Homework 1

1-1 Simulate a Function

GitHub link: https://github.com/harsha-mangena/DL-CPSC-8430/blob/main/Home-Work-1/HW-1_simulate_function.ipynb

The descriptions provided outline the architecture and training configurations of three distinct neural network models, each designed with specific characteristics in mind. These models incorporate a weight decay parameter as a form of regularization to enhance generalization by penalizing large weights during the training process. Below is an elaborated description of each model:

Model 1: Comprehensive Multi-layer Architecture

- **Architecture**: Model 1 is constructed with seven dense (fully connected) layers, demonstrating a more complex network design intended to capture intricate patterns in the data. This complexity allows the model to learn a variety of high-level features at different layers.
- **Parameters**: The model comprises 571 parameters in total.
- **Loss Function**: It employs Mean Squared Error Loss (MSELoss), which is standard for regression tasks, indicating that this model is likely aimed at predicting continuous values.
- **Optimizer**: RMSProp is used, a choice that suggests an emphasis on adaptive learning rates, helping in converging faster and more effectively in the context of complex landscapes.
- **Learning Rate**: Set to 0.001 (1e-3).
- **Activation Function**: LeakyReLU is chosen for its ability to allow a small, non-zero gradient when the unit is not active, potentially mitigating the vanishing gradient problem.
- **Weight Decay**: A weight decay of 0.0001 (1e-4) is applied.

Model 2: Balanced Mid-sized Network

- **Architecture**: This model features four dense layers. With 572 parameters, it's designed to be somewhat complex but not as deep as Model 1, offering a balanced approach to learning representations.
- **Loss Function**: Also uses MSELoss, fitting for regression problems where the goal is to minimize the difference between the predicted and actual values.
- **Optimizer**: Adopts RMSProp, beneficial for its adaptive learning capabilities, which is particularly useful in networks with a moderate number of layers.

Vamsi Sai Ranga MANGINA
C89194110

- **Learning Rate**: Maintains a learning rate of 0.001.
- **Activation Function**: LeakyReLU is used, providing the benefits of maintaining gradient flow in layers that might otherwise become inactive and halt learning.
- **Weight Decay**: Implements the same level of weight decay as Model 1, aiming to keep the model weights small to combat overfitting.

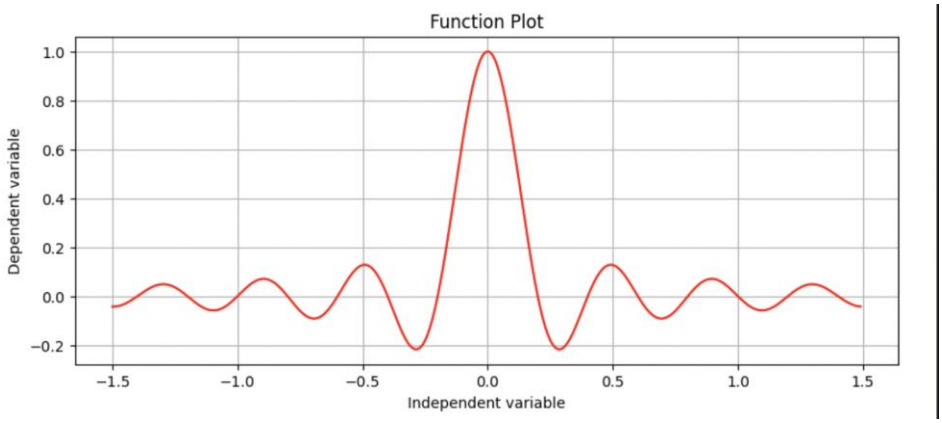
Model 3: Simplified Single-layer Design

- **Architecture**: Consists of a single dense layer, with a total of 571 parameters.
- **Loss Function**: Utilizes MSELoss, indicating its application towards regression tasks, focusing on minimizing the square of errors between targets and predictions.
- **Optimizer**: Uses RMSProp, ensuring that even simpler models benefit from adaptive learning rates for faster convergence.
- **Learning Rate**: The learning rate is set at 0.001.
- **Activation Function**: Implements LeakyReLU, which is especially useful in a single-layer model to avoid dead neurons and ensure some level of non-linearity in predictions.
- **Weight Decay**: A weight decay of 0.0001 is applied.

