CPSC 8430 : Deep Learning

Homework 1 Report

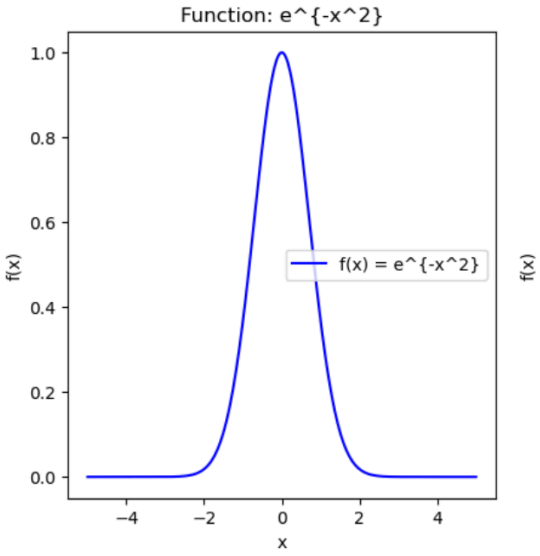
Submitted by: Shwetha Sivakumar
CUID: C20140199

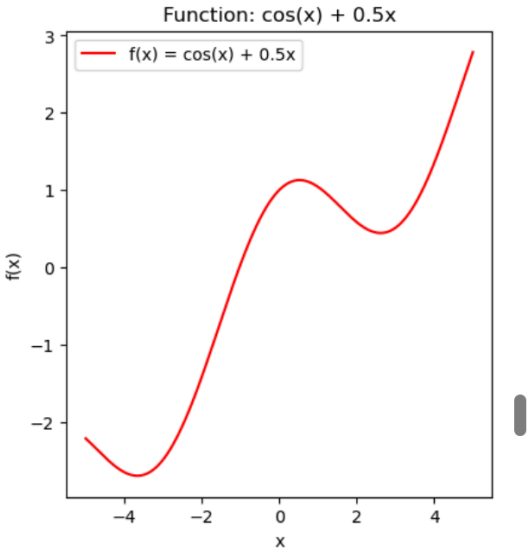
HW 1-1
Deep Vs Shallow:
1.1 Stimulate a Function

This section shows the evaluation of various deep neural network (DNN) models trained on both simulated functions and the MNIST dataset. The primary objective is to compare shallow and deep models on their ability to approximate mathematical functions and perform well on a real-world dataset.

The two non-linear single-input, single-output functions chosen are :

1. f(x)=e–x^2
2. f(x)=cos(x)+0.5x





The 3 models evaluated include ShallowCNN, MediumCNN, and DeepCNN. The models were compared based on their performance in approximating these functions. The following configurations were used:

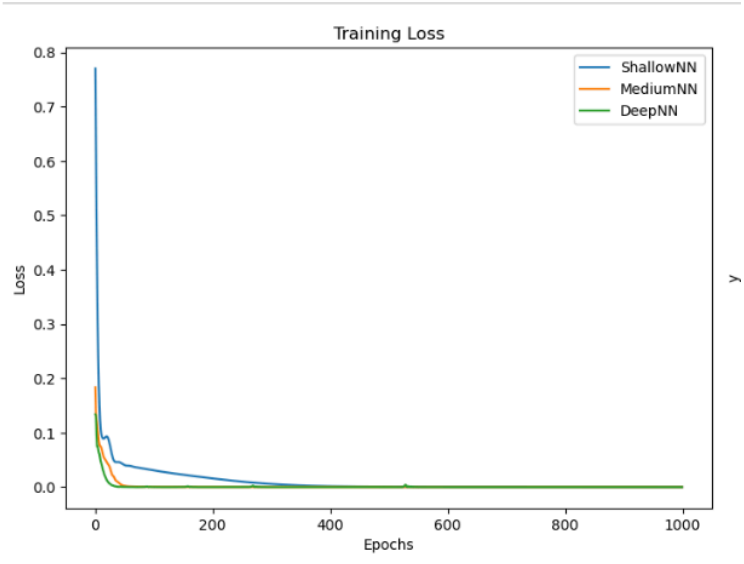
Models and Architectures

ShallowNN: The ShallowNN model for simulating the functions consists of two fully connected layers. The first layer transforms 1 input feature into 10 hidden features, and the second layer maps these 10 features to a single output feature. This architecture results in 21 parameters (10 weights and 10 biases in the first layer, 10 weights and 1 bias in the second layer). The ReLU activation function introduces non-linearity. The model is trained using Mean Squared Error (MSELoss) with the Adam optimizer (learning rate of 0.01).

