CPSC-8430: Deep Learning - Homework 1

Task1: Deep Vs Shallow

Task 1-1: Simulate a Function

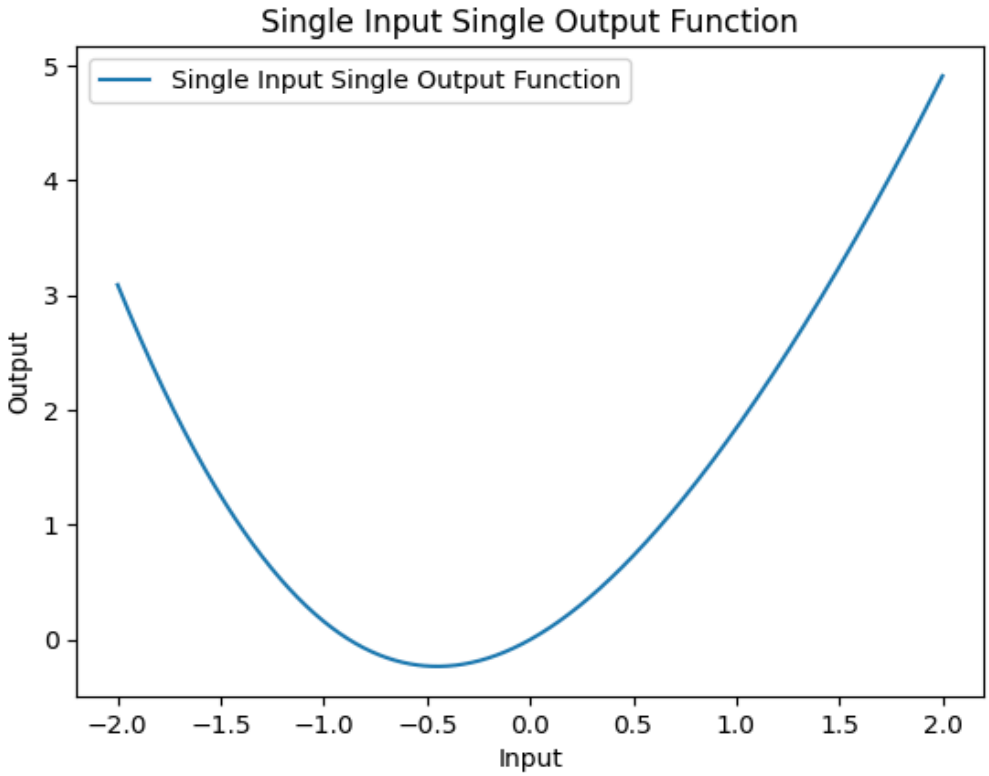
https://github.com/laxman2405/DeepLearning_HW1/blob/main/Laxman_Madipadige_HW1_Simulated_Function.ipynb

Common Properties for all Models in both functions:
We have used **ReLU** as the activation function because of its simplicity and capability to learn complex patterns and it is applied at each layer in the forward function.
We have used the pytorch **Adam** algorithm for computing the optimizer and **nn.MSELoss()** to compute the loss function for the models. Total number of epochs is **20000** with a learning ratio of **0.001.**

Function1: sin(x) + x2**
Below is the screenshot of the function defined, which takes in a single argument **'x'** and returns the computed result.

```
def simulationFunction(x):
    return np.sin(x) + x**2

x_train = np.linspace(-2,2,4000).reshape(4000, 1)
y_train = simulationFunction(x_train)

x_actual = np.linspace(-2,2,2000).reshape(2000,1)
y_actual = simulationFunction(x_actual)
```

We generate 2 sets of values for training and validation. X_train has 4000 values spaced between -2 to +2 and x_Actual has 2000 values spaced between -2 to +2. Y_train and y_actual will get the values from above function. Below is the graph for the function which shows a U-shaped curve, having minimum value at -0.5 and maximum value as the input value increases.



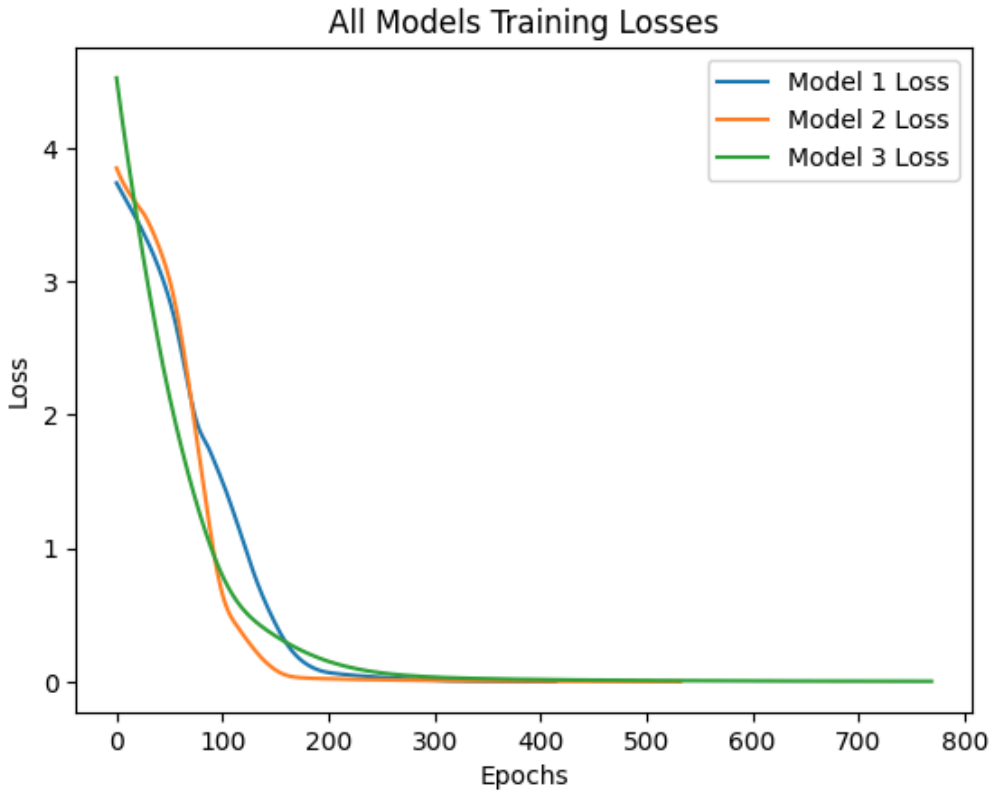
First set of Deep Neural Network Models for Function-1:
Below are the defined 3 DNN models which are used for approximating the proposed function mentioned above.

DNN_1 Model description:
One input layer with **10** neurons, **6** hidden layers with **20** neurons each and one output layer mapping to 1-D output. As for Convergence, we can see that Model 1 has reached its convergence at epoch **414** with a loss of **0.0012362** for total parameters of **2361.**

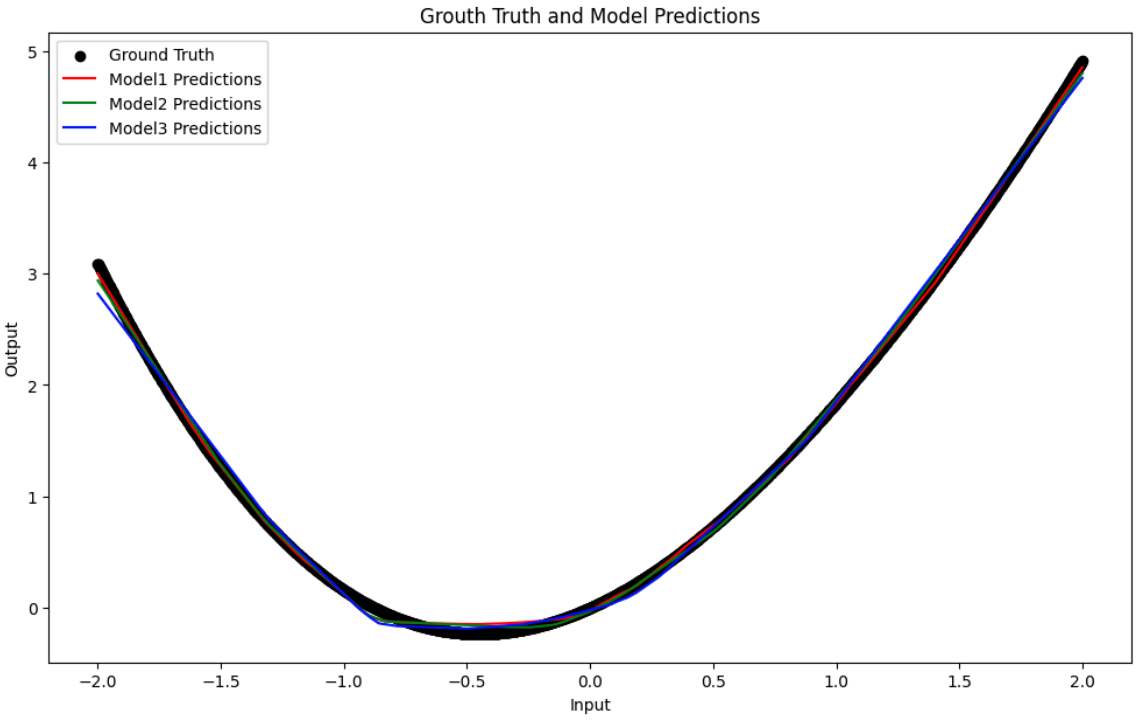
DNN_2 Model description:
One input layer with **10** neurons, **3** hidden layers with **varying neuron** sizes each and one output layer mapping to 1-D output. As for Convergence, we can see that Model 2 has reached its convergence at epoch **532** with a loss of **0.0017591** for total parameters of **1511.**