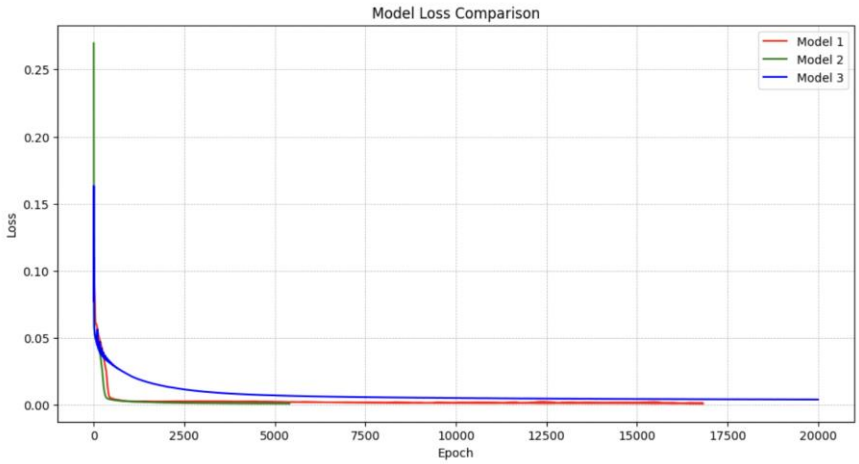
Function 1

Function: sin(5*pi*x) / 5*pi*x

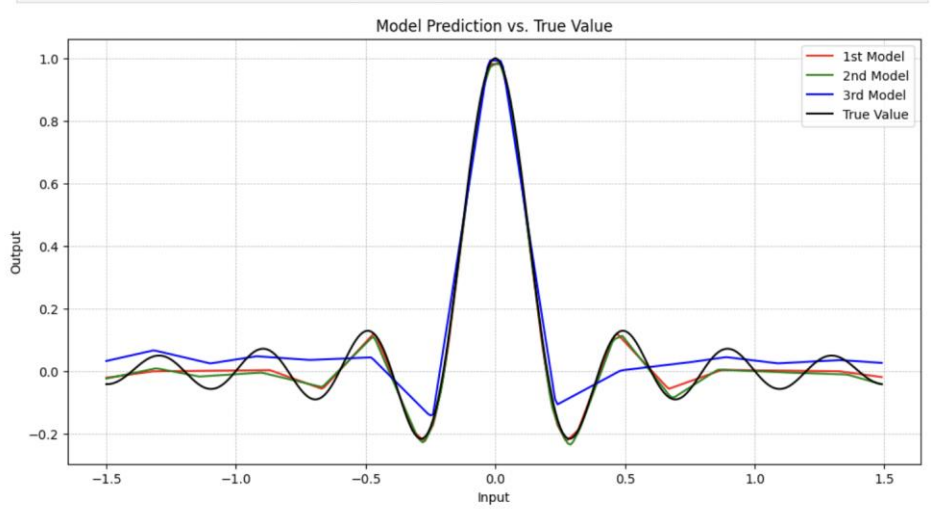
Below is the function plot for the same function.



Each model achieves convergence upon reaching the maximum number of epochs designated for training or when the pace of learning diminishes significantly.



The following graph illustrates the comparison between the actual values (Ground Truth) and the predictions made by the three models.



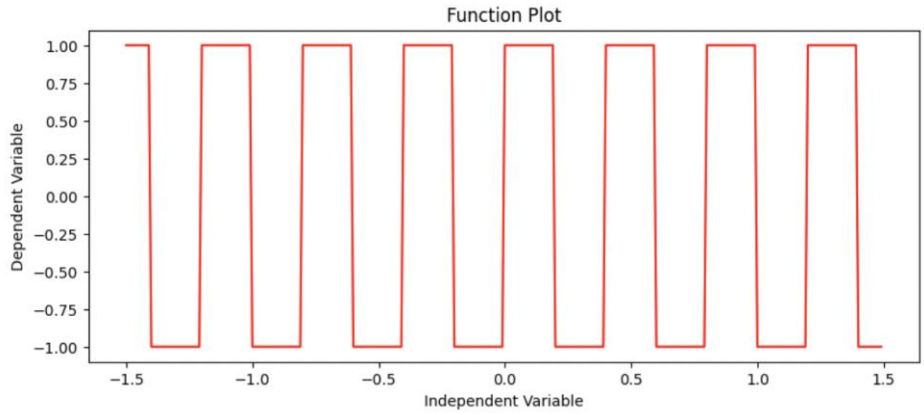
Observed Results

The outcomes reveal that Models 1 and 2 achieve convergence at a notably quicker pace in contrast to Model 3, which requires the full span of allotted epochs to approach convergence. As highlighted by the graph, Models 1 and 2 exhibit lower loss values, indicating a superior ability to learn the underlying function compared to Model 3. This phenomenon underscores the significance of the models' layered architectures, where the additional layers in Models 1 and 2 evidently contribute to their enhanced learning efficiency and overall performance.

Function 2

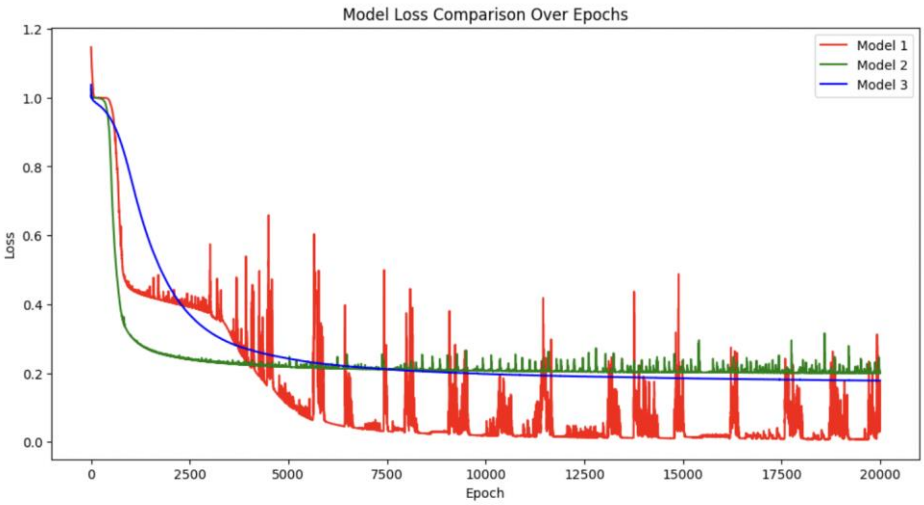
Function: sgn(sin(5*pi*x) / 5*pi*x)

Below is the function plot for the same function:

