Deep Learning Homework – 1

Name: Usha Sri Gunduboina

Course: CPSC 8430 Deep Learning

GitHub: https://github.com/ushasrigunduboina/Homework1-DeepLearning.git

Part 1: Deep vs Shallow

Simulate the function:

In this simulation, the goal was to model two distinct non-linear functions using Deep Neural Networks (DNNs) over 20,000 epochs. The objective was to train two different DNN architectures with the same number of parameters and compare their performance based on the loss over epochs and the quality of their predictions. The learning rate for both models is 0.001

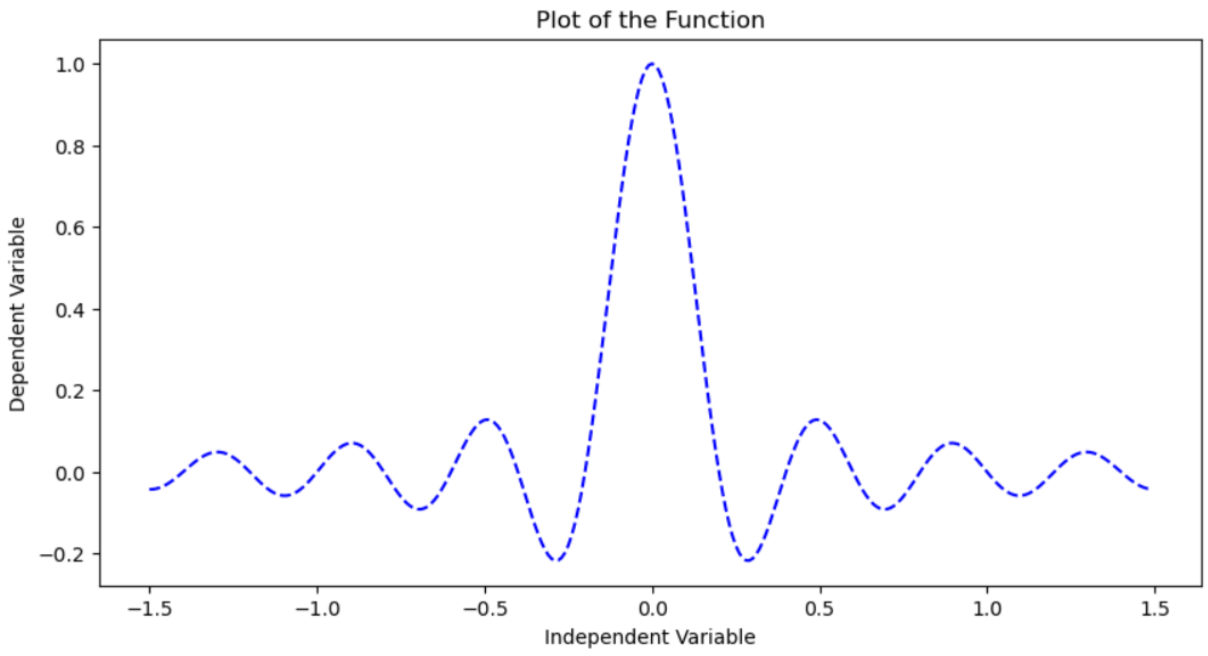
The two functions used in this study are:

1) y= sin(5πx)/5πx

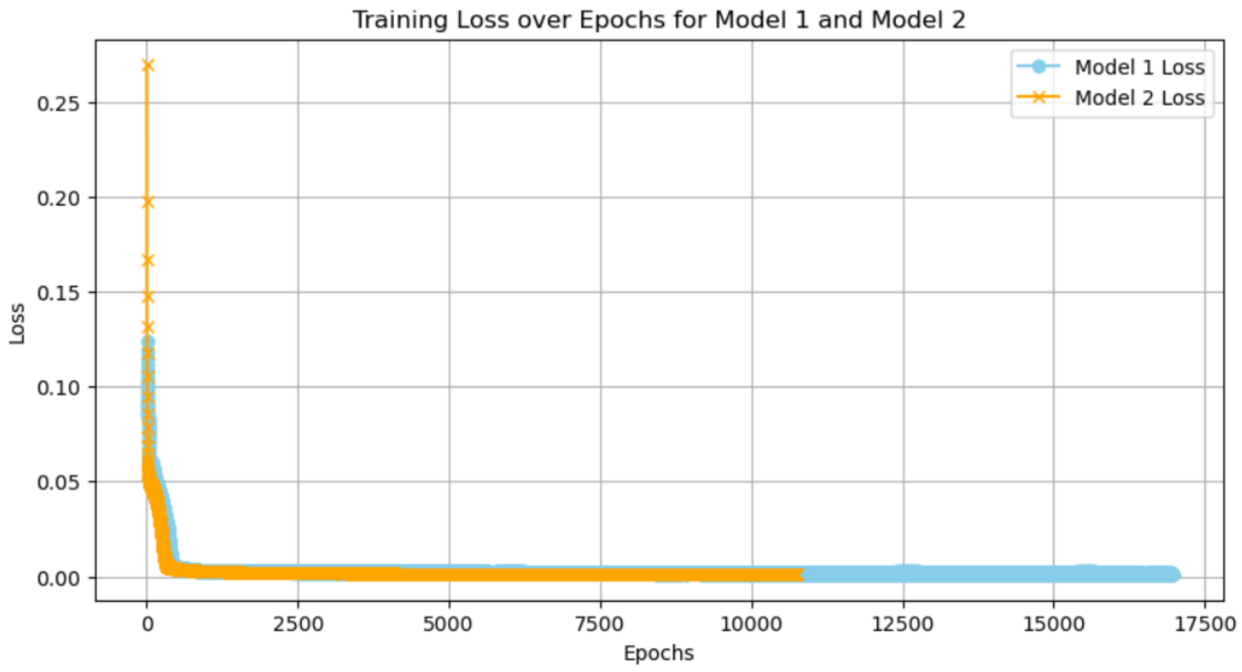
2) y= sgn(sin(5πx))

1st FUNCTION:

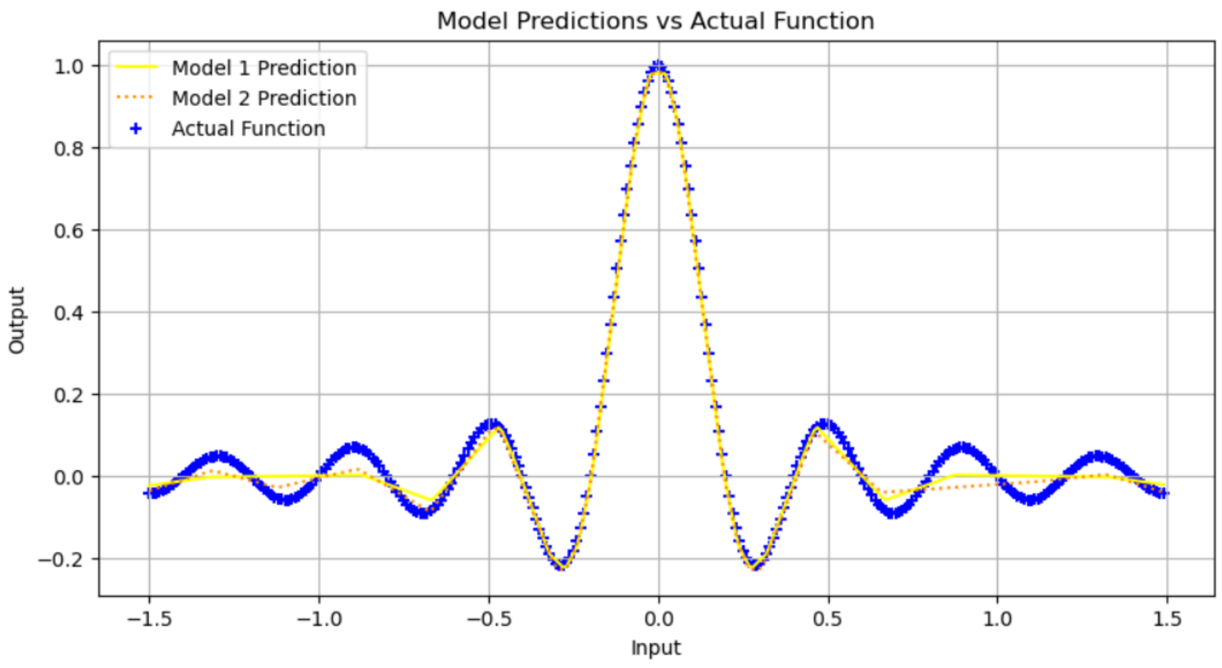
The function I used here is sin(5πx)/5πx. There are multiple graphs related to the training and prediction process of the models for Function 1. Image 1, this graph visualizes the input vs. the target data, showing the dampened sine wave generated by the function.



```
Convergence reached for loss: 0.0009993407875299454
```



