Pattern Recognition & Machine Learning
Lab-9 Assignment

Name: Tanish Pagaria
Roll No.: B21AI040

Question-1

Loading MNIST dataset and splitting into train-validation-test sets

Performed using `torchvision.datasets.MNIST`. The dataset was split into training, validation, and test sets using `torch.utils.data.random_split`. The train-to-validation ratio was set at 50000:10000.

Applying augmentions to images

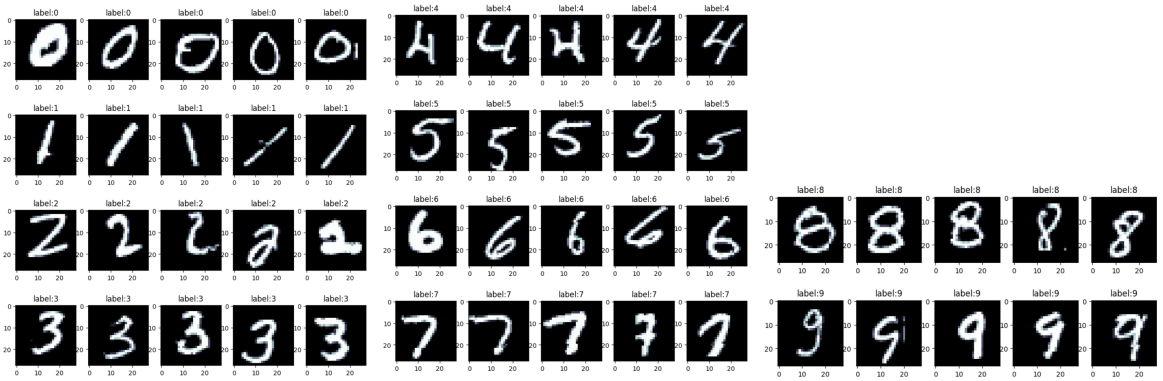
Using `torchvision.transforms`, functions were created to apply the following transformations:-

- *Training dataset:* RandomRotation(5 degrees), RandomCrop(size=28, padding=2), ToTensor and Normalize
- *Testing dataset and validation dataset:* ToTensor and Normalize

The values **0.1307** and **0.3081** used for the `Normalize()` transformation are the global mean and standard deviation of the MNIST dataset.

Plotting images from each class and creating data loaders for the training and testing datasets

By iterating over the dataset and storing tensors and labels in a dictionary, some images from each class were plotted, as shown.



Some images from each class

Data loaders were created for the train and test datasets using `torch.utils.data.DataLoader` with batch sizes of 64 and 1000.

3-Layered Multilayer Perceptron using PyTorch (all linear layers)

We created a class called `MLP_PyTorch`. It inherited its methods from `torch.nn.Module`. The constructor for the class took in the number of nodes in each layer of the MLP. It would also print the number of trainable parameters upon initialization.

The `forward` method was defined to perform forward propagation on the input and return the output layer. The activation function used in the hidden layers was ReLU. Also, `save` and `load` methods were defined to save the model in any specified directory and load the model from that directory, respectively.

The `fit` method was defined, which trained the model, displayed the training loss and the validation accuracy after each loss, and also saved the best model after each epoch. It took in the following parameters: dataloaders of training and validation sets; optimizer (default: `Adam`); loss function (default: `CrossEntropyLoss`); learning rate (default: 0.005); and the number of epochs (default: 5).

The `test` method was defined, which took in the dataloader of the testing dataset and displayed the model accuracy on the test set. Helper methods were defined to plot Loss-Epoch graphs for the training set and Accuracy-Epoch graphs for the validation set. Another helper method was defined to visualize correct and incorrect predictions on the test set.

The following results were obtained using the above implemented class on the MNIST dataset.

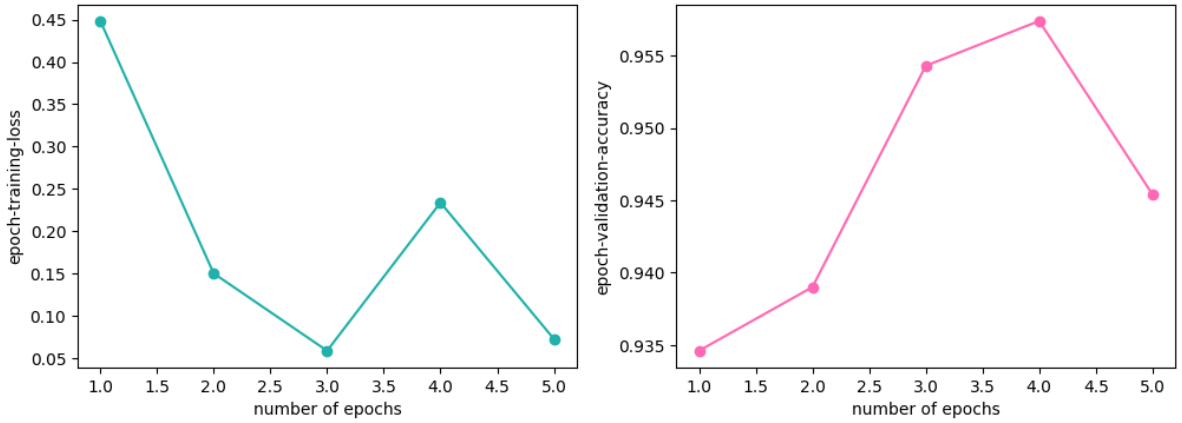
Initialization of the model

```
model = MLP(num_input_nodes=28*28, num_hidden_nodes_1=256, num_hidden_nodes_2=256, num_output_nodes=10)
```

```
Number of trainable parameters:-
200704
256
65536
256
2560
10
```

```
epoch: 1, loss: 0.44838428497314453
validation accuracy: 0.9346
-----------------------------------------
epoch: 2, loss: 0.15029026567935944
validation accuracy: 0.939
-----------------------------------------
epoch: 3, loss: 0.058974701911211014
validation accuracy: 0.9543
-----------------------------------------
epoch: 4, loss: 0.23393379151821136
validation accuracy: 0.9574
-----------------------------------------
epoch: 5, loss: 0.07298774272203445
validation accuracy: 0.9454
-----------------------------------------
```

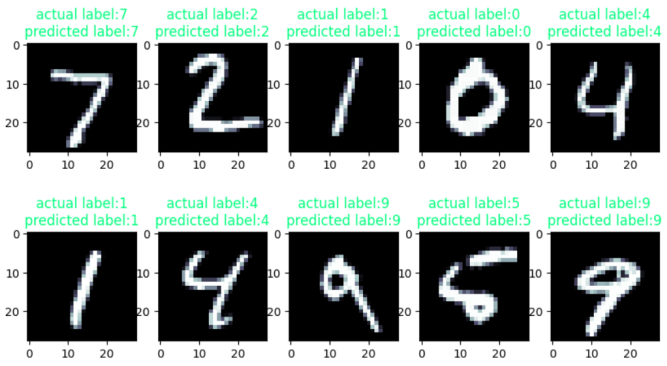
Training and visualization of the Loss-Epoch and Accuracy-Epoch curves



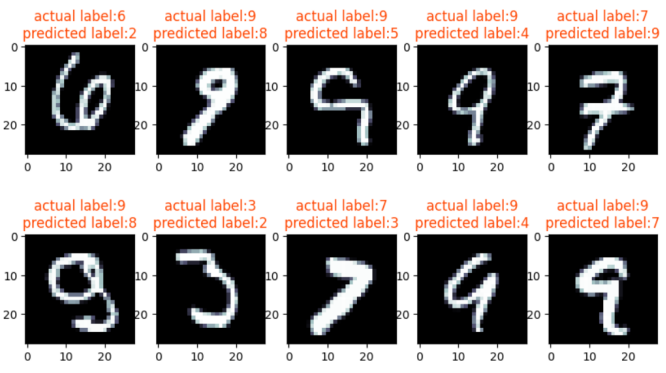
Testing results and visualization of the correct and incorrect predictions

```
Testing accuracy: 0.9466
-----------------------------------------------------------------------------------------------------------
```

Some of the correct predictions
(in testing dataset)



Some of the incorrect predictions
(in testing dataset)



(The testing accuracy was obtained using the best model saved in all the epochs)

Question-2

Loading the dataset, Preprocessing and Data Visualization

The following exploratory analysis was performed on the “Abalone” dataset.

