Jordyn Brooks
CPSC 8430 - Deep Learning
Homework 1
Github link: https://github.com/brooksjk/Deep_Learning - under folder HW1

1-1: Deep vs Shallow

Github file: HW1_1-1.ipynb

A. Simulate a function

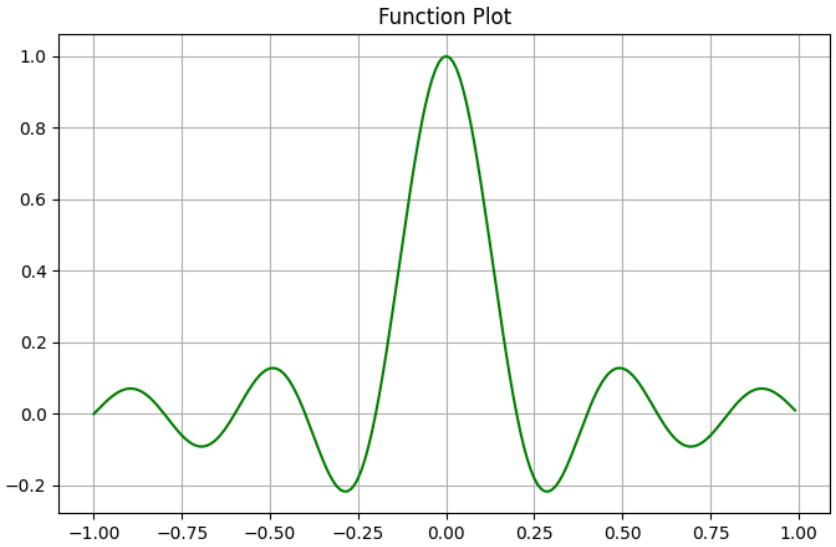
Models:

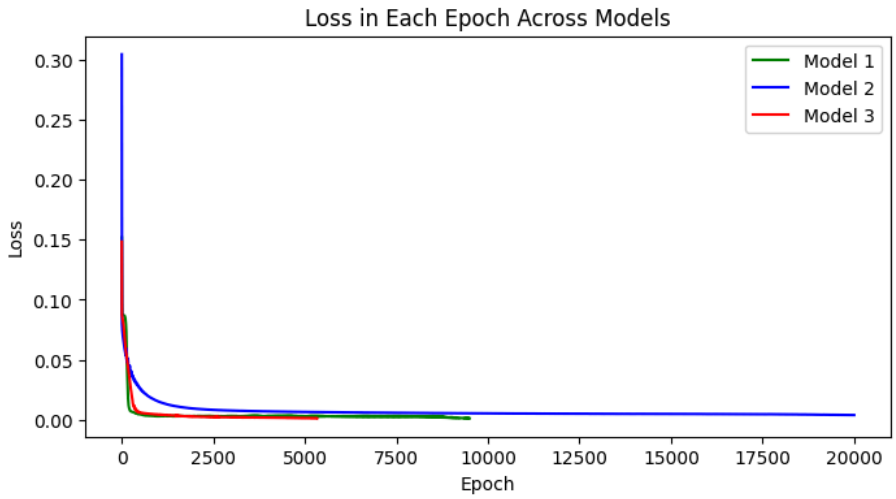
- Model 1
  - Number of parameters: 571
  - Architecture:
    - Input Layer: 1 input feature
    - Hidden Layers:
      - Layer 1: 5 neurons
      - Layer 2: 10 neurons
      - Layers 3–6: 10 neurons
      - Layer 7: 5 neurons
    - Output Layer: 1 output feature
  - Activation Function: Leaky ReLU applied after every hidden layer
- Model 2
  - Number of parameters: 571
  - Architecture:
    - Input Layer: 1 input feature
    - Hidden Layer: 190 neurons
    - Output Layer: 1 output feature
  - Activation Function: Leaky ReLU applied after the hidden layer
- Model 3
  - Number of parameters: 572
  - Architecture:
    - Input Layer: 1 input feature
    - Hidden Layers:
      - Layer 1: 10 neurons
      - Layer 2: 18 neurons
      - Layer 3: 15 neurons
      - Layer 4: 4 neurons
    - Output Layer: 1 output feature
  - Activation Function: Leaky ReLU applied after every hidden layer
- All Models:

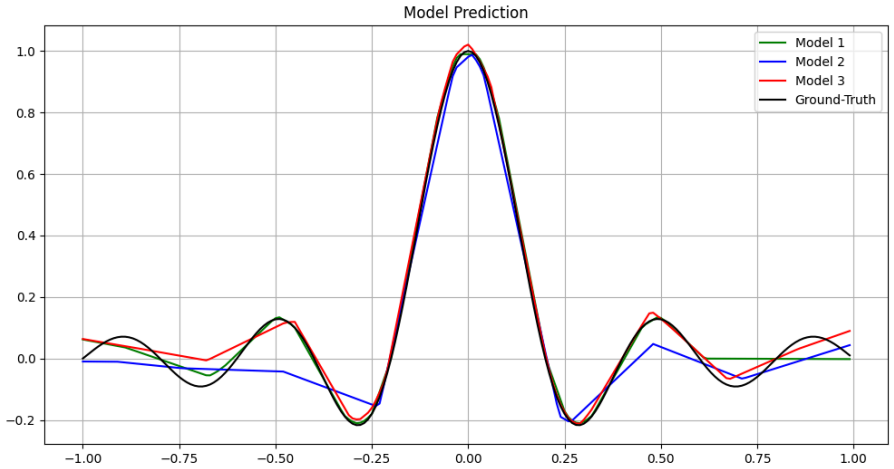
- Hyperparameters:
  - Learning Rate: 1e-3
  - Weight Decay: 1e-4
  - Loss Function: MSELoss
  - Optimizer: RMSProp
  - Maximum number of epochs: 20000

Functions:

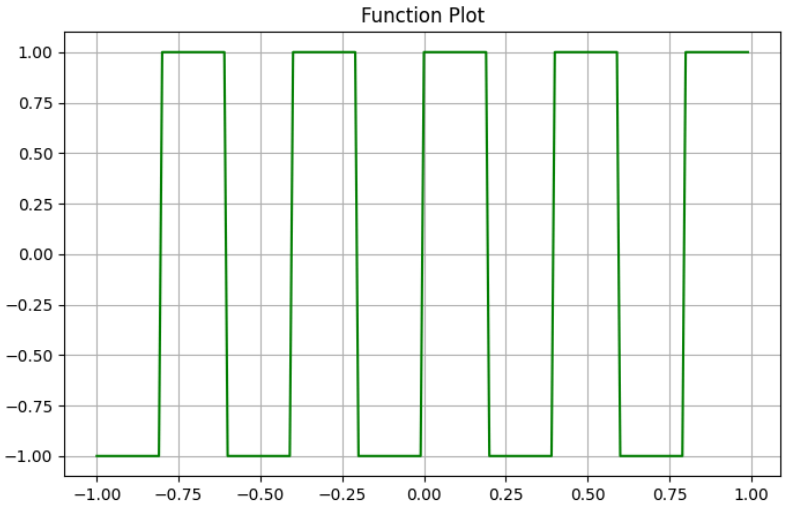
- Function 1: $\frac{sin(5\pi x)}{5\pi x}$