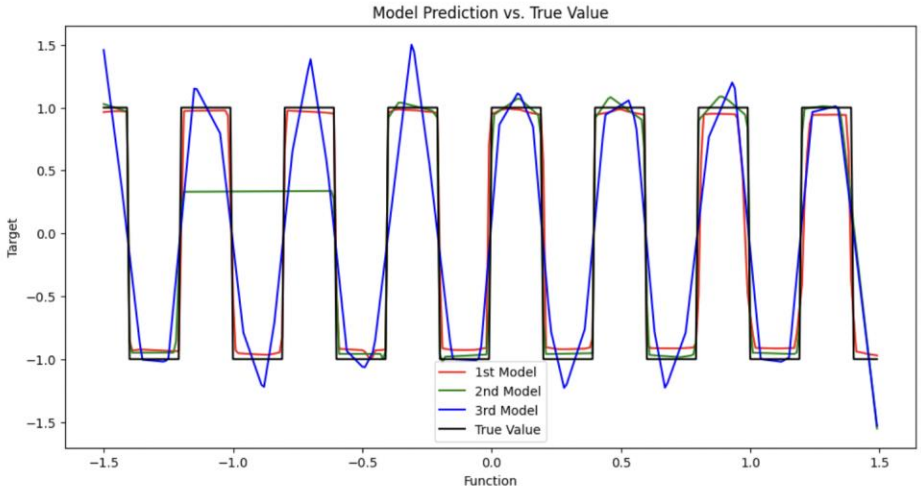


Simulation Insights

Each model reaches a point of convergence either upon completing the designated maximum number of epochs or when the rate of learning significantly slows down. The accompanying graph provides a visual representation of the loss encountered by these models throughout the training phase.



The graph below shows the Ground Truth vs the Prediction for the 3 models.



Observed Results

All models struggled to fully grasp the complex function, hitting the maximum number of epochs without fully converging. Model 1 emerged as the top performer with the lowest loss, slightly ahead of Model 2, demonstrating its superior learning capability. Meanwhile, Model 3 lagged, unable to significantly lower its loss or achieve convergence, highlighting the advantage of having more layers in a neural network for better and faster learning.

1-1 Train on Actual Tasks

GitHub link: https://github.com/harsha-mangena/DL-CPSC-8430/blob/main/Home-Work-1/HW-1_COMMNIST.ipynb

The models described below were trained using the MNIST dataset, which consists of 60,000 training images and 10,000 test images. Each model employs a different architecture to recognize handwritten digits.

Model 1: CNN (LeNet)

- 2D convolution layer: apply a 2D max pooling -> ReLu
- 2D convolution layer: apply a 2D max pooling -> ReLu
- 2D Dense Layer: ReLu
- 2D Dense Layer: ReLu

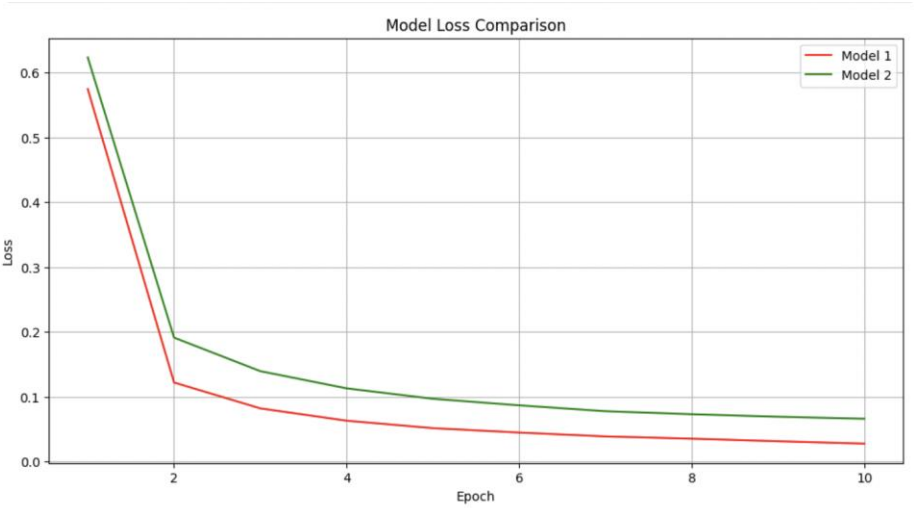
Model 2: CNN (custom)

- 2D convolution layer: ReLu
- 2D convolution layer: apply a 2D max pooling -> ReLu -> Dropout
- 2D convolution layer: apply a 2D max pooling -> ReLu -> Dropout
- 2D Dense Layer: ReLu -> Dropout
- 2D Dense Layer: Log_softmax (Output)

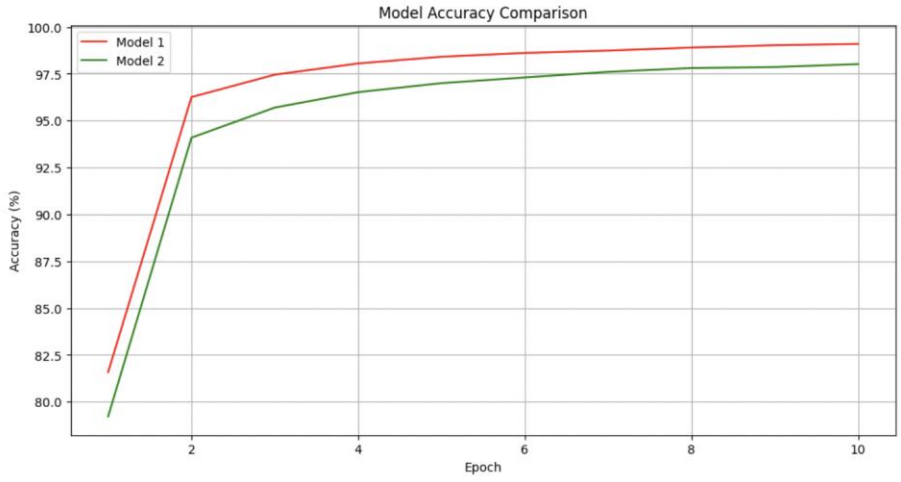
Hyperparameters for Training

- Learning rate is set to 0.01, with a momentum of 0.5.
- The models use Stochastic Gradient Descent (SGD) as their optimizer.
- Training is performed with a batch size of 64 over 10 epochs.
- The loss function used is Cross Entropy, suitable for classification tasks.