DNN_3 Model description:
One input layer with **20** neurons, **1** hidden layer with **30** neurons and one output layer mapping to 1-D output. As for Convergence, we can see that Model 3 has reached its convergence at epoch **769** with a loss of **0.004319** for total parameters of **701.**

After training the model, we plot the graph for training loss for all the three models. As we see in the graph below, as the number of epoch increases the loss decreases for all models, showing that training is done for the data and at convergence point, the model has very less rate to learn by training.



Below graph shows the plot graph with ground truth and the predictions that have been made by all the constructed three models.

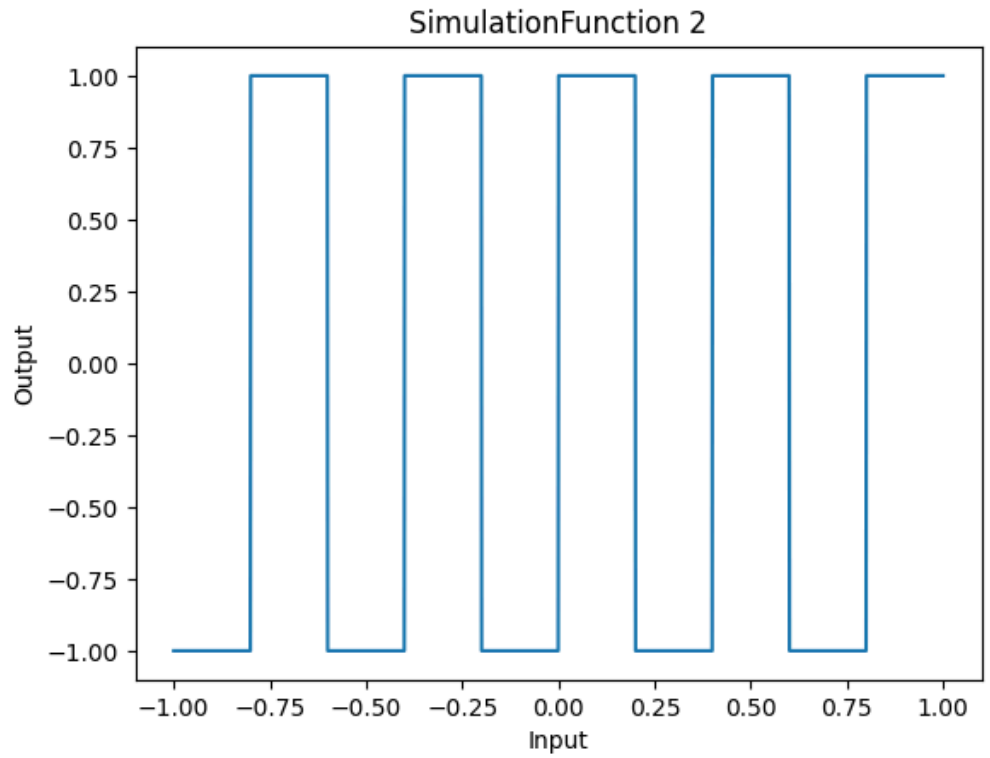


Overall Comments:
If we observe the graph, along with the losses at convergence point, we can see that the Model1 has a graph which is more accurate to the actual graph because of its more number of hidden layers. Since Model2 and Model3 have less number of hidden layers, it was unable to achieve the accuracy that model1 has achieved.

Function2: sign(sin(5*pi*x)
Below is the screenshot showing a new function that again takes in a single argument **'x'** and returns the computed result.

```
def simulationFunction(x):
    return np.sign(np.sin(5*np.pi*x))

x_train = np.linspace(-1,1,6000).reshape(6000, 1)
y_train = simulationFunction(x_train)

x_actual = np.linspace(-1,1,2000).reshape(2000,1)
y_actual = simulationFunction(x_actual)
```

We generate 2 sets of values for training and validation. X_train has 6000 values spaced between -1 to +1 and x_Actual has 2000 values spaced between -1 to +1. Y_train and y_actual will get the values from above function. Below is the graph for the function which is discrete and step-wise because of the **sign** function along with logarithmic **sin** function.



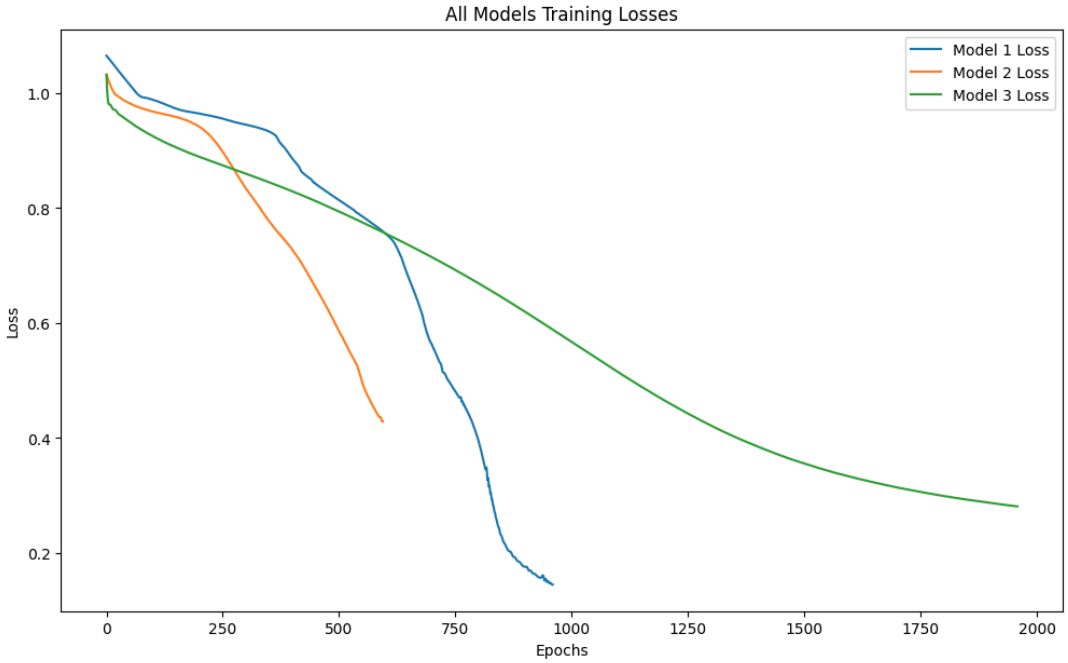
Second set of Deep Neural Network Models for Function-2:
Below are the defined 3 DNN models which are used for approximating the proposed function2 mentioned above.