From the above graph, a plot of training loss over epochs revealed that Model 2 steadily outperformed Model 1 as training progressed. Function 1 was a continuous, smooth function, and both models were able to approximate it effectively. The deeper model, with more layers, converged faster and achieved a lower loss. The predicted values closely matched the true values, especially in regions of high oscillation.



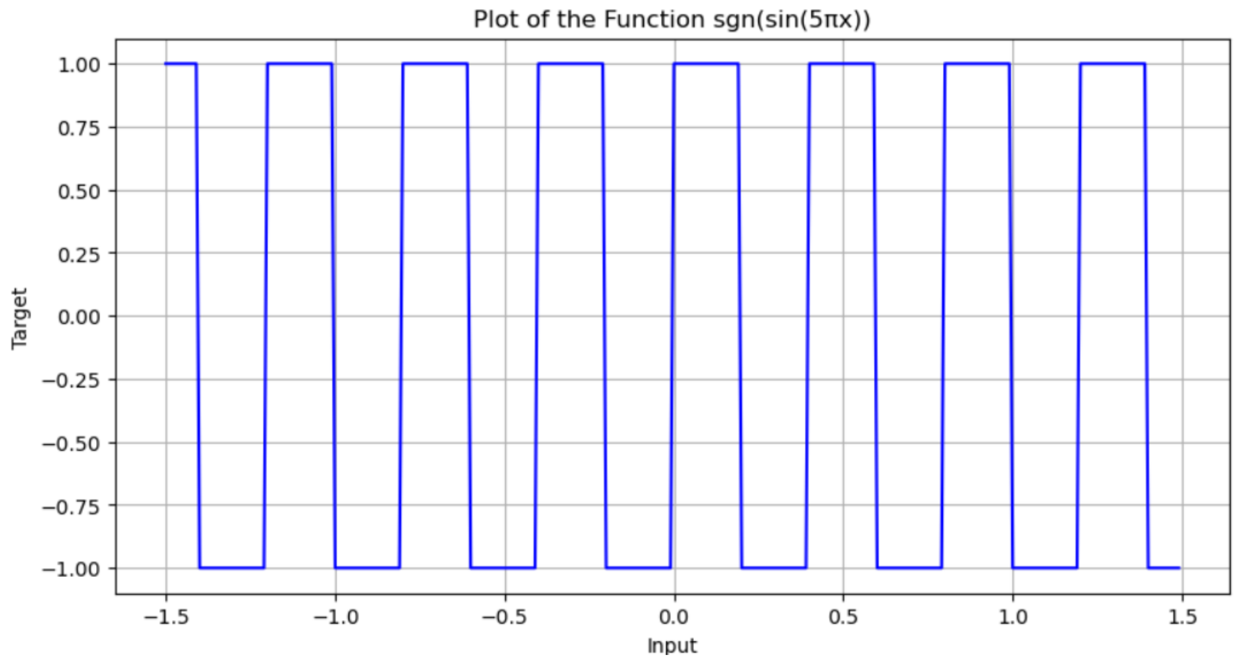
Result:

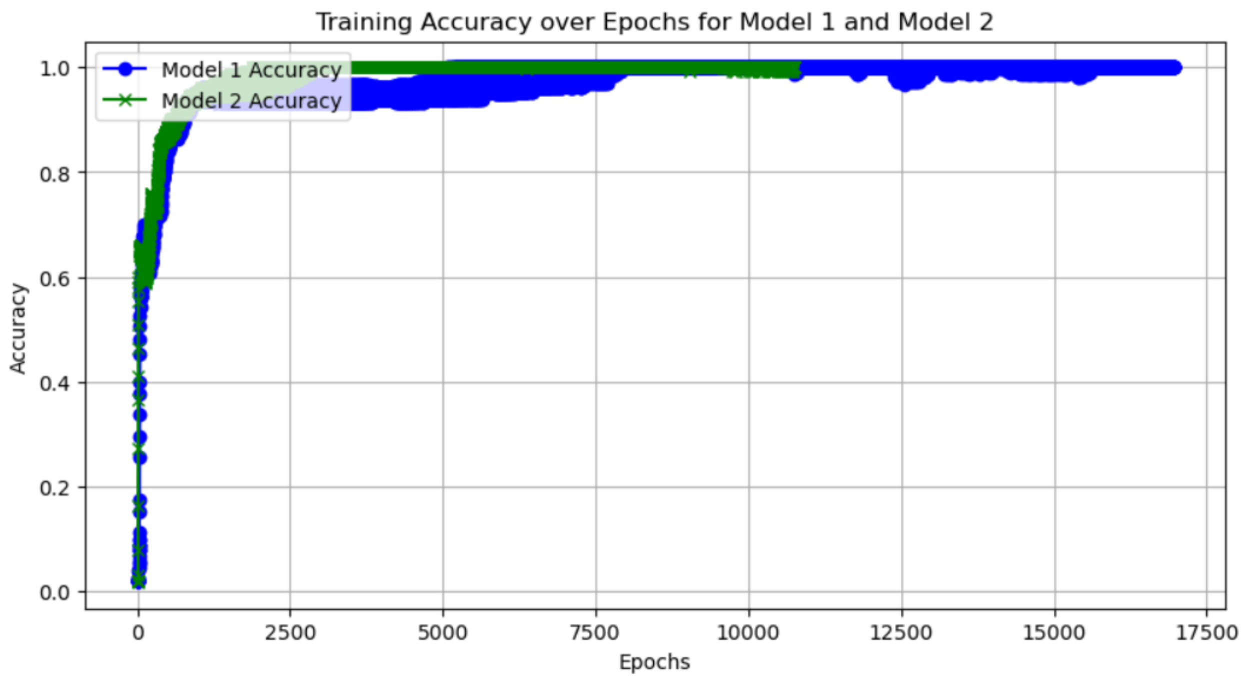
Both models, but the deeper Model 2 reached a slightly lower loss than the simpler Model 1. The loss decreased steadily throughout training.

Both models produced smooth approximations of the ground truth function. Model 2 captured more of the nuances in the curve, especially around critical points where the sine function transitions.

2nd FUNCTION:

Now I am using the 2nd function which is sgn(sin(5πx)). Below image gives the function plot of x and y. Both models were trained on these functions for 20,000 epochs each, and the training loss and accuracy were monitored. training the models on a signum function presents additional challenges due to the discontinuous nature of the function.