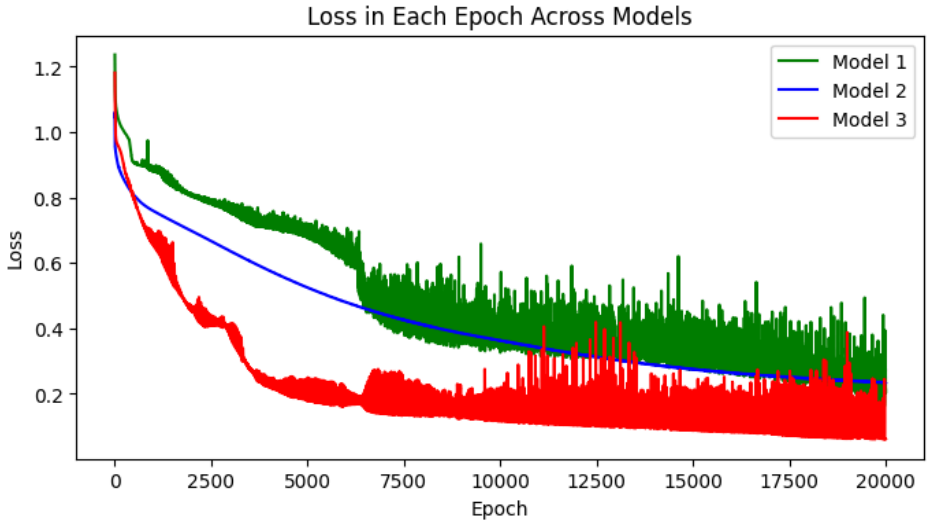


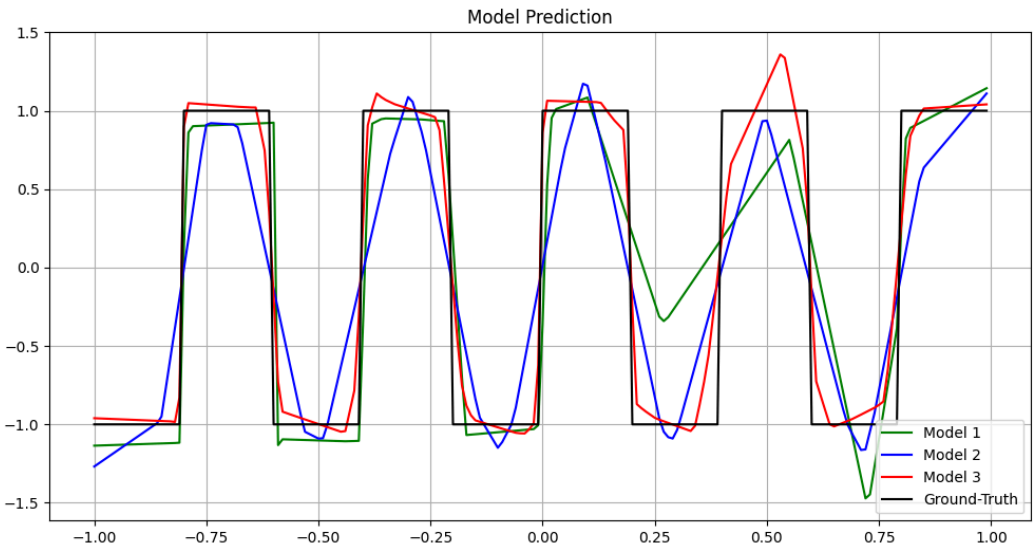




- Function 2: $sin(5\pi x)$







Results:

For both functions, the losses for each model either converge as the maximum number of epochs is reached or start to plateau as learning slows down. For Function 1, Models 1 and 3 converge relatively quickly, indicating they effectively learned the target function. In contrast, Model 2 reached the maximum number of epochs without fully converging, showing a higher final loss.

The models that converge earlier also exhibit lower loss values compared to the model that doesn't, reinforcing that more complex architectures tend to perform better. For Function 2, all models reached the maximum number of epochs before convergence, but Model 3 had the lowest final loss, demonstrating the closest fit to the ground-truth function.

Overall, these results suggest that increasing the number of layers and neurons generally improves a model's learning capacity and accuracy, allowing it to fit more complex functions better than simpler architectures.

B. Train a Task

Models:

- CNN
    - Architecture:
        - Convolutional Layers:
            - Conv Layer 1: 3 input channels (for RGB images) and hidden1_dim output channels, with a kernel size of 5x5
            - Conv Layer 2: hidden1_dim input channels and 32 output channels, with a kernel size of 5x5
        - Fully Connected Layers:

                - Fully Connected Layer 1 (fc1): The output size from the convolutional layers is flattened and connected to 256 neurons
                - Fully Connected Layer 2 (fc2): 256 neurons connected to 10 output classes
        - Activation Functions:
            - ReLU activation is applied after each convolutional and fully connected layer
            - LogSoftmax applied to the final output for multi-class classification
        - Pooling: Max-pooling with a kernel size of 2 applied after the second convolutional layer
        - Dynamic Output Size: The _get_conv_output_size method is used to compute the size of the output from the convolutional layers before feeding it into the fully connected layers
        - Adjustable Parameters: The model can be customized by changing the number of filters in the first convolutional layer (e.g., 16, 44, 80), which affects the total number of parameters and the capacity of the model to capture features.
            - This is how I obtained my 3 models used
- DNN
    - Architecture:
        - Input Layer:
            - Input size is configurable (input_size) based on the dimensions of the input data.
        - Fully Connected Layers:
            - Fully Connected Layer 1 (fc1): Configurable hidden layer size (hidden1_dim), for example, 128, 200, or 260 neurons
            - Fully Connected Layer 2 (fc2): 256 neurons connected to 10 output classes
        - Fully Connected Layer 3 (fc3):
            - Input: 256 neurons.
            - Output: 10 neurons (for classification into 10 classes).
            - Activation: Log Softmax for final classification.
    - Adjustable Parameters: The model allows for tuning the size of the first hidden layer (hidden1_dim), which affects its capacity to capture patterns. Example models include hidden sizes of 128, 200, and 260 neurons.
    - Activation Function: ReLU is applied after the first two fully connected layers to introduce non-linearity, while Log Softmax is used at the output layer for multi-class classification.