NonLinear_DNN_1 Model description:
One input layer with **5** neurons, **6** hidden layers with **10** neurons each and one output layer mapping to 1-D output. As for Convergence, we can see that Model 1 has reached its convergence at epoch **959** with a loss of **0.144835** for total parameters of **571.**

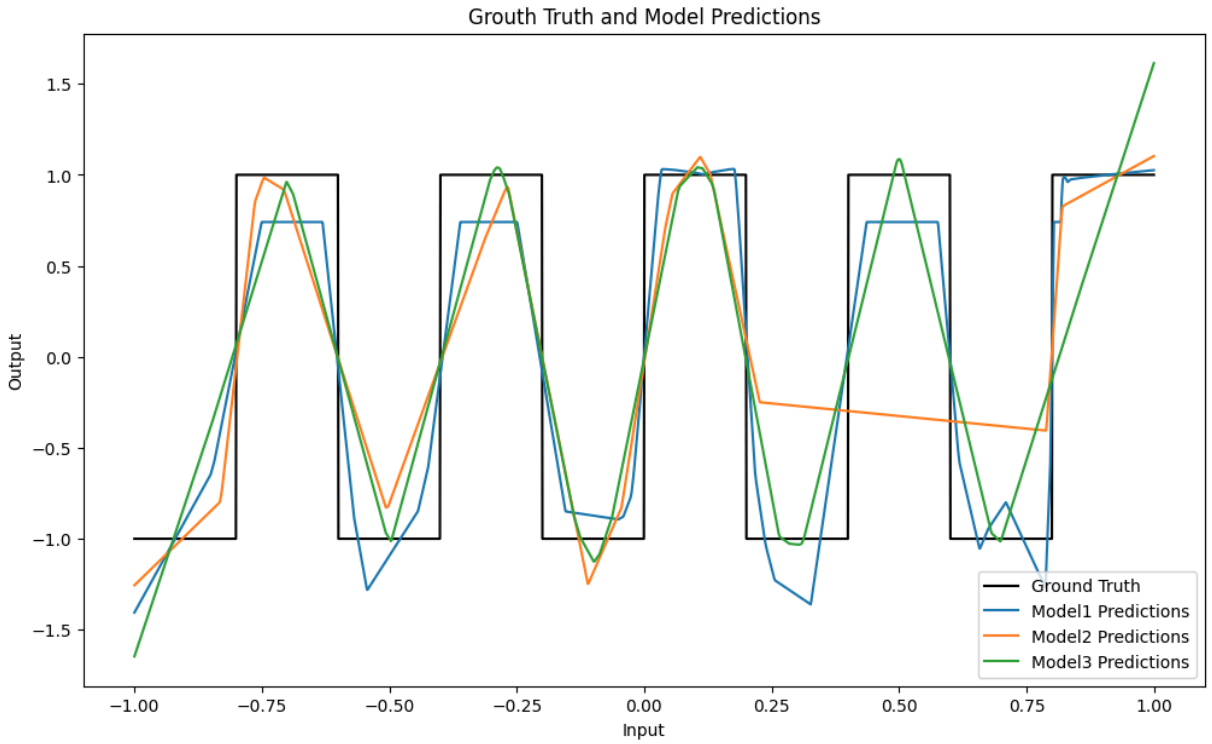
NonLinear_DNN_2 Model description:
One input layer with **10** neurons, **3** hidden layers with **varying neuron** sizes each and one output layer mapping to 1-D output. As for Convergence, we can see that Model 2 has reached its convergence at epoch **594** with a loss of **0.429038** for total parameters of **572.**

NonLinear_DNN_3 Model description:
One input layer with **190** neurons. We didn't add any hidden layers and one output layer mapping to 1-D output. As for Convergence, we can see that Model 3 has reached its convergence at epoch **1958** with a loss of **0.281065** for total parameters of **571.**

After training the model, we plot the graph for training loss for all the three models. As we see in the graph below, as the number of epoch increases the loss decreases for all models, showing that training is done for the data and at convergence point, the model has very less rate to learn by training.



Below graph shows the plot graph with ground truth and the predictions that have been made by all the constructed three models.



Overall Comments:
As we can see from the above plot, though all the models are trained with almost the same parameters, model1 plots the most accurate curve with less loss and high accuracy because of the many hidden layers along with the complex architecture it has. Next, followed by Model3 and Model2 have predicted the ground truth accurately.

Task 1-2: Train on Actual Tasks

https://github.com/laxman2405/DeepLearning_HW1/blob/main/Laxman_Madipadige_HW1_Train_Actual_Task.ipynb

We used MNIST data set to get the training and testing data and using a data loader, loaded 128 and 64 batch size train and test data. Constructed 3 CNN models for image classification which are described below.

Common Properties for Models:
For all the CNN models, we have used the loss function as **nn.CrossEntropyLoss()** and the activation function as **ReLU** along with log softmax for the final output layer. The optimizer algorithm used is **Adam** with a learning ratio of **0.001.** Since we are using the MNIST data set, the standard input image is **28x28** pixels.

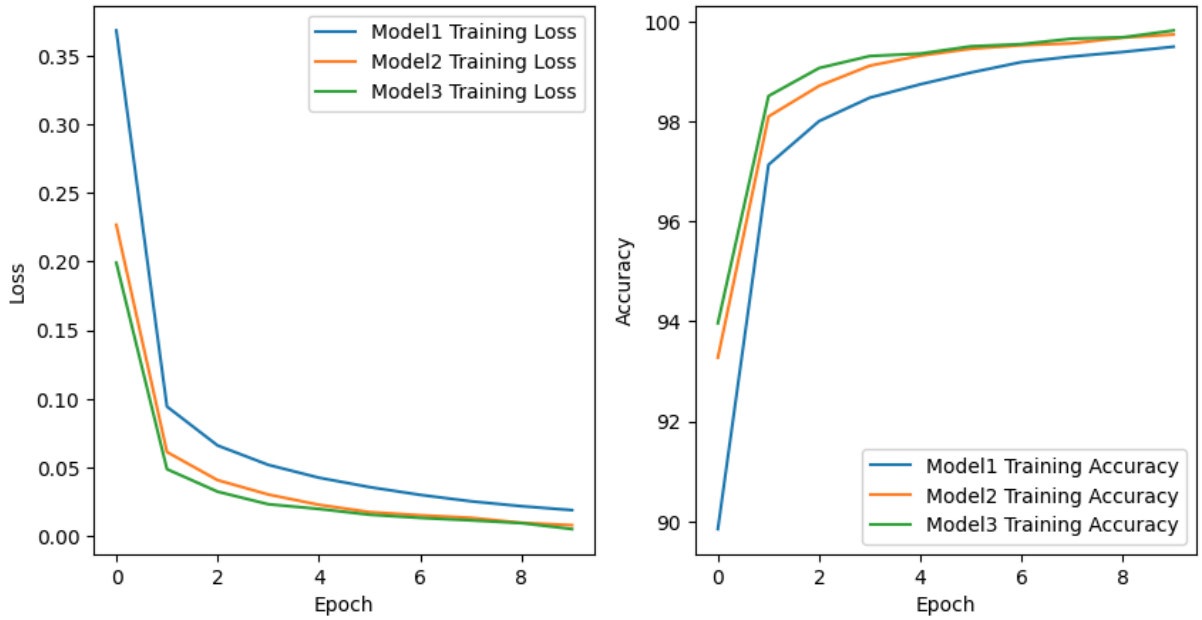
CNN_1 Model description:
Defined **two convolutional layers** and **two fully connected layers**. **Conv1** has **5** and **conv2** has **15** filters each with kernel size of 3. Based on the convolutional layers, the input image pixels will change. Defined a max pooling layer of size **2x2** which reduces the image pixels to half. Next, **fc1 and fc2** have **128 and 10** filters respectively.

CNN_2 Model description:
Defined **two convolutional layers** and **two fully connected layers**. **Conv1** has **16** and **conv2** has **32** filters with kernel size of **3** on each layer. Since, we defined max poling size of **2x2**, the image pixels will reduce to half. **Fc1** and **fc2** are defined with **128** and **10** filters respectively.

CNN_3 Model description:
Defined **two convolutional layers** with max pooling of size **2x2** followed by **three fully connected layers**. **Conv1** has **32** and **conv2** has **64** filters with kernel size of **3**, which are used to change the image pixels when applied. **Fc1**, **fc2** and **fc3** have **256, 128** and **10** filters respectively.

When the MNIST data set is used on each of the above models, the initial image pixels will change. For example, consider model3, after the first **conv1**, the size becomes **28x28x32**. Then since we are applying pooling, the size is reduced to half as **14x14x32**. Then, after **conv2** the size becomes **14x14x64**. After second max pooling, it reduces to **7x7x64 = 3136**. So, **3136** is given as input size to the **first fully connected layer** which reduces dimensions to reach a final output layer for **10** classes.