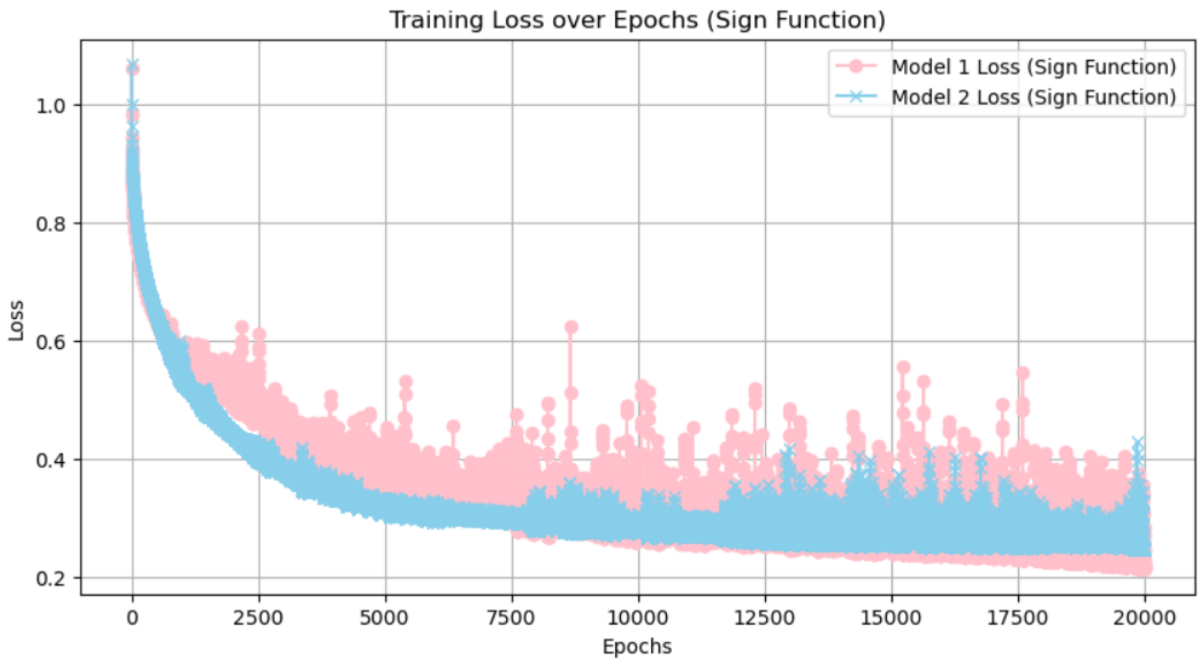
Results:

The models faced more difficulty with this function, as discontinuities in the target function make it harder for neural networks to achieve low error rates. However, Model 2 still outperformed Model 1, achieving a lower loss and more stable training behavior.

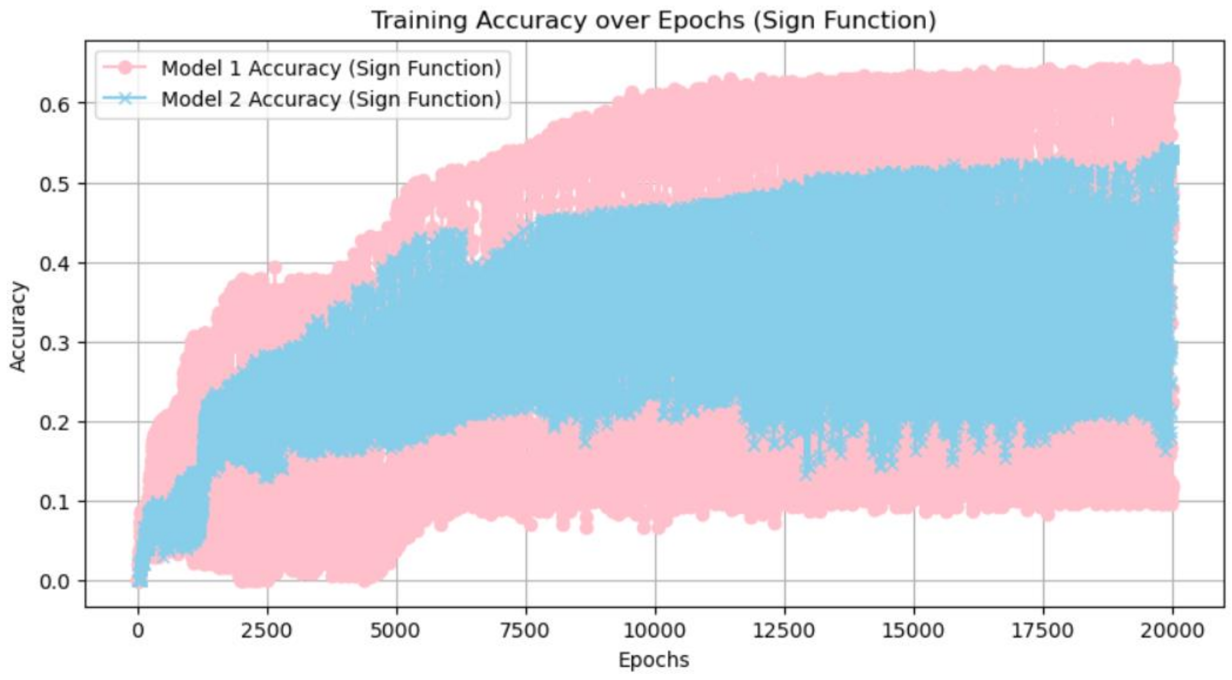
All models 1 and 2 found that the accuracy function could yield the required outcomes if we were given an unknown input. Overall, the models in the center of the graph performed well. Deeper models experience reduced failure at the graph's edges.

Train on Actual Tasks:

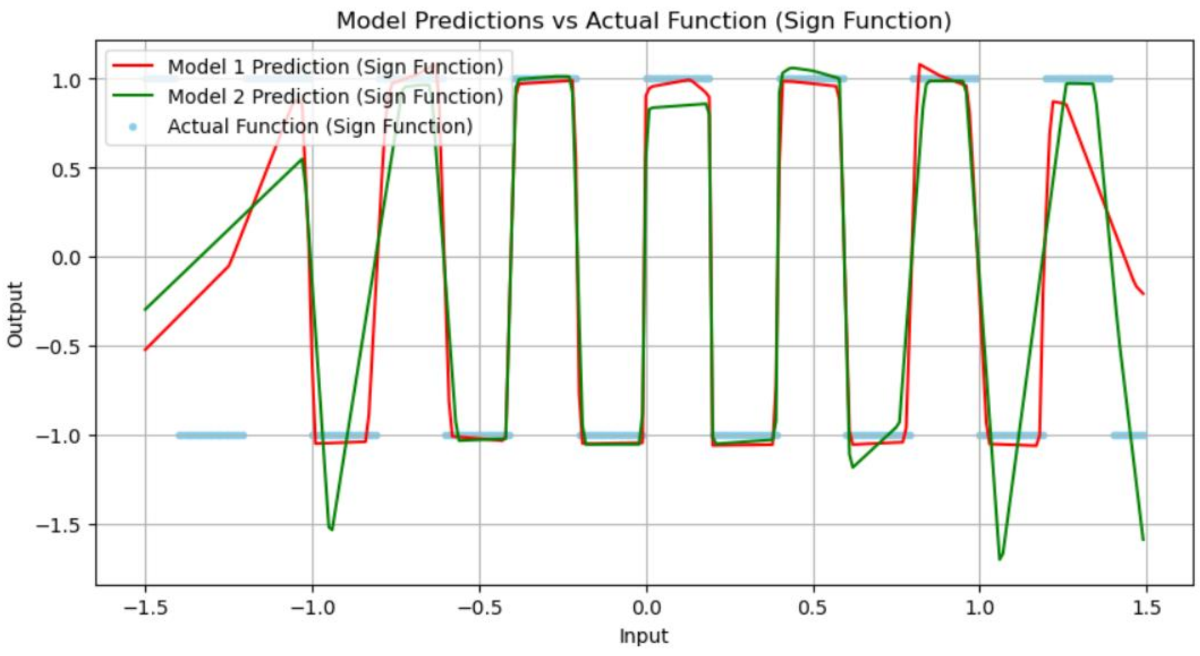
Here, I trained deep neural networks (DNNs) on actual tasks involving non-linear function approximation. The focus was on training two different models on the same task to compare their performance, using datasets with well-known structures. And max epochs reached without convergence for both training loss and accuracy.



