CPSC 8430
DEEP LEARNING
HOMEWORK 1 REPORT
SREERAM PALADUGU

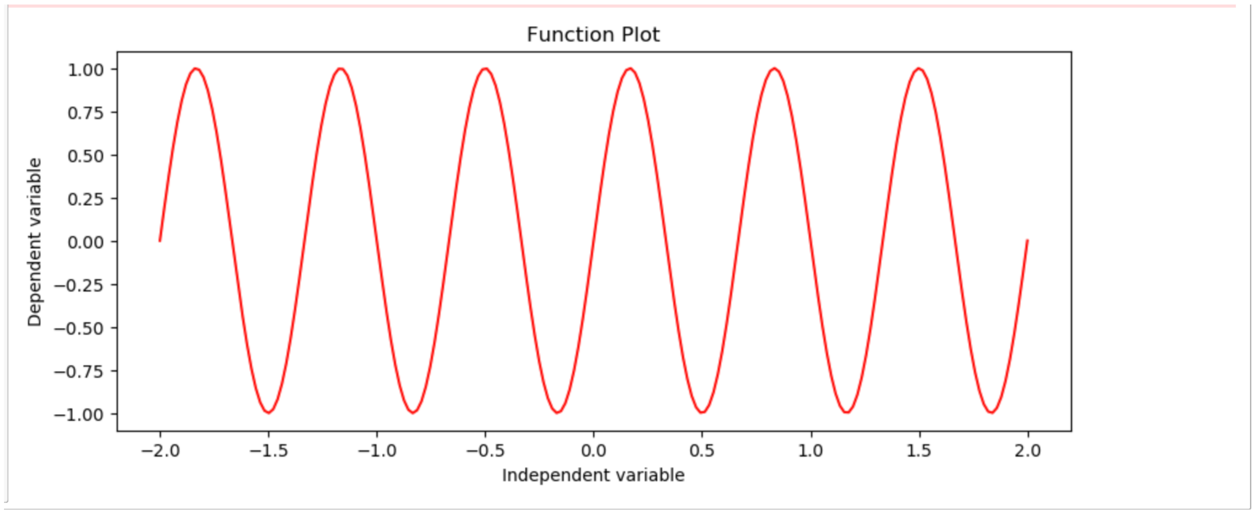
Github link- https://github.com/sreerampaladugu10/deeplearning

Part-1
Deep vs. shallow learning

simulating function 1 -

Firstly, I used PyTorch to generate a sine wave for the function, $\frac{\sin(5\pi x)}{5\pi x}$ using a random number generator with a seed value of 0 to ensure reproducible results. The random number generator is used to generate x and y data which is then converted into PyTorch tensors and plotted into a graph.

(Img graph1)



Neural network models-

For this task, I have simulated the following function $\frac{\sin(5\pi x)}{5\pi x}$ with three different Multi-Layer Perceptron (MLP) or Deep Neural Networks (DNN) models. Described below

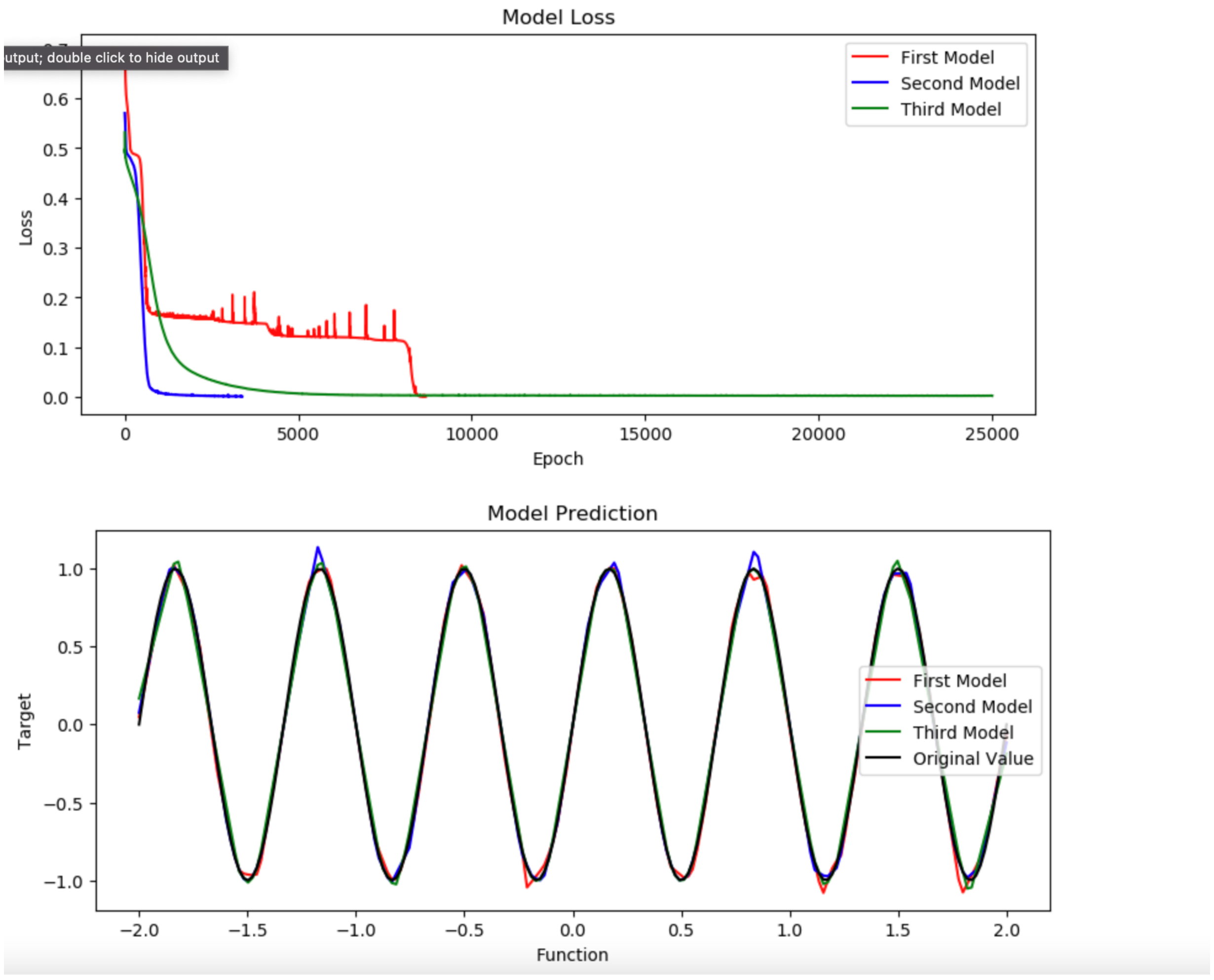
In terms of the number of parameters, model 1 has 571 parameters, model 2 has 572, and model 3 has 571.

In terms of architecture, all three models use “leaky_relu” as their activation function, “MSELoss” as their loss function, and “Adam” as their Optimizer Function.