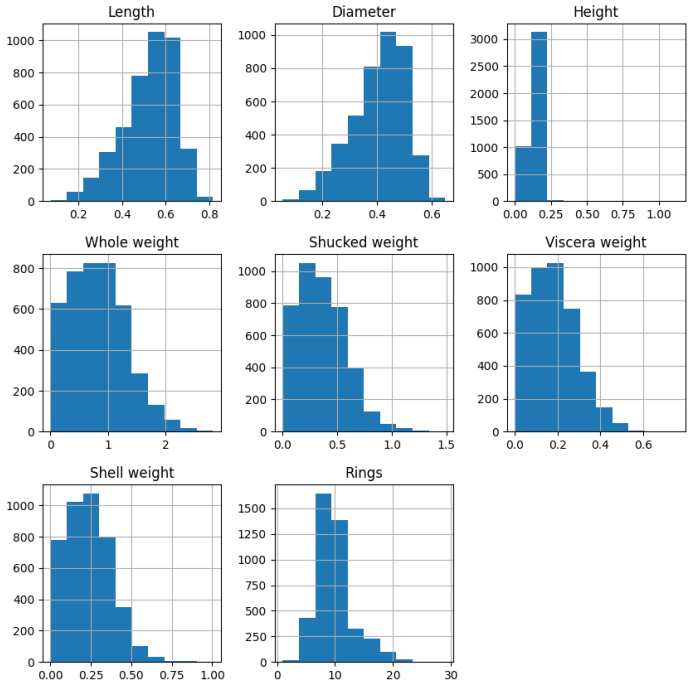
	Sex	Length	Diameter	Height	Whole weight	Shucked weight	Viscera weight	Shell weight	Rings
0	M	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.1500	15
1	M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.0700	7
2	F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.2100	9
3	M	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.1550	10
4	I	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.0550	7
...	...	...	...	...	...	...	...	...	...
4172	F	0.565	0.450	0.165	0.8870	0.3700	0.2390	0.2490	11
4173	M	0.590	0.440	0.135	0.9660	0.4390	0.2145	0.2605	10
4174	M	0.600	0.475	0.205	1.1760	0.5255	0.2875	0.3080	9
4175	F	0.625	0.485	0.150	1.0945	0.5310	0.2610	0.2960	10
4176	M	0.710	0.555	0.195	1.9485	0.9455	0.3765	0.4950	12

4177 rows × 9 columns

```
RangeIndex: 4177 entries, 0 to 4176
Data columns (total 9 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Sex             4177 non-null   object
 1   Length          4177 non-null   float64
 2   Diameter        4177 non-null   float64
 3   Height          4177 non-null   float64
 4   Whole weight    4177 non-null   float64
 5   Shucked weight  4177 non-null   float64
 6   Viscera weight  4177 non-null   float64
 7   Shell weight    4177 non-null   float64
 8   Rings           4177 non-null   int64
dtypes: float64(7), int64(1), object(1)
memory usage: 293.8+ KB
```

The dataset did not contain any NULL values or NaN entries. There were 29 unique classes in the dataset.