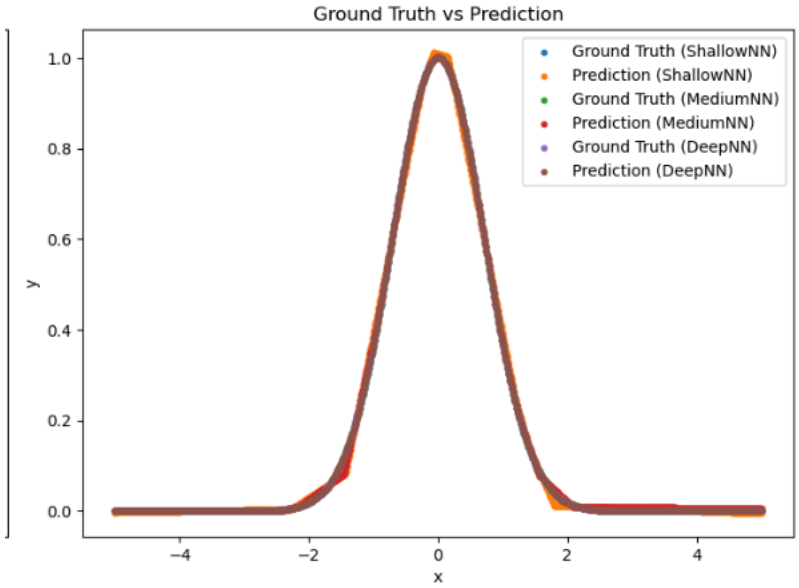
MediumNN: The MediumNN model for this function has three fully connected layers. It maps 1 input feature to 20 hidden features in the first layer, 20 features to 10 in the second layer, and 10 features to 1 output feature in the third layer. This setup results in 221 parameters (20 weights and 20 biases in the first layer, 200 weights and 10 biases in the second layer, 10 weights and 1 bias in the third layer). The ReLU activation function is used, and the model is evaluated using MSELoss and optimized with Adam (learning rate of 0.01).

DeepNN: The DeepNN model features four fully connected layers. It has 1 input feature transformed to 40 hidden features in the first layer, 40 to 30 in the second, 30 to 20 in the third, and 20 to a single output feature in the fourth layer. This model has 1,561 parameters (40 weights and 40 biases in the first layer, 1,200 weights and 30 biases in the second layer, 600 weights and 20 biases in the third layer, 20 weights and 1 bias in the final layer). It uses ReLU activation, MSELoss for evaluation, and is trained with the Adam optimizer (learning rate of 0.01).

Training loss and Ground Truth Vs Prediction for f(x)=e−x^2:

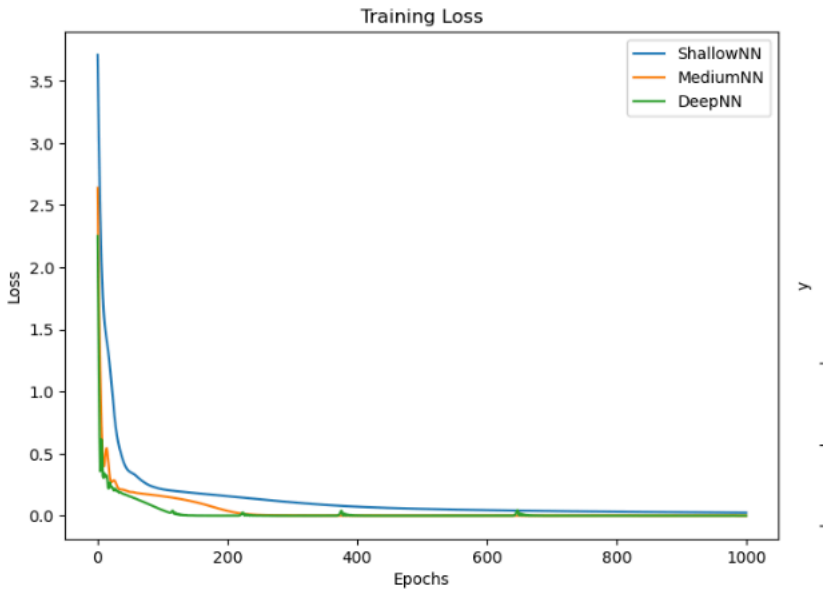


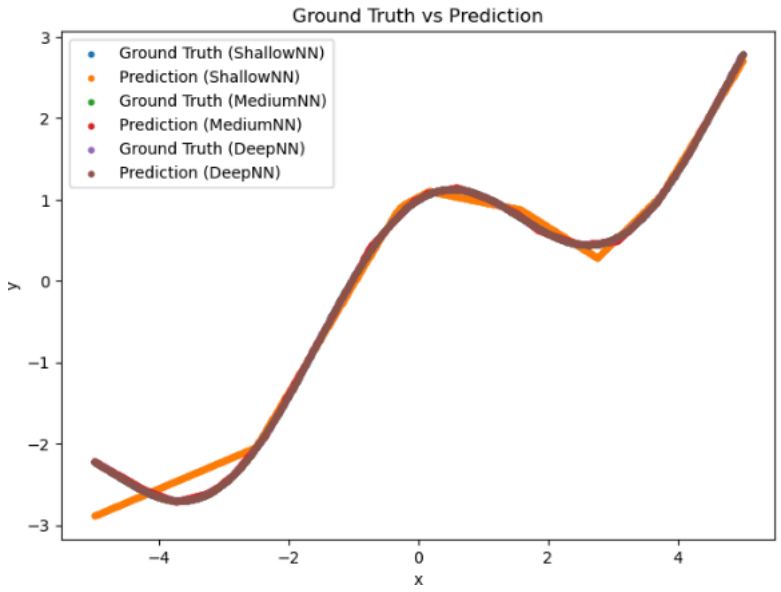
3.