The above image shows training loss of Model 1 and Model 2. The below image shows training accuracy of Model 1 and Model 2.



The graph below shows model prediction and actual function. This image shows the training loss for both Model 1 and Model 2 over 20,000 epochs, comparing their performance on both functions. Model 2 consistently performed better (lower loss) for both functions.



Results:

Learning a discontinuous function like the signum function was more challenging for both models. Model 2 achieved a lower loss overall but struggled near the transition points where the function abruptly changes between -1 and 1.

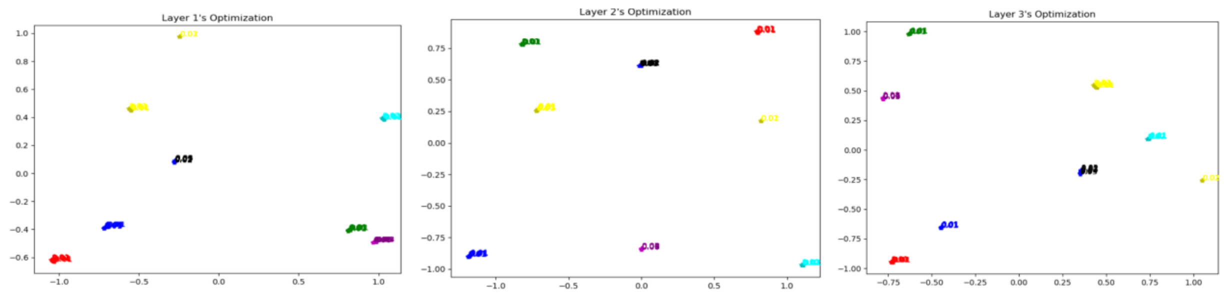
Model 2 gave a more stable and accurate fit, while Model 1 showed more oscillations near transition points, although it was able to capture the general shape of the function.

Part 2: Optimization

Visualize the Optimization Process:

In this process, I trained a deep neural network (DNN) on a non-linear function task using a model with multiple layers. The task involved learning a non-linear relationship between the input and output using a custom architecture. The architecture of the model included multiple fully connected layers with varying numbers of neurons at each layer, starting from 4 neurons in the first layer and ending with 12 neurons in the fourth layer, followed by a single-output neuron for the final prediction.

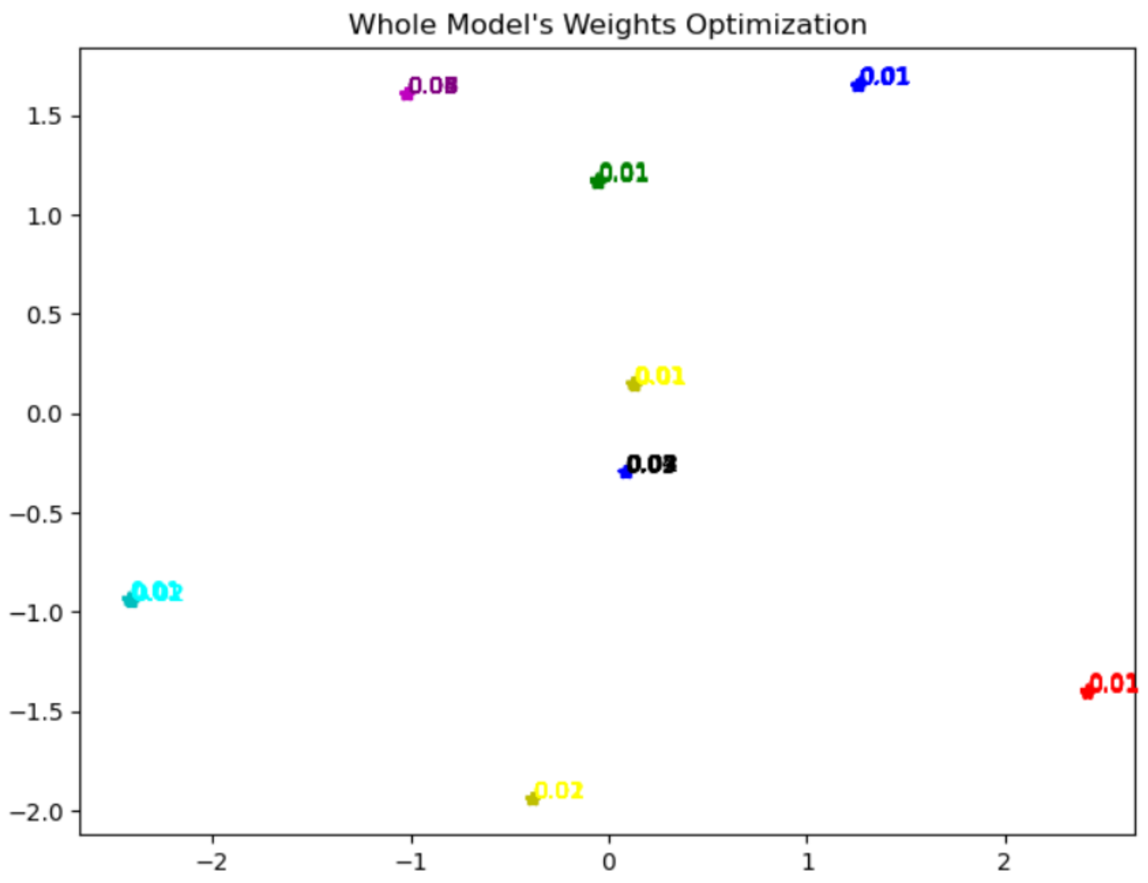
The total number of parameters in the model was 199, which allowed it to have sufficient capacity to learn the complex non-linear function.



In the above we have taken 3 layers optimization. Below image shows Whole model’ Weight Optimization.

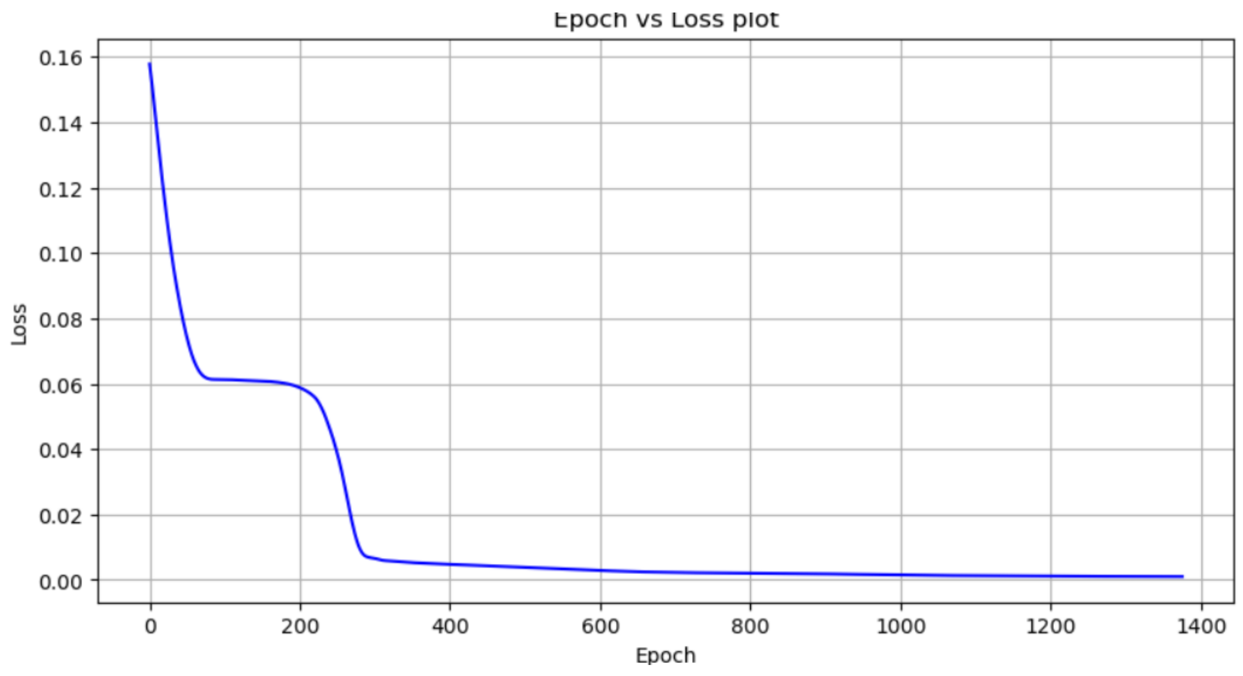
The function PcaImplem takes in an input array (inp_arr) and a target number of dimensions to retain (dim_to_ret). PCA is applied to the weight tensors of three layers (`lay1`, `lay2`, and `lay3`) and the entire model's weights (`whole_mod_red`).