Below is the training loss for Model 1 and Model 2.



Below is the training accuracy for Model 1 and Model 2.



Observed Results

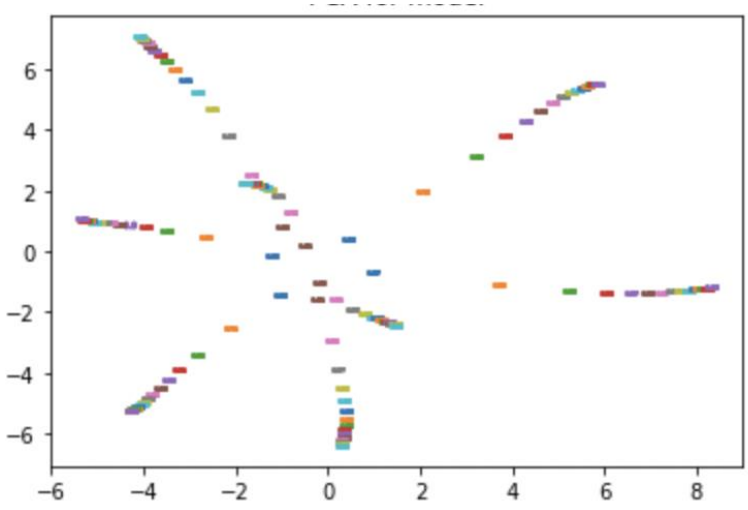
Model 1, with its architecture inspired by the well-established LeNet design, achieves a lower loss and superior performance compared to Model 2. This optimized CNN model significantly

surpasses the custom-built CNN across all epochs. Similarly, when it comes to training accuracy, Model 1 also leads, demonstrating higher effectiveness in recognizing handwritten digits from the MNIST dataset.

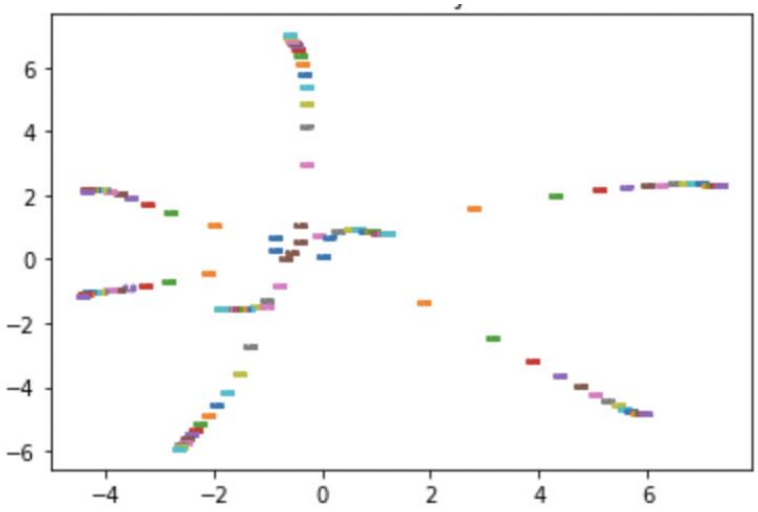
1-2 Visualize the optimization process

GitHub Link: https://github.com/harsha-mangena/DL-CPSC-8430/blob/main/Home-Work-1/HW-1_PCA.ipynb

- Dataset: MNIST
  - Training Set Size: 60,000 images
  - Testing Set Size: 10,000 images
- Model Configuration:
  - Dense Layers: 3
  - Activation Function: ReLu
- Training Specifications:
  - Loss Function: Cross Entropy Loss
  - Optimizer: Adam
  - Learning Rate: 0.0004
  - Batch Size: 1000
- Additional Procedure:
  - Collect model weights every 3 epochs during training.
  - Conduct training for a total of 8 cycles.
  - Post-training, apply PCA to reduce the weight dimensions to 2.



PCA for Model.



PCA for 1-Layer.

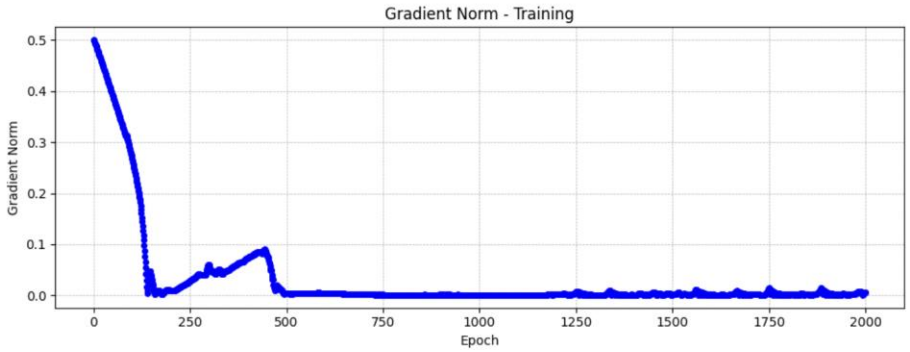
The graph above shows the model weights after we've used PCA to simplify them. Originally, each model had 8,240 parameters. After training the models 8 times for 45 epochs and applying PCA, we see how the weights are reduced to just two dimensions in the graph.