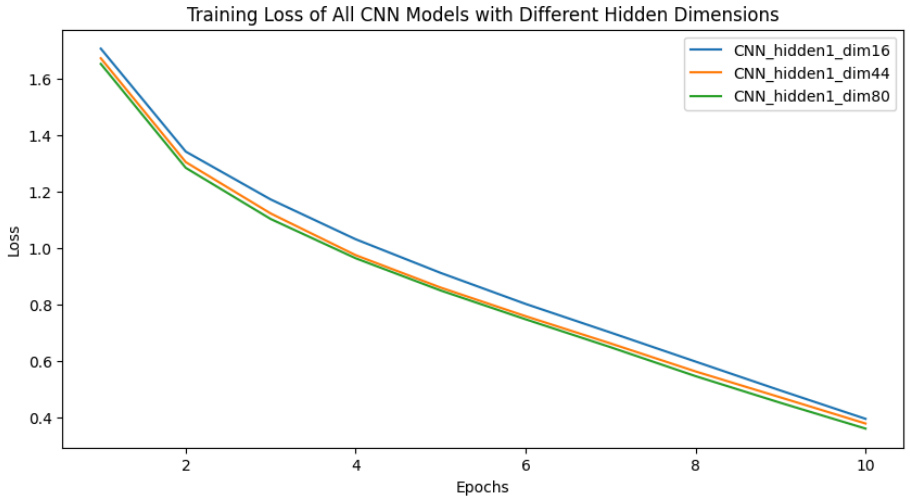
- Hyperparameters
    - Learning rate: 0.01
    - Optimizer: SDG
    - Loss function: MSELoss
    - Max epochs: 10

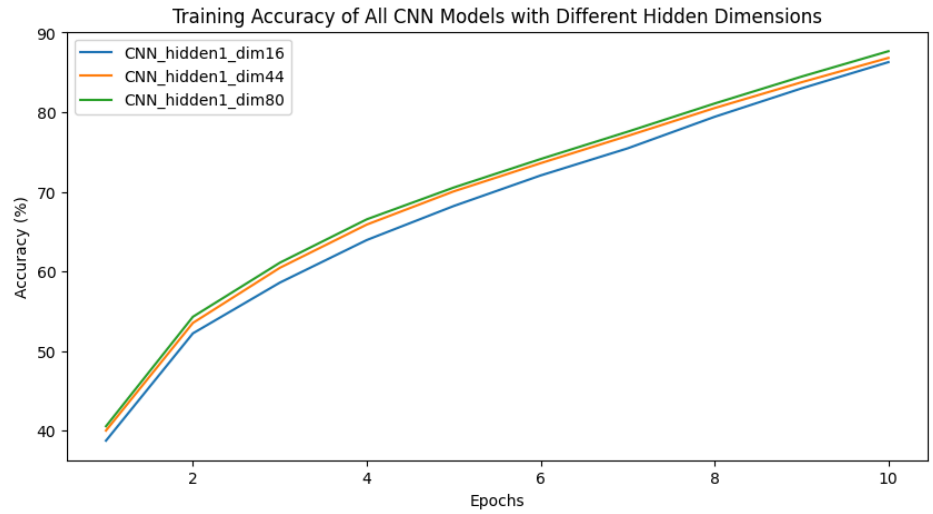
Task: CIFAR-10

The CIFAR-10 dataset is a popular benchmark dataset for image classification tasks in deep learning. It has 60,000 color photos with a resolution of 32x32 pixels, organized into ten unique categories: airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. Each class consists of 6,000 photos, with 50,000 in the training set and 10,000 in the test set.

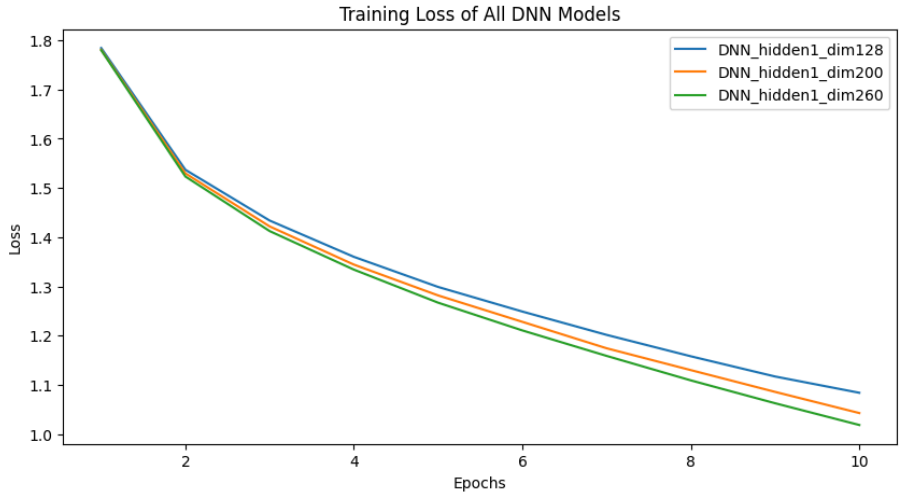
Results:

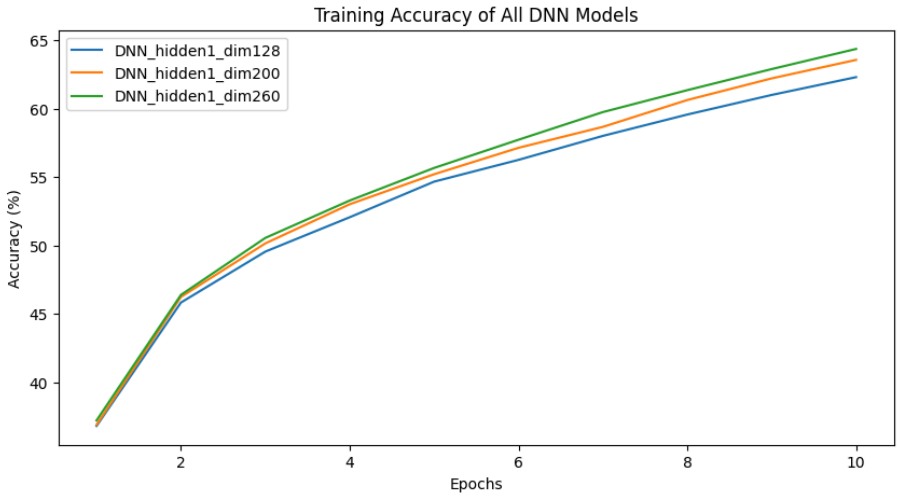
- CNN





- DNN





Results:

The CNN and DNN models with more hidden dimensions performed better in terms of loss and accuracy. This is because having more neurons enables the models to learn more intricate patterns. When comparing the two models, CNNs outperformed DNNs in terms of loss and accuracy. This is due to the CNN models' capacity to better collect picture properties such as edges and textures using convolutional layers created specifically for image input. In contrast, DNNs process each pixel independently, making it more difficult for them to learn spatial patterns and resulting in lesser performance than CNNs.