In addition, models 1 and 2 have the same hyperparameters, with their Learning Rate being “0. 0012” and Weight Decay being “1e-4” whereas model 3 has a Learning Rate being of “0. 0011” and a weight Decay of “1e-4”

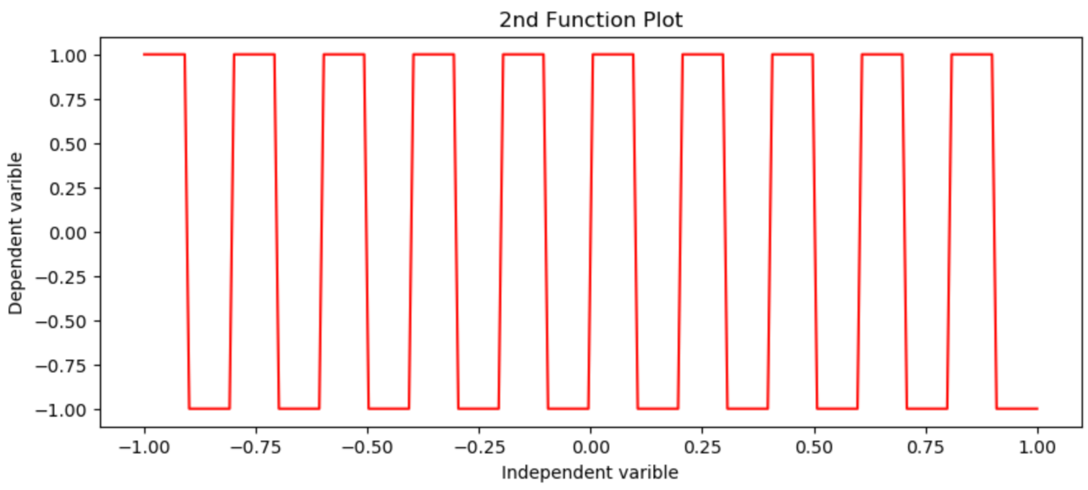
In terms of the performance of the models -

The graphs below compare the (model loss vs. epoch) and model predictions for all three models.

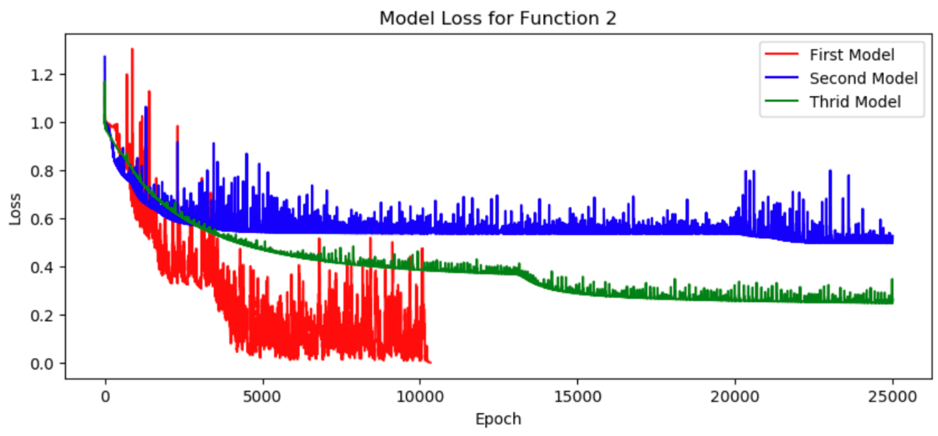


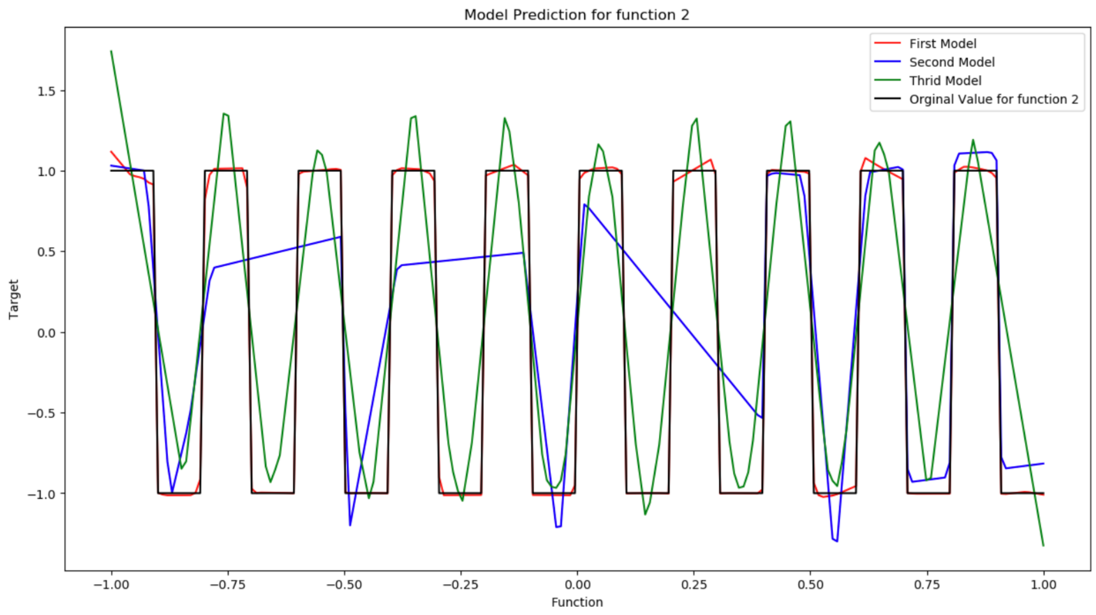
Simulating Function 2 -

Similar to the first function, to simulate this function, $\operatorname{sgn}(\sin(5\pi x))$ I utilized PyTorch to generate a sine wave function. I did so by using a random number generator with a seed value of 1 to ensure consistent results. This random number generator was utilized to produce the x and y data points. These data points were then converted into PyTorch tensors and were plotted onto a graph for visualization.



The models for this function more or less have the same parameters but a different hyperparameter learning rate of 0.009.





The graphs above show the (loss vs. epoch) graph and (prediction vs. function) graph for the second function, comparing the performance of all three models.

Observations-

Based on the observation of the two prediction graphs, it can be concluded that the models performed well on the first graph with little deviation from the actual values. Still, there were noticeable deviations in the second graph. The first model seems to have the most accurate prediction among the three models in the second graph, while the third model had the least accurate prediction.

1.2 Optimization (training on minst)

About the dataset-

Firstly we set the manual seed for the PyTorch random number generator to 0. Then, it loads the MNIST dataset for training and testing using the torchvision module. The MNIST dataset is a set of handwritten digits and is commonly used as a benchmark for image classification tasks.

The train_dataset loads the training data, while the test_dataset loads the testing data. The data is transformed into tensors using the transforms.ToTensor() function, which converts the data into PyTorch tensors.

CNN Models -

For this task, I have created 3 CNN models which have some similarities and differences in terms of architecture.

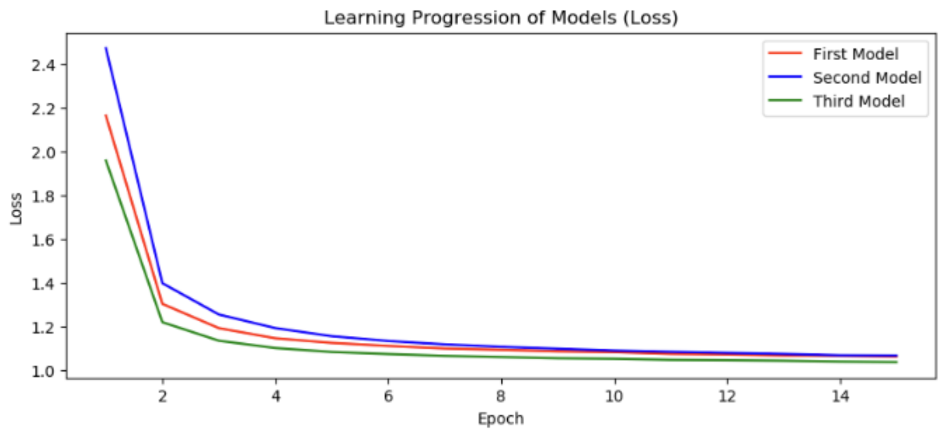
CNN-1 is a Convolutional Neural Network model that contains 2 Convolution layers with a kernel size of 4, followed by Max Pooling of size 2 and stride 2. The model has 2 fully connected layers with the ReLU activation function, and the total number of parameters is 25550. The loss function