1-2 Observe gradient norm during training

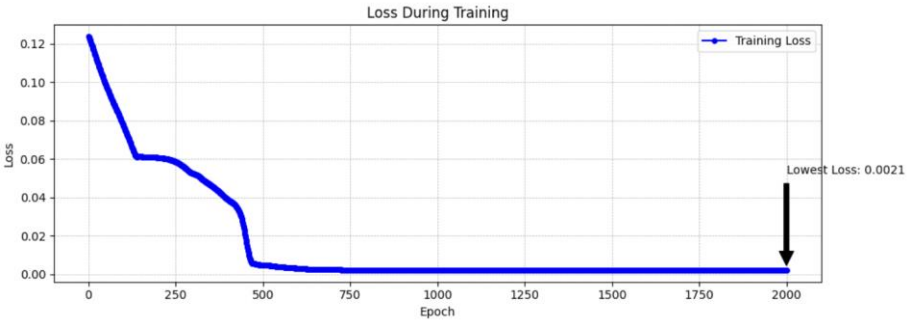
Github: https://github.com/harsha-mangena/DL-CPSC-8430/blob/main/Home-Work-1/HW-1_grad_norm.ipynb

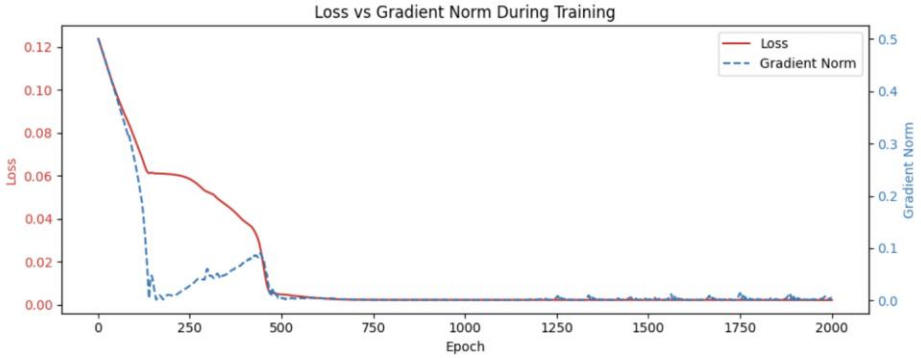
- Function Used for Calculations: Gradient norm and loss calculated using the function sin(5*pi*x) / 5*pi*x.
- Training Approach: The model is trained across epochs instead of iterations due to the small size of the input data.
- Visualization: A graph displaying the gradient norm throughout the epochs is provided below.

Below is a graph for gradient norm across the epochs.



Below is a graph for loss across the epochs.





Observed Result:

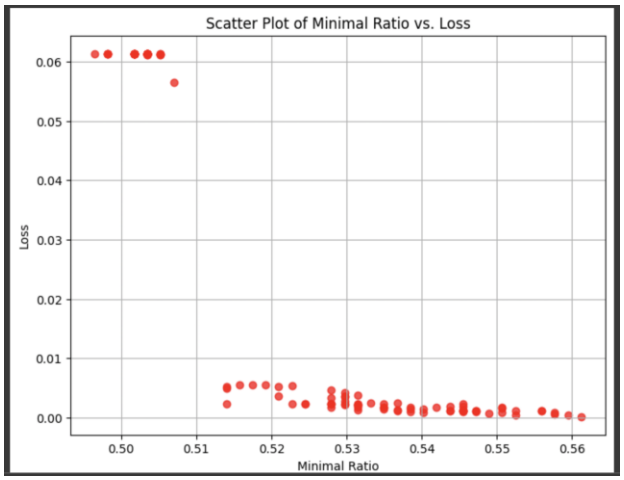
- Training Outcome: The model successfully trained and reached convergence.
- Gradient Trend: Post-100 epochs, there's a noticeable uptick in the gradient, mirroring trends seen in the loss graph.
- Loss Behavior: Initially, the loss levels off, then it starts to decline more slowly before eventually stabilizing after approximately 500 epochs.

1-2 When Gradient is Almost Zero

Github: https://github.com/harsha-mangena/DL-CPSC-8430/blob/main/Home-Work-1/HW-1_min_grad.ipynb

- The function train_model trains a given neural network model (model) using the provided input data (x) and target labels (y).
- It iterates over a fixed number of epochs (4000) to train the model, updating its weights based on the calculated loss.
- During training, it tracks the loss values, epoch numbers, and gradient norms.
- If the gradient norm falls below a threshold or the maximum number of epochs is reached, the training loop terminates. It then calculates the minimal ratio based on the eigenvalues of the Hessian matrix and returns the final loss value and minimal ratio.