1-2: Optimization

Github files:

A. HW1_1-2_PCA.py

B & C: HW1_1-2_GradNorm.py

A. Visualize the Optimization Process

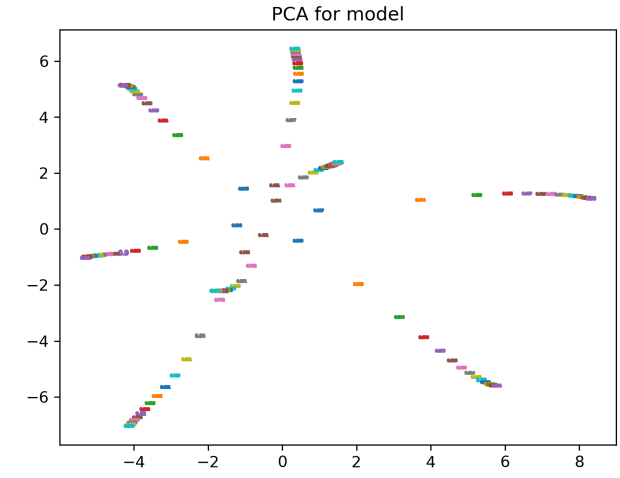
Task: MNIST

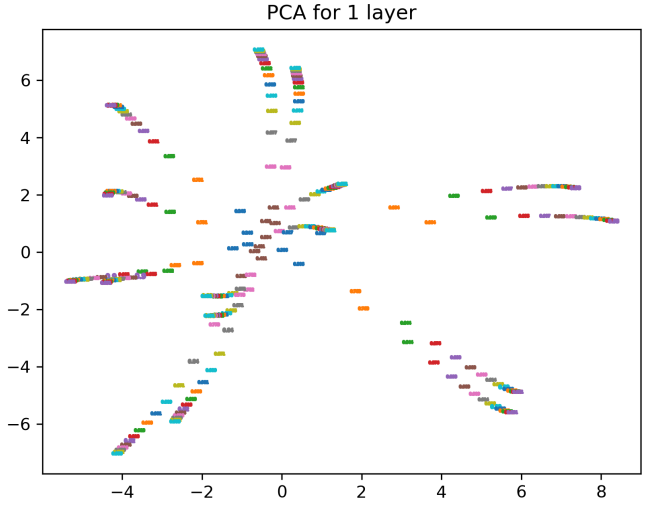
The MNIST dataset is a widely used benchmark for image classification tasks, consisting of 70,000 grayscale images of handwritten digits (0-9) in 28x28 pixel resolution. It includes 60,000 images for training and 10,000 for testing, with each image represented as a 1D vector of 784 pixels. Labels corresponding to the digits help train machine learning models, making MNIST a popular choice for testing algorithms, particularly in deep learning and computer vision. Its simplicity and versatility make it an ideal dataset for building and evaluating neural networks.

Model:

- DNN
  - Architecture:
    - Input layer (fc1): accepts a vector of size 784 (representing a flattened 28x28 image)
    - 1st linear layer (fc1): transforms the input into a 10-dimensional space.
    - 2nd linear layer (fc2): takes the output of the first layer and increases the dimensionality to 20.
    - 3rd linear layer (fc3): reduces the dimensionality back to 10, the number of output classes
  - Activation:
    - Between each layer, a ReLU (Rectified Linear Unit) activation function is applied, introducing non-linearity to the model. However, the output layer does not use an activation function, which suggests that this model is intended for a classification task where further processing (such as softmax) might be applied externally.
  - Hyperparameters:
    - Learning Rate: 0.0004
    - Weight Decay: $1^{e^{-4}}$
    - Optimizer: Adam
    - Loss Function: Cross Entropy Loss

The model was trained 8 times, with weights being recorded every 3 epochs for a maximum of 45 epochs. Dimensions of the weights were reduced to two via PCA. The resulting data was plotted as shown below:



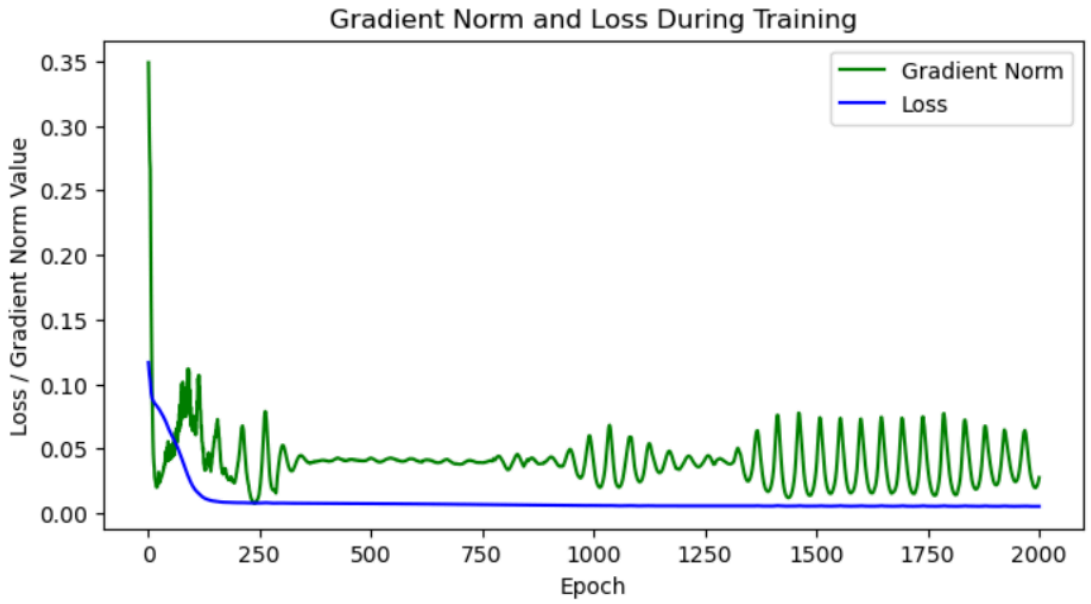


Results:

The PCA plots for both the entire model and a single layer reveal a radial, star-like distribution of data, with clusters extending in multiple directions. This pattern indicates that key features of the model's activations or weights are captured along the principal components, showing how variance is distributed in a lower-dimensional space. The top plot, representing the whole model, shows a slightly broader spread of points, suggesting a more diverse variance distribution, while the single-layer plot shows a similar structure, indicating that the layer captures a focused but significant portion of the model's overall variance.

B. Observe Gradient Norm During Training

Model: Model 3 from part 1-1 A.



Results:

The plot displays the gradient norm and loss during the model's training process over 2000 epochs. The loss, represented by the blue line, starts high and decreases rapidly within the first 100 epochs, indicating that the model is learning effectively and optimizing its parameters early in the training. After around 200 epochs, the loss stabilizes at a low value and remains constant for the remainder of the training, suggesting that the model has converged and is no longer making significant improvements in minimizing the error.

