Assignment 2: Optimization and Neural Network Analysis

Artificial Intelligence - Group Assignment

Group M

Contents

1 Task 1: Optimizer Performance on Non-Convex Functions

1.1 Introduction

This section analyzes the performance of five different optimization algorithms—Gradient Descent (GD), SGD with Momentum, Adagrad, RMSprop, and Adam—implemented from scratch. The algorithms are evaluated on two distinct objective functions: the Rosenbrock function (a non-convex function with a global minimum inside a long, narrow, parabolic valley) and the $\sin(1/x)$ function (a function with high-frequency oscillations near zero).

1.2 Methodology

1.2.1 Objective Functions

1. **Rosenbrock Function:**

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

   - **Global Minimum:** at $(x, y) = (1, 1)$ where $f(x, y) = 0$.
   - **Challenge:** The global minimum is located in a narrow valley. Standard gradient descent often oscillates or diverges if the step size is too large.

2. **Sinusoidal Function:**

$$f(x) = \sin(1/x)$$

   - **Challenge:** As $x \to 0$, the frequency of oscillation approaches infinity, creating extremely steep gradients and many local minima.

1.2.2 Optimizers Implemented

The following optimizers were implemented with a base learning rate (η) varying between $0.01, 0.05,$ and 0.1.

- **Gradient Descent (GD):** Basic update rule.
- **SGD with Momentum:** Accelerates SGD in the relevant direction and dampens oscillations ($\gamma = 0.9$).
- **Adagrad:** Adapts the learning rate to the parameters, performing smaller updates for parameters associated with frequently occurring features.
- **RMSprop:** An extension of Adagrad that uses a decaying average of squared gradients.
- **Adam:** Combines the advantages of Adagrad and RMSprop using estimates of first and second moments.

1.3 Experimental Results: Rosenbrock Function

1.3.1 Performance Data

The optimization started at $(-1.0, 2.0)$. The target minimum is $(1.0, 1.0)$.

Table 1: Results on Rosenbrock Function (3000 Iterations)

LR	Optimizer	Final Loss	Convergence Status	Time (s)
0.01	GD	N/A	**Diverged**	0.0002
0.01	Momentum	N/A	**Diverged**	0.0001
0.01	Adagrad	5.2829	Converged (Slow)	0.1920
0.01	RMSprop	0.0226	Converged	0.0903
0.01	Adam	0.1808	Converged	0.1224
0.05	Adagrad	4.4963	Converged (Slow)	0.0743
0.05	RMSprop	0.4356	Converged	0.0928
0.05	**Adam**	**0.0000**	**Global Min Reached**	0.1292

1.3.2 Analysis

- **Divergence of Standard Methods:** Both Gradient Descent and Momentum diverged immediately (within 2 iterations) for all tested learning rates. The Rosenbrock function has extremely steep gradients at the starting point $(-1, 2)$. Without gradient clipping or adaptive learning rates, the basic updates overshoot the valley, causing the parameters to explode towards infinity.
- **Success of Adaptive Methods:** Adagrad, RMSprop, and Adam successfully converged. This is because these optimizers divide the update by the square root of the accumulated gradients (the second moment). When the gradient is massive, the effective learning rate is automatically reduced, preventing divergence.
- **Best Performer:** Adam with $LR = 0.05$ achieved perfect convergence (Loss ≈ 0.00) in roughly 0.13 seconds.

