried different values of hyper parameters and trained the neural network for the pair of best hyperparameters and hence calculated the loss and accuracies.

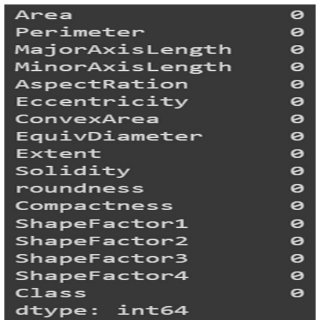
For this trained MLP model, I got the best accuracy at particular epoch on the test set after training as 54.18%.

Question - 2:

Task - 1: Pre-process & visualize the data. Create train, val, and test splits but take into consideration the class distribution (Hint: Look up stratified splits).:-

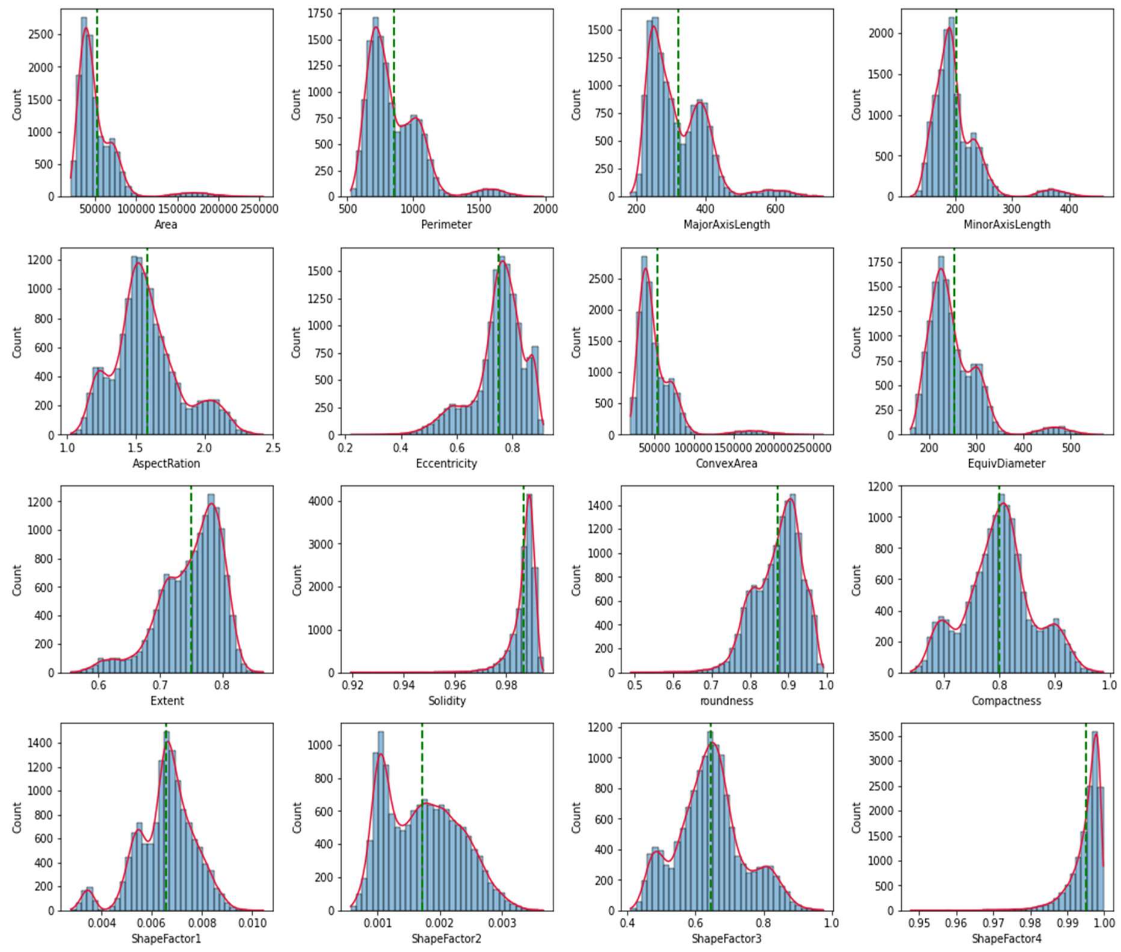
I first took the input of the given dry bean dataset. Then, I performed the data pre-processing on the given dataset.