Below plot graph shows the Training loss and Training accuracy for all models described above:



Overall Comments:
All the models have reduced loss rate on increasing epoch values. Model1, though it started off with higher loss and lower accuracy, as the training is going on both loss and accuracy have improved. Model2 performs well by improving its efficiency from the beginning itself compared to model1. Model3 is the most efficient because it has the lowest loss and highest accuracy from the first compared to other models showing very less overfitting.

Task-2: Optimization

Task 2-1: Visualize Optimization Process

https://github.com/laxman2405/DeepLearning_HW1/blob/main/Laxman_Madipadige_HW2_Visualize_Optimization.ipynb

We load the MNIST data set by applying normalization and getting the train loader and test loader for batches of sizes 128 and 64.

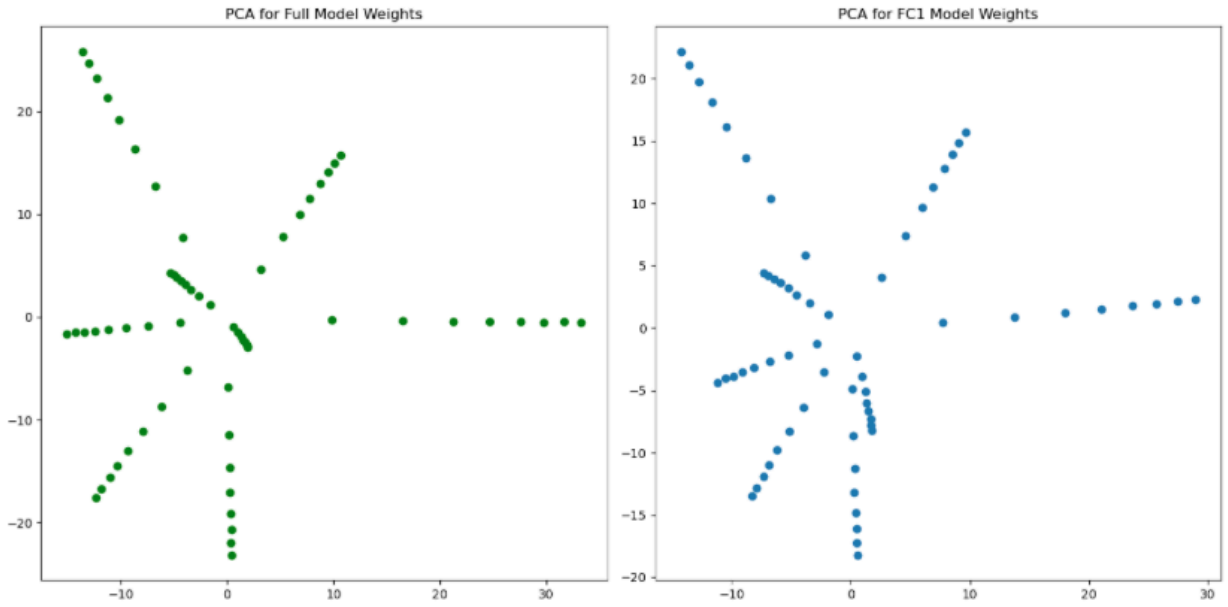
DNN_1 Model description:
Defined a DNN with one input layer, 2 hidden layers and an output layer. Since we use the **MNIST** standard data set, the initial input size is 28x28 **(784)**. Two hidden layers of 128 and 64 neurons along with a final output layer of **10** neurons (because digital classification in MNIST data set is 10).

Training and Calculating Weights:
The train model will collect the weights for every epoch and train the model **8** times. Each of the training loops will first invoke the DNN model, then define an optimizer with **Adam** algorithm and **learning ratio of 0.001**, loss function using **CrossEntropyLoss()** and repeat for 24 training times. This will result in collecting **8** weights for one of the fully connected layers using the condition.

When training the model, for every 3 iterations we collect the weights of the fc1 (first fully connected layer) and the weights of the full mode and store them in the arrays along with the computed losses.

PCA Components:
Then, using the weights for full model and fc1, we reduce the dimensionality to 2 components using PCA.

The below graph shows the PCA for full model weights and first fully connected model weights.



Overall Comments:
Left graph shows how the weights are clustered near the origin, showing low variance while many other weights are spread significantly while training the model. Right graph shows how the weights would change based on the PCA and model parameters like optimizer and learning ratio. Both the graphs show a similar pattern, but the points are closely clustered for the full model weights near origin and the diverged points show high variance which may impact the training performance.

Task 2-2: Observe gradient norm during training

https://github.com/laxman2405/DeepLearning_HW1/blob/main/Laxman_Madipadige_HW2_Gradient_Norm.ipynb

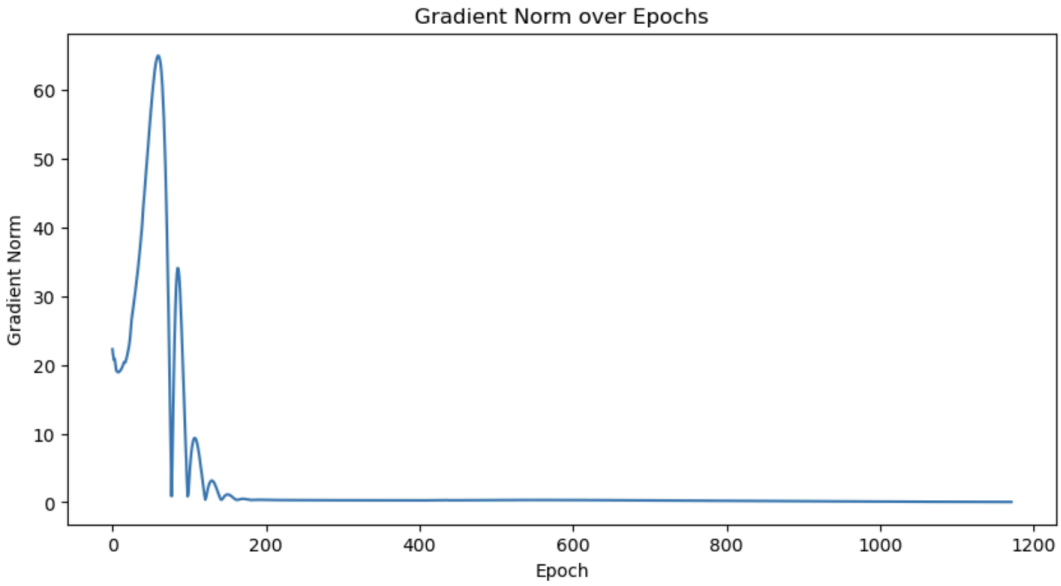
In this task, I have chosen the target function to be sin(x) + x.This function generates **100000** values for x_train ranging from 1 to 10 and **50000** values for x_actual ranging from 1 to 10 . Both the y_actual and y_train will get values by invoking the function.

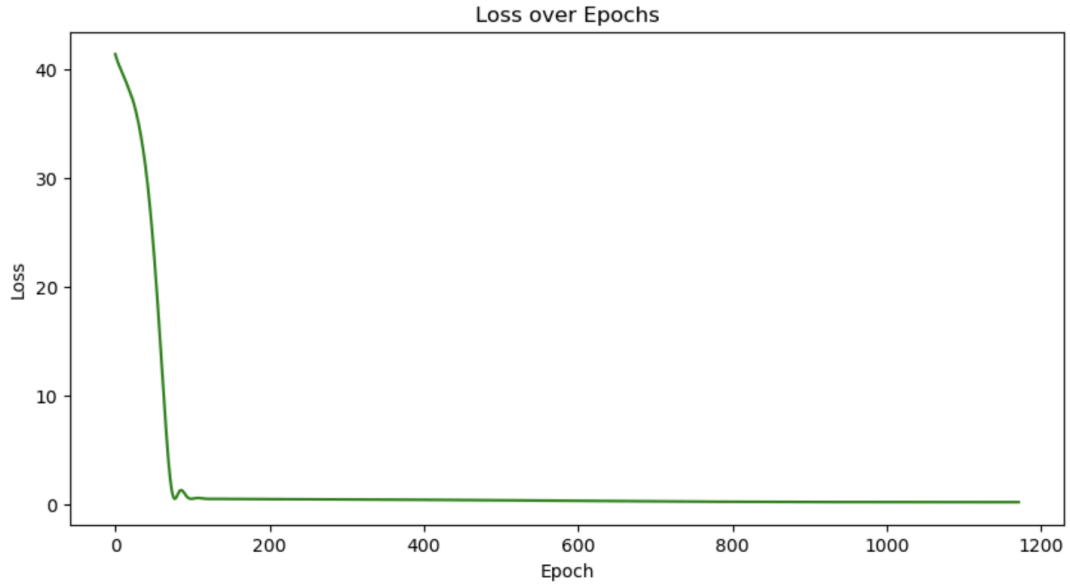
```
def single_input_single_output(x):
    return np.sin(x) + x

x_train = np.linspace(1,10,100000).reshape(100000, 1)
y_train = single_input_single_output(x_train)

x_actual = np.linspace(1,10,50000).reshape(50000, 1)
y_actual = single_input_single_output(x_actual)
```

Defined a model with an input layer, three hidden layers and one output layer with **ReLU** activation function. Also, having **Adam** for optimizer and using **nn.MSELoss()** to compute the loss function.