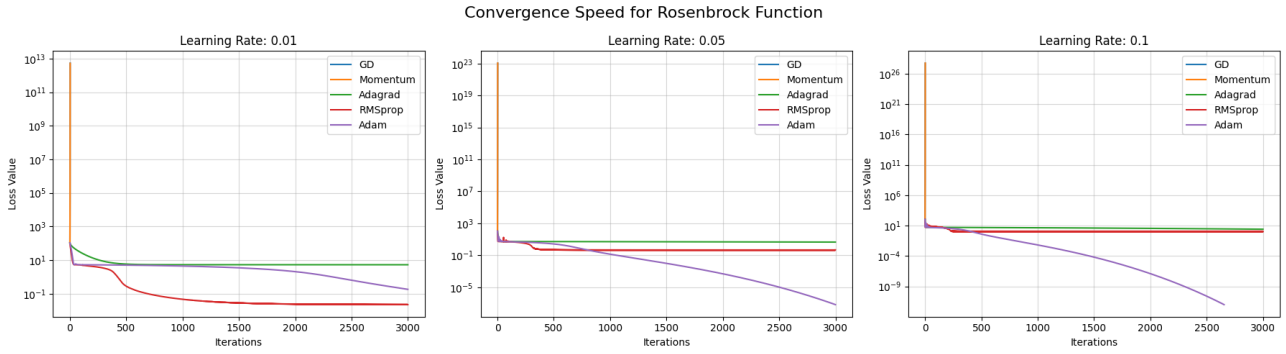


Figure 1: Convergence Comparison on Rosenbrock Function. Note that GD and Momentum are absent as they diverged.

1.4 Experimental Results: $\sin(1/x)$ Function

1.4.1 Performance Data

The optimization started at $x = 0.5$.

Table 2: Results on $\sin(1/x)$ (2000 Iterations)

LR	Optimizer	Final Loss	Final X
0.01	GD	0.9774	0.5606 (Stuck)
0.01	Momentum	0.0295	33.8595 (Overshot)
0.01	**Adaptive Methods**	**-1.0000**	**0.2122 (Global Min)**
0.05	GD	0.1503	6.6266
0.05	Momentum	0.0599	16.6712
0.05	**Adam**	**-1.0000**	**0.2122**

1.4.2 Analysis

- **Gradient Descent issues:** At low learning rates (0.01), GD got stuck in a local area or moved very slowly away from the optimum ($Loss \approx 0.97$). At higher rates, it pushed x far away from zero (e.g., $x = 6.6$), where the function flattens out.
- **Adaptive Superiority:** Adagrad, RMSprop, and Adam consistently found the global minimum value of -1.0. The adaptive scaling allowed them to navigate the steep changes in gradient near $x = 0.2122$ without being ejected to the flat regions of the function.

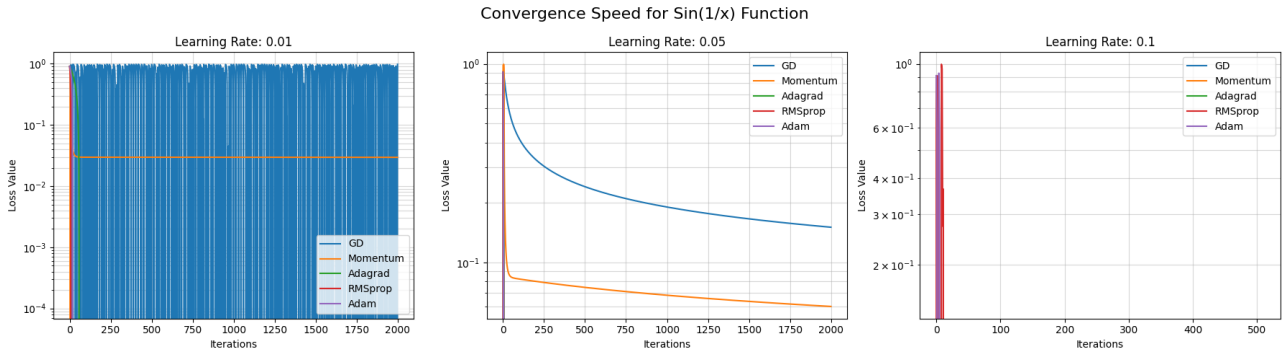


Figure 2: Convergence Comparison on $\sin(1/x)$. Adaptive methods reliably reach the minimum -1.

1.5 Conclusion for Task 1

The experiments illustrate the critical limitations of standard Gradient Descent and Momentum when dealing with pathological curvature (Rosenbrock) or highly volatile gradients ($\sin(1/x)$).