The convergence analysis reveals that **Model 1** achieved convergence at epoch 1500 with a loss of 0.00088, making it the fastest to converge among the models. **Model 2** followed, reaching convergence at epoch 2500 with a slightly higher loss of 0.0009, while **Model 3** took the longest, converging at epoch 20000 with a loss of 0.0022. This trend suggests that a model with fewer layers can converge more quickly and potentially achieve a lower loss, as seen with **Model 1**. Conversely, **Model 3**, despite its increased complexity, faced longer training times and higher final loss, which may indicate issues such as overfitting or inefficiencies in the training process. Overall, the results suggest that while increasing the number of layers can enhance model capacity, it does not necessarily lead to faster or better convergence. The optimal number of layers depends on various factors, including the nature of the problem, data characteristics, computational resources, and strategies to mitigate challenges like vanishing/exploding gradients and overfitting.

Training loss and Ground Truth Vs Prediction for f(x)=cos(x)+0.5x:





Model 1 demonstrated a consistent and rapid decrease in loss throughout the epochs, starting from 0.2234 at epoch 100 and reaching a final loss of 0.0043 at epoch 1000. This steady reduction indicates effective learning and convergence within the given training period.

Model 2 showed an initial high loss of 0.7410 at epoch 100 but improved significantly, converging to a final loss of 0.0041 by epoch 1000. The loss trajectory suggests a slow start followed by a steady decline, with the final loss comparable to **Model 1**. This model also shows that even with a challenging start, consistent training can lead to convergence similar to that of simpler models.

Model 3 exhibited a highly efficient training process, starting from a loss of 0.0429 at epoch 100 and rapidly decreasing to a final loss of 0.0001 by epoch 1000. This model converged quickly and achieved a very low final loss, demonstrating high efficiency in learning and generalization.

Overall, all three models reached similar final losses, but their convergence rates varied significantly. **Model 3** had the most rapid improvement, suggesting it was highly effective in learning from the data. **Model 1** showed a steady learning curve, while **Model 2** needed more epochs to achieve a similar final performance. These observations underscore the importance of evaluating both the convergence rate and the final loss to assess model performance effectively.

1.2 Training on Actual Tasks:

MNIST dataset was used with 3 CNN models with varying architecture complexity. The 3 models are:

SimpleCNN

The SimpleCNN model is the most basic of the three. It consists of a single convolutional layer that applies 16 filters of size 3x3 to the input image, with padding to maintain the spatial dimensions. This layer is followed by a ReLU activation function and a max pooling operation to reduce the dimensionality of the feature maps. The output from the convolutional layer is then

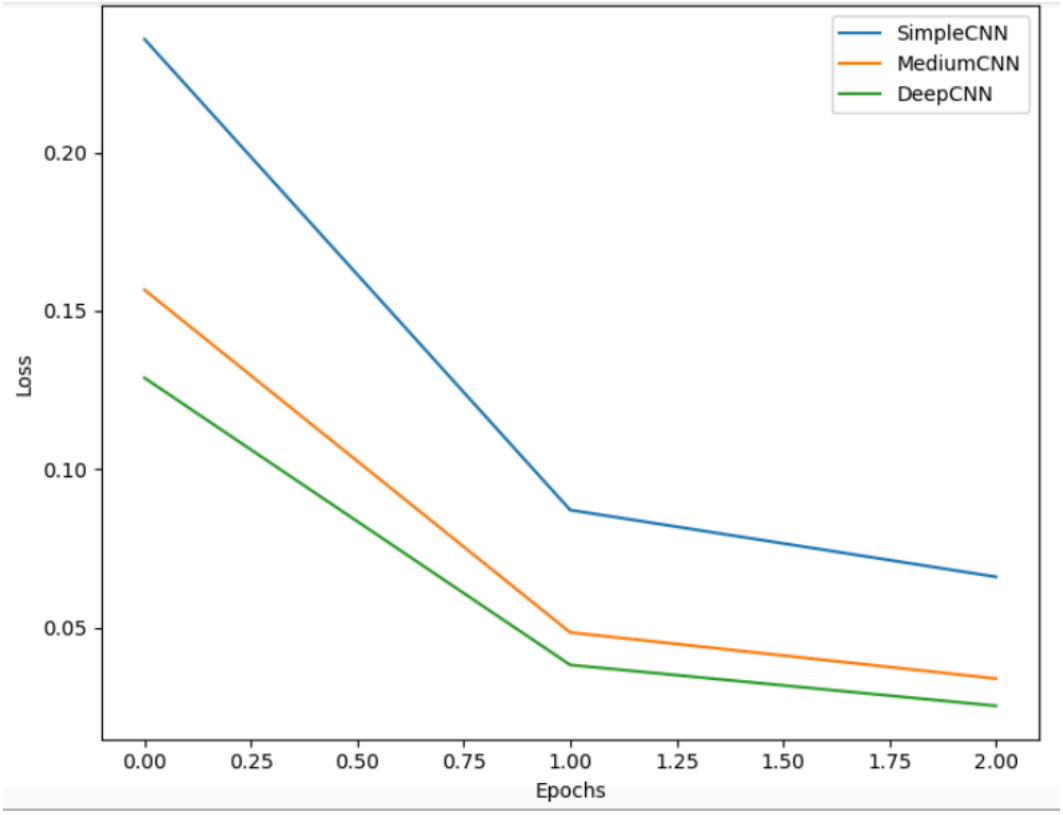
flattened into a one-dimensional vector and passed through a fully connected layer with 10 output units, corresponding to the 10 classes of the MNIST dataset. This straightforward architecture is designed to capture fundamental features from the images and is suitable for basic image classification tasks.