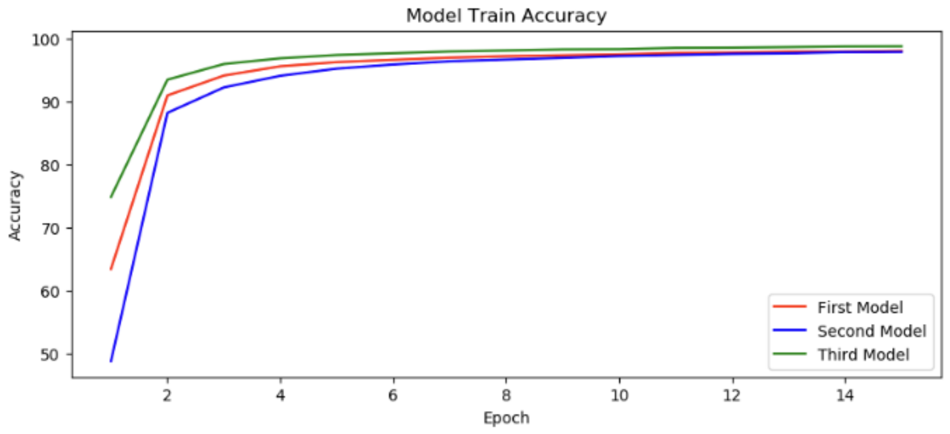
CNN-2 is a Convolutional Neural Network with two Convolution layers with a kernel size of 4 and uses Max Pooling with a pool size of 2 and strides of 2. It has 4 fully connected dense layers and uses the "relu" activation function. The total number of parameters is 25570. The loss function used is "CrossEntropyLoss", and the optimizer is "Adam". The hyperparameters include a learning rate of 0.0001, weight decay of 1e-4, and drop out of 0.25.

CNN-3 has two convolutional layers with kernel size 5, and max pooling with a pool size of 2 and strides of 2. This model has only one dense layer, and the activation function used is ReLU. This model's total number of parameters is 25550, and the loss function used is "CrossEntropyLoss." The optimization function used is "Adam." The model's hyperparameters include a learning rate of 0.0001, weight decay of 1e-4, and drop out of 0.25.

Training the model -

the model is trained by performing forward and backward passes in each iteration. The number of iterations is defined by the num_epochs input argument. The function returns lists of epochs, training losses, training accuracy, and average training loss per epoch.





```
CNN1 Test Accuracy: 98.78 %
CNN2 Test Accuracy: 97.92 %
CNN3 Test Accuracy: 98.94 %
```

The graph above depicts the (loss vs. epoch) for the models, and the second graph shows the (accuracy vs. epoch) for all three models.

Observation-

Overall, the differences in the number of layers, the number of neurons in each layer, and the parameters in the convolutional layers can affect the model's ability to learn and generalize to new data.

In our case, we can observe that CNN3 has the highest accuracy of 98.94%, followed by CNN1, which has an accuracy of 98.78%, and CNN2 is the least accurate of all 3 models, with an accuracy of 98.66%.

The loss factor for CNN3 was 0.0744, the loss factor for CNN1 was 0.0880, and the loss factor for CNN2 was 0.0242. These loss factors indicate how well the model fits the training data, with lower loss factors indicating a better fit. The difference between the loss factors may be due to differences in the architectures of the three models or due to other hyperparameters such as the learning rate or optimizer used.

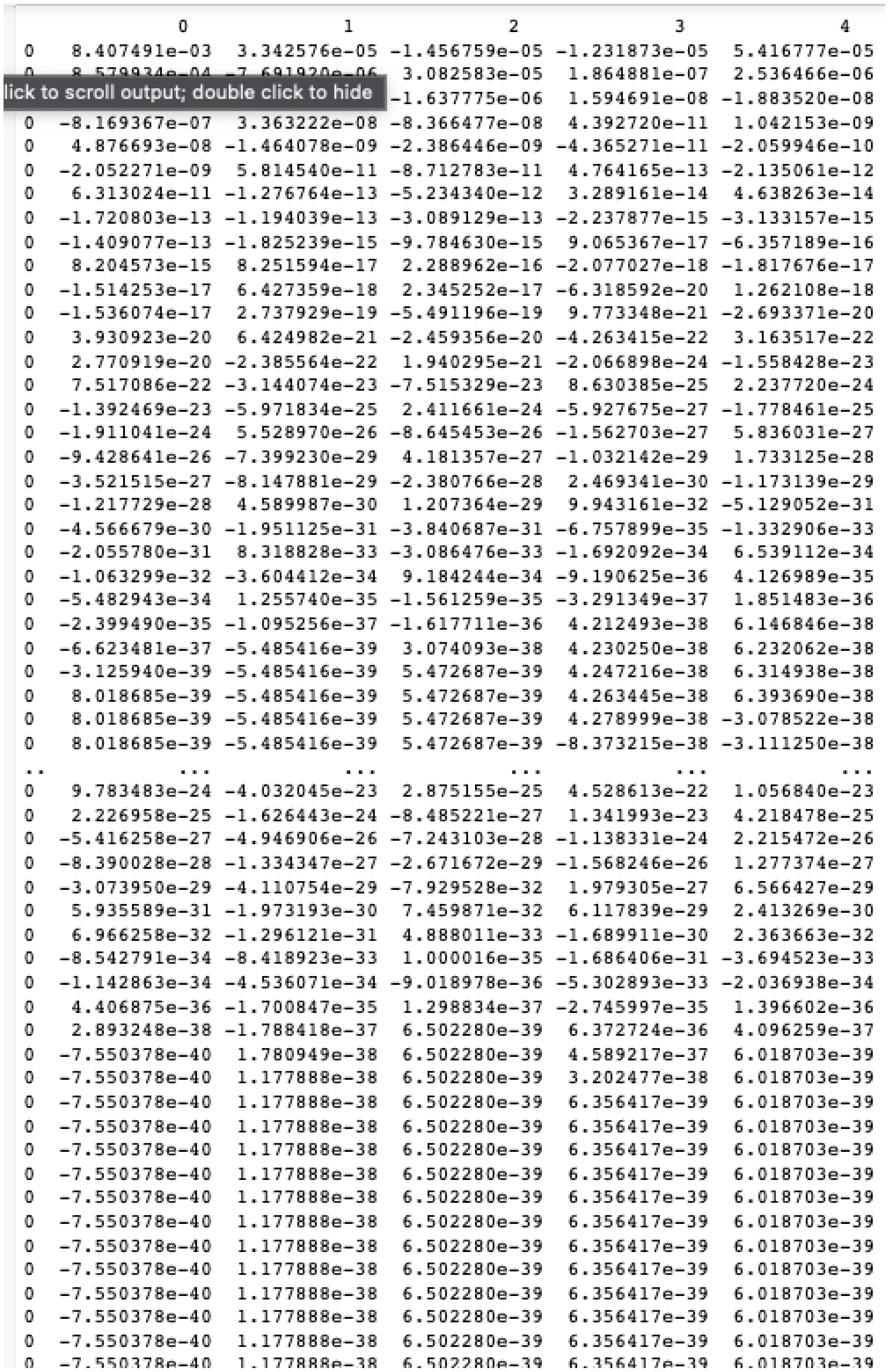
2.1 Optimizing the visualization process-

Firstly we implement a simple deep neural network (DNN) in PyTorch. It has three fully connected (fc) layers, each using a ReLU activation function. The input layer has 784 neurons, representing the flattened image size of 28x28 pixels. The first hidden layer (fc1) has 500 neurons, the second hidden layer (fc2) has 50 neurons, and the output layer (fc3) has 10 neurons, which is used for the 10 classes of the MNIST dataset. The total number of parameters in this model is 397510.