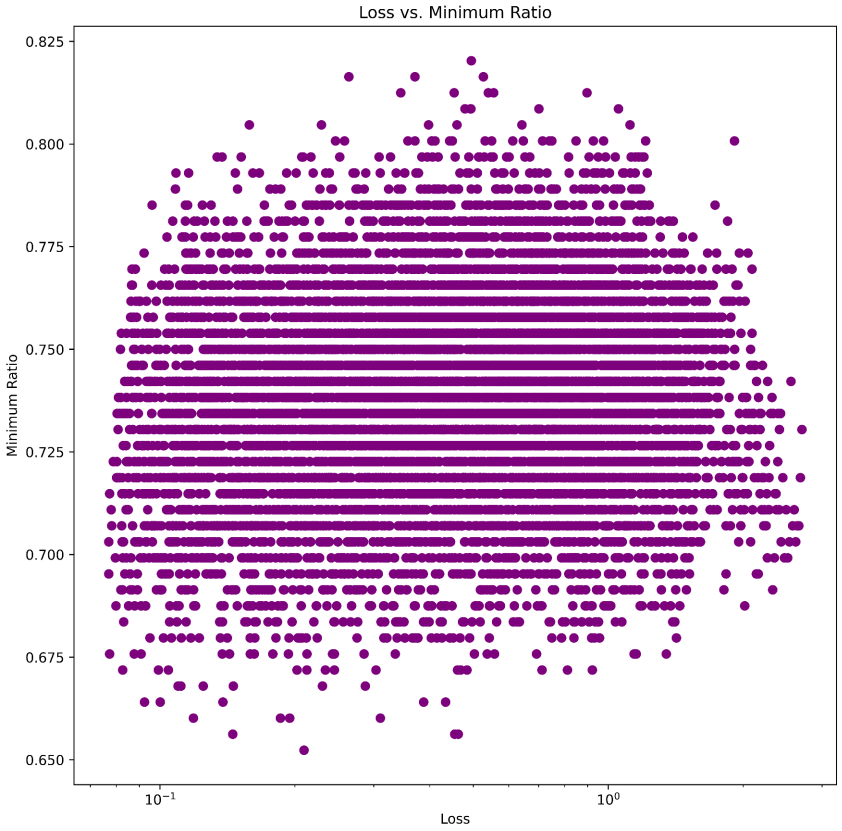
The gradient norm, represented by the green line, begins with large fluctuations in the first 200 epochs, reflecting significant updates to the model's weights as it learns. Following this, the gradient norm stabilizes but exhibits periodic oscillations between epochs 500 and 2000, indicating that while the loss remains stable, the model parameters are still being adjusted in smaller steps. These oscillations could be a result of the learning rate or the specific optimization algorithm used. Overall, the plot demonstrates successful early learning and convergence, though the gradient behavior suggests further fine-tuning continues throughout the later stages of training.

C. What Happened When Gradient is Almost Zero

Model: Model 3 from part 1-1 A.

To capture the weights when the gradient norm is zero, I recorded the model's parameters after each epoch and tracked the gradient norm throughout the training process. The weight configuration when the gradient norm approaches 0 can be obtained from the final set of weights at the end of training. This reflects the model's state at a local minimum, where no additional gradient-based optimization can improve performance.

The minimum ratio is a notion that refers to the proportion of positive eigenvalues in the model's Hessian matrix. The Hessian matrix describes the loss function's second-order curvature, and its eigenvalues provide information about that curvature. A larger fraction of positive eigenvalues indicates a more stable and convex region of the loss surface.



Results:

The creation of horizontal bands in the data demonstrates that the minimum ratio remains impressively constant throughout a wide range of loss values. The majority of the data points are centered in a small band of 0.7 to 0.775, implying that fluctuations in the loss have little impact on the minimum ratio. This consistency shows that the minimal ratio is most likely governed by internal model dynamics, such as gradient norms or weight modifications, which are unaffected by changes in loss.

This stability emphasizes the minimal ratio as a reliable optimization metric. Its invariance across different loss levels indicates that the training process is well-regulated, limiting substantial fluctuations within internal parameters despite loss variations. While the loss metric focuses on the model's prediction errors, the minimum ratio may represent a more stable component of the model's internal optimization dynamics, allowing for smoother, more controlled training. This could lead to more consistent performance and reduced overfitting, making the minimum ratio a useful measure of model regularization throughout training.

1-3: Generalization

Github files:

A. HW1_1-3_RandLabel.py
B. HW1_1-3_NumParams.py
C-a. HW1_1-3_FvG_train.py
C-b. HW1_1-3_FvG_sens.py

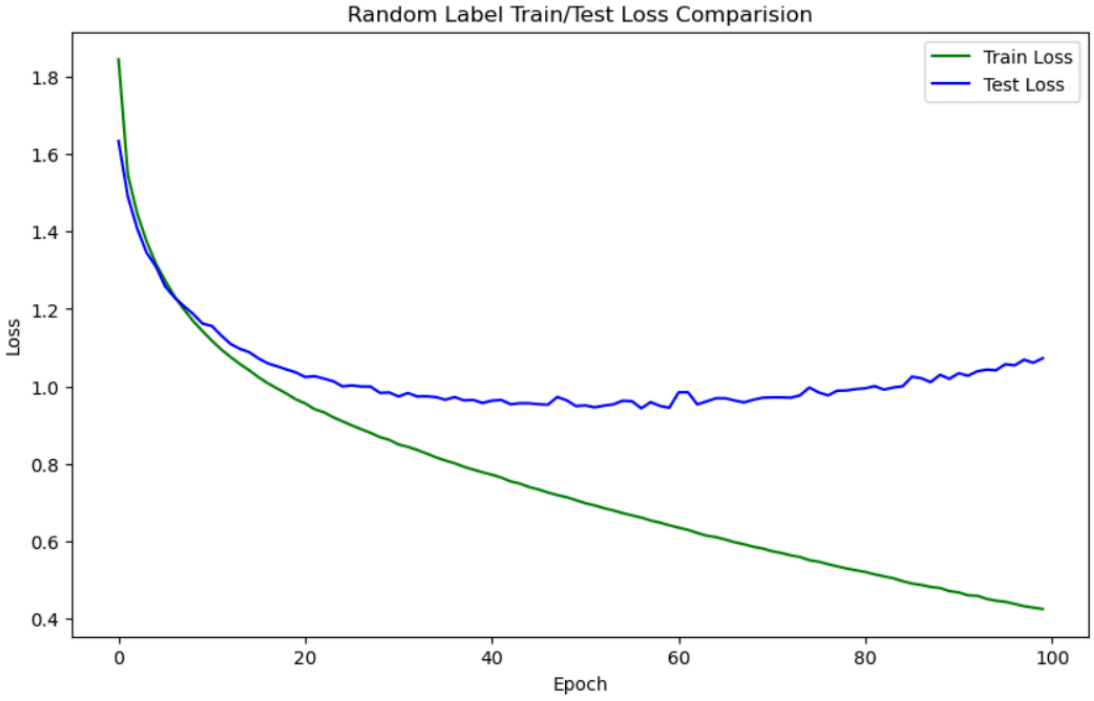
A. Can a network fit random labels?

Task: CIFAR-10

Model: CNN

- Architecture:
    - Input Layer: Input size is 32×32×3 (3 color channels for the RGB image).
    - Convolutional Layers:
        - Convolutional Layer 1 (conv1):
            - Input: 3 channels (RGB image).
            - Output: 16 feature maps generated by applying 16 filters of size 3x3
            - Activation: ReLU followed by a 2x2 max-pooling operation to reduce the spatial dimensions.
        - Convolutional Layer 2 (conv2):
            - Input: 16 feature maps from conv1.3x3

            - Output: 32 feature maps generated by applying 32 filters of size
            - Activation: ReLU followed by a 2x2 max-pooling operation.
    - Fully Connected Layers:
        - Fully Connected Layer 1 (fc1):
            - Input: Flattened feature maps (32 feature maps of size 6×6, resulting in 1152 inputs).
            - Output: 128 neurons.
            - Activation: ReLU applied to introduce non-linearity.
        - Fully Connected Layer 2 (fc2):
            - Input: 128 neurons from fc1.
            - Output: 10 neurons, representing the 10 output classes for classification.
    - Hyperparameters:
        - Learning rate: 0.0001
        - Optimizer: Adam
        - Loss function: Cross-Entropy Loss
        - Number of epochs: 100