MediumCNN

The MediumCNN model introduces more complexity and capacity compared to the SimpleCNN. It starts with a convolutional layer that applies 32 filters of size 3x3, followed by another convolutional layer with 64 filters of the same size. Each convolutional layer is followed by ReLU activations and max pooling operations to progressively reduce the spatial dimensions of the feature maps while increasing the number of filters. After the feature extraction, the output is flattened and passed through a fully connected layer with 10 units. This model can capture more intricate patterns and features due to its deeper and more complex architecture.

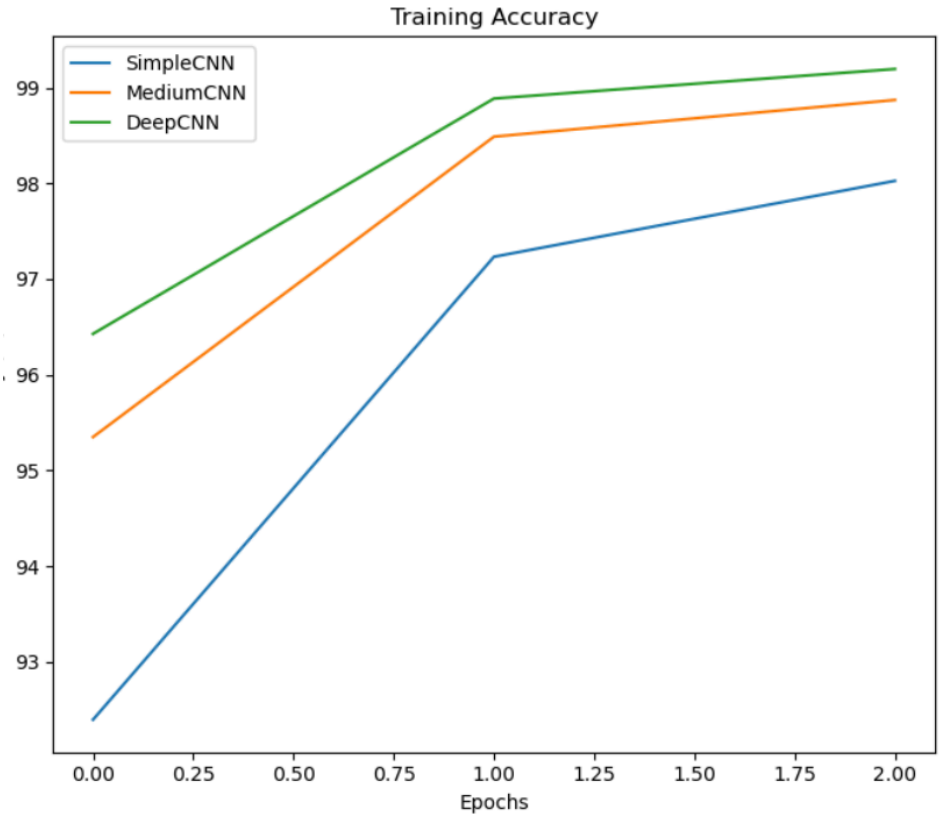
DeepCNN

The DeepCNN model is the most advanced among the three. It features three convolutional layers with increasing numbers of filters: 64, 128, and 256, respectively. Each convolutional layer is followed by a ReLU activation and max pooling to downsample the feature maps. The depth of this model allows it to learn very detailed and abstract features from the input images. After the final convolutional layer, the output is flattened and fed into a fully connected layer with 10 output units. This architecture is designed to handle more complex patterns and is expected to perform better on tasks requiring detailed feature extraction, albeit with increased computational cost.