In the forward method, the input data is first flattened into a one-dimensional tensor, and then passed through the three fully connected layers with ReLU activation functions. Finally, the output from the last fully connected layer (fc3) is returned as the final prediction.

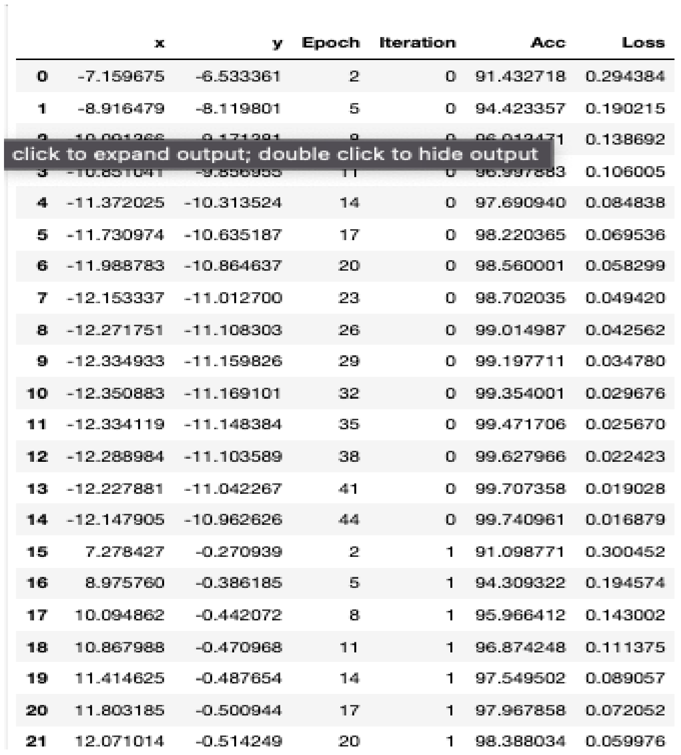
Now we implement a training loop for a DNN model using the PyTorch library. It trains the model 6 times, using the same model structure but with different random initializations. CrossEntropyLoss is used as the loss function and Adam optimizer is used with a learning rate of 0.0004 and weight decay of 1e-4. The results from each training iteration are stored.

For each of the 6 runs of the model, the weights are collected for every epoch for a total of 45 epochs. The results are stored in a pandas DataFrame shown below.

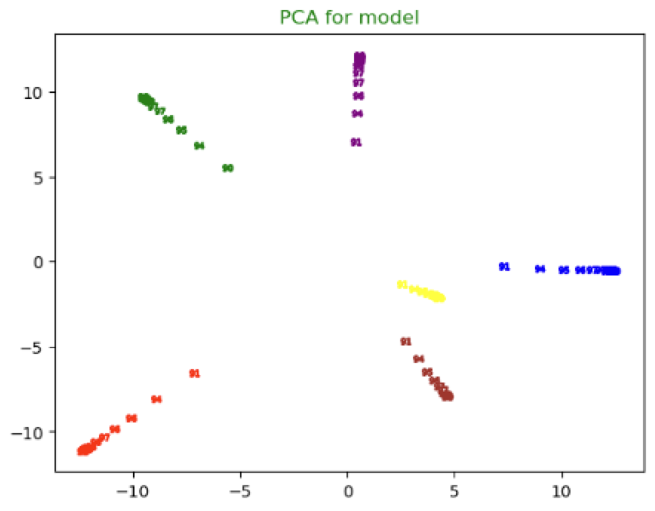


```
                0              1              2              3              4
0    8.407491e-03   3.342576e-05  -1.456759e-05  -1.231873e-05   5.416777e-05
0    8.579934e-04  -7.691920e-06   3.082583e-05   1.864881e-07   2.536466e-06
lick to scroll output; double click to hide   -1.637775e-06   1.594691e-08  -1.883520e-08
0   -8.169367e-07   3.363222e-08  -8.366477e-08   4.392720e-11   1.042153e-09
0    4.876693e-08  -1.464078e-09  -2.386446e-09  -4.365271e-11  -2.059946e-10
0   -2.052271e-09   5.814540e-11  -8.712783e-11   4.764165e-13  -2.135061e-12
0    6.313024e-11  -1.276764e-13  -5.234340e-12   3.289161e-14   4.638263e-14
0   -1.720803e-13  -1.194039e-13  -3.089129e-13  -2.237877e-15  -3.133157e-15
0   -1.409077e-13  -1.825239e-15  -9.784630e-15   9.065367e-17  -6.357189e-16
0    8.204573e-15   8.251594e-17   2.288962e-16  -2.077027e-18  -1.817676e-17
0   -1.514253e-17   6.427359e-18   2.345252e-17  -6.318592e-20   1.262108e-18
0   -1.536074e-17   2.737929e-19  -5.491196e-19   9.773348e-21  -2.693371e-20
0    3.930923e-20   6.424982e-21  -2.459356e-20  -4.263415e-22   3.163517e-22
0    2.770919e-20  -2.385564e-22   1.940295e-21  -2.066898e-24  -1.558428e-23
0    7.517086e-22  -3.144074e-23  -7.515329e-23   8.630385e-25   2.237720e-24
0   -1.392469e-23  -5.971834e-25   2.411661e-24  -5.927675e-27  -1.778461e-25
0   -1.911041e-24   5.528970e-26  -8.645453e-26  -1.562703e-27   5.836031e-27
0   -9.428641e-26  -7.399230e-29   4.181357e-27  -1.032142e-29   1.733125e-28
0   -3.521515e-27  -8.147881e-29  -2.380766e-28   2.469341e-30  -1.173139e-29
0   -1.217729e-28   4.589987e-30   1.207364e-29   9.943161e-32  -5.129052e-31
0   -4.566679e-30  -1.951125e-31  -3.840687e-31  -6.757899e-35  -1.332906e-33
0   -2.055780e-31   8.318828e-33  -3.086476e-33  -1.692092e-34   6.539112e-34
0   -1.063299e-32  -3.604412e-34   9.184244e-34  -9.190625e-36   4.126989e-35
0   -5.482943e-34   1.255740e-35  -1.561259e-35  -3.291349e-37   1.851483e-36
0   -2.399490e-35  -1.095256e-37  -1.617711e-36   4.212493e-38   6.146846e-38
0   -6.623481e-37  -5.485416e-39   3.074093e-38   4.230250e-38   6.232062e-38
0   -3.125940e-39  -5.485416e-39   5.472687e-39   4.247216e-38   6.314938e-38
0    8.018685e-39  -5.485416e-39   5.472687e-39   4.263445e-38   6.393690e-38
0    8.018685e-39  -5.485416e-39   5.472687e-39   4.278999e-38  -3.078522e-38
0    8.018685e-39  -5.485416e-39   5.472687e-39  -8.373215e-38  -3.111250e-38
..            ...            ...            ...            ...            ...
0    9.783483e-24  -4.032045e-23   2.875155e-25   4.528613e-22   1.056840e-23
0    2.226958e-25  -1.626443e-24  -8.485221e-27   1.341993e-23   4.218478e-25
0   -5.416258e-27  -4.946906e-26  -7.243103e-28  -1.138331e-24   2.215472e-26
0   -8.390028e-28  -1.334347e-27  -2.671672e-29  -1.568246e-26   1.277374e-27
0   -3.073950e-29  -4.110754e-29  -7.929528e-32   1.979305e-27   6.566427e-29
0    5.935589e-31  -1.973193e-30   7.459871e-32   6.117839e-29   2.413269e-30
0    6.966258e-32  -1.296121e-31   4.888011e-33  -1.689911e-30   2.363663e-32
0   -8.542791e-34  -8.418923e-33   1.000016e-35  -1.686406e-31  -3.694523e-33
0   -1.142863e-34  -4.536071e-34  -9.018978e-36  -5.302893e-33  -2.036938e-34
0    4.406875e-36  -1.700847e-35   1.298834e-37  -2.745997e-35   1.396602e-36
0    2.893248e-38  -1.788418e-37   6.502280e-39   6.372724e-36   4.096259e-37
0   -7.550378e-40   1.780949e-38   6.502280e-39   4.589217e-37   6.018703e-39
0   -7.550378e-40   1.177888e-38   6.502280e-39   3.202477e-38   6.018703e-39
0   -7.550378e-40   1.177888e-38   6.502280e-39   6.356417e-39   6.018703e-39
0   -7.550378e-40   1.177888e-38   6.502280e-39   6.356417e-39   6.018703e-39
0   -7.550378e-40   1.177888e-38   6.502280e-39   6.356417e-39   6.018703e-39
0   -7.550378e-40   1.177888e-38   6.502280e-39   6.356417e-39   6.018703e-39
0   -7.550378e-40   1.177888e-38   6.502280e-39   6.356417e-39   6.018703e-39
0   -7.550378e-40   1.177888e-38   6.502280e-39   6.356417e-39   6.018703e-39
0   -7.550378e-40   1.177888e-38   6.502280e-39   6.356417e-39   6.018703e-39
0   -7.550378e-40   1.177888e-38   6.502280e-39   6.356417e-39   6.018703e-39
0   -7.550378e-40   1.177888e-38   6.502280e-39   6.356417e-39   6.018703e-39
0   -7.550378e-40   1.177888e-38   6.502280e-39   6.356417e-39   6.018703e-39
0   -7.550378e-40   1.177888e-38   6.502280e-39   6.356417e-39   6.018703e-39
0   -7.550378e-40   1.177888e-38   6.502280e-39   6.356417e-39   6.018703e-39
```