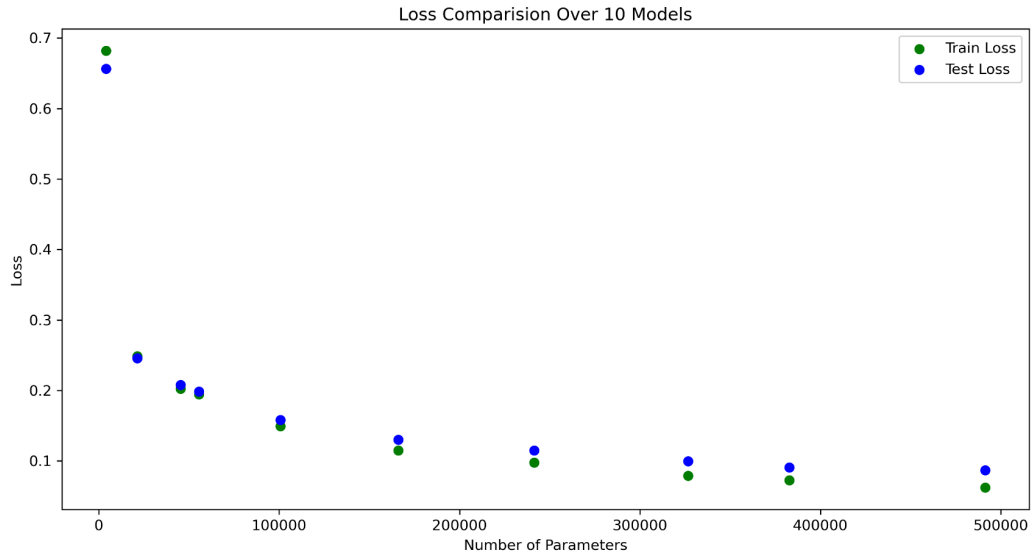


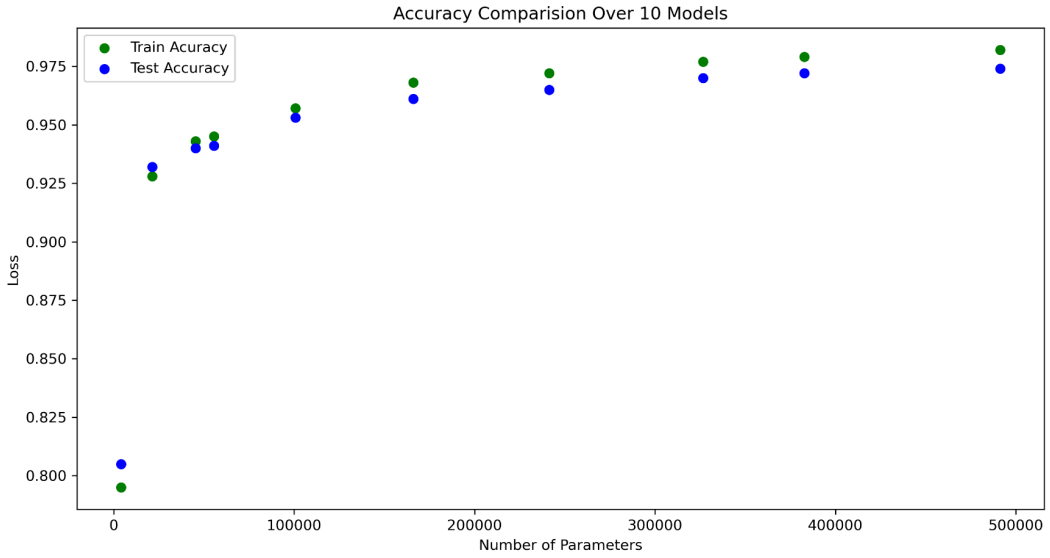
B. Number of Parameters vs. Generalization

Models:

All models follow a DNN architecture with:

- Input Layer: All models have the same input size (784), which indicates that they are designed for inputs like flattened 28x28 images (such as those from the MNIST dataset).
- Fully Connected (Linear) Layers: Each model consists of three fully connected (Linear) layers, though the number of units in the hidden layers differs.
- Activation Function: All models use the ReLU activation function (F.relu()) after the first two layers, promoting non-linearity between layers.
- Output Layer: The final output layer in each model has 10 units, which is typical for classification tasks with 10 classes (e.g., MNIST).
- Hyperparameters:
    - Learning rate: 0.0001
    - Optimizer: Adam
    - Activation Function: ReLU
    - Number of epochs: 10
- Number of parameters:
    - Model 1 - 45360
    - Model 2 - 4079
    - Model 3 - 21435
    - Model 4 - 55630
    - Model 5 - 382770
    - Model 6 - 166060
    - Model 7 - 241410
    - Model 8 - 100710
    - Model 9 - 491360
    - Model 10 - 326760





Results:

As the number of parameters increased, both training and testing loss reduced dramatically, indicating that larger models fit the data more accurately. The sharp drop in loss between low-parameter and mid-range parameter models implies that smaller models may suffer from underfitting because they lack the ability to grasp the underlying patterns in the data. Beyond a certain threshold, roughly 300,000 parameters, the reduction in loss becomes modest, indicating diminishing returns as model complexity increases. Notably, the difference between training and testing loss is

minor across all models, indicating that the models do not overfit despite growing complexity.

Similarly, the accuracy comparison showed that models with more parameters perform better, with training and testing accuracy seeming to increase dramatically for models with fewer than 100,000 parameters. Accuracy plateaus for models with over 300,000 parameters, reaching near-optimal performance of ~98% in both training and testing. The little difference in training and testing accuracy shows that the models generalize effectively to new data, preventing overfitting. Overall, increasing model complexity up to a certain point results in large performance advantages, but adding parameters beyond 300,000 yields only moderate improvements. This balance of increasing accuracy while keeping low loss without overfitting emphasizes the importance of determining the optimal model complexity.