A s

expected, increasing model complexity from SimpleCNN to DeepCNN leads to better performance on the MNIST dataset. DeepCNN performs the best, achieving the lowest loss, which is indicative of its superior feature extraction and learning capabilities. DeepCNN demonstrates the best generalization ability, which is consistent with the expectation that deeper models can capture more complex features.The MediumCNN and DeepCNN models show more stable loss values compared to SimpleCNN, reflecting their better capacity to generalize from the training data. The additional layers and parameters in MediumCNN and DeepCNN help in achieving lower and more stable losses.



All models perform well, but the DeepCNN has the highest accuracy, followed by MediumCNN and SimpleCNN. Each model improves over the epochs, with DeepCNN achieving the best final results. he trend suggests that increasing model complexity (from SimpleCNN to DeepCNN) improves accuracy, as deeper models tend to capture more features and patterns from the data.

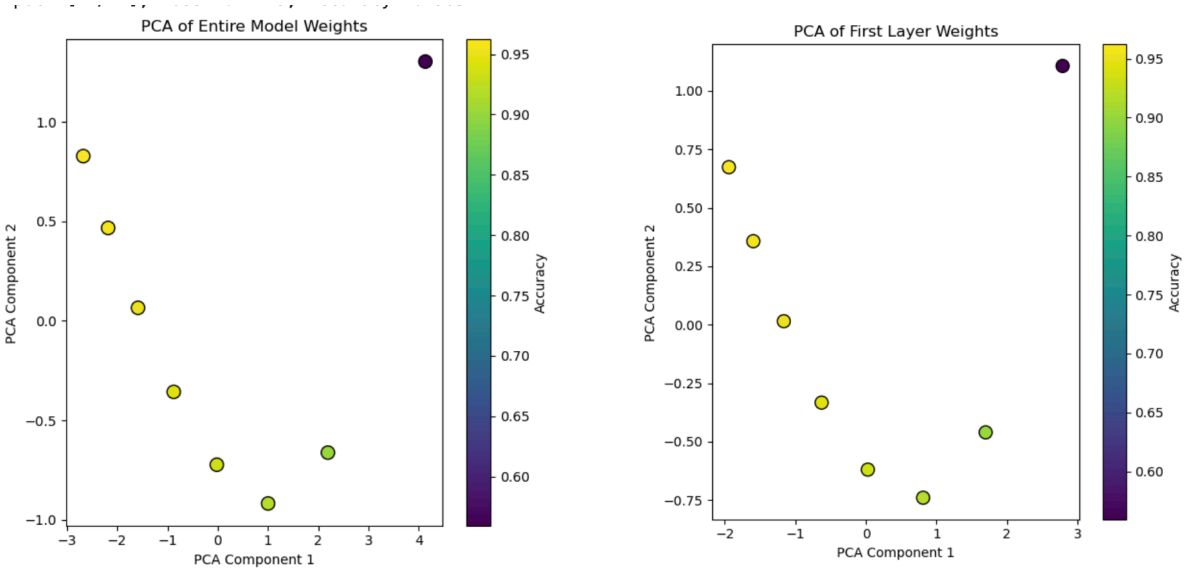
Github Link: https://github.com/shwethasivakumar/Deep-Learning/blob/main/HW1-1.ipynb

HW 1-2
Optimization
2.1 Visualize the Optimization Process

In the experiment, a SimpleNN model was trained on the MNIST dataset for 24 epochs, with model parameters recorded every 3 epochs, totaling 8 collections. The training utilized Stochastic Gradient Descent (SGD) with a learning rate of 0.01. Principal Component Analysis (PCA) was used to reduce the dimensionality of the weights to two dimensions. The model consists of three fully connected layers: fc1 (128 units), fc2 (64 units), and fc3 (10 units), with ReLU activation functions and CrossEntropyLoss as the loss function. The training involved 8 separate runs, with weights collected and plotted to analyze their evolution and impact on accuracy.

PCA of Entire Model Weights: The figure below shows the PCA of weights for the entire model across epochs. The reduction to 2 dimensions helps visualize how the weights evolve during training.