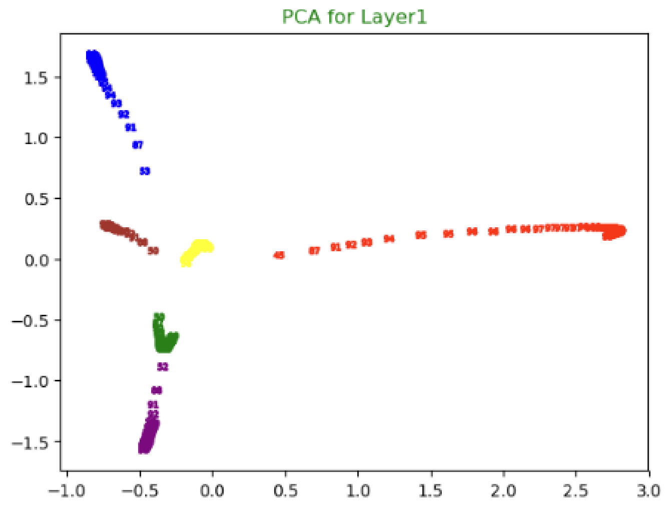
We reduced the dimensionality of the weight data by applying PCA after every 3rd epoch of sorting. This transformation yielded a 2-dimensional representation that facilitated the extraction of valuable insights from the otherwise high-dimensional data. The result of this process, which aims to retain as much information as possible, is presented in the form of a table and accompanying graphs below.


click to expand output; double click to hide output

	x	y	Epoch	Iteration	Acc	Loss
0	-7.159675	-6.533361	2	0	91.432718	0.294384
1	-8.916479	-8.119801	5	0	94.423357	0.190215
2	-10.091366	-9.171381	8	0	96.012471	0.138692
3	-10.851041	-9.856955	11	0	96.997883	0.106005
4	-11.372025	-10.313524	14	0	97.690940	0.084838
5	-11.730974	-10.635187	17	0	98.220365	0.069536
6	-11.988783	-10.864637	20	0	98.560001	0.058299
7	-12.153337	-11.012700	23	0	98.702035	0.049420
8	-12.271751	-11.108303	26	0	99.014987	0.042562
9	-12.334933	-11.159826	29	0	99.197711	0.034780
10	-12.350883	-11.169101	32	0	99.354001	0.029676
11	-12.334119	-11.148384	35	0	99.471706	0.025670
12	-12.288984	-11.103589	38	0	99.627966	0.022423
13	-12.227881	-11.042267	41	0	99.707358	0.019028
14	-12.147905	-10.962626	44	0	99.740961	0.016879
15	7.278427	-0.270939	2	1	91.098771	0.300452
16	8.975760	-0.386185	5	1	94.309322	0.194574
17	10.094862	-0.442072	8	1	95.966412	0.143002
18	10.867988	-0.470968	11	1	96.874248	0.111375
19	11.414625	-0.487654	14	1	97.549502	0.089057
20	11.803185	-0.500944	17	1	97.967858	0.072052
21	12.071014	-0.514249	20	1	98.388034	0.059976



```
<Figure size 5000x2500 with 0 Axes>
```

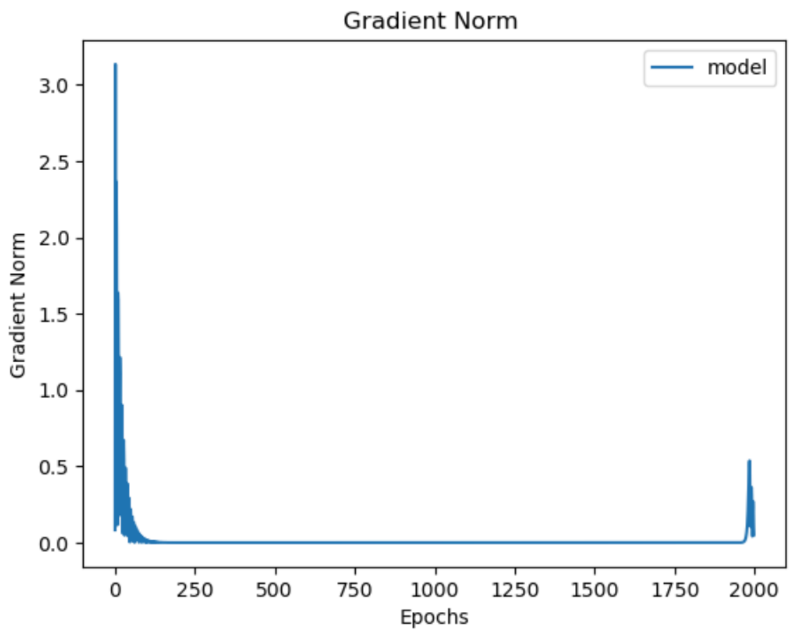


Observations-

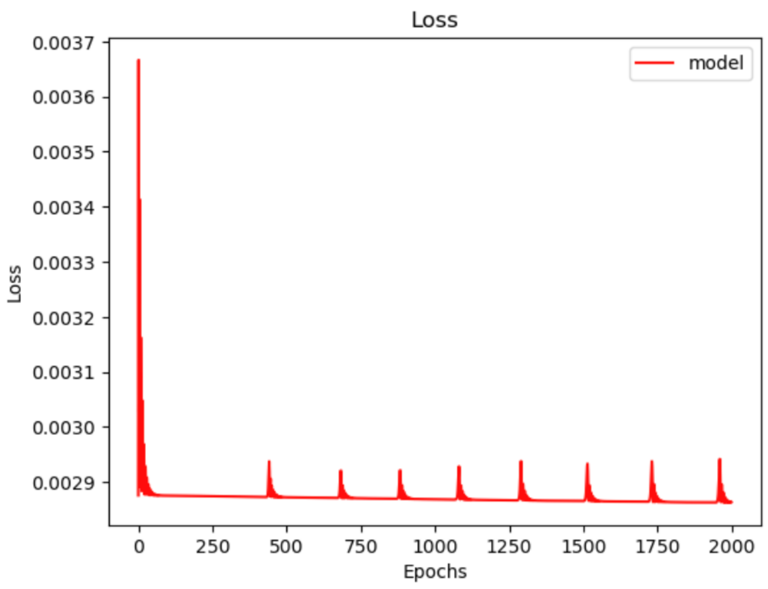
The model weights were collected after every 3rd epoch of training (45 epochs in total), resulting in 8 sets of weights with 417500 values each. To make the data easier to visualize, PCA was used to reduce the dimensionality of the weights to 2 dimensions. The 2-dimensional representation of the data was then plotted and visualized in the form of graphs, allowing researchers better to understand the results of the model's training process.