I checked for any nan/missing values present in the dataset. It appears that there are no missing values in the dataset.

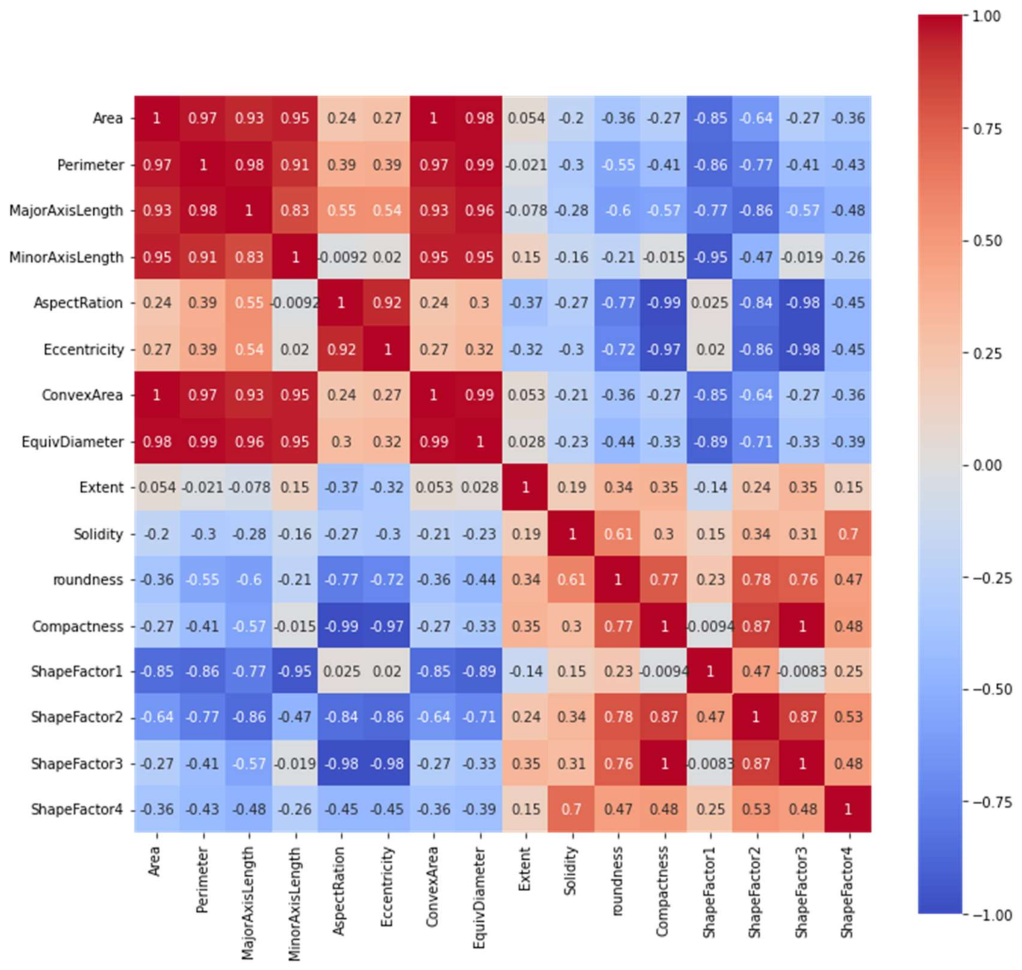


```
Area               0
Perimeter          0
MajorAxisLength    0
MinorAxisLength    0
AspectRation       0
Eccentricity       0
ConvexArea         0
EquivDiameter      0
Extent             0
Solidity           0
roundness          0
Compactness        0
ShapeFactor1       0
ShapeFactor2       0
ShapeFactor3       0
ShapeFactor4       0
Class              0
dtype: int64
```

For Visualization, I plotted the numerical features of the given dataset. From this plot it is observed that some distributions have long tails and most are bi-modal which means that some bean classes should be quite distinct from others.