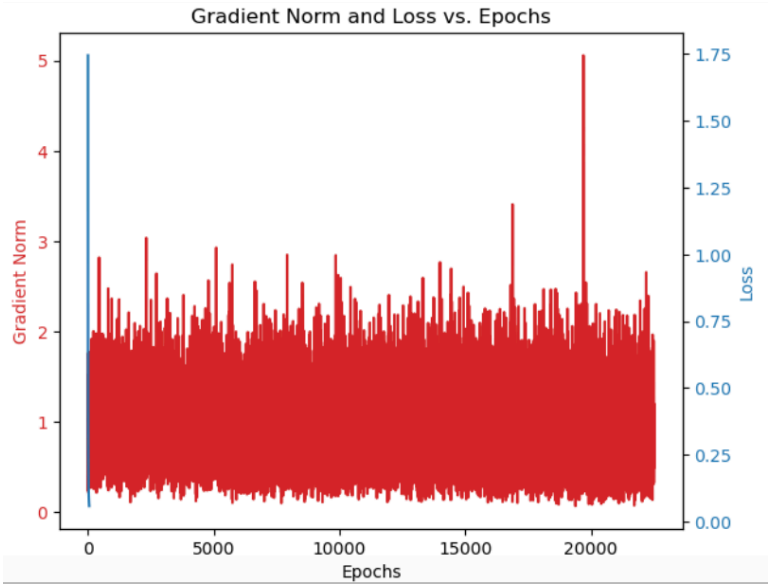
PCA of First Layer Weights: The PCA plot for weights of the first layer reveals changes in the weight distribution of the initial layer, giving insight into how early layers adapt during training.

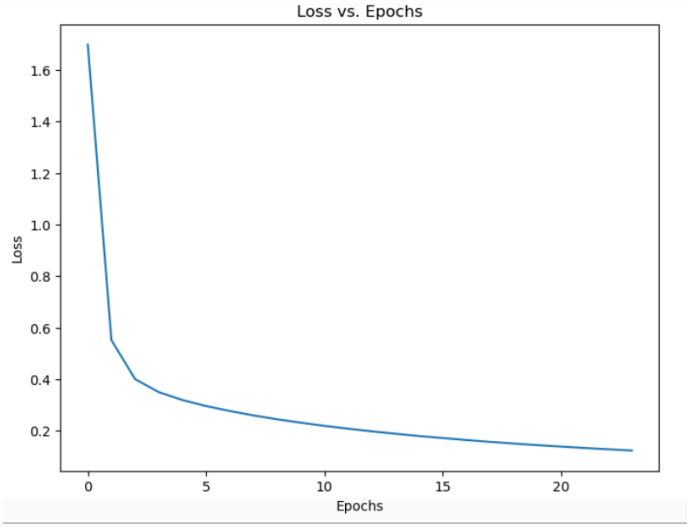


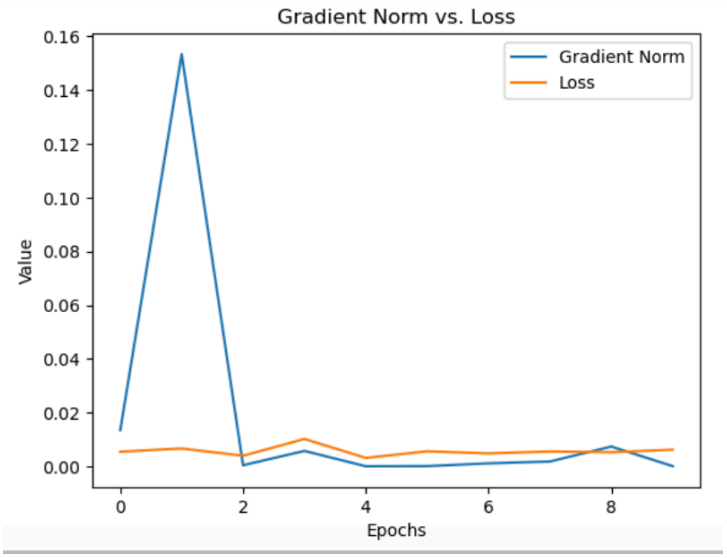
The deep neural network (DNN) was trained for 24 epochs, with weights collected every 3 epochs for a total of 8 collections. The weights, initially high-dimensional, were reduced to two dimensions using Principal Component Analysis (PCA). Loss and accuracy metrics showed significant improvement: from a loss of 1.8025 and accuracy of 48.55% at epoch 1, the model achieved a final loss of 0.1245 and accuracy of 96.46% by epoch 24. PCA visualization helped track how weights evolved over time, providing insights into the model's learning patterns and effectiveness. This analysis is essential for optimizing the training process and enhancing model performance.

2.2 Observe Gradient Norm During Training

The model used for the experiment is a SimpleNN, a neural network with three fully connected layers. The training was conducted using Stochastic Gradient Descent (SGD) with a learning rate of 0.01 and CrossEntropyLoss as the loss function. The model was trained for a total of 24 epochs.