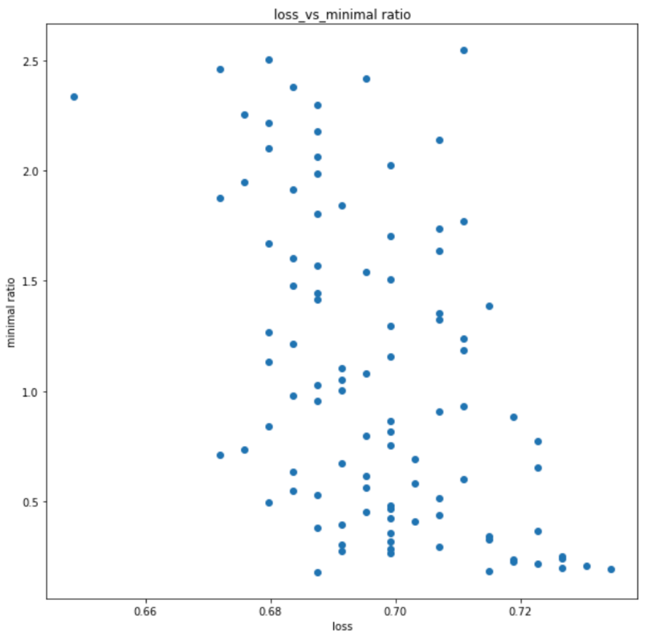
2.2 - Observe Gradient Norm During Training

A Deep Neural Network with three fully connected layers was trained to simulate the function (sin(5*(pix)))/((5(pi*x)). The network has 57 parameters. The Adam optimizer was utilized for optimization. Eight training series, each lasting 30 epochs, were conducted, during which the model weights were recorded. The second layer of the network is depicted in Figure 7, while the entire optimization process is illustrated in Figure 8. PCA was employed to achieve dimension reduction, and the learning rate for all models was set at 0.001.



The figure above shows the (gradient norm vs. epoch) and the figure below shows the(loss vs. epochs)





The figure above shows(the loss vs. minimal ratio)

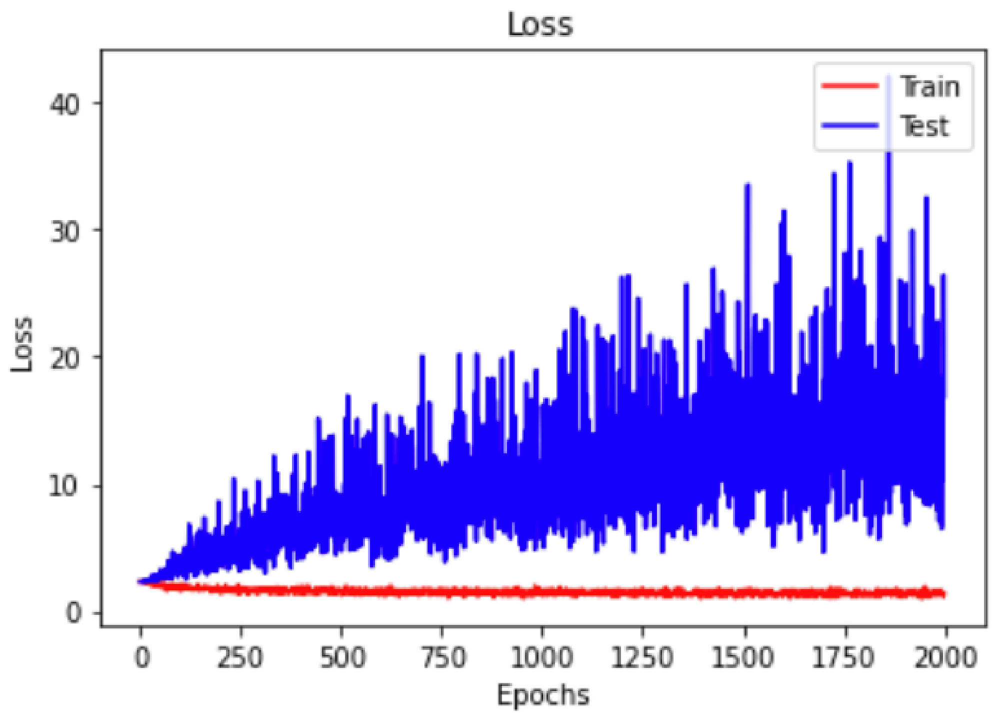
3.1 Can networks fit random labels

Firstly we create a PyTorch neural network model named "DNModel". The model has an input layer with size 784, which is the flattened size of the 28x28 images in the MNIST dataset. The model has three hidden layers with sizes 120, 120, and 16, respectively, and uses ReLU activation functions. The output layer has size 10, which corresponds to the number of classes in the MNIST dataset, and uses a linear activation function.

In the experiment, the model is trained and evaluated on the MNIST test dataset, which consists of 10,000 data points. The test data from the MNIST dataset is loaded into the test_loader DataLoader object with a batch size of 50. The model is then trained for 2000 epochs, with each epoch consisting of training on the batches of data loaded from the train_loader DataLoader object and evaluating the model on the test data batches loaded from the test_loader.

In each training epoch, the model's parameters are updated using the Adam optimizer with a learning rate of 0.001. The training loss for each epoch is calculated using the CriterionLoss function and stored in the train_losses list. Similarly, after training the model on the batches, the model is evaluated on the test data batches and the test loss for each epoch is calculated and stored in the test_losses list.

It is worth noting that the model inputs are flattened using before being passed through the model for prediction.



Observation-

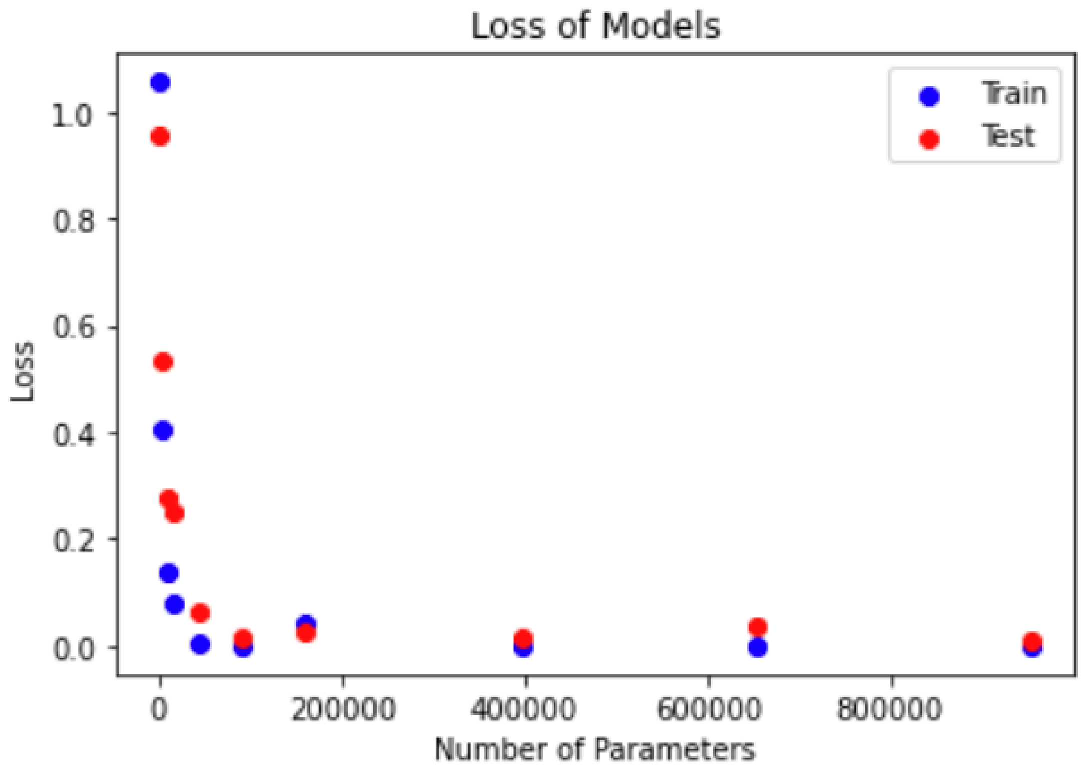
In the graph above it appears that the network is not fitting random labels. The fact that the training loss is low but the test loss is high suggests that the network is overfitting to the training data and not generalizing well to unseen data.