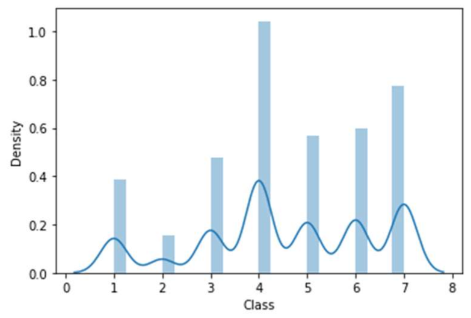
Also, I plotted the Pearson linear correlation. From this plot it is observed that there are lots of highly correlated features.



Then, I checked for the datatypes of the columns present in the dataset. Since, the column ‘Class’ was categorical I applied the categorical encoding on that column with the help of Label Encoder.

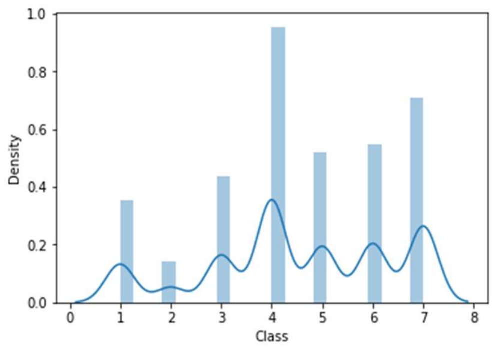
Then, I scaled the entire dataset with the help of StandardScaler(). As, Data scaling is a recommended pre-processing step when working with deep learning neural networks.

Then, I plotted the distribution of class in the given dataset.

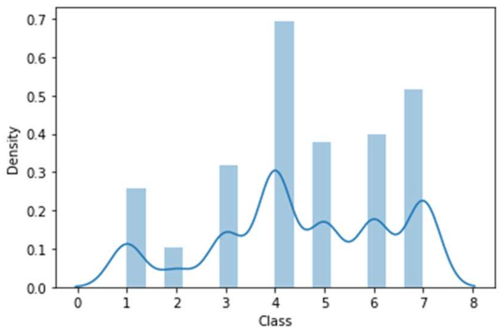


Then, I split the data into training and testing datasets. I made sure that while splitting the data the class distribution remains uniform in the training and testing sets. For that I applied the principles of stratified splits.

I also plotted the distribution of target column in training and testing dataset. The plots clearly shows that the classes distribution in training and testing sets is uniform.



Training



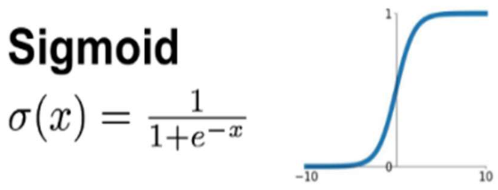
Testing

<u>Task - 2:</u> Implement a multi-layer perceptron from scratch. This would include the following.

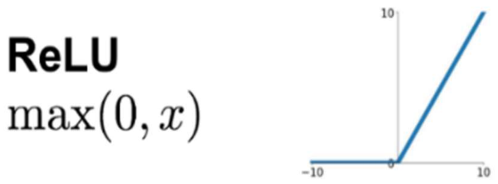
a. Write activation functions.