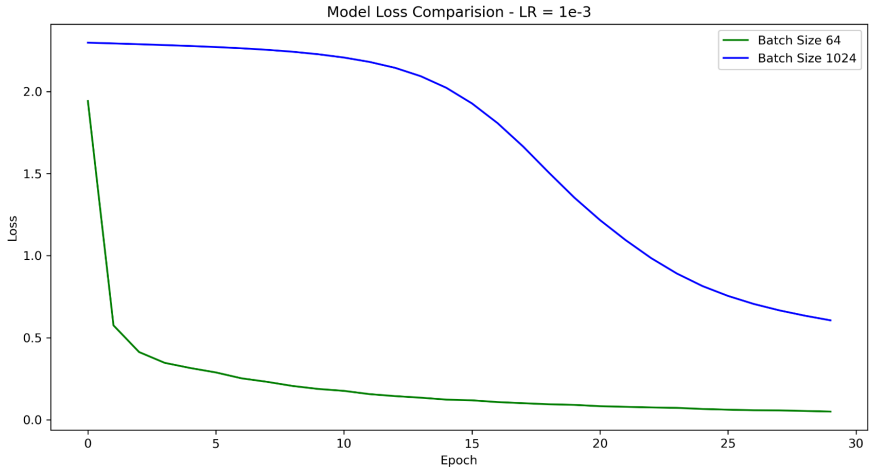
C. Flatness vs. Generalization

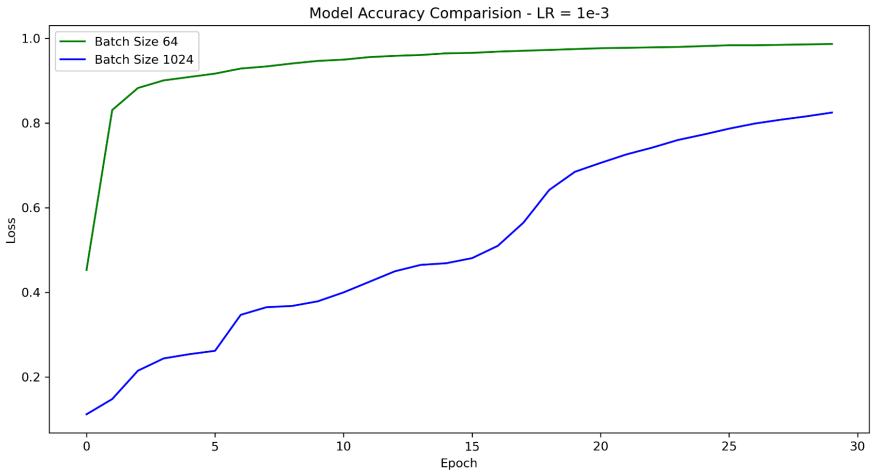
Task: MNIST

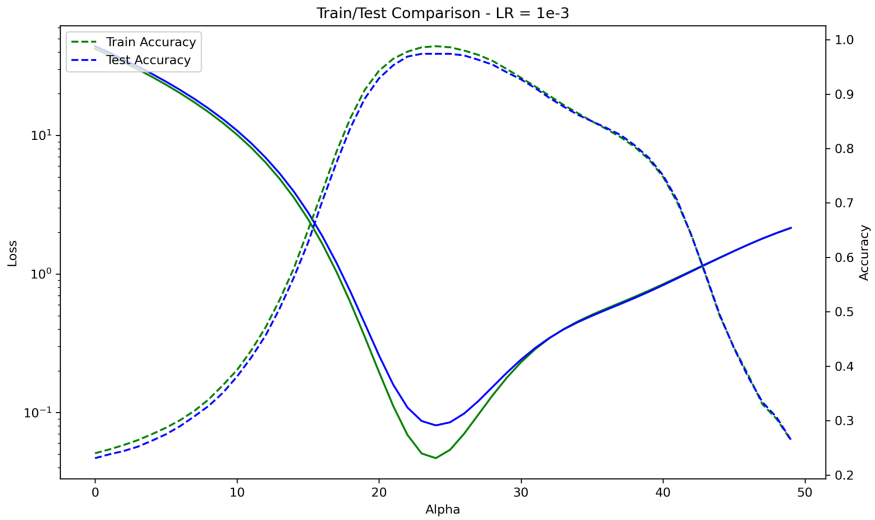
Models: DNN

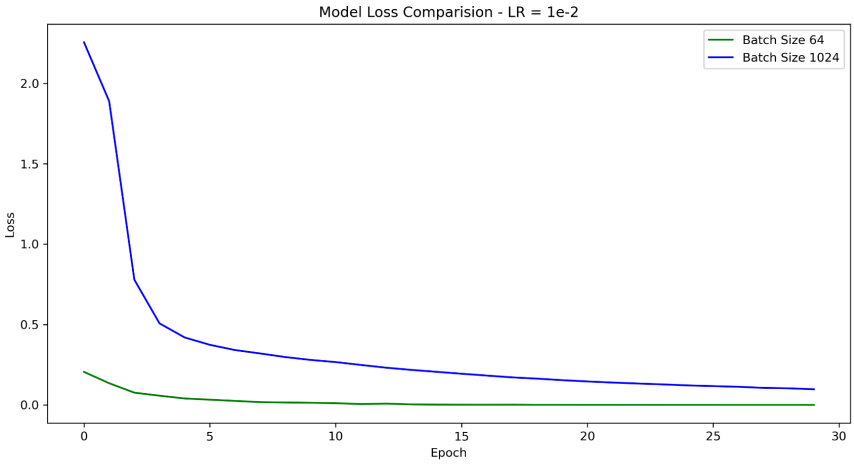
- Architecture
  - Input Layer: images of size 28x28
  - First Fully Connected Layer (fc1):
    - Input: 784 neurons
    - Output: 500 neurons
  - Second Fully Connected Layer (fc2):
    - Input: 500 neurons
    - Output: 100 neurons
  - Third Fully Connected Layer (fc3):
    - Input: 100 neurons
    - Output: 50 neurons
  - Output layer: 10 possible classes
- All layers used ReLU as the activation function
- Hyperparameters:
  - Learning rate: (two tested)
    - 1e-3
    - 1e-2
  - Optimizer: Adam
  - Loss function: Cross-Entropy Loss
  - Number of epochs: 30
  - Batch size:
    - M1 + M1_2 = 64
    - M2 + M2_2 = 1024