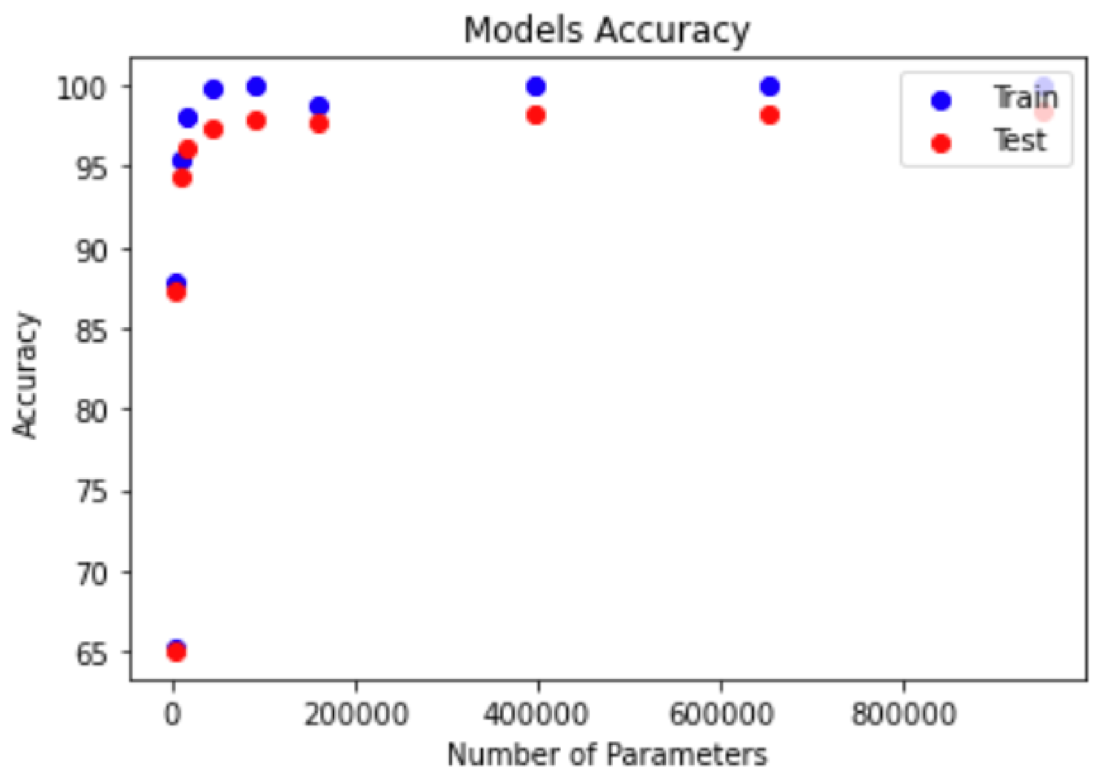
3.2 parameters vs generalization

Model1 has 784 input features, 128 hidden units in the first layer and 10 output features. The activation function used in this model is the ReLU activation function. This model has 15,114 parameters, which is a relatively large number of parameters for this type of task. With more parameters, the model has the ability to fit the training data very well, but there is also a risk of overfitting.

Model2, on the other hand, has 784 input features, 4 hidden units in the first layer, 6 hidden units in the second layer and 10 output features. The activation function used in this model is also the ReLU activation function. This model has only 3,240 parameters, which is a relatively small number of parameters compared to Model1. With fewer parameters, the model may not fit the training data as well, but there is a reduced risk of overfitting.

Both models were trained using 50 epochs, and their accuracy was recorded for both the training and testing data for each epoch. The training data was divided into batches of 500 samples, and the testing data was divided into batches of 100 samples. The criterion used for both models is the Cross-Entropy Loss, and the optimizer used for both models is the Stochastic Gradient Descent (SGD) optimizer.





Observations-

In conclusion, based on the observationfrom the graphs above, it can be seen that increasing the number of parameters in the model decreases its loss and increases its accuracy. However, after a certain number of parameters, the improvement from iteration to iteration becomes negligible, and adding additional parameters barely improves the model. Also, it can be observed that the models achieve higher accuracy and lower loss values when run on the training dataset compared to the testing dataset.

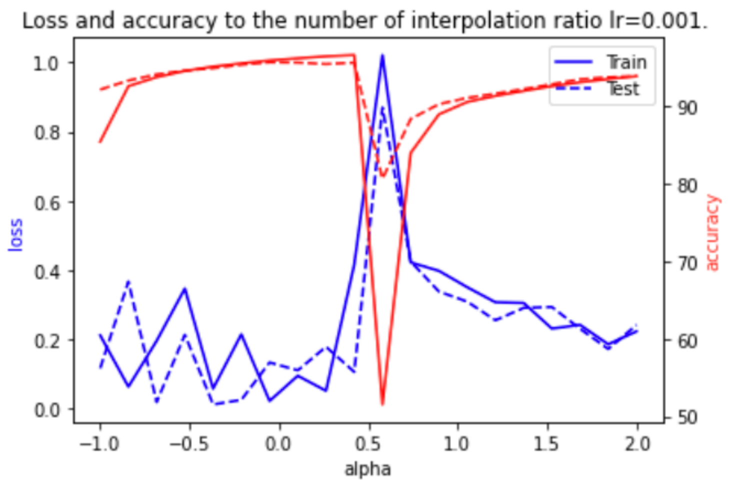
This suggests that the models may be overfitting to the training data and may not generalize well to new, unseen data.

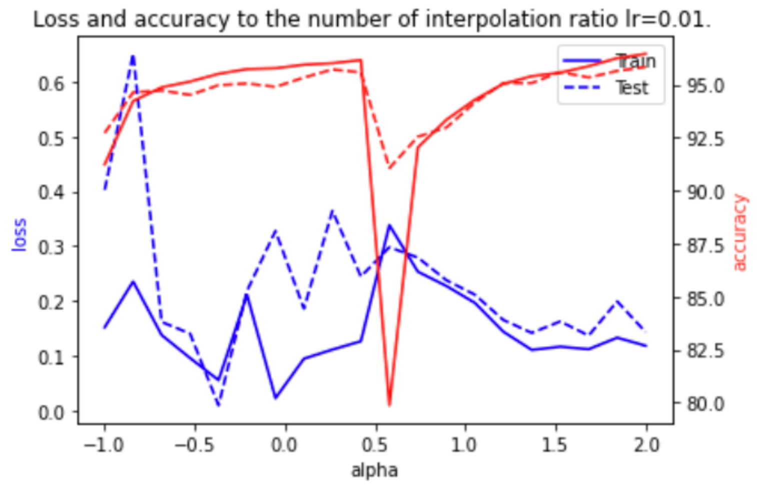
3.3 part 1 - Flatness v.s. Generalization

The experiments were conducted on the MNIST dataset for handwritten digit classification task. Five deep neural networks were implemented with two different batch sizes, 64 and 1024. The learning rate for all models was set to 0.001 and the Adam Optimizer was used for optimization. The models were trained using ReLU activation function and Cross-Entropy loss function was used as the criterion for computing the loss.