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 29])
```

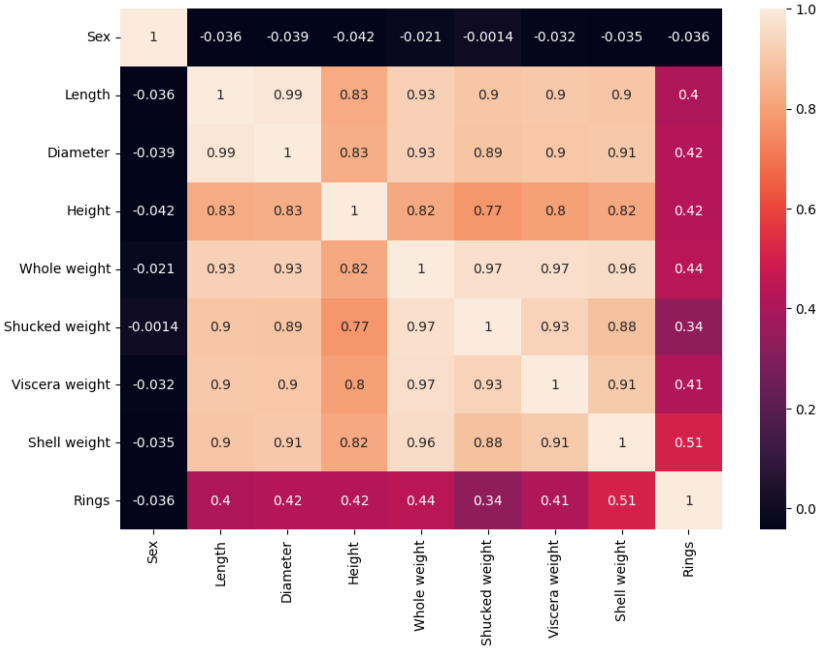


Dataset visualization using histogram

We encoded any categorical features and applied standard scaling to the dataframe. As mentioned previously, there were 29 distinct classes in the dataset. Some of these classes had only single data points, so the dataset could not be split into training-validation-testing sets by stratified splitting. It was handled by binning the dataset. The number of rings added to 1.5 would give the age of the abalone. This was used to discretize the data by making 3 classes based on the age, as shown below.

Number of Rings	Class
1 - 10	Young Age
11 - 20	Middle-Age
21 - 30	Old Age

The labels were encoded again, and the correlation heatmap of the dataset was plotted. A stratified split was performed afterwards on the dataset using a self-implemented function.



Correlation heatmap of the "Abalone" dataset (after preprocessing)

Implementation of Multi-Layer Perceptron from scratch

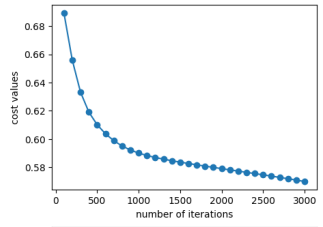
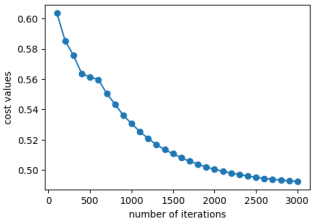
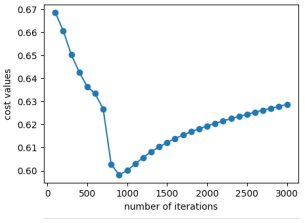
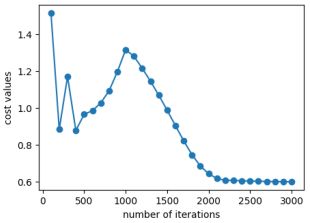
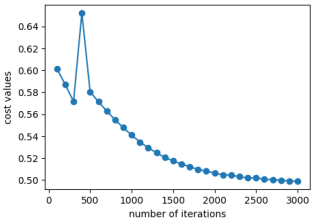
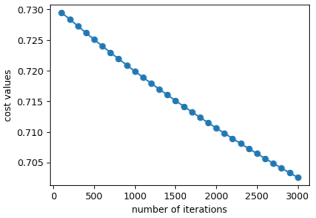
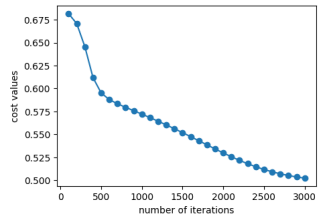
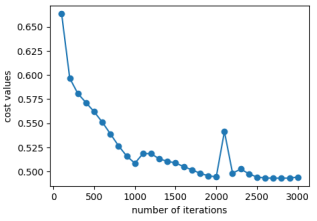
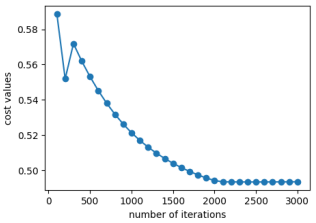
We created a class called `MLP`. The constructor took in the parameters, including the training dataset and labels, information regarding the hidden layers (number of hidden layers and hidden nodes), the activation function to be used, the learning rate, the weight initialization type, and the number of iterations. The class was flexible and could be used to deal with any number of hidden layers with any number of nodes. The weights and biases were initialized in the constructor based on the weight initialization type: **"Random"**, **"Constant"** or **"Zero"**.
Activation functions and their derivatives were implemented from scratch as helper methods. These included **Sigmoid**, **tanH**, and **ReLU** for the hidden layers and **Softmax** for the output layer. **Categorical Cross Entropy Loss** was also defined as the cost function for the multi-class classification.