The reduced-dimension data helps in simplifying the model weights while retaining most of the significant features, making it easier to analyze the weights and potentially reducing computational complexity. The `loss_vector` represents the loss values, likely for later comparison or visualization.

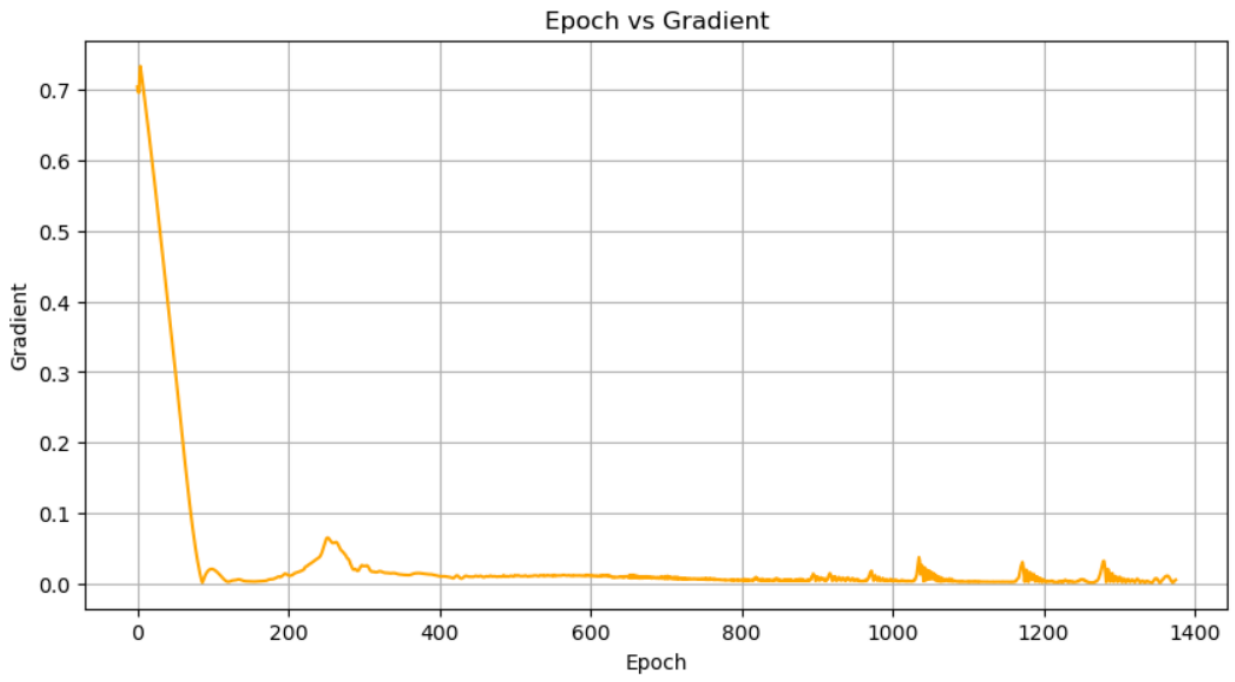


Observe Gradient Norm during Training:

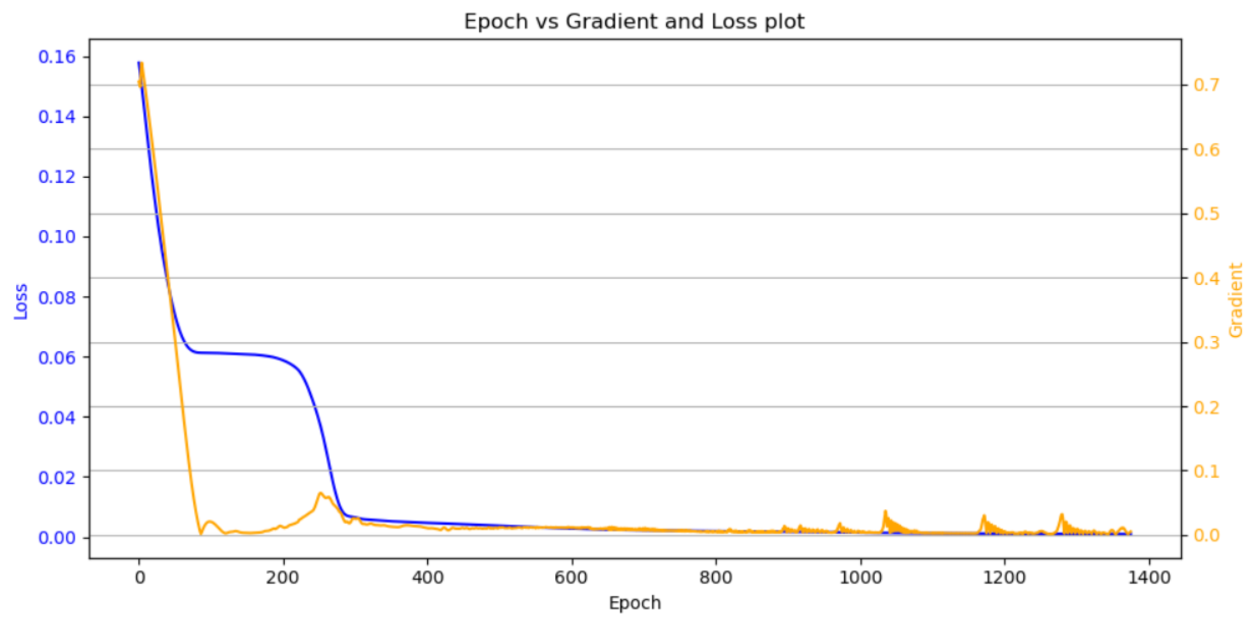
Here, the model's performance was measured by tracking the loss and gradient norms over 2500 epochs, using Mean Squared Error (MSE) as the loss function and the Adam optimizer to update the weights.



The image shows the reduction in loss over time, starting with a higher initial value and steadily decreasing as the model learns the target function. The loss plateaus after several hundred epochs, indicating that the model is approaching convergence, where further training yields diminishing improvements.