Observed Result

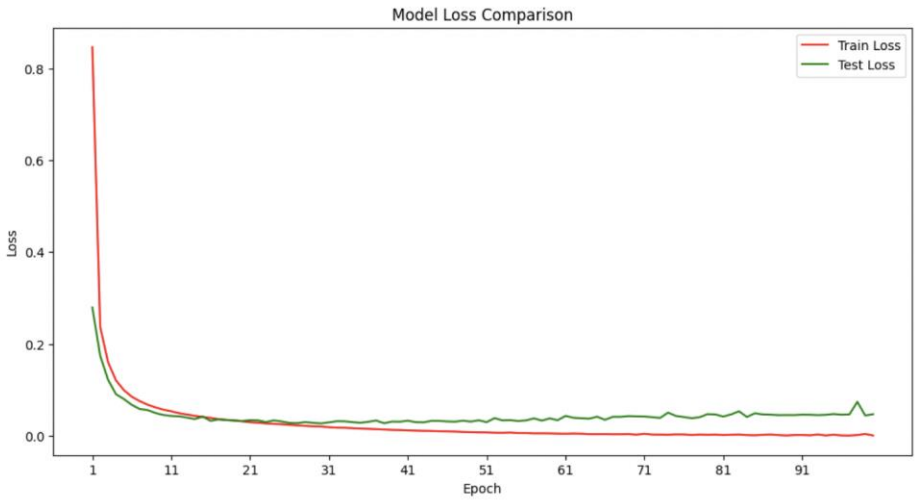


1-3 Can network fit random labels?

Github: https://github.com/harsha-mangena/DL-CPSC-8430/blob/main/Home-Work-1/HW-1_rand_label_fit.ipynb

- Dataset: MNIST
  - Training Set Size: 60,000 images
  - Testing Set Size: 10,000 images
- Model Configuration:
  - Dense Layers: 3
  - Activation Function: ReLU
- Training Specifications:
  - Loss Function: Cross Entropy Loss
  - Optimizer: Adam
  - Learning Rate: 0.0004
  - Batch Size: 1000

- Additional Procedure:
    - Collect model weights every 3 epochs during training.
    - Conduct training for a total of 8 cycles.
    - Post-training, apply PCA to reduce the weight dimensions to 2.



The model was trained on randomly assigned labels, making the training slow as it tries to memorize these random labels. Over epochs, it reduces the loss by essentially memorizing the training data. However, since the labels are random, the test loss increases gradually as the model struggles to generalize to unseen data. This results in a widening gap between the training and test losses as epochs increase.

1-3 Number of parameters vs Generalization

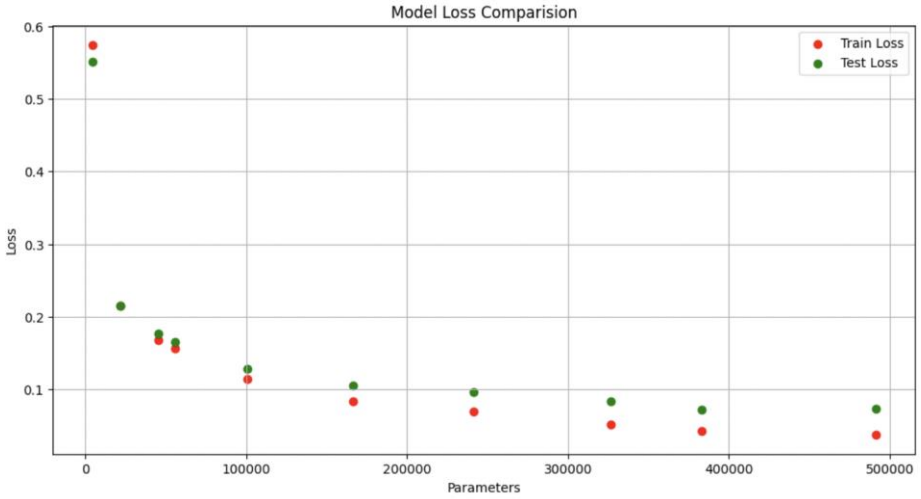
Github: https://github.com/harsha-mangena/DL-CPSC-8430/blob/main/Home-Work-1/HW-1_param_compare.ipynb

- Dataset: MNIST
    - Training Set: 60,000 images
    - Testing Set: 10,000 images
- Model Configuration:
    - Dense Layers: 3
    - Activation Function: ReLU
- Training Specifications:
    - Loss Function: Cross Entropy Loss

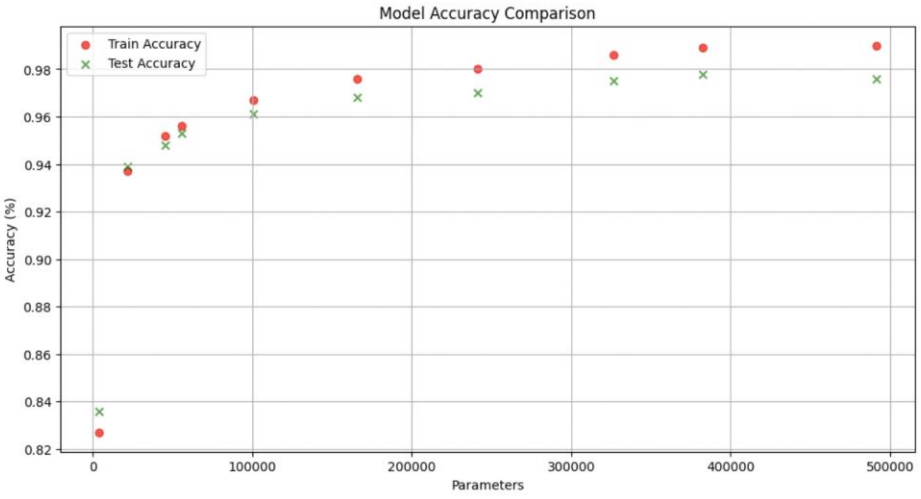
- Optimizer: Adam
- Learning Rate: 1e-3
- Batch Size: 50

We vary the size of the models by approximately doubling the inputs and outputs at every dense layer, this increases the number of parameters for training.