The `forward_propagation` method dealt with the forward propagation algorithm of the input. It initialized the layers and also performed matrix multiplications and applied activation functions. The layers were stored as Numpy arrays in lists.
The `backward_propagation` method dealt with the back propagation of the error. The derivatives of the activation functions and matrix multiplication calculations were used for calculating the *stochastic gradient,* which was used to update the weights and bias terms.

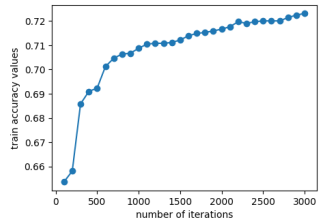
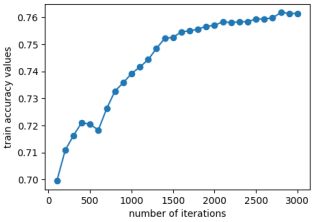
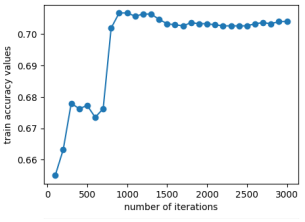
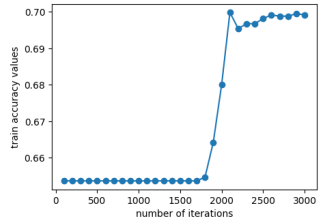
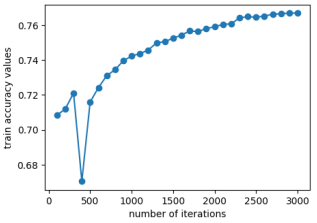
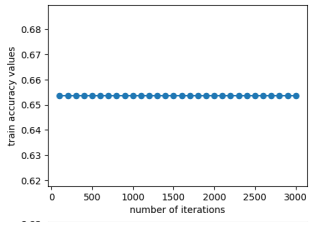
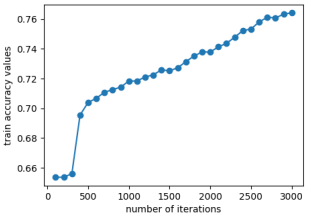
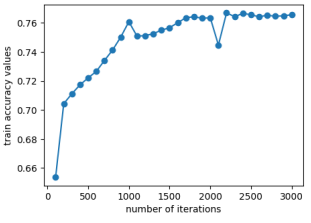
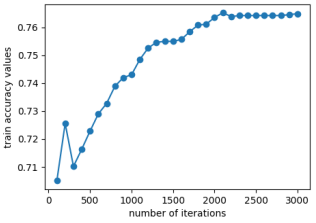
Other helper methods were defined for the initialization of the algorithm, performing testing and returning results, and also, visualization using graph plots of the cost v/s the number of iterations curve, validation score v/s the number of iterations curve etc.

The training, validation, and testing sets were used in the model evaluation. The following results and visualization plots were obtained by varying the activation functions and the types of weight initializations, as shown. The learning rates and the number of hidden layers and nodes were varied in the below cases in order to improve the performance of the model.

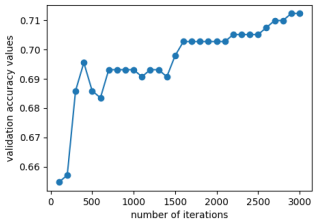
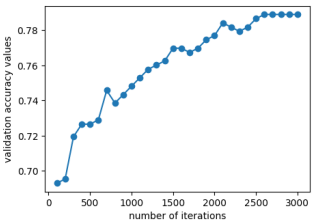
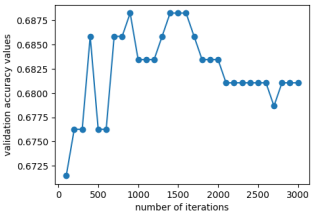
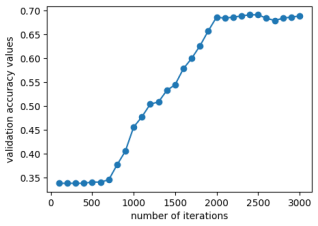
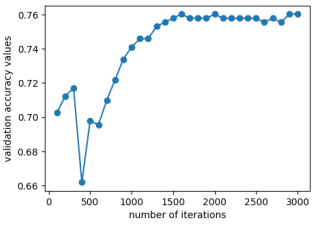
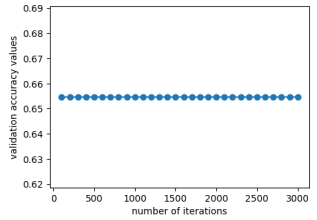
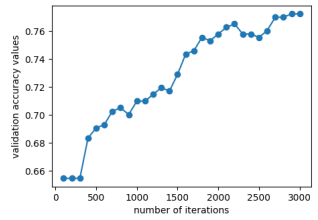
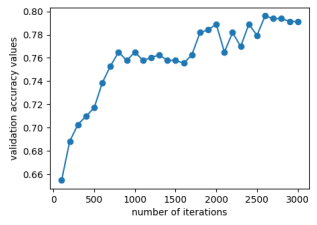
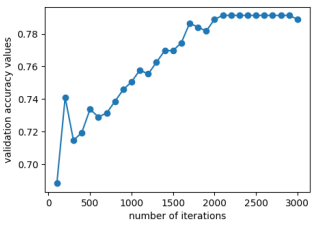
Cost Values v/s Number of iterations

Activation Function → **Weight Initialization ↓**	Sigmoid	ReLU	tanH
Random			
Constant			
Zero			

Training accuracy values v/s Number of iterations

Activation Function → **Weight Initialization ↓**	Sigmoid	ReLU	tanH
Random			
Constant			
Zero			

Validation accuracy values v/s Number of iterations