While training the model, at every iteration we calculate the gradient norm (using L2 form) and append it to the gradient_arr array. Gradient norms will help us understand how steep the changes are during the training phase.

Below are the graphs showing Gradient norm over epochs and loss over epochs.





Overall Comments:
If we observe both the plot graphs above, the gradient norm gradually increases at the beginning of the training, indicating that the model is exploring the parameters but later it decreases and stabilizes at a point. If we observe the loss, it starts off with a high value and gradually decreases indicating an effective training model and then stabilizes as well. At some point both gradient norm and loss reach a convergence point, and further training yields no improvement in results.

Task 2-3: What happened when Gradient is Almost Zero

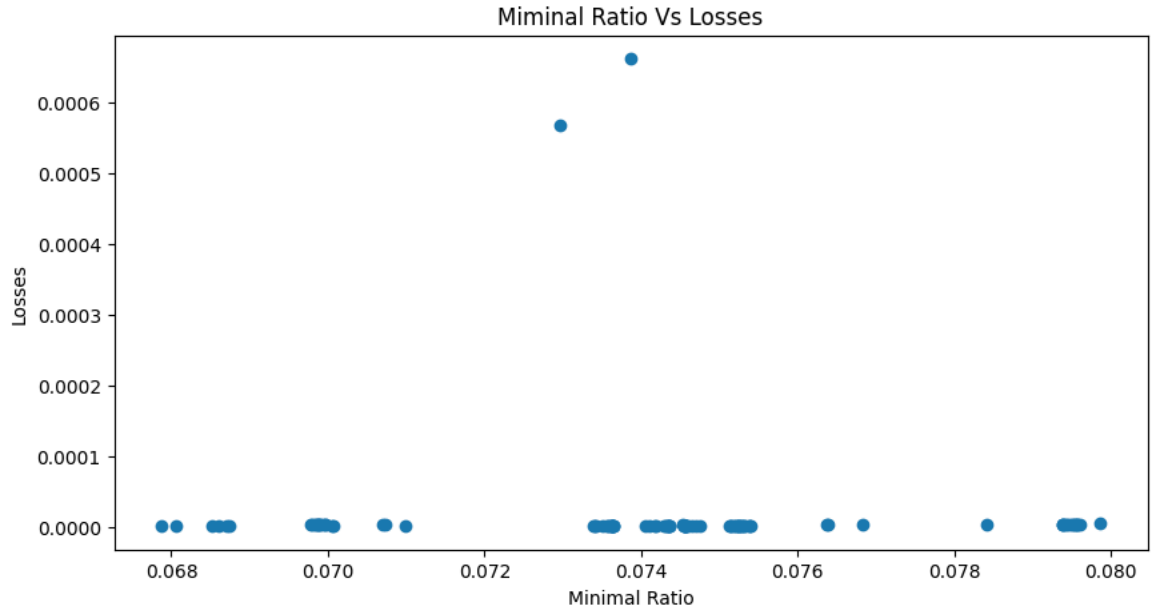
https://github.com/laxman2405/DeepLearning_HW1/blob/main/Laxman_Madipadige_HW2_At_Gradient_Zero.ipynb

We use the function as cos(x)/x and created a neural network model to train against data. We constructed a function compute_minimal_ratio_losses and compute_gradient. Compute_gradient will calculate the L2 norm of the gradient for the given model parameters.

Minimal ratio is a metric used to find the local minimum while the model is training. Here, the function compute_minimal_ratio_losses takes the model, loss function, x and y data as input and computes the minimal ratio based on second derivatives. We find the minimal ratio by finding the ratio between the number of positive curvatures to the total number of curvatures.

And, to compute a minimal ratio, we invoke the function only when the gradient norm is below the threshold(0.0001 - almost zero) in order to assess if the point is likely to be minimum.

Below graph shows the plot between minimal ratio and loss when gradient is almost zero.



Overall Comments:
As we observe in the above plot, as the minimal ratio is increasing its value, the losses changes are very minimal. Although, there are a couple of points where the losses are high, but since we trained the model for 100 times and only twice the loss was high. Also, at the end of minimal ratio at around 0.080, the loss recorded was close to zero, resembling that the model is well trained and performed.

Task-3: Generalization

Task 3-1: Can the network fit random labels?

https://github.com/laxman2405/DeepLearning_HW1/blob/main/Laxman_Madipadige_HW3_Random_Labels.ipynb

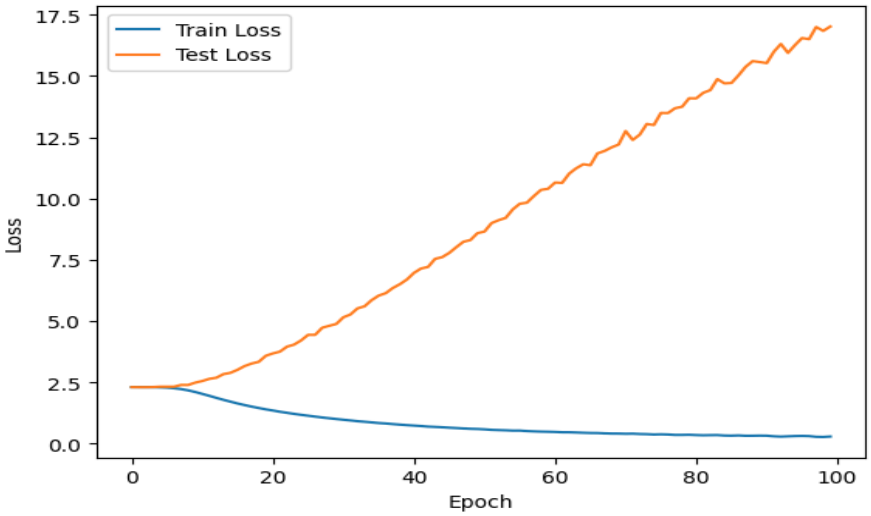
In this task, we aim for randomly generated labels for the MNIST dataset and train the neural network model.

We use transforms.Compose to normalize the data with a mean of **0.13** and standard deviation of **0.30**. Then after normalizing we download the dataset into current ./data folder into the training data. We replace the actual integer labels of the training set with the randomly generated integer labels**.** Train and test loader are generated for batch processing with batch sizes of **128** and **64** respectively.

Defined a CNN model with **2 convolutional layers and 2 fully connected layers**. First **Conv1** has **8** filters and **Conv2** has **16** filters with a kernel size of **3x3**. Since we use MNIST data set and after passing through all convolutional layers and max pooling which reduces the size to half, the feature maps are reduced to size of **784** and passed to the initial fully connected layer. The final fully connected layer maps it to the **10** output values.

Regarding the training and testing model, Optimizer **Adam** with a learning ratio of **0.001** is used along with **CrossEntropyLoss** loss function. To compute training_loss, we iterate over the train loader and compute total_train_loss using the loss function. Then, evaluate the model to compute the testing loss using the test loader and append it to the losses array. This process is repeated for **100 epochs** to calculate the losses.

Below graph has epochs on the x-axis with losses (both training and testing) plotted on the y-axis.



Overall Comments:
From the above graph, we can see that train loss has a decrease in it's values which signifies that the model is training the random labels correctly, but the test loss goes on increasing signifies the inability to generalize the random data. This is an example of overfitting, because the model was unable to learn patterns from the training set resulting in poor performance on the test set for randomly generated labels.