I wrote the code for 4 activations functions from scratch i.e., Sigmoid, ReLu, Tanh and LeakyReLu. For this I just implemented their formulas in my code i.e.,

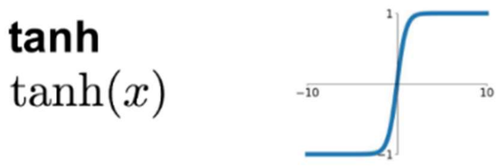
For Sigmoid:



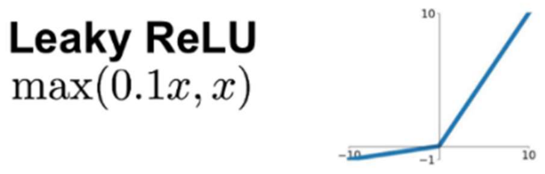
For ReLu:



For Tanh:



For LeakyReLu:



b. Forward propagate the input.

For this task, I used the implemented function “productOfWeightAndInputs” that helps compute the product that is the inputs are combined with the initial weights in a weighted sum.

Then, to forward propagate the input. I wrote a function “forwardPropagate” in which each linear combination is propagated to the next layer subjected to the provided activation function. So, I iterated over each row and neuron in the layers and computed the inputs subject to the provided activation function.

c. Backward propagate the error.

For this task, I wrote a function “backwardPropagateTheError” that allows the Multilayer Perceptron to iteratively adjust the weights in the network, with the goal of minimizing the cost function.

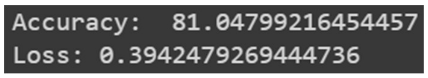
So, I calculated the error of each layer form the initialized network, then applied the delta rule for perceptron to multilayer feedforward neural networks and based on the provided activation function. I calculated the product of error and that provided activation function derivative.

d. Train the network using stochastic gradient descent.

For this task, I wrote a function "trainNetwork" and "updateweights". So, the function "updateweights" helps to update the weight of the inputs based on the error and the learning rate. The function "trainNetwork" then trains the network by calculating the best minimum loss by simulatenously updating weights and I iterated this in range of number of epochs provided. It basically uses the principles of stochastic gradient descent to train the network i.e., by replacing the actual gradient by an estimate thereof (calculated from a randomly selected subset of the data).

e. Predict the output for a given test sample and compute the accuracy.

For this task, I predicted the output for the given test sample and computed the accuracy. I kept two hidden layers, 0.001 as my learning rate and 'ReLu' as my activation function. So, for these parameters I got the accuracy and loss as:

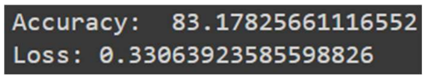


```
Accuracy:  81.04799216454457
Loss: 0.3942479269444736
```

<u>Task - 3:</u> Now experiment with different activation functions (at least 3 & to be written from scratch) and comment (in the report) on how the accuracy varies. Create plots to support your arguments.

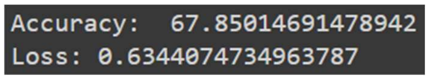
For this task, I took three of my implemented activation functions from scratch in the previous task. Then, I trained the Multi-Layer Perceptron for each activation function setting same hyper parameters (i.e., 0.001 as learning rate and two hidden layers) and computed the accuracies for each activation functions.

For *ReLu* as activation function, I got the accuracy and loss as:



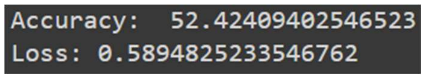
```
Accuracy:  83.17825661116552
Loss: 0.33063923585598826
```

For *Sigmoid* as activation function, I got the accuracy and loss as:



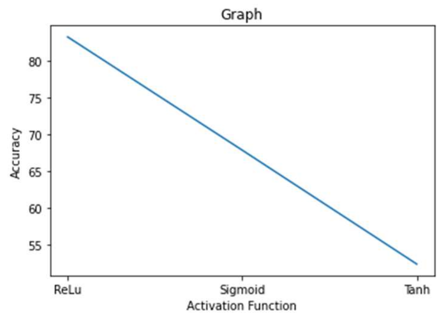
```
Accuracy:  67.85014691478942
Loss: 0.6344074734963787
```

For *Tanh* as activation function, I got the accuracy and loss as:



```
Accuracy:  52.42409402546523
Loss: 0.5894825233546762
```

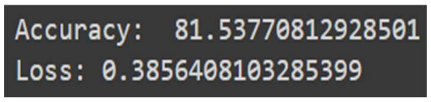
I also plotted these accuracies on a single plot to visualize them. Hence, it is clearly seen from the plot that for selected set of hyperparameters, "ReLu" activation function gave the best accuracy and minimum loss.