Activation Function → **Weight Initialization ↓**	Sigmoid	ReLU	tanH
Random			
Constant			
Zero			

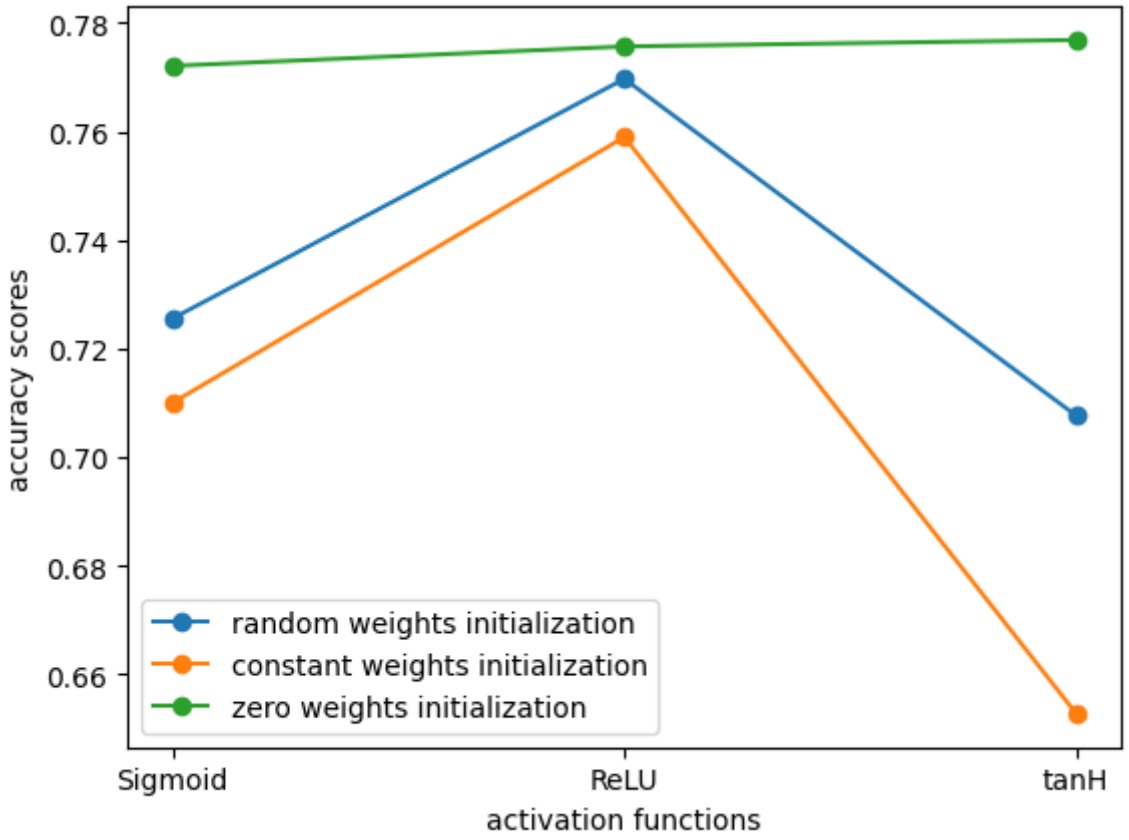
Testing accuracy scores

Activation Function → **Weight Initialization ↓**	Sigmoid	ReLU	tanH
Random	0.7255369928400954	0.7696897374701671	0.7076372315035799
Constant	0.7100238663484487	0.7589498806682577	0.652744630071599
Zero	0.7720763723150358	0.7756563245823389	0.7768496420047732

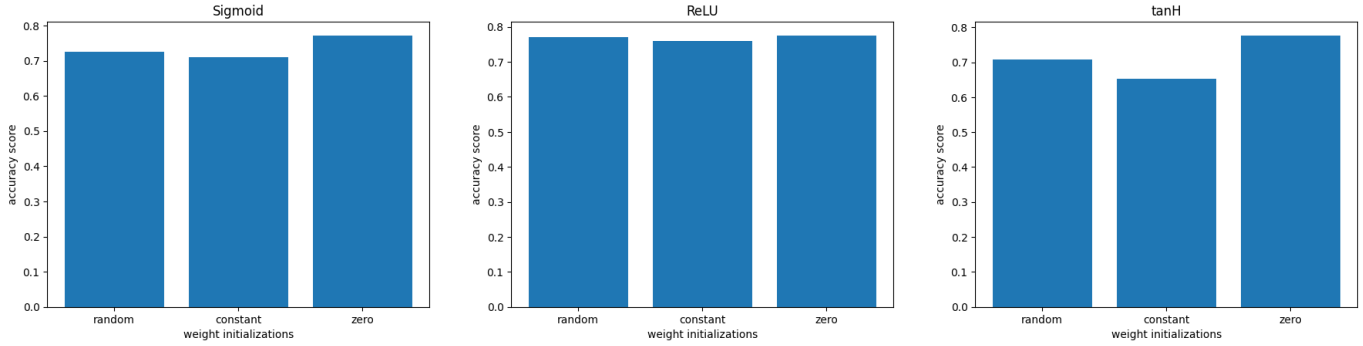
Based on the above plots and accuracy scores, the following observations were made:-
(All these experiments were performed on a single hidden layer)

- All the activation functions were able to achieve the highest accuracy of around 0.77 when initialized with “Zero” weights.
- ReLU (Rectified Linear Unit) function gave a high accuracy score for all the weight initialization types.
- tanH function performed poorly when initialized with constant weights, and did not show any significant change in cost or accuracy scores.
- Zero weight initialization also showed high performance in the validation scores for all the three activation functions.

Comparison among different activation functions based on the test accuracy scores



Comparison among different weight initializations



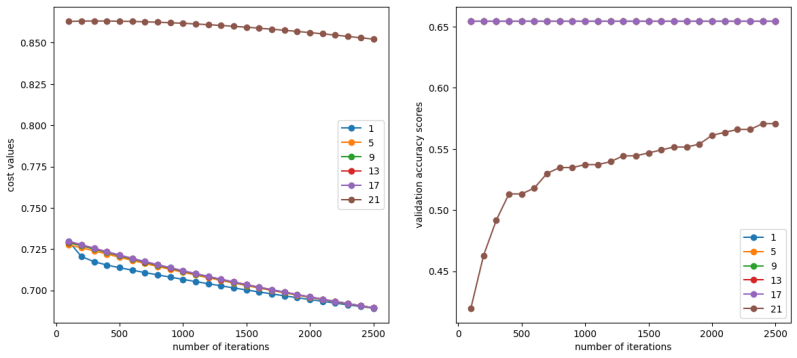
`Sigmoid` `ReLU` `tanH`

Varying the number of hidden nodes (assuming a single hidden layer)

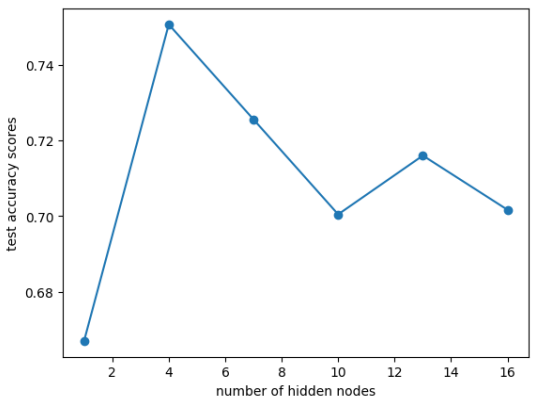
Plots for all the activation functions for all the weight initialization types were plotted for cost values v/s the number of iterations and also for validation accuracy values v/s the number of iterations for different values of hidden nodes.

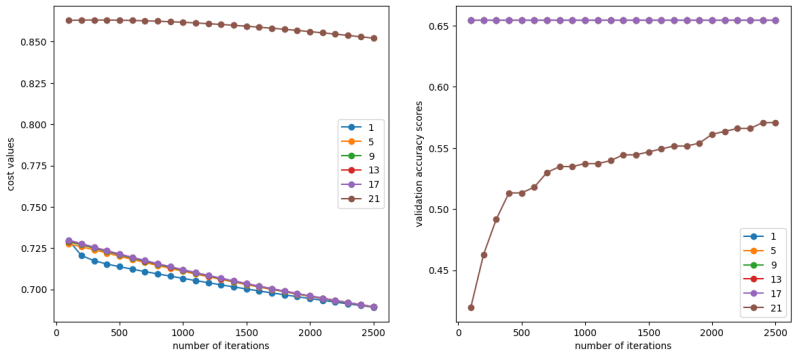
Plots for test accuracy values v/s the number of hidden nodes were also plotted for the same.

Sigmoid

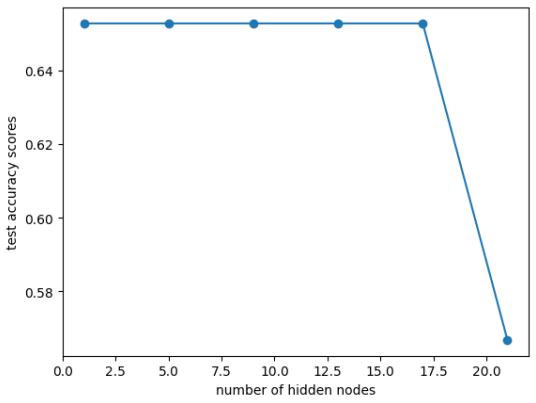


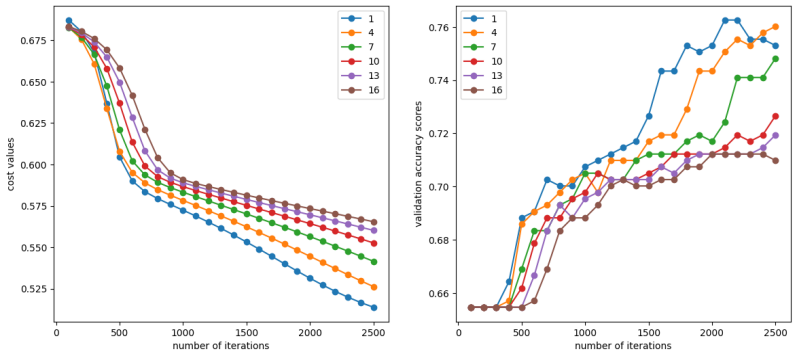
Random Weight Initialization



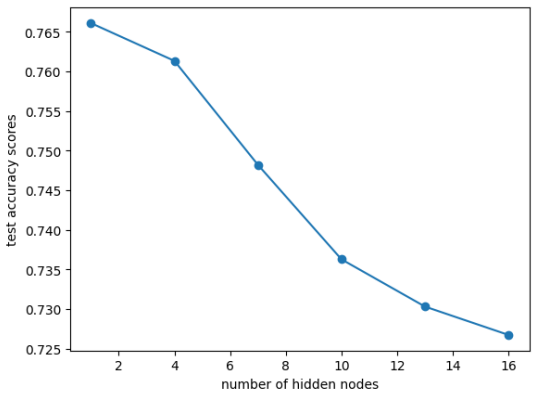


Constant Weight Initialization





Zero Weight Initialization

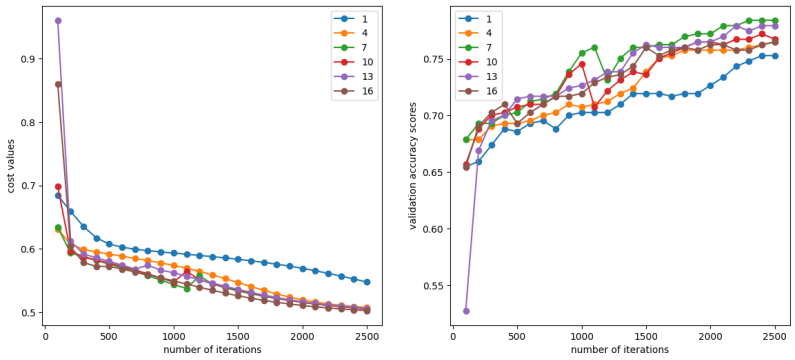
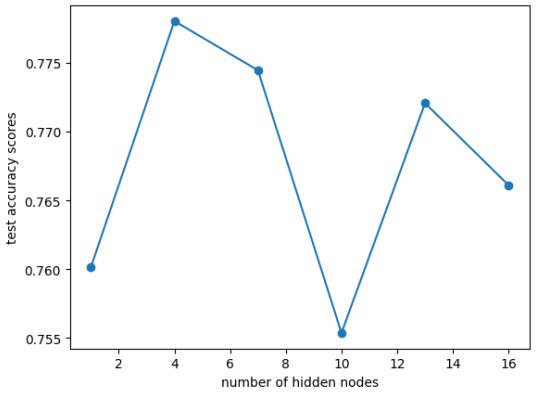
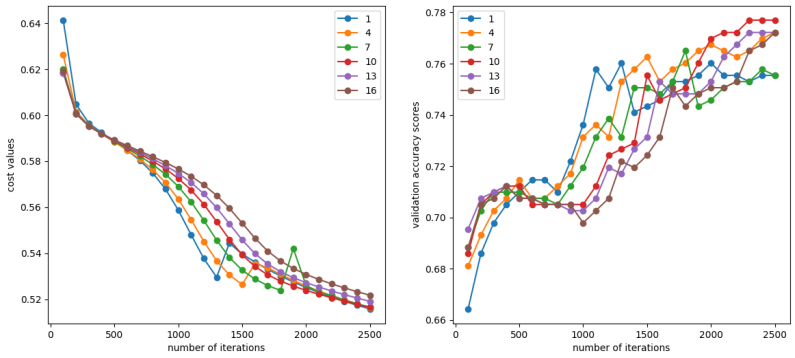
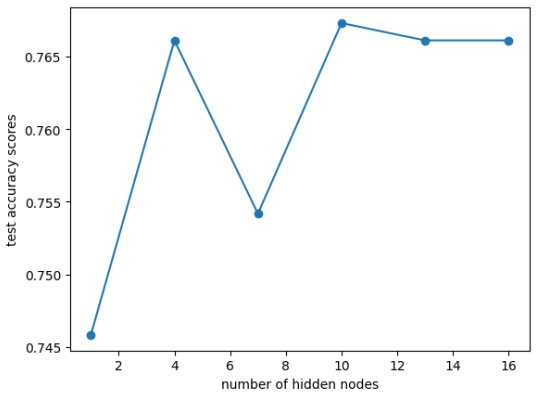