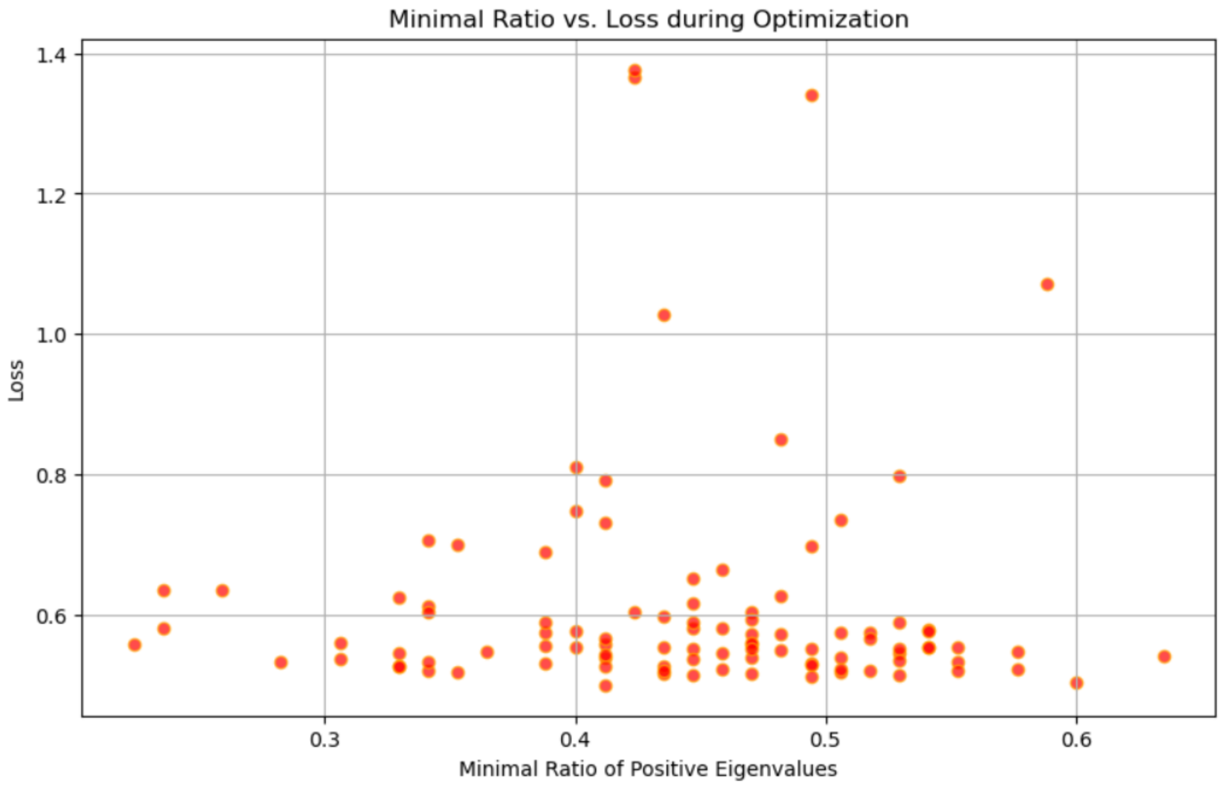
The graph tracks the magnitude of weight updates during training. Early in the training process, gradients are large as the model adjusts its weights significantly. As training progresses, the gradient values decrease, reflecting smaller weight updates and a more stable learning process.



This graph overlays the loss and gradient norm on the same plot, highlighting the relationship between the two. As the loss decreases, the gradient also diminishes, signifying that the model is making more precise adjustments to its weights as it converges.

What Happened when Gradient is Almost Zero:

In this experiment, a neural network was trained to approximate the cosine function, using a shallow architecture with two hidden layers. The goal was to monitor the relationship between the loss function and the minimal ratio of positive eigenvalues during the optimization process.



The resulting graph plots the minimal ratio of positive eigenvalues from the Hessian matrix against the loss. As the model trains and the loss decreases, the minimal ratio of positive eigenvalues tends to increase, suggesting that the model is approaching a more stable local minimum.

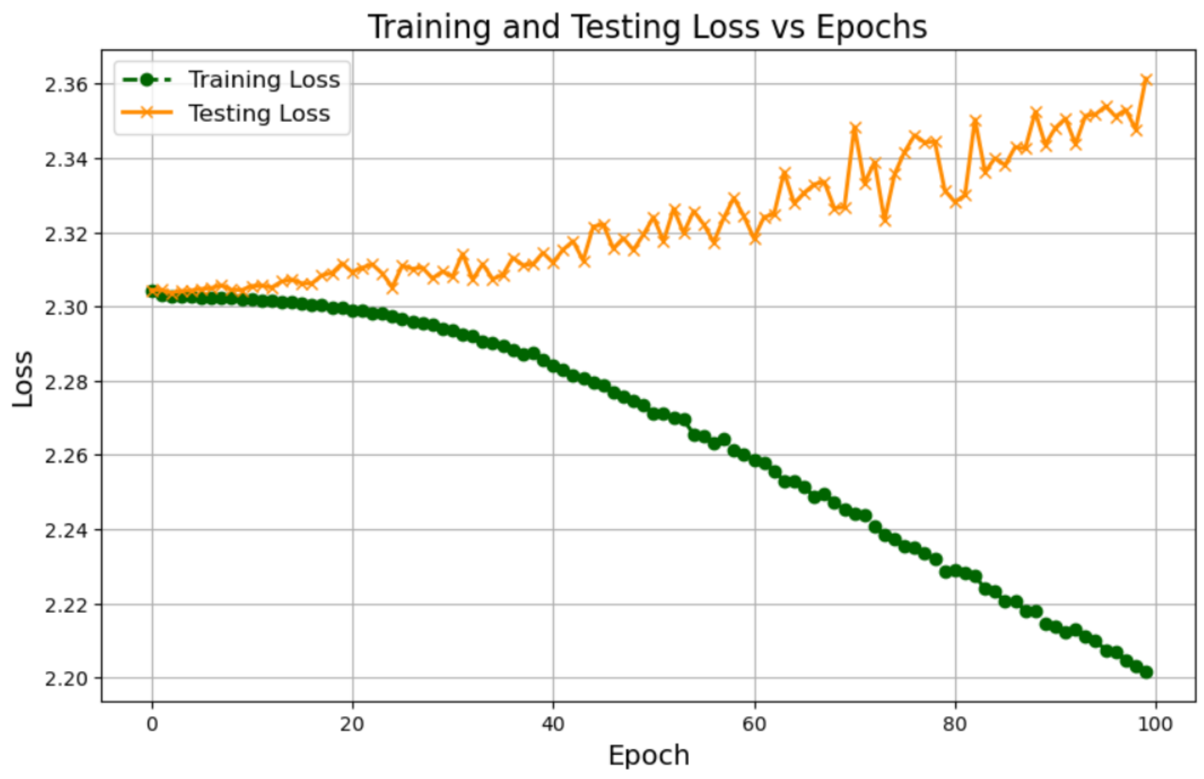
A higher ratio of positive eigenvalues indicates a convex region of the loss surface, where the model is more likely to converge to an optimal solution.

The loss values reduce as training progresses, reflecting the model's ability to fit the cosine function better over time.

Part 3: Generalization

Can Network fits Random Labels:

Here, a modified convolutional neural network (CNN) with two dropout layers was trained on the MNIST dataset. The dataset was altered by assigning random labels to the training set, while the testing set retained the correct labels. The model was trained for 100 epochs using the Adam optimizer, with a learning rate of 0.0001, and the CrossEntropyLoss function was used to compute the loss.