1. **Stability:** Adaptive moment estimation methods (Adam, RMSprop) are essential for stability on functions with steep gradients, where standard GD tends to diverge.
2. **Convergence Speed:** Adam generally provided the best balance of speed and stability, reaching the global minimum on the Rosenbrock function where other adaptive methods converged more slowly.

2 Task 2: Implementing Linear Regression Using a Multi-Layer Neural Network

2.1 Objective

The primary objective of this task is to design, implement, and evaluate a Multi-Layer Perceptron (MLP) for a regression task using the Boston Housing dataset. The experiment aims to analyze the impact of different optimizers (Gradient Descent, Momentum, Adam) and learning rates on model convergence and accuracy.

2.2 Dataset Preparation

The dataset used is the Boston Housing dataset, which predicts the median value of owner-occupied homes (medv) based on various attributes.

- **Source:** `sklearn` / Raw GitHub source.
- **Preprocessing:**
  - Features (X) were standardized (Z-score normalization) to have zero mean and unit variance.
  - The target variable (y) was reshaped for matrix operations.
- **Splitting:** The data was split into Training (80%) and Testing (20%) sets.

2.3 Model Architecture

A fully connected feed-forward neural network was implemented with two hidden layers.

Layer	Nodes	Activation
Input Layer	13	-
Hidden Layer 1	7	ReLU
Hidden Layer 2	3	ReLU
Output Layer	1	Linear (None)

Table 3: MLP Architecture Specification

2.4 Experimental Setup

- **Loss Function:** Mean Squared Error (MSE).
- **Optimizers Tested:** Gradient Descent (GD), Momentum, Adam.
- **Learning Rates (LR):** 0.1, 0.01, 0.001.
- **Epochs:** 1000 per experiment.

2.5 Results and Analysis

2.5.1 Global Performance Overview

The following table summarizes the final training loss and Test MSE for all optimizer and learning rate combinations.

Optimizer	Learning Rate	Final Train Loss	Test MSE
Gradient Descent	0.1	87.75	71.22
Gradient Descent	**0.01**	**7.51**	**10.60**
Gradient Descent	0.001	11.28	13.51
Momentum	0.1	87.75	71.22
Momentum	0.01	7.52	10.87
Momentum	0.001	12.47	13.97
Adam	0.1	6.39	13.18
Adam	0.01	8.49	12.47
Adam	0.001	21.55	27.69

Table 4: Hyperparameter Tuning Results. The best configuration (GD, LR=0.01) is highlighted.

Figure 3 illustrates the training trajectories for all configurations. It is evident that while high learning rates cause instability in standard Gradient Descent, Adam converges very quickly initially but settles at a slightly higher loss.