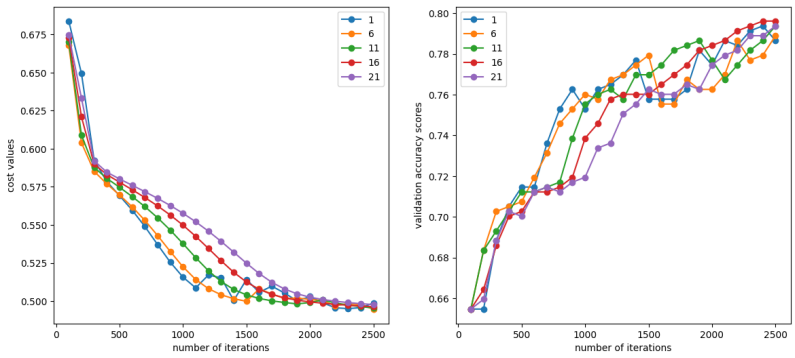
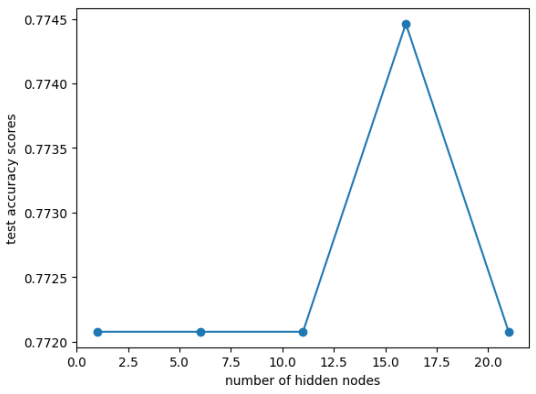


Observations for Sigmoid:-

With random weight initialization, the best performance was observed at around 4 hidden nodes, and the performance decreased on decreasing or increasing the number of nodes further.

For constant weight initialization, high performance was obtained for nodes lying in the range between 1 and 18, and decreased afterwards.

For zero-weight initialization, the best performance was seen for a lower number of nodes and decreased on increasing the number.

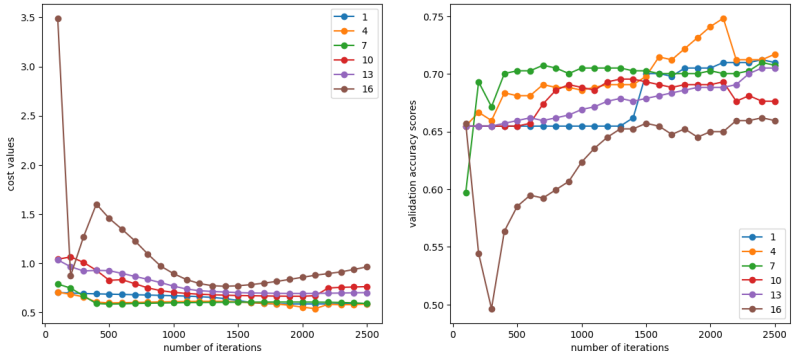
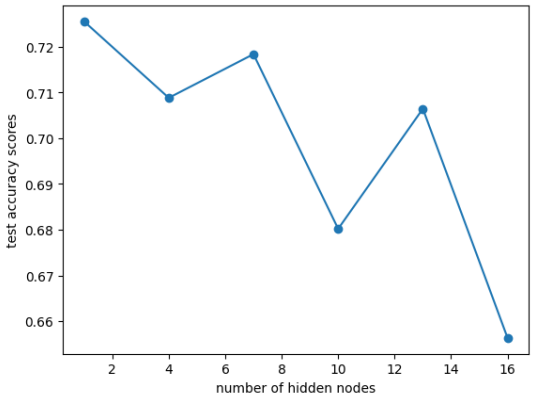
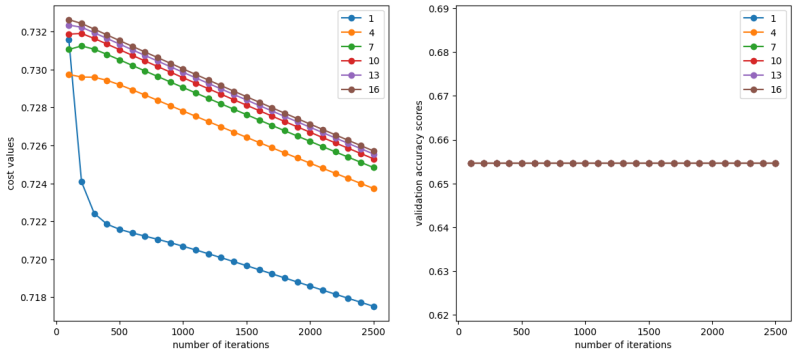
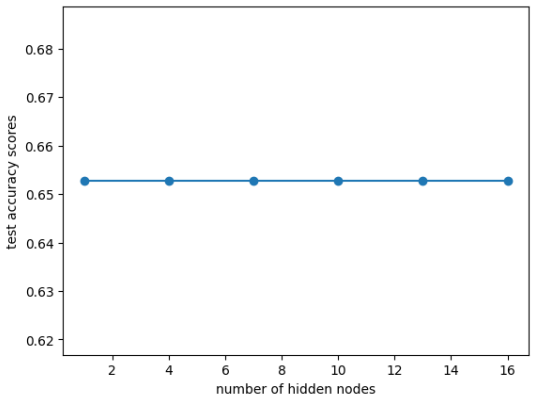
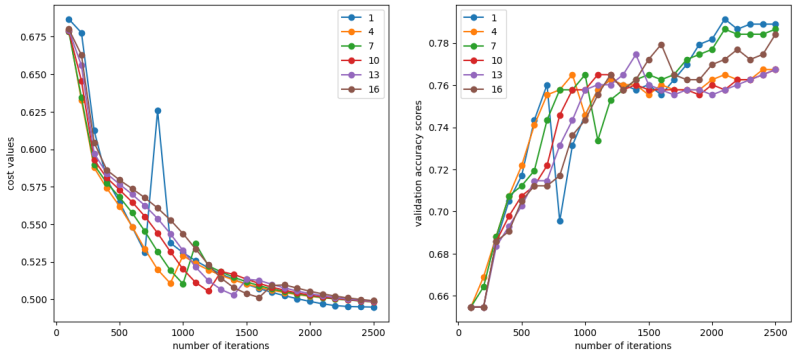
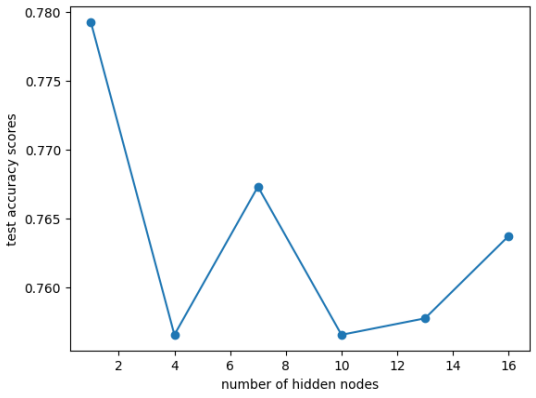
ReLU	
 *Random Weight Initialization*	
 *Constant Weight Initialization*	
 *Zero Weight Initialization*	

Observations for ReLU:-

With random weight initializations, the best performance was observed at around 4 hidden nodes, and the performance decreased on increasing or decreasing the number of nodes with a significant amount of fluctuation in between.

For constant weight initialization, high performance was observed at around 4 nodes and then again at 10 nodes, after which there was only a slight decrease.

For zero-weight initialization, the highest performance was seen at 16 nodes. The performance was constant initially and then suddenly increased and then decreased suddenly again.

tanH		
 *Random Weight Initialization*		
 *Constant Weight Initialization*		
 *Zero Weight Initialization*		

Observations for tanH:-

With random and zero-weight initializations, the best performance was observed at around 1-2 hidden nodes, and the performance decreased on increasing the number of nodes further with a few fluctuations in between.

For constant weight initialization, low performance was observed, which did not change with changing the number of nodes. It can be said that convergence has taken place.

Saving and loading the weights of our model (by Pickling)

It was performed using the `Pickle` Python module. The pickle file for the weights got saved in the specified directory and could be loaded again.

Saving the weights