Graph for Loss comparison for the various models.



Graph for Accuracy comparison for the various models



The graphs show that as the number of parameters increases, the gap between the training and test loss/accuracy widens. The test loss stabilizes earlier than the training loss, indicating

overfitting. With more parameters, the model becomes more complex and can memorize the training data well, leading to high accuracy on the training set but poorer generalization to unseen data. To mitigate overfitting, it's crucial to minimize the difference between train and test loss/accuracy. Unfortunately, due to computational constraints, I couldn't increase the number of parameters further, but the trend suggests the impact of overfitting.

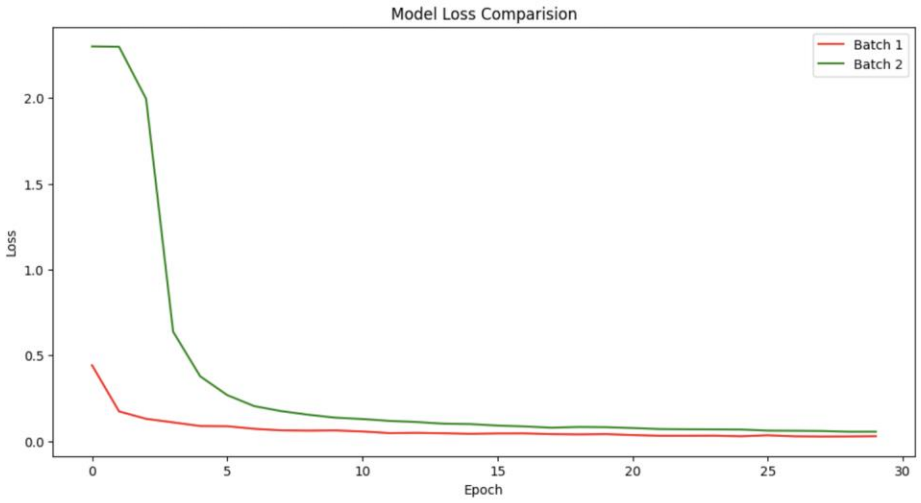
1-3 Flatness vs Generalization

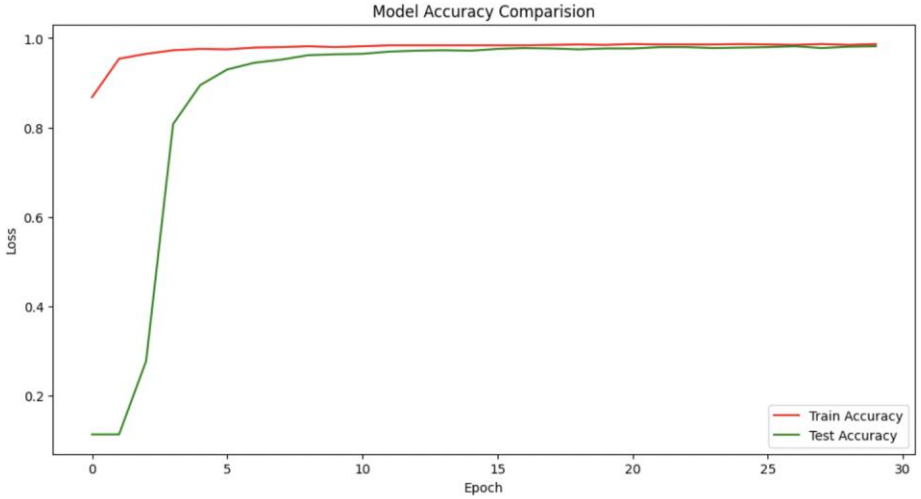
Part 1

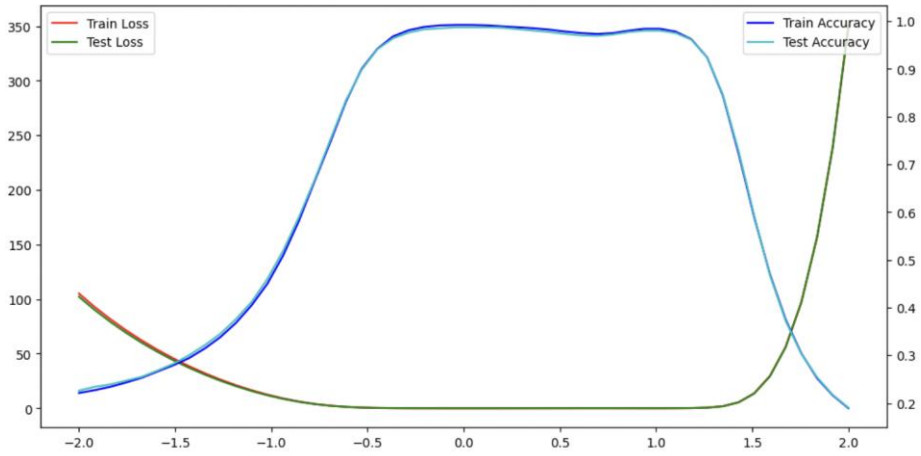
Github: https://github.com/harsha-mangena/DL-CPSC-8430/blob/main/Home-Work-1/HW-1_Flat_vs_General_interpolation.ipynb

- Dataset: MNIST
  - Training Set: 60,000 images
  - Testing Set: 10,000 images
- Model Configuration:
  - Dense Layers: 3
  - Convolutional Layers: 2
  - Activation Function: ReLU
- Training Specifications:
  - Loss Function: Cross Entropy Loss
  - Optimizer: SGD
  - Learning Rates: 1e-3, 1e-2
  - Batch Sizes: 100, 500
- Process Overview:
  - Train models with different batch sizes and collect their weights.
  - Use interpolation to combine weights from different batch sizes.
  - Create 50 models with new interpolated weights.
  - Capture loss and accuracy of new models to plot them against original models in a single graph.