Task 3-2: Number of Parameters Vs Generalization

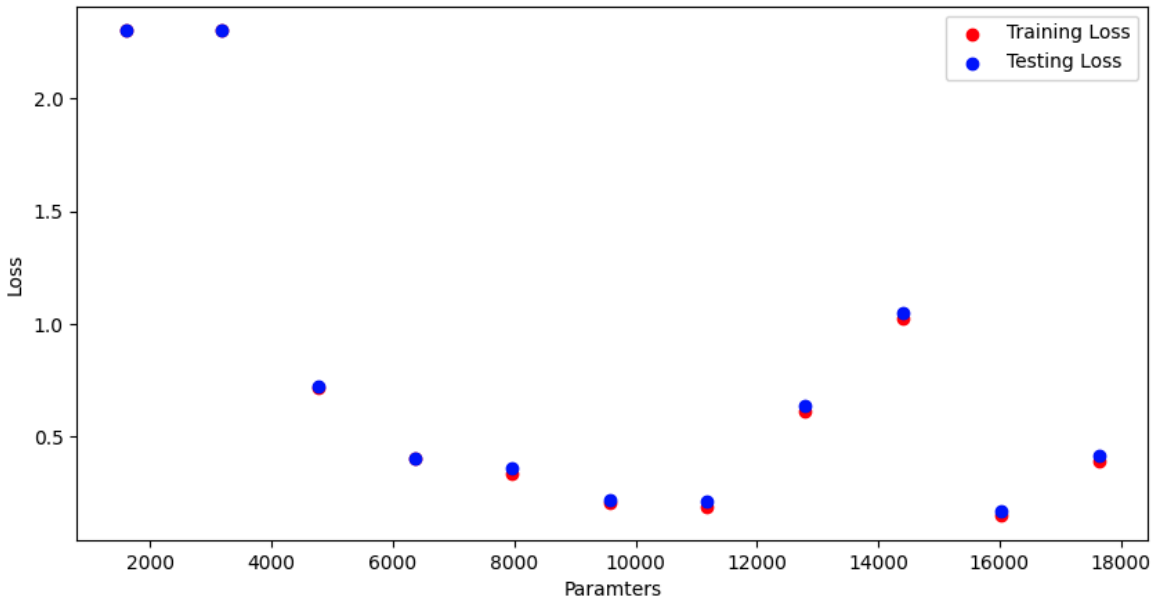
https://github.com/laxman2405/DeepLearning_HW1/blob/main/Laxman_Madipadige_HW3_Parameters_Vs_Generalization.ipynb

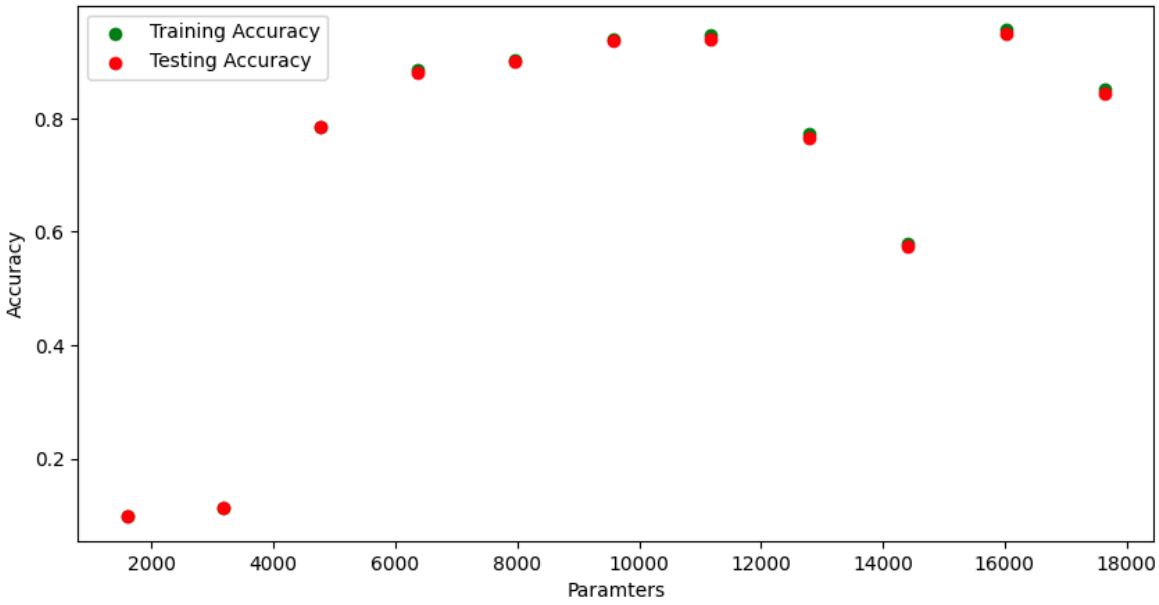
We load the MNIST dataset by applying normalization and loading the train and test loader with batch size of **128** each. I have defined a total of 11 models to train with one input layer with size **784**, one hidden and one output layer mapped to **10** neurons. Below are the characteristics of each model.

Model	1	2	3	4	5	6	7	8	9	10	11
Neurons	2,1	4,2	6,3	8,4	10,5	12,6	14,7	16,8	18,9	20,10	22,11

Total number of parameters range from 1593 to 17643 for a total of 11 models. Defined a function to compute metrics for losses and accuracy which constructs loss function as **nn.CrossEntropyLoss()** and optimizer as **Adam** with learning ratio of **0.0001.** The model is trained for **10 epochs.**

Below is the graph which shows the Training, Testing loss and Training, Testing accuracy against total number of parameters for each model.





Overall Comments:

If we observe the loss and accuracy until the parameter range of 10000, the loss is gradually decreasing and accuracy is increasing after each model, which means the model is trained well around the range of 6000 to 10000 parameters. Beyond this range, when the complexity increases, there is no much increase in the accuracy or decrease in loss because the model

started to experience overfitting or it has reached a saturation level and both loss and accuracy are experiencing fluctuations in values.

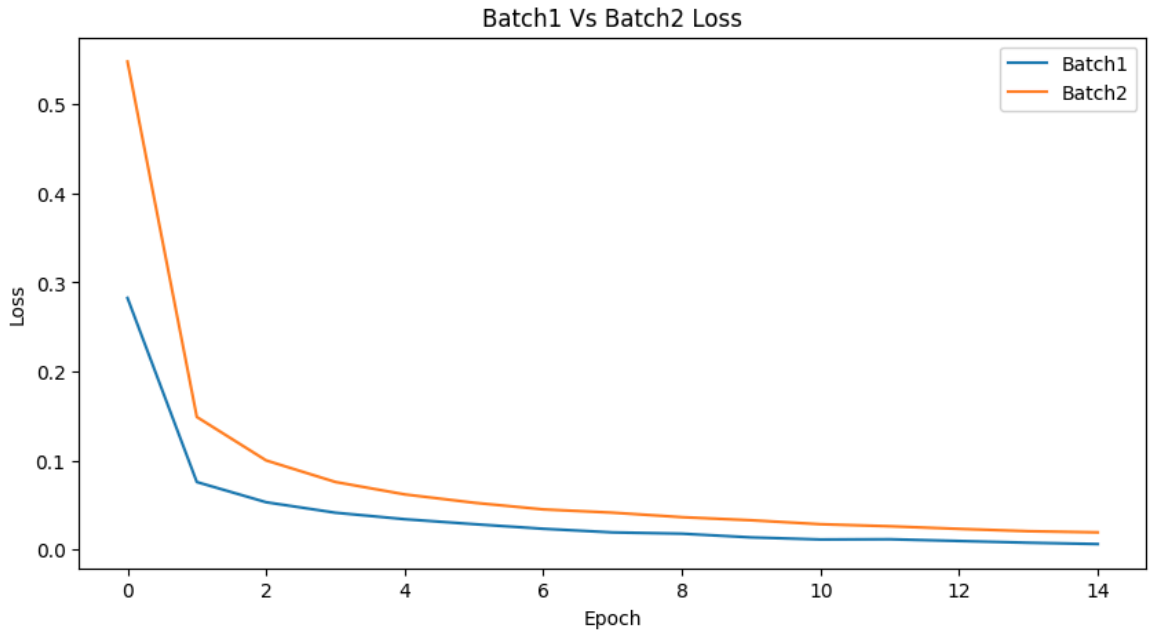
Task 3-3-1: Flatness Vs Generalization - Part: 1

https://github.com/laxman2405/DeepLearning_HW1/blob/main/Laxman_Madipadige_HW3_Flatness_Vs_Generalization_1.ipynb