The sensitivity of each model was calculated by finding the Frobenius norm of gradients. The models were compared based on their train and test accuracy as well as their sensitivity values.

The 5 models are Deep Neural Networks with different architectures. The differences between the models include the number of hidden layers, the number of neurons in each hidden layer, and the activation function used in each layer. Model1 and Model2 have different batch sizes and Model3, Model4, and Model5 have the same architecture but with different numbers of neurons in the hidden layers.





Summary-

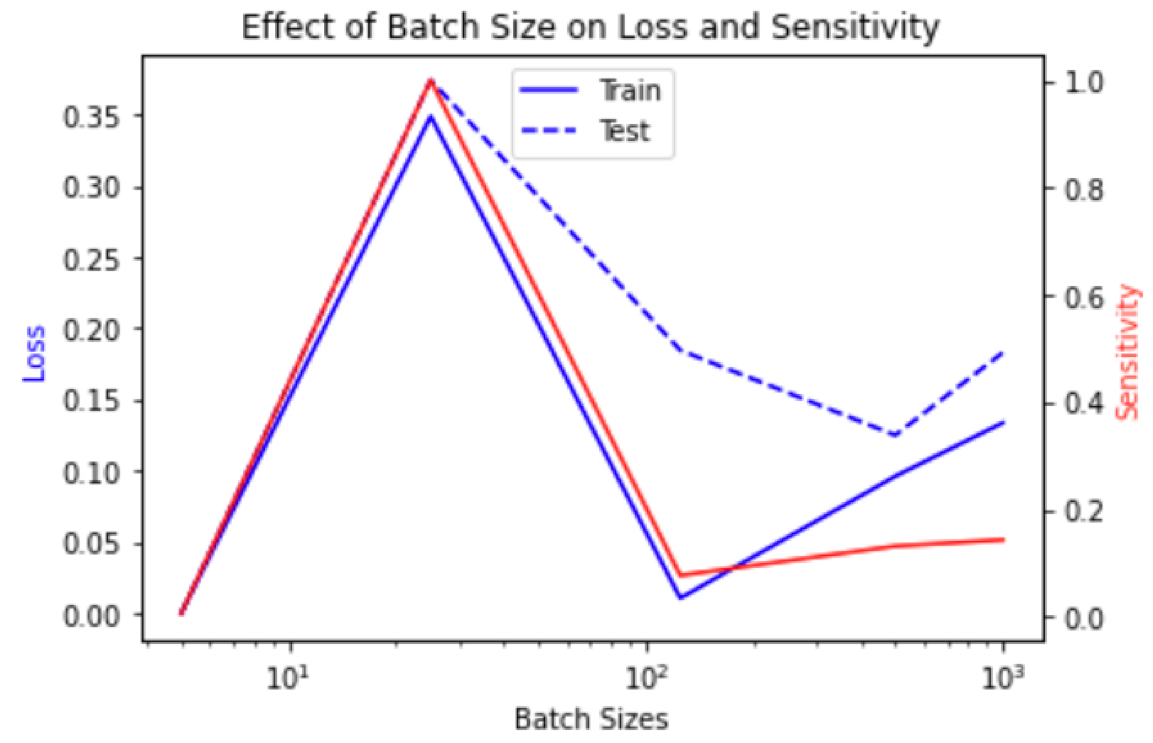
The five models had different architectures with different number of hidden layers and neurons. The results, shown in the graphs above , indicate that the loss, accuracy, and linear interpolation alpha were different for two models with learning rates of 0.001 and 0.01, respectively.

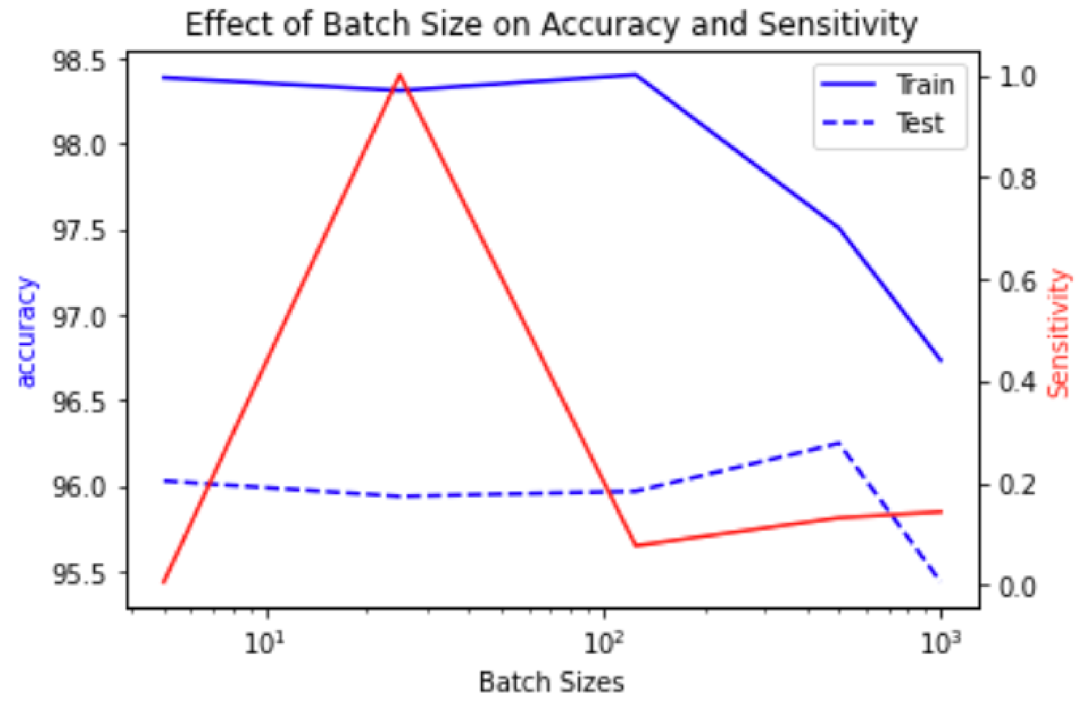
The results suggest that the learning rate is an important factor in designing and training effective deep neural networks, and that different learning rates can lead to significant differences in the performance of the models.

3.3 part-2 Flatness v.s. Generalization

Firstly I created Five identical Deep Neural Networks, consists of two hidden layers with 16630 parameters, which were were training different batches. The Adam Optimizer is used for optimization process and all the models have learning rate has 0.001.

All the models were trained on the MNIST dataset with a batch size of 1000 and 50 epochs. The training and testing accuracy and loss values were recorded for each model.


Effect of Batch Size on Loss and Sensitivity
Train
Test
Loss
Sensitivity
Batch Sizes


Effect of Batch Size on Accuracy and Sensitivity
Train
Test
accuracy
Sensitivity
Batch Sizes

Conclusion -

Based on the graphs above, it can be concluded that the accuracy and sensitivity of the deep neural networks are influenced by the batch size and loss. As the batch size increases, the sensitivity decreases. Hence, the best results for the network can be obtained when the batch size is in the range of 100 to 1000. This suggests that there is a trade-off between accuracy and sensitivity, and choosing the right batch size is crucial for optimizing the performance of the network.