```
save_name = 'model_weights.pkl'
with open(save_name, 'wb') as file:
    pickle.dump(mlp_sigmoid_random.weights, file)
```

Loading the weights

```
with open(save_name, 'rb') as file:
    pickled_weights = pickle.load(file)

pickled_weights
```

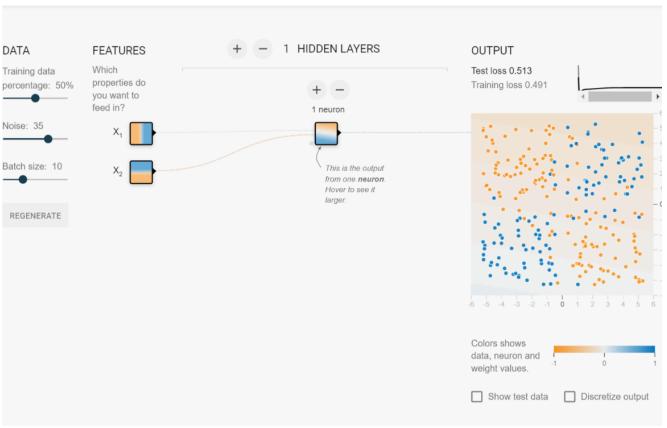
```
[array([[0.6653447482229055, 0.6301511293695229, 1.0396235691976377,
         0.3538781584376763, 0.5035681538066819, 0.5109019033780688,
         0.44200001236525416, 0.030721649658227355],
        [0.2474806479838925, 0.489488157743387, 0.2575069908362247,
         0.8069430595301148, 0.9913474563089504, -0.3241267397700564,
         0.13822237850548133, 1.0581444414010837],
        [0.24118483562167797, 0.9935841099252846, 0.9161484695051303,
         0.32293716254663457, 0.9985914696920786, 0.39349676631060604,
         0.39535877053344787, 0.7736319974253744],
        [0.10663018307229079, 0.0870822122783988, 0.18800308630069512,
         0.4472542053905035, -0.2514708558083048, 0.5992928341227215,
         0.7207298757514962, 0.5936592209788302]], dtype=object),
 array([[ 1.1183309 , -0.53283945,  0.11516306,  1.10773921],
        [ 0.21138236,  0.87542436,  1.14238485,  0.38988974],
        [-0.18622078,  1.18694167,  0.08541588,  0.63290946]])]
```

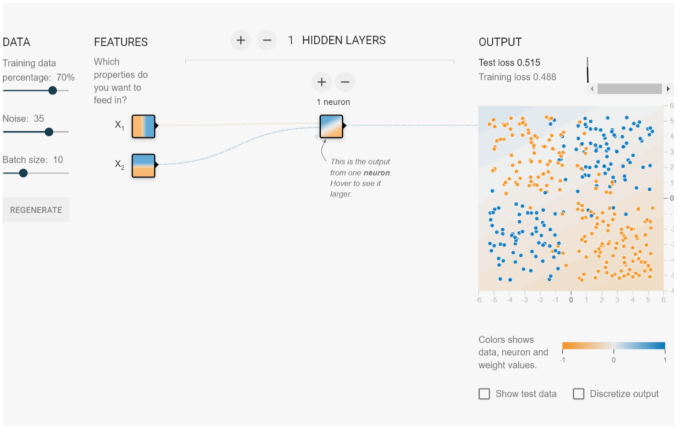
(Example of saving and loading our model weights using Pickle module)

Question-3

Google Playground

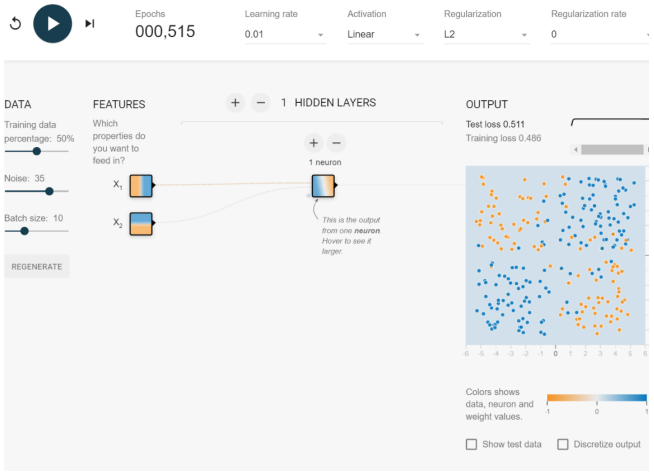
Increasing the model size does not improve the fit, and it starts converging more quickly, as shown below. The model is underfitting and gives a high loss value. The linear activation function does not work well in this case.

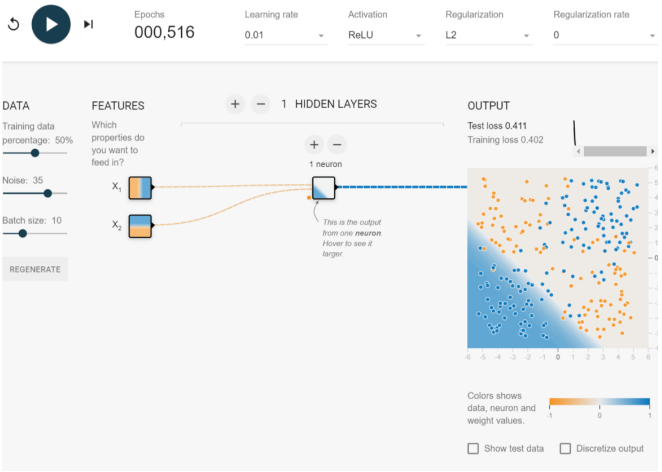




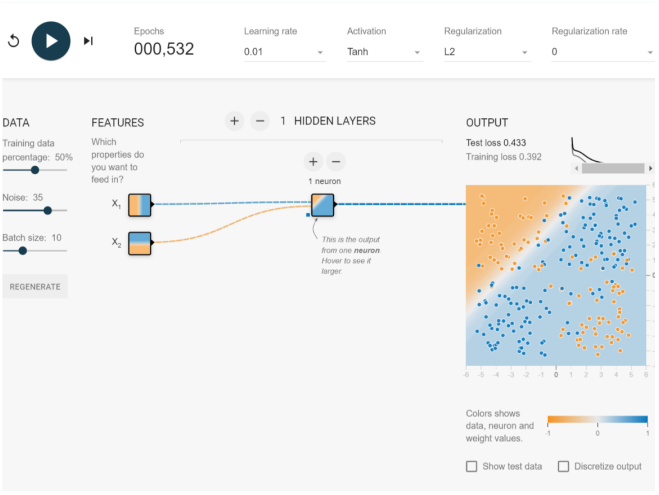
After hitting the Reset the Network button to get a new random initialization about 4-5 times, we observe that the outputs converge to a similar shape.