<u>Task - 4:</u> Experiment with different weight initialization: Random, Zero & Constant. Create plots to support your arguments.

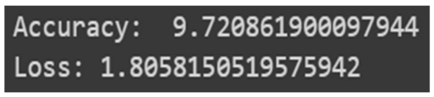
For this task, I experimented with the different activation functions i.e., Random, Zero and Constant. I used the implemented MLP model from scratch and updated the weight initialization technique for each experiment. So, for this I took 'ReLu' as my activation function, 0.001 as my learning rate and two hidden layers. Hence, I kept these hyperparameters fixed and changed the weight initialization and computed the accuracies and loss.

For *Random* weight initialization, I got the accuracy and loss as:



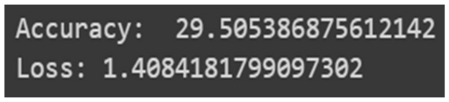
```
Accuracy:  81.53770812928501
Loss: 0.3856408103285399
```

For *Zero* weight initialization, I got the accuracy and loss as:



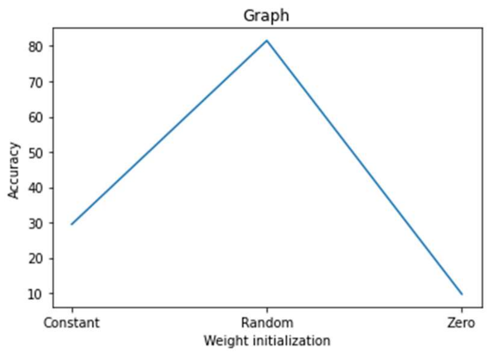
```
Accuracy:  9.720861900097944
Loss: 1.8058150519575942
```

For *Constant* weight initialization, I got the accuracy and loss as:



```
Accuracy:  29.505386875612142
Loss: 1.4084181799097302
```

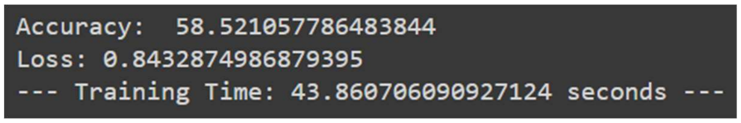
I also plotted these accuracies on a single plot to visualize them. Hence, it is clearly seen from the plot that for the selected set of hyperparameters, Random weight initialization gave the best accuracy and minimum loss.



<u>Task - 5:</u> Change the number of hidden nodes and comment upon the training and accuracy. Create plots to support your arguments.

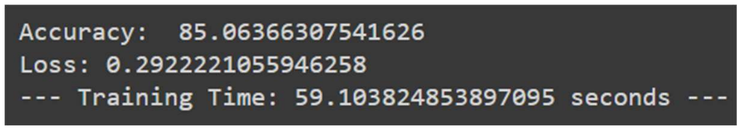
For this task, I again trained the MLP model implemented from scratch and kept the hyperparameters as 0.001 as learning rate, “ReLu” as my activation function. I then varied the number of hidden layers and computed the accuracies, loss and training time for each case:

For *1* Hidden Node Layer:



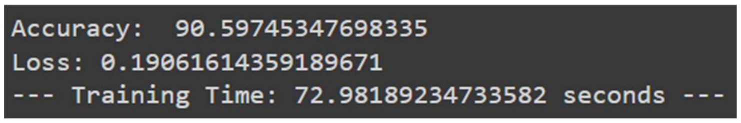
```
Accuracy:  58.521057786483844
Loss: 0.8432874986879395
--- Training Time: 43.860706090927124 seconds ---
```