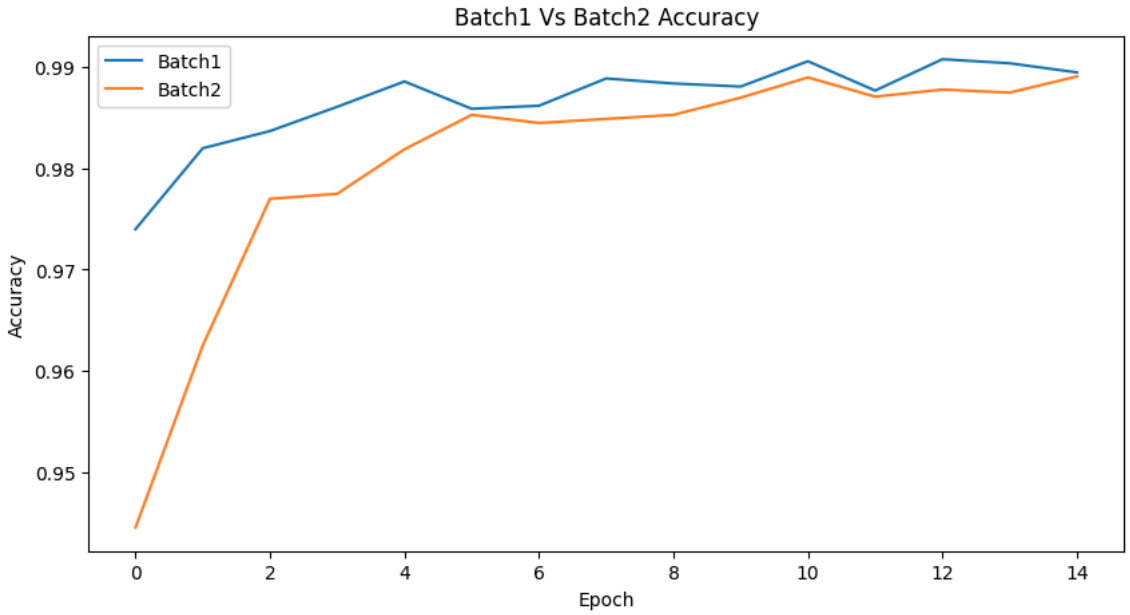
Loaded the MNIST data set by normalizing the set with mean of 0.1307 and standard deviation of 0.3081 and since we have to train the model for 2 different batches, we defined the train and test loader for 2 batch sizes of 128 and 512.

Defined the CNN model with **2** convolutional network layers and **2** fully connected layers. First conv1 has **8** filters and second has **16** filters, having kernel size of **3**, with max pooling size of **2x2** applied on both the convolutional layers will reduce the size to **784**. The final output layer has **10** filters. **ReLU** activation function is used and returns the final value using log_softmax. Constructed a training model to compute the losses and accuracies. The training model is run for **15** epochs on both the batches of the training set. Optimizer **Adam** with learning ratio of **1e-2 and crossEntropyLoss()** loss function is passed to the function as parameters.

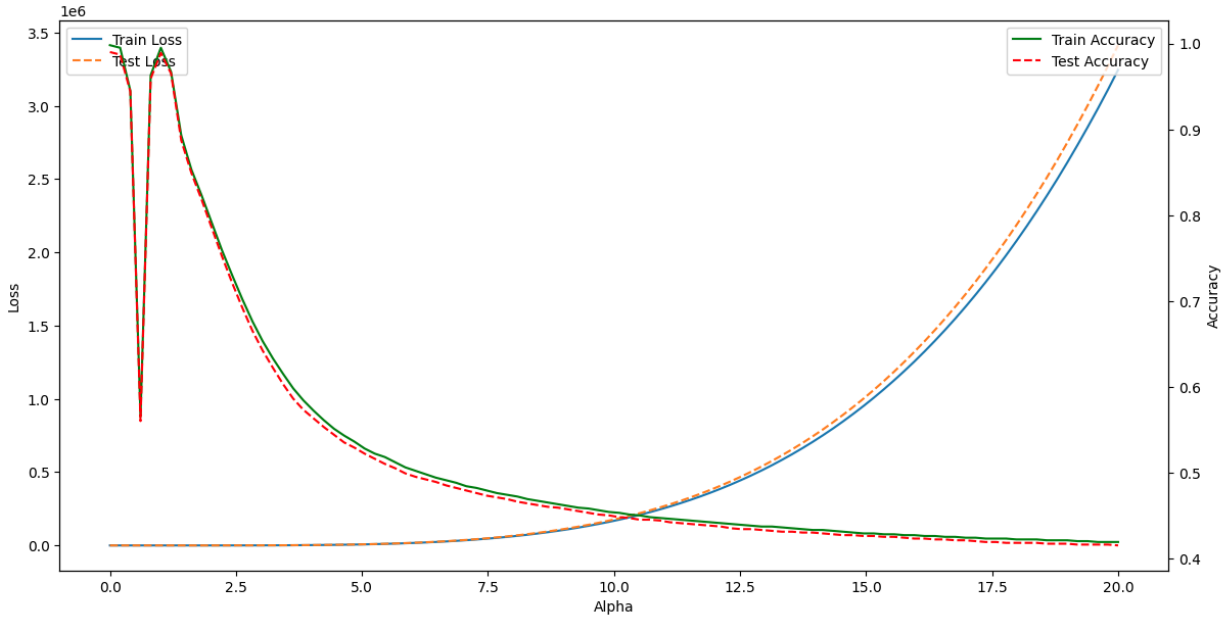
Next, we use both batches params to compute the **interpolation** between those 2 sets. For this, we generate **100** alpha values spaced between **0** and **20.** We generate each interpolated set and store it in variables and using that model we find the training and testing loss and accuracy for each set in the full interpolated set. Below graph shows the epoch values on the x-axis and loss (both batch 1 and batch 2) on the y-axis:



Below graph shows the epoch values on the x-axis and accuracy (both batch 1 and batch 2) on the y-axis:



Below graph shows the alpha values on the x-axis and Train, Test loss and Train, Test accuracy on the y-axis:



Overall Comments:
As we can see, the model is well trained for both the batches. Because, though the loss and accuracy on both the batches started off rough, the loss gradually reduced resulting in increase in accuracy for both the batches. As the alpha values approaches closely we can observe that loss is reduced and accuracy is increased, showing the model is well trained and both train and test data move very closely indicating the model generalizes well and there is negligible chance of overfitting.

Task 3-3-2: Flatness vs Generalization Part-2:

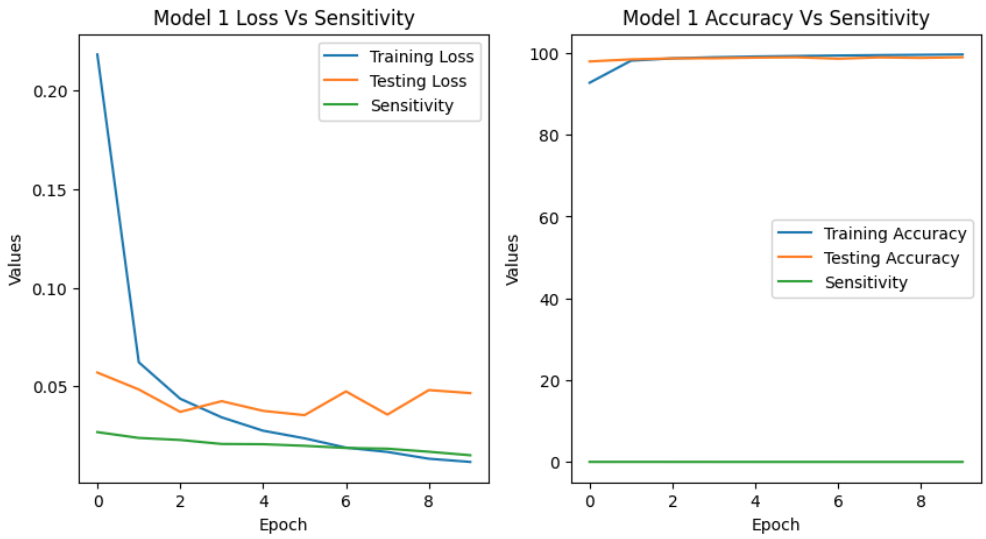
https://github.com/laxman2405/DeepLearning_HW1/blob/main/Laxman_Madipadige_HW3_Flatness_Vs_Generalization_2.ipynb

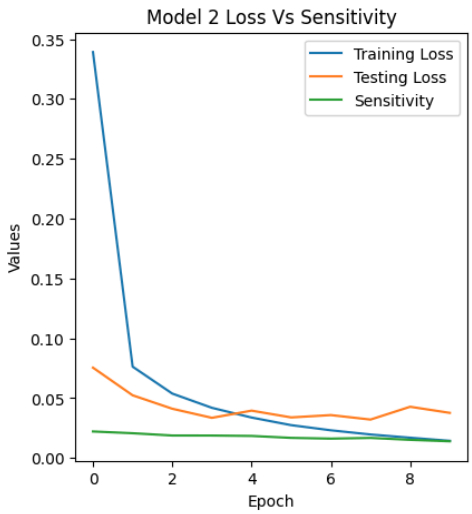
Constructed the CNN model with 2 convolutional layers, 2 fully connected layers. First **conv1** has **8 filters** and second one has **16** filters, with kernel size as **3**. Input to the first fully connected layer is reduced to **784** and the final fully connected layer outputs **10** filters. Both the convolutional layers go through a **2** max poolings size of **2x2** which reduces the size by half.