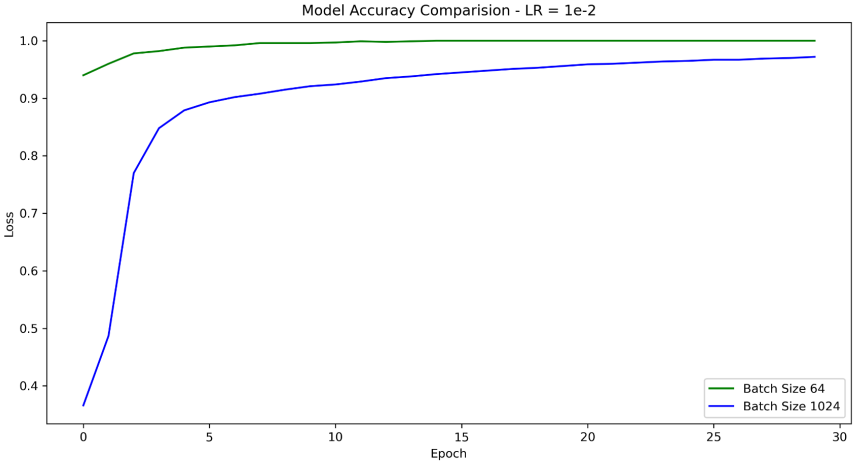
Part 1 - Interpolation

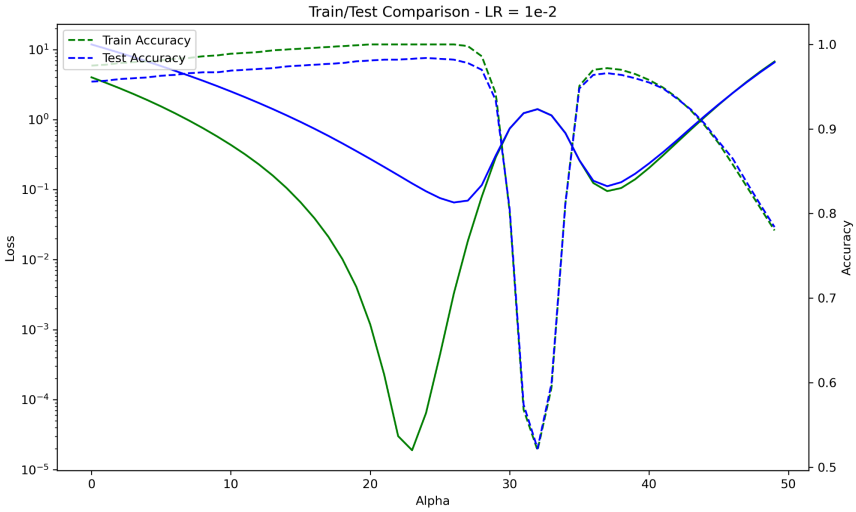







Model Loss Comparision - LR = 1e-2
Batch Size 64
Batch Size 1024
Loss
Epoch


Model Accuracy Comparision - LR = 1e-2
Batch Size 64
Batch Size 1024
Loss
Epoch

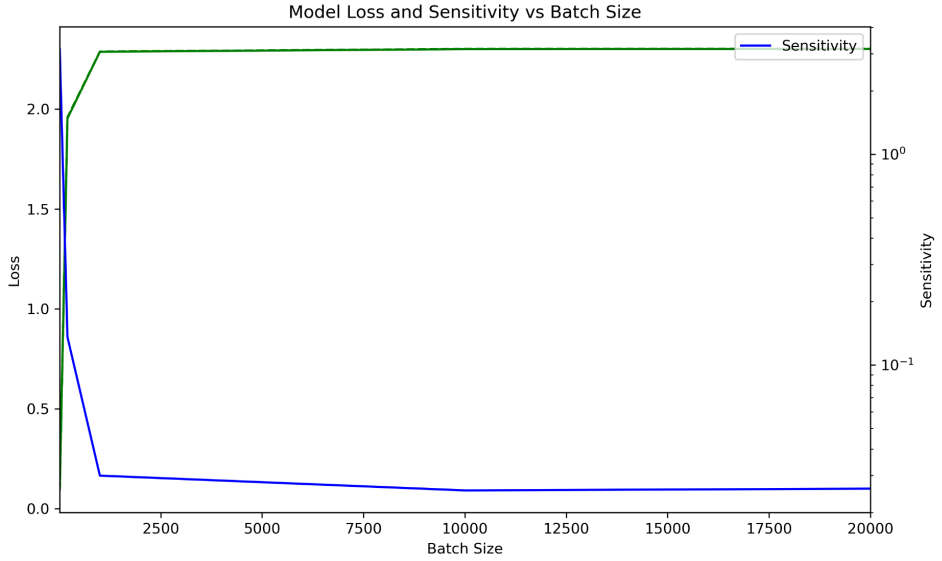

Train/Test Comparison - LR = 1e-2
Train Accuracy
Test Accuracy
Loss
Accuracy
Alpha

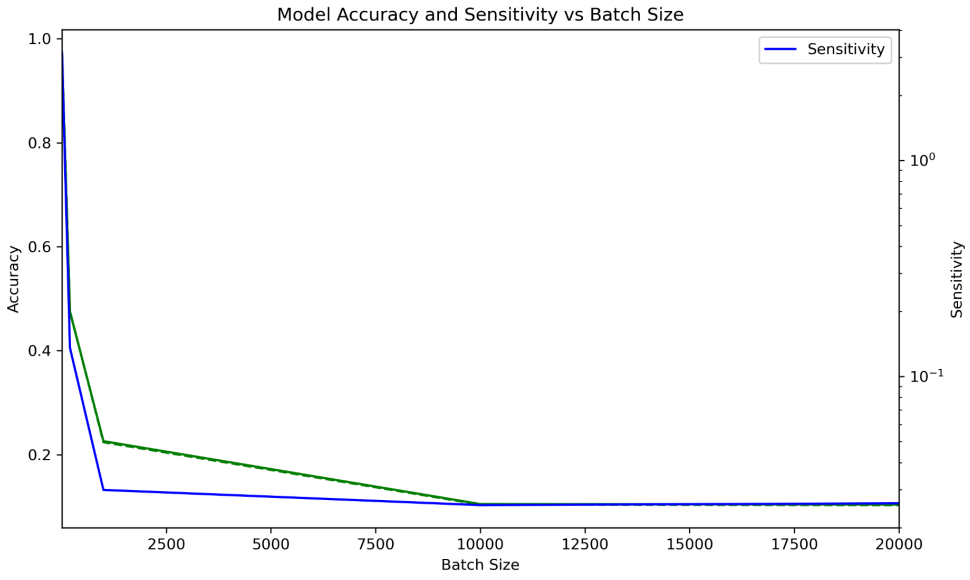
Results:

The training loss for both batch sizes decreased as the interpolation ratio increased. However, the batch size of 64 resulted in a smoother and more consistent drop in loss than the batch size of 1024, which saw swings, particularly around alpha = 25 and 35. Training accuracy continuously improved as alpha increased, with the lower quantity of batches model obtaining higher accuracy faster than the bigger batch model. The testing loss fluctuated dramatically with the bigger batch size (1024), with spikes around alpha = 25 and 35. The test accuracy also followed these oscillations, demonstrating that generalization is unstable. The lower batch size (64), on the other hand, followed a more constant path, resulting in smoother test accuracy increases. Overall, the results reveal that, in this case, reduced batch sizes resulted in more consistent and predictable performance, particularly in terms of testing accuracy and loss. The increased batch size caused instability and oscillations, limiting the model's capacity to generalize successfully.

Both models performed better throughout training when their learning rate was lower at 1e-2 vs at 1e-3. The smaller batch size showed more consistent performance, with a continuous decrease in loss and an increase in accuracy over the interpolation range. The larger batch size model showed a slower decrease in loss and more variations in accuracy, although the fluctuations were less apparent than in the prior experiment with a higher learning rate. Similar to the training loss, the testing loss and accuracy for the smaller batch size model were more stable, with a clear improvement in performance as the interpolation ratio increased. The larger batch size model showed instability, particularly around alpha = 30, where a sharp drop in accuracy and an increase in loss were observed.

Part 2 - Sensitivity





Results:

This experiment used the same model from part one with a learning rate of 1e-4 to test the model's sensitivity to different batch sizes. The batch sizes evaluated included 10, 200, 1000, 10,000, and 20,000. The results show that as batch size increased, the model's loss values increased, but accuracy decreased. Smaller batch sizes, notably 10 and 200, resulted in reduced training and testing losses and higher accuracy, indicating superior generalization performance than larger batch sizes. The model showed significant sensitivity at smaller batch sizes, but performance stabilized as batch sizes approached 10,000 or greater. This pattern suggests that, while bigger batch sizes improve computing efficiency, they can limit the model's ability to generalize, as seen by increased loss and reduced accuracy. Overall, smaller batch sizes enable more exact gradient updates, which improves learning dynamics and generalization.