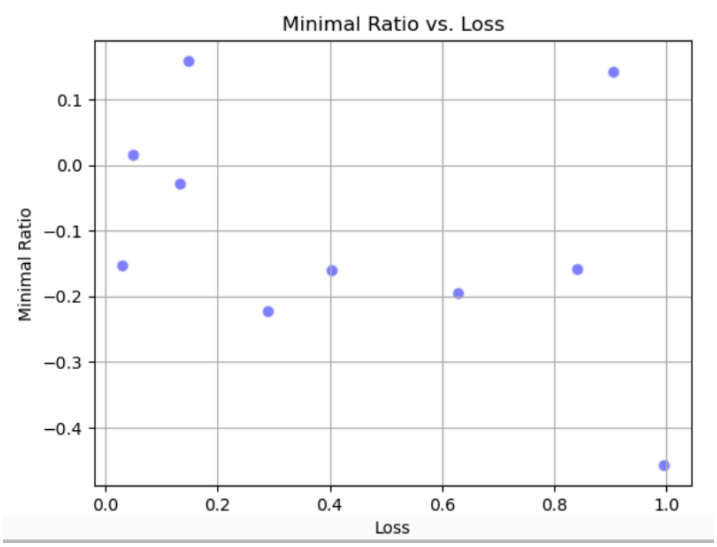


The training loss decreases consistently, reflecting effective learning. The gradient norm, which measures the magnitude of gradients, starts high and decreases over time, typical of a model approaching convergence. The gradual decline in gradient norm indicates improved optimization and better model fit. A sudden drop in the gradient norm might signal convergence to a local minimum or a plateau where additional training yields minimal gains.

2.3 What Happens When the Gradient is Almost Zero?

The model, SimpleNN, was trained using Stochastic Gradient Descent (SGD) with a learning rate of 0.01 and optimized with the CrossEntropyLoss function. The training process spanned 100 epochs, allowing for comprehensive evaluation and tuning of the model's performance over an extended period. To find weights with a zero gradient norm, compute the gradient norms of model parameters during training and identify those with a norm of zero, indicating no gradient change. The minimal ratio is calculated as the ratio of the smallest to the largest eigenvalue of the Hessian matrix of the loss function. A low minimal ratio suggests a highly elongated loss surface, while a ratio close to one indicates a more uniform curvature.

Minimal Ratio vs. Loss: The plot shows the relationship between the minimal ratio (indicating curvature) and the loss. When the gradient norm approaches zero, the minimal ratio also tends to increase, reflecting that the model is likely near a local minimum.



Comments: When the gradient norm approaches zero, it suggests that the model has reached a point where it can no longer learn effectively, as it is in a local minimum or saddle point. The minimal ratio provides additional insight into the curvature of the loss surface, showing that small gradients often coincide with flatter regions in the error landscape.

Github Link: https://github.com/shwethasivakumar/Deep-Learning/blob/main/HW1-2.ipynb

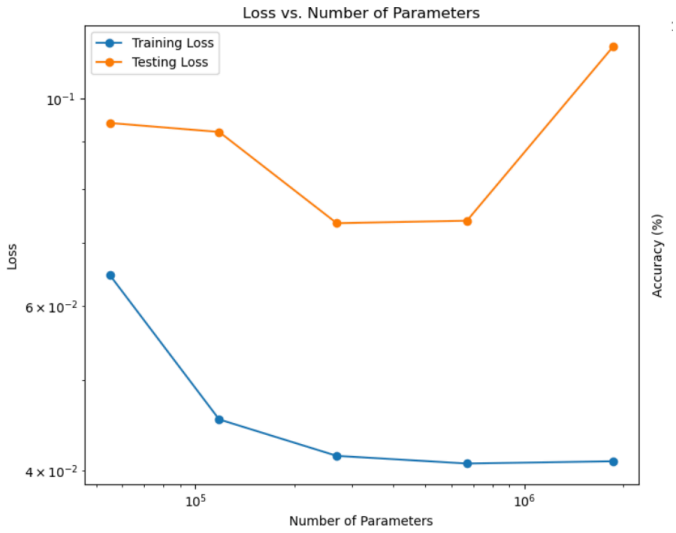
HW 1-3
Generalization
3.1 Can Network Fit Random Labels?

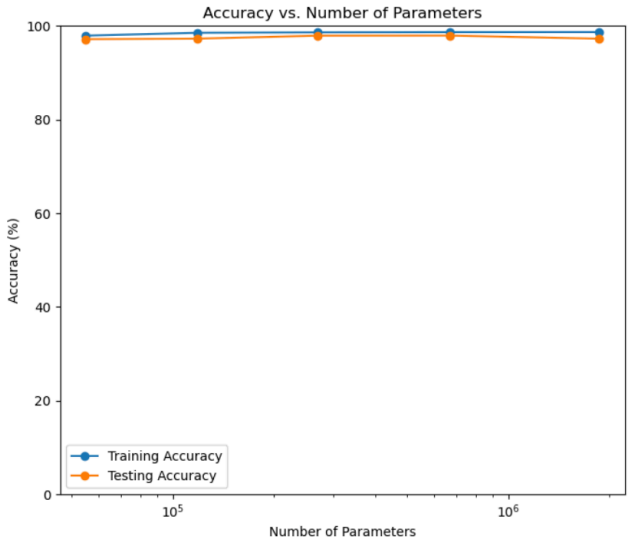
In this task, we will train a neural network with a single hidden layer and approximately 397,510 parameters on the MNIST dataset with randomized labels. Using CrossEntropyLoss and the Adam optimizer with a learning rate of 0.0001, we'll evaluate how the model's performance evolves over epochs. A plot will be generated to show training and testing loss versus epochs, illustrating how the model performs when learning from data with meaningless, randomized labels.