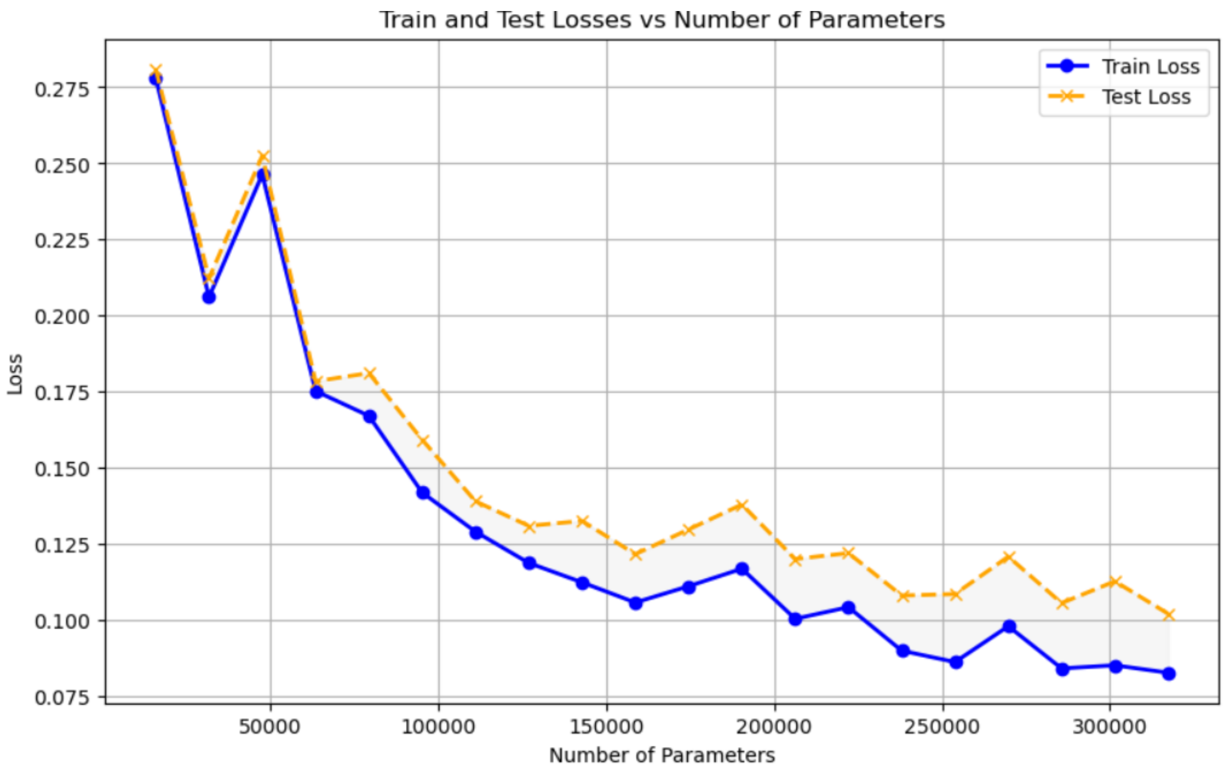


From the above image we can say that, green dashed line represents the training loss. As expected, it decreases over time, as the model memorizes the random labels.

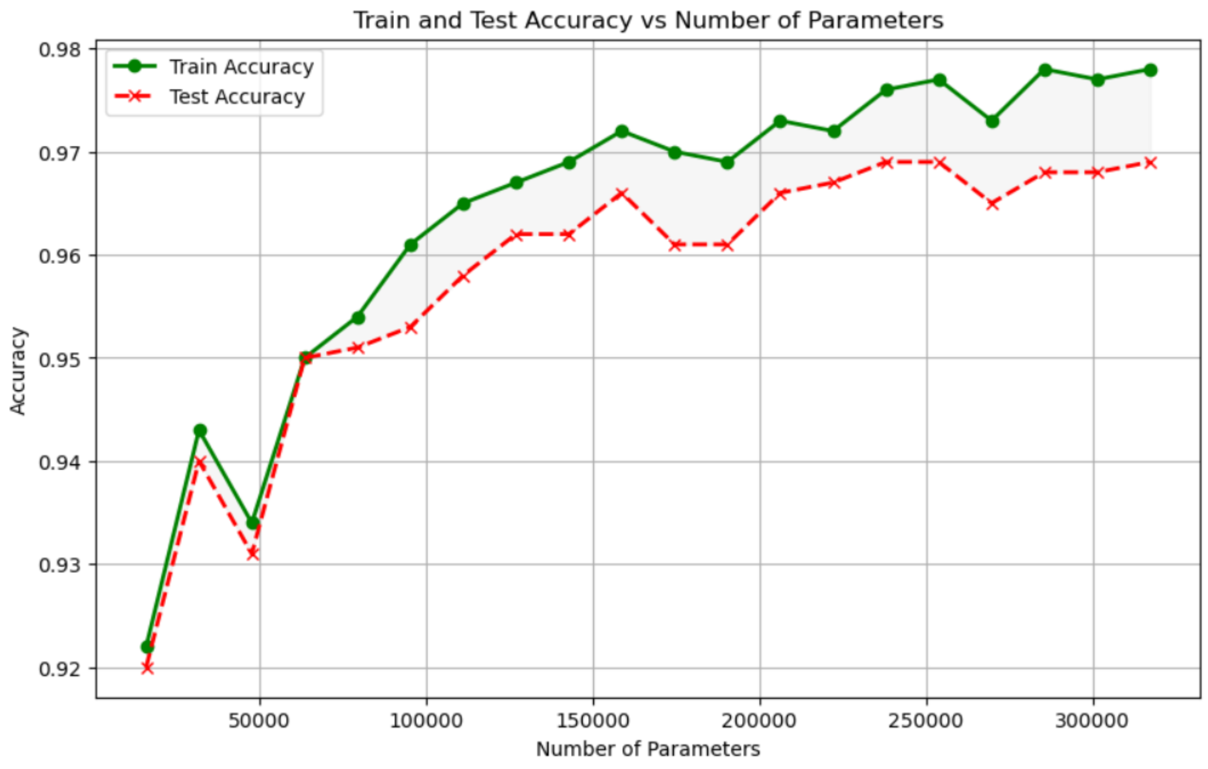
And the orange solid line represents the testing loss, which remained high and stable, reflecting the inability of the model to generalize from the random labels it learned during training to the correctly labeled test set.

Number Parameters vs Generalization:

In here, 20 different neural network models were trained on the dataset. Each model had a varying number of neurons in the first hidden layer (ranging from 20 to 400 neurons in increments of 20), while the second hidden layer was fixed at 8 neurons. The objective was to observe the effect of the number of parameters on the model's training and testing loss, as well as its accuracy.



The above image shows Train and Test Loss vs. Number of Parameters. As the number of parameters increased, the training loss generally decreased, showing that models with more parameters could fit the training data more effectively.



Training accuracy improved with larger models, suggesting that increased model capacity helped in learning the training data. However, test accuracy did not consistently improve with more parameters, indicating the risk of overfitting in larger models.

Flatness vs Generalization [Part 1]:

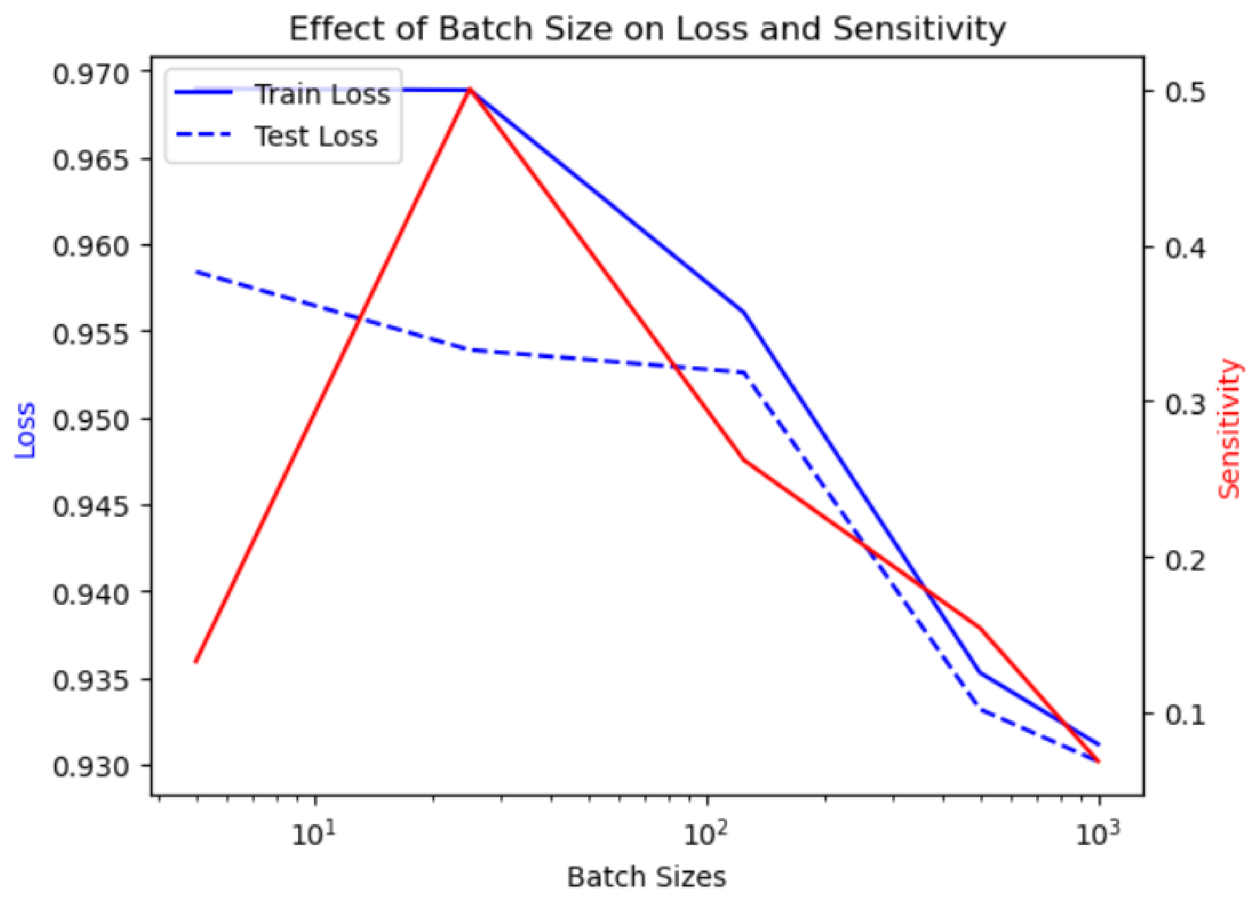
In this part of the experiment, two models (m1 and m2) were trained using different batch sizes to observe their performance in terms of loss and accuracy. After training both models, a linear interpolation was performed between the parameters of these two models to study how the interpolation ratio affects the performance of the models in terms of loss and accuracy.

The batch sizes I used were 64 and 1024.

Where batch size 64 showed faster convergence during training but also displayed higher variance in its test accuracy.

And with batch size 1024 had a smoother training curve but converged more slowly.

The below image depicts the loss and accuracy and also the linear interpolation ratio with a learning rate of 0.001 and an Adam optimizer was used.



Results:

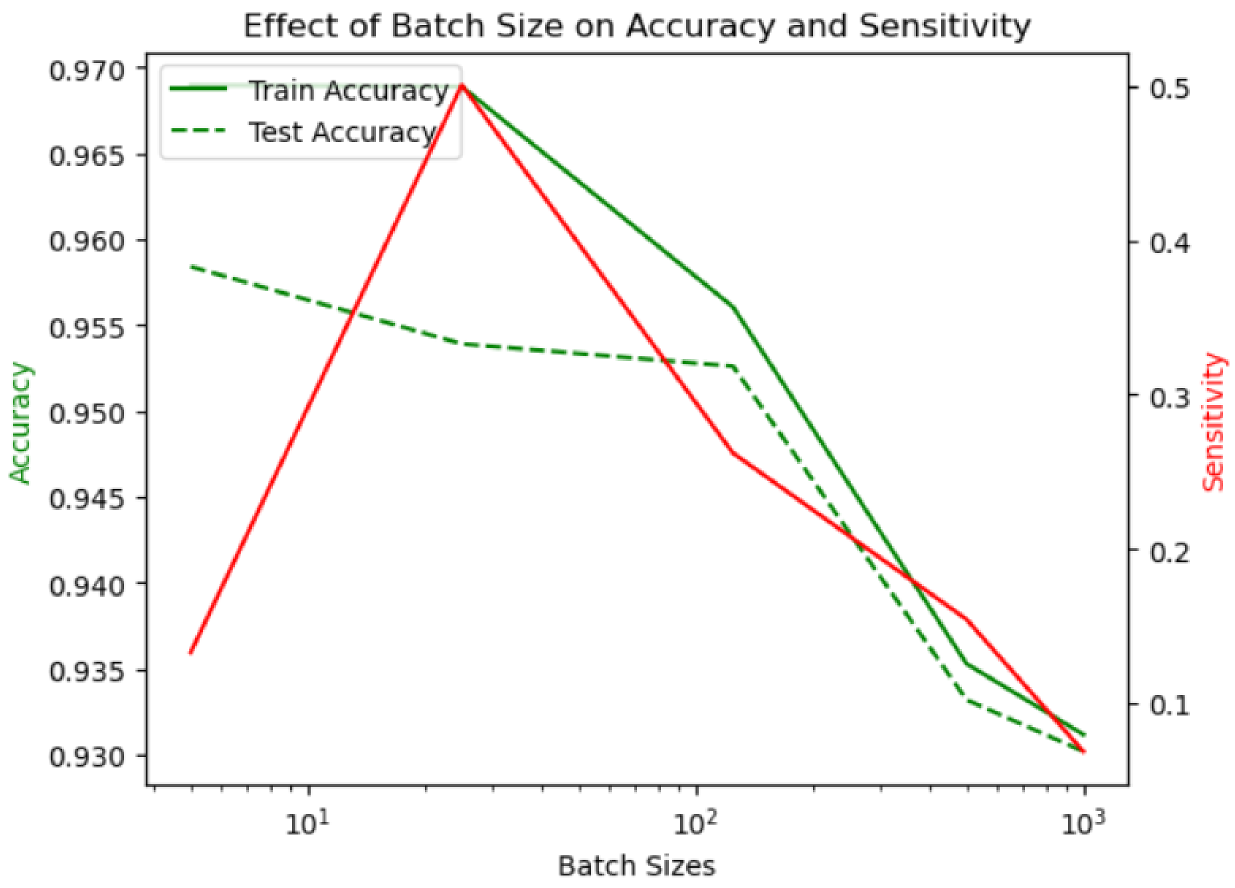
Here, after performing linear interpolation, a range of values for both loss and accuracy were recorded. The interpolation ratio played a significant role in how well the combined models generalized to unseen test data.

The intermediate models performed somewhere between the performance, with varying degrees of success depending on the interpolation ratio.

Flatness vs Generalization [Part 2]:

In this task, five models were trained with different training approaches. The goal was to record the loss, accuracy, and sensitivity of each model and analyze how the batch size affects each metric. Sensitivity was calculated as the Frobenius norm of the gradients during backpropagation.

- The model was trained on MNIST.
- Batch sizes used for training the models were: 5, 25, 125, 500, and 1000.
- Here Adam Optimizer was used and the learning rate was 0.001



Results:

The above images tell that the Training and testing, accuracy and loss, and sensitivity can be seen in the best results between 10^ 1 and 10^3 of batch sizes.

Smaller batch sizes showed high sensitivity and quicker initial gains in accuracy, but larger batch sizes resulted in more stable, though slightly slower, accuracy improvement.

Models with smaller batch sizes showed higher fluctuations in their loss values, while larger batch size showed smoother loss curves but slower convergence.

Accuracy: Models with moderate batch sizes, achieved the best balance between convergence speed and generalization, with higher accuracy on the test set.

Sensitivity: Models trained with smaller batch sizes exhibited higher sensitivity, indicating that they were more reactive to changes in the training data. Larger batch sizes showed lower sensitivity, reflecting more stable but slower learning.