Learning Rate 1e-2

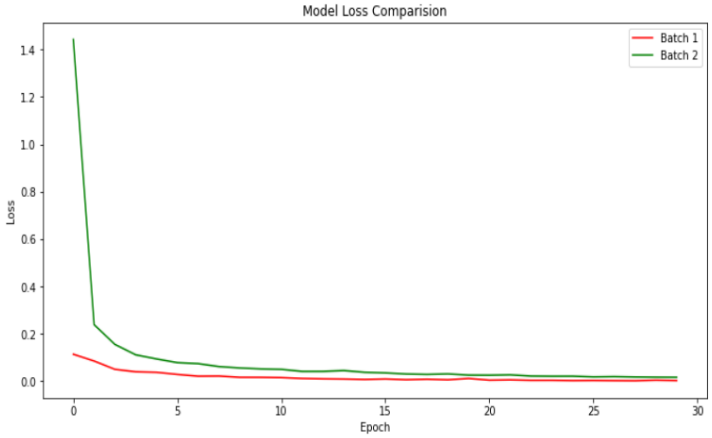


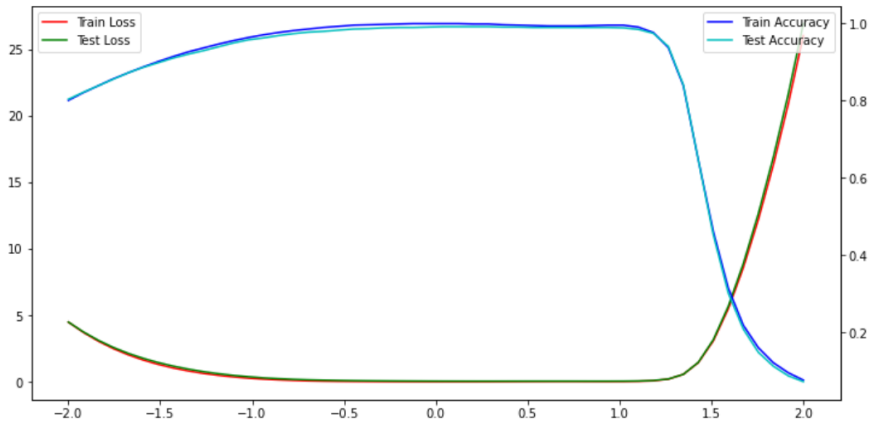




Vamsi Sai Ranga MANGINA
C89194110

Learning Rate 1e-3





Observed Result

The accuracy of both models with varying learning rates drops around an alpha value of 1.5, and then steeply increases. Alpha represents the interpolation ratio calculated using the formula ***"(1-alpha) * batch1_param + alpha * batch2_param"***. This suggests that machine learning algorithms tend to interpolate between data points. However, if the model has more parameters than data, it effectively memorizes the data and interpolates between them.

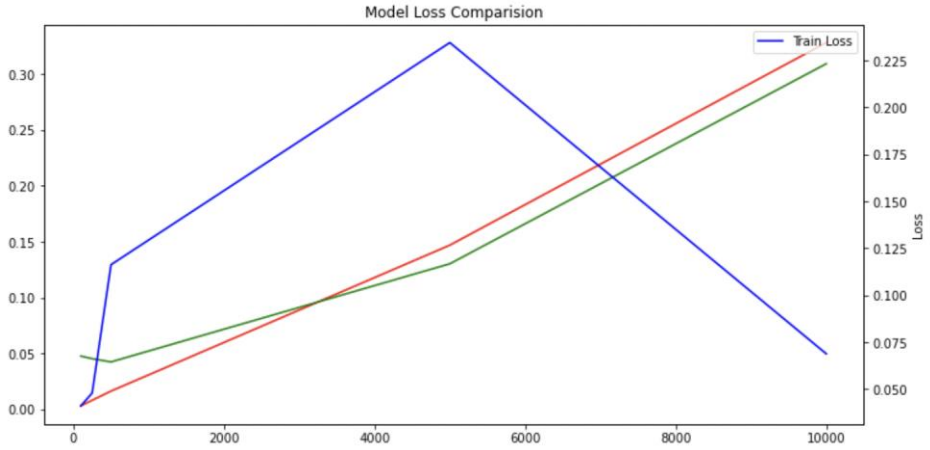
Part 2

Flatness vs Generalization

Github: https://github.com/harsha-mangena/DL-CPSC-8430/blob/main/Home-Work-1/HW-1_Flat_vs_General_sensitivity.ipynb

- Dataset: MNIST

    - Training Set: 60,000 images
    - Testing Set: 10,000 images
- Model Configuration:
    - Dense Layers: 3
    - Convolutional Layers: 2
    - Activation Function: ReLU
- Training Specifications:
    - Loss Function: Cross Entropy Loss
    - Optimizer: SGD
    - Learning Rate: 1e-3
    - Batch Size: 50



- Legend in graphs incorrectly displays sensitivity as blue line, while red/green lines represent model accuracy and losses.
- X-axis denotes batch size.
- Y-axis in the top graph represents loss, while Y-axis in the bottom graph represents accuracy.