As the labels are randomized, the model fails to learn meaningful patterns from the data. The training and test losses are high and do not improve significantly, reflecting the model's inability to generalize from random noise.

3.2 Number of Parameters vs. Generalization

The task involves evaluating the impact of varying the number of parameters on the performance of 10 CNN models, ranging from 39,760 to 1,590,010 parameters. Each model features two convolutional layers with a 4x4 kernel size, ReLU activation, and uses CrossEntropyLoss for training. The Adam optimizer with a learning rate of 0.0001 is employed, alongside a dropout rate of 0.25 for regularization.



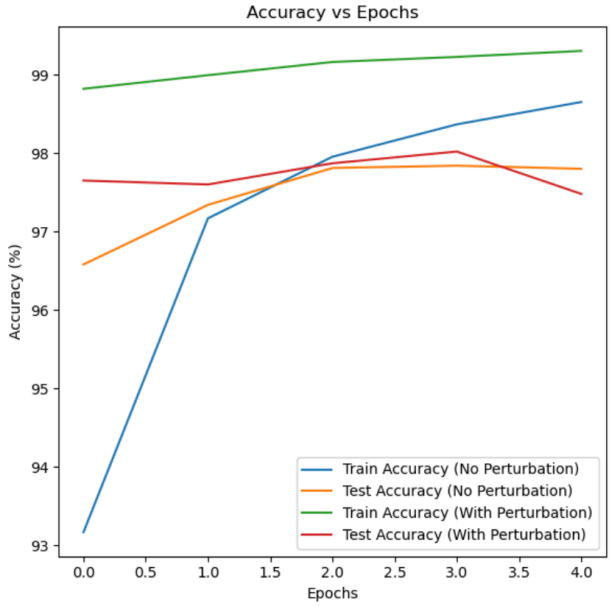


As the number of parameters increases, the loss decreases, and accuracy improves. However, larger models may overfit, showing higher accuracy on training data but not necessarily on testing data.

3.3 Flatness vs. Generalization

Part 1: Batch Size Impact

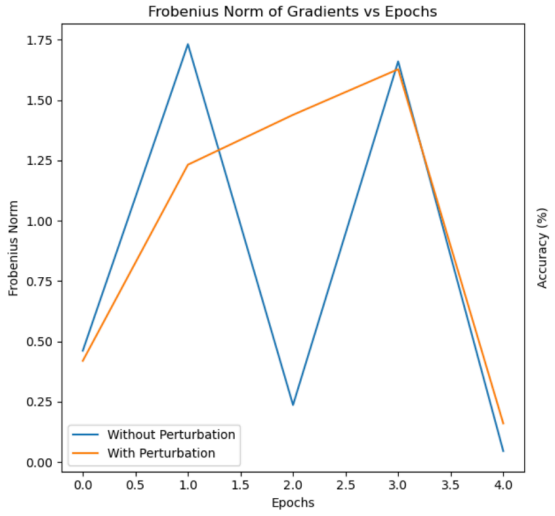
The task involves comparing the performance of two models with identical architectures but different batch sizes. Model 1 uses a batch size of 64 with 6,748 parameters, CrossEntropyLoss, and the Adam optimizer set at a learning rate of 0.001 and weight decay of 1e-4. Model 2 operates with a larger batch size of 1024 and also has 6,748 parameters, but utilizes a weight decay of 1e-2 while maintaining the same learning rate and loss function. The objective is to assess how varying batch sizes and weight decay impact model performance and training dynamics.



Larger batch sizes generally lead to smoother training dynamics but might not always improve generalization compared to smaller batch sizes.

Part 2: Batch Size Sensitivity

In this experiment, five neural network models are trained on a dataset with varying batch sizes: 10, 30, 120, 500, and 800. Each model is configured with a neural network architecture consisting of 6,748 parameters, utilizing ReLU activation functions, and optimized with the Adam optimizer set at a learning rate of 0.001. The loss function used is CrossEntropyLoss, and each model is trained for 20 epochs. This setup allows for an evaluation of how different batch sizes impact the training dynamics and overall performance of the model.



The choice of batch size affects the convergence rate and generalization ability. Smaller batch sizes may provide more noise and lead to better generalization, while larger batch sizes might converge faster but could overfit.
Github Link: https://github.com/shwethasivakumar/Deep-Learning/blob/main/HW1-3.ipynb