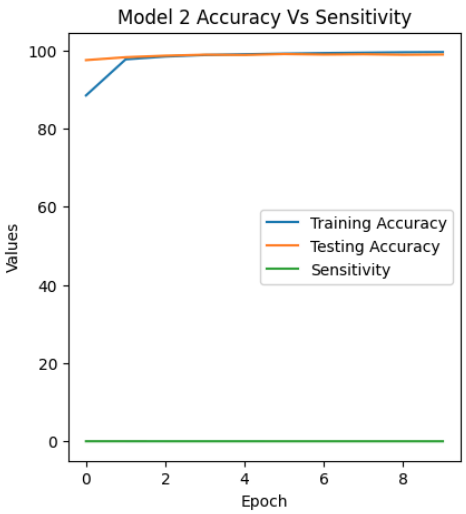
Training and testing models are defined to compute the losses and accuracies and run for **10** epochs. We load the MNIST data set by performing normalization and generating **5** train and test loaders of different batch sizes. For few batch sizes we use optimizer **Adam** with **learning ratio of 0.001** and for few we used optimizer **SGD** with **learning ratio of 0.01.**

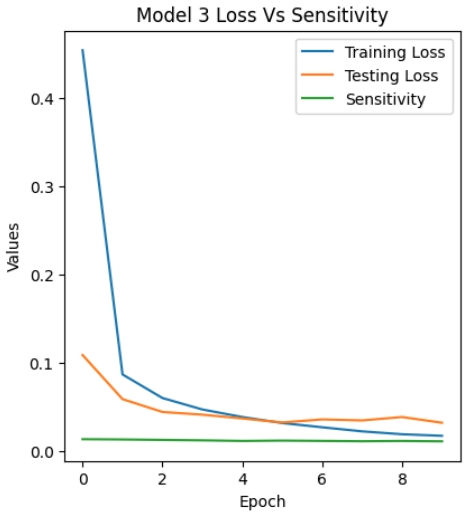
For each batch, we compute metrics of loss, accuracy as well as sensitivity and plot them in the graph.

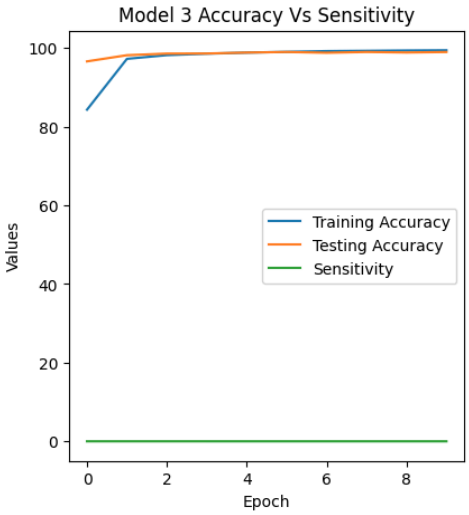
Below figures show the value ranges of training, testing loss and training, testing accuracy and sensitivity for a wide range of epoch values.

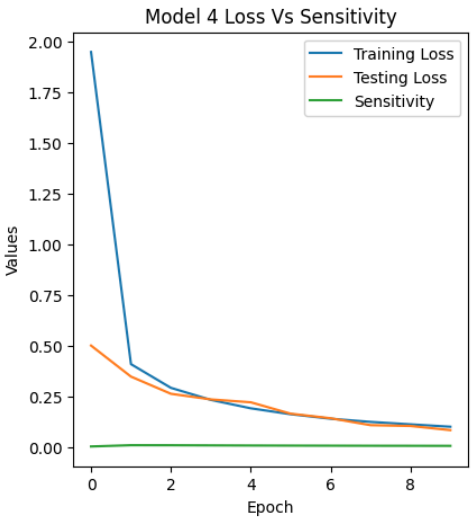


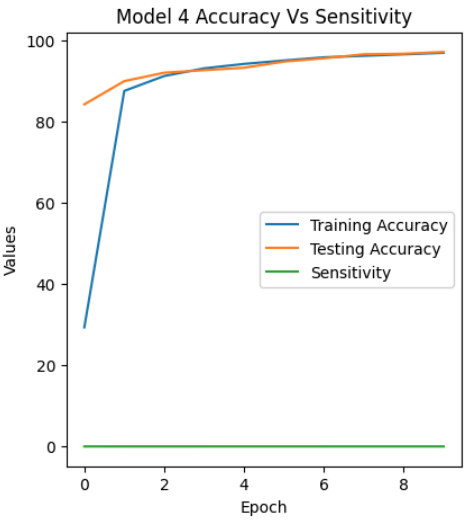

Model 2 Loss Vs Sensitivity
Training Loss
Testing Loss
Sensitivity
Values
Epoch

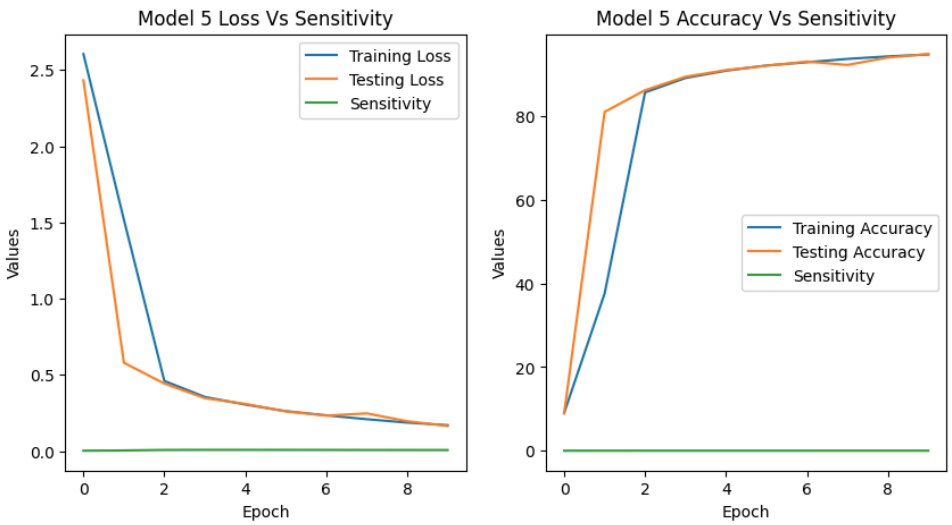

Model 2 Accuracy Vs Sensitivity
Training Accuracy
Testing Accuracy
Sensitivity
Values
Epoch


Model 3 Loss Vs Sensitivity
Training Loss
Testing Loss
Sensitivity
Values
Epoch


Model 3 Accuracy Vs Sensitivity
Training Accuracy
Testing Accuracy
Sensitivity
Values
Epoch


Model 4 Loss Vs Sensitivity
Training Loss
Testing Loss
Sensitivity
Values
Epoch


Model 4 Accuracy Vs Sensitivity
Training Accuracy
Testing Accuracy
Sensitivity
Values
Epoch



Overall Comments:

Since the first 3 models use different parameters compared to the last 2 models in terms of optimizer and learning ratio, we can clearly observe the difference in loss and accuracy. For the first 3 models, the loss started off high, but decreased gradually resembling the efficient training model. Even the accuracy, though didn't start off high, it maintained high accuracy in further epochs. If we take the last 2 models, loss remains to be reduced at the end, but accuracy started off very low in the beginning, but later on boosted to a high value. Sensitivity remains almost constant among all the 5 models.