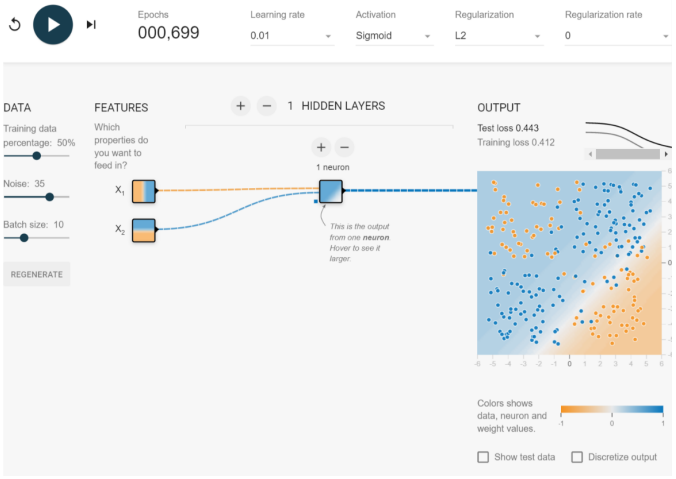
The role of initialization in non-convex optimization is significant because it can impact the quality of the solution obtained by the optimization algorithm. In neural networks, the optimization problem is typically non-convex, which means that it has multiple local minima and a single global minimum. Different initializations of the model parameters can lead to different local minima, which may have significantly different performance on the validation or test data.





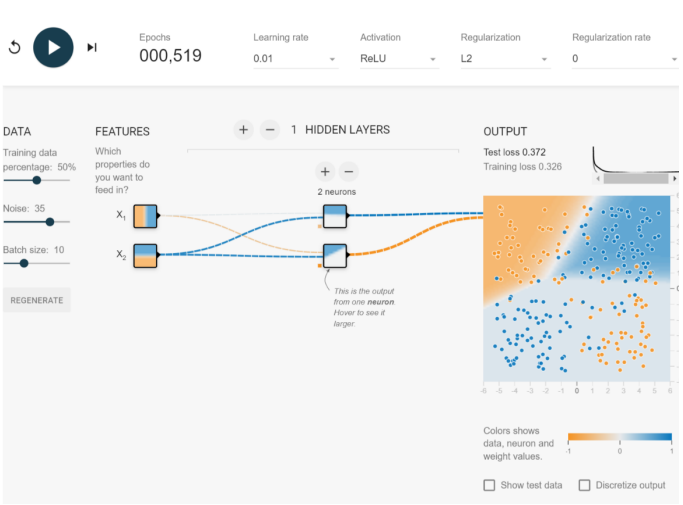
Linear *ReLU*

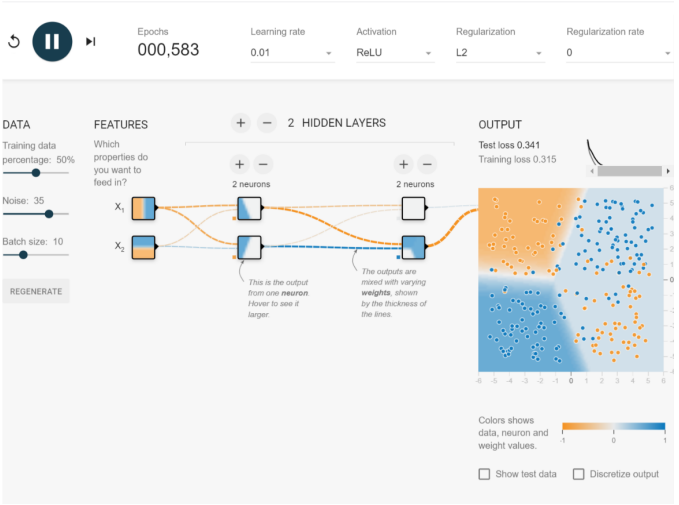




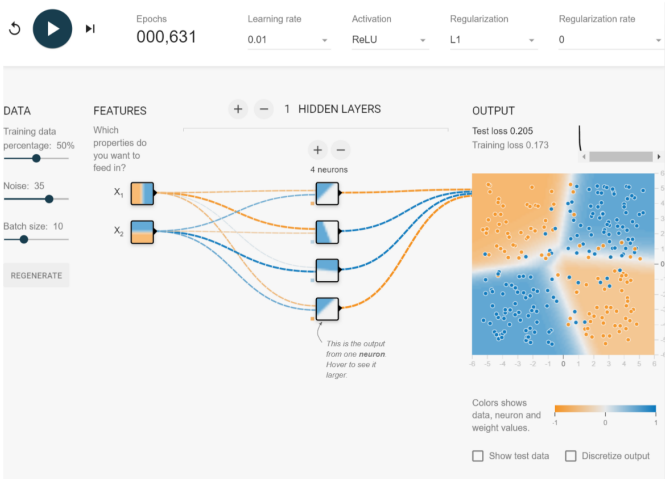
tanH *Sigmoid*

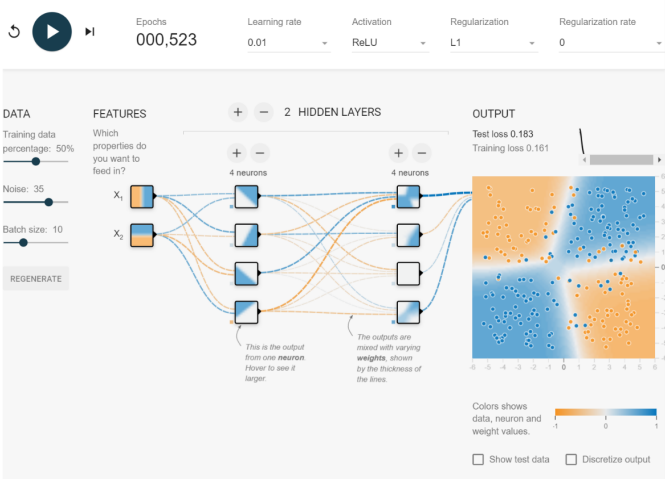
By changing the number of layers or nodes, extra stability is added to the output. The loss is decreased.





Changing the number of nodes to 4 gave better results.



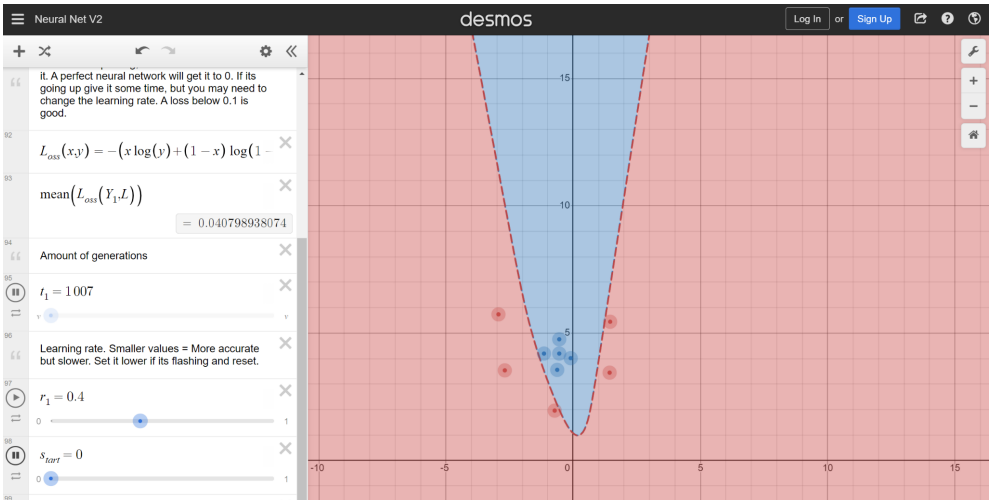


The best test loss observed was around 0.180. Not much of a significant difference in the test loss was observed after adding 4 nodes to one of the hidden layers.

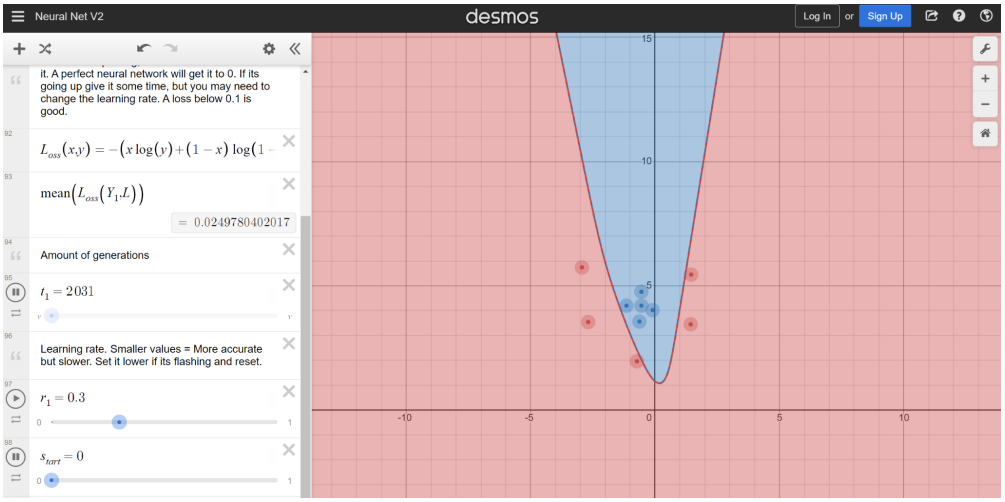
Neural Nets V2 Desmos Visualized

Training stability observations by varying the learning rate:-

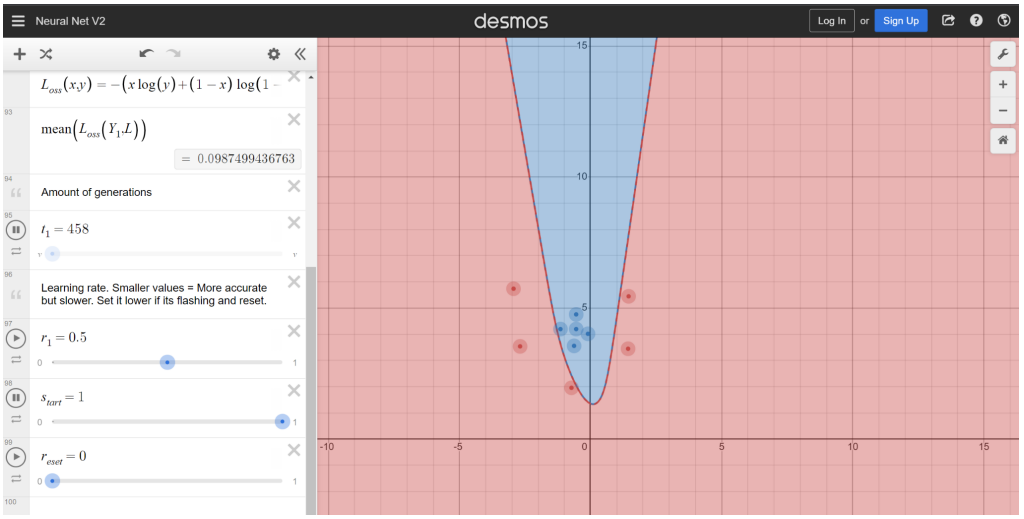
A loss of around 0.04 was obtained at a learning rate of 0.4 at around 1000 sec.



A loss of 0.025 was observed at the learning rate of 0.3 in around 2000 sec.
(very slow as compared to other learning rates)



At learning rate = 0.5, flashing was observed after around 450 seconds, with a loss of around 0.098.



Tweaking the learning rate can affect the stability of the training process in several ways. A high learning rate can cause the optimization algorithm to overshoot the minimum, leading to oscillations or divergence. On the other hand, a low learning rate can cause optimization to get stuck in local minima or plateaus, slowing down the convergence.