For *2* Hidden Node Layer:



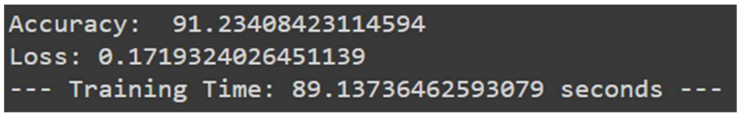
```
Accuracy:  85.06366307541626
Loss: 0.2922221055946258
--- Training Time: 59.103824853897095 seconds ---
```

For *3* Hidden Node Layer:



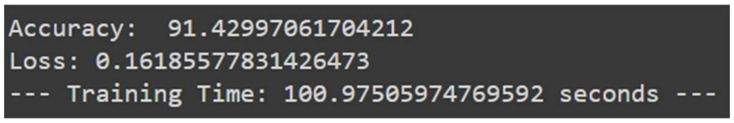
```
Accuracy:  90.59745347698335
Loss: 0.19061614359189671
--- Training Time: 72.98189234733582 seconds ---
```

For *4* Hidden Node Layer:



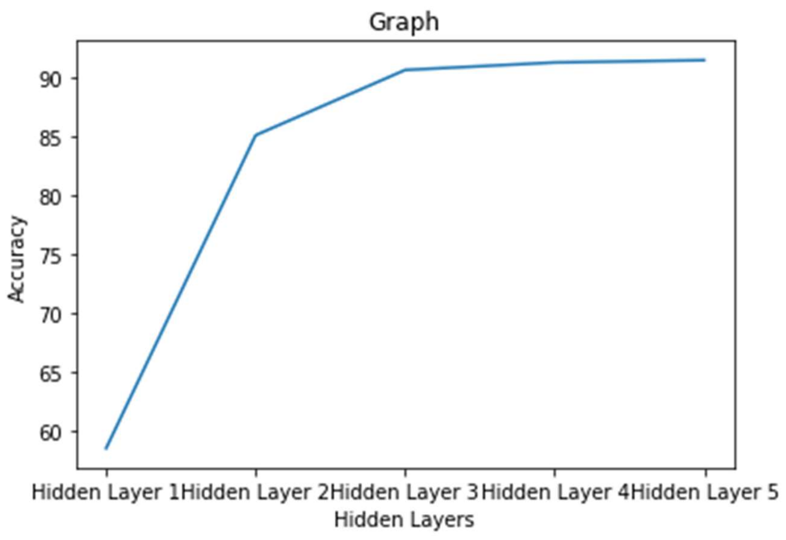
```
Accuracy:  91.23408423114594
Loss: 0.1719324026451139
--- Training Time: 89.13736462593079 seconds ---
```

For *5* Hidden Node Layer:



```
Accuracy:  91.42997061704212
Loss: 0.16185577831426473
--- Training Time: 100.97505974769592 seconds ---
```

I also plotted these accuracies on a single plot to visualize them. Hence, it is clearly seen from the plot that initially on increasing the number of hidden layers accuracies increase but as we further increase the number of hidden layers the test set accuracy tends to decrease because for large number of hidden layers, model tends to overfit the data that leads to lesser accuracies and increased loss on the testing sets. Also, it is observed that on increasing the number of hidden layers training time increases significantly.



<u>Task - 6:</u> Add a provision to save and load weights in the MLP.

For this task, I added a provision in the function "trainNetwork" that helps to store the weights in the MLP that gave the best accuracies and minimum loss. It helps in decreasing computational cost as now there is no need to again and again calculate the updated weights as we can use the stored weight which gave the best accuracy. Now this function also returns the stored weights. Hence, now we can save and load weights in the MLP.