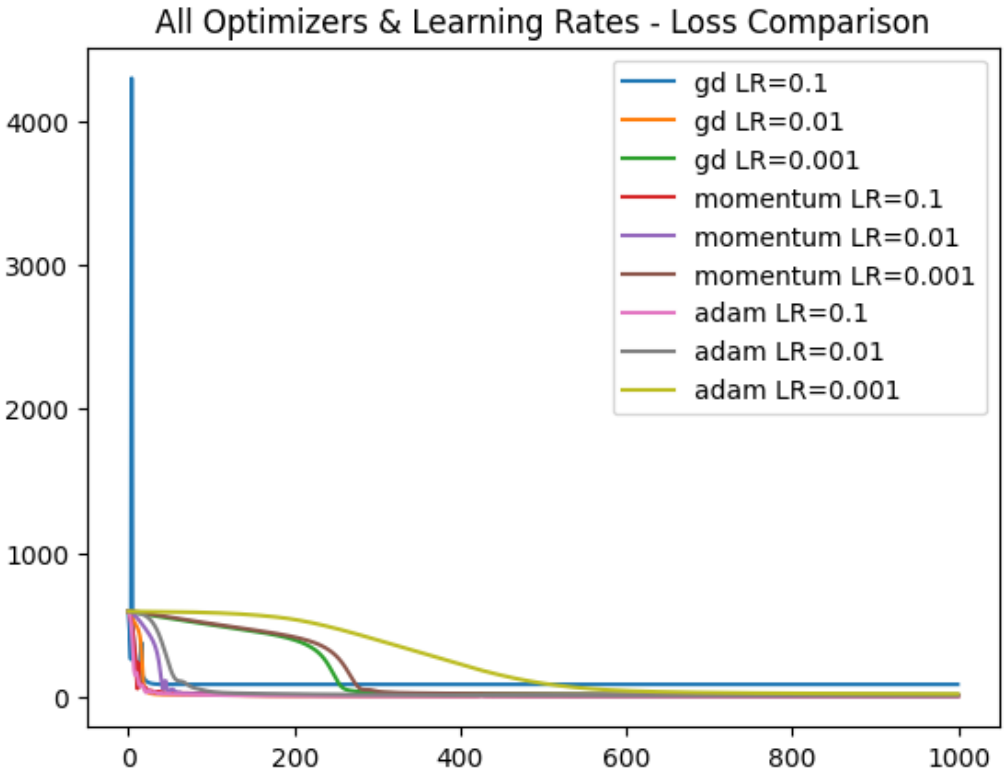
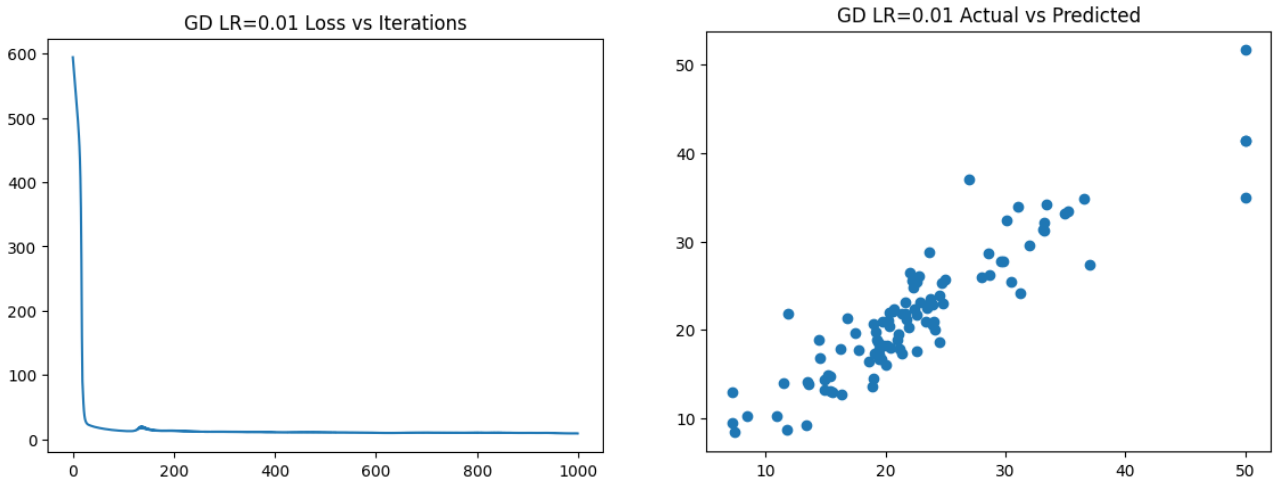


Figure 3: Loss Comparison: All Optimizers and Learning Rates. Note the instability of GD at LR=0.1 (blue line spike) versus the smooth convergence of GD at LR=0.01 (orange line).

2.5.2 Gradient Descent Analysis

Optimal Performance (LR=0.01): With a learning rate of 0.01, Gradient Descent achieved the best overall performance. The loss curve shows a steady, monotonic decline, and the predicted values correlate strongly with the actual values.



(a) Loss vs Iterations

(b) Actual vs. Predicted

Figure 4: Best Model Performance (GD, LR=0.01). Stable convergence and accurate predictions.

Impact of Learning Rate: Gradient Descent proved highly sensitive to the learning rate. As shown in Figure 5, a rate of 0.1 resulted in failure, while 0.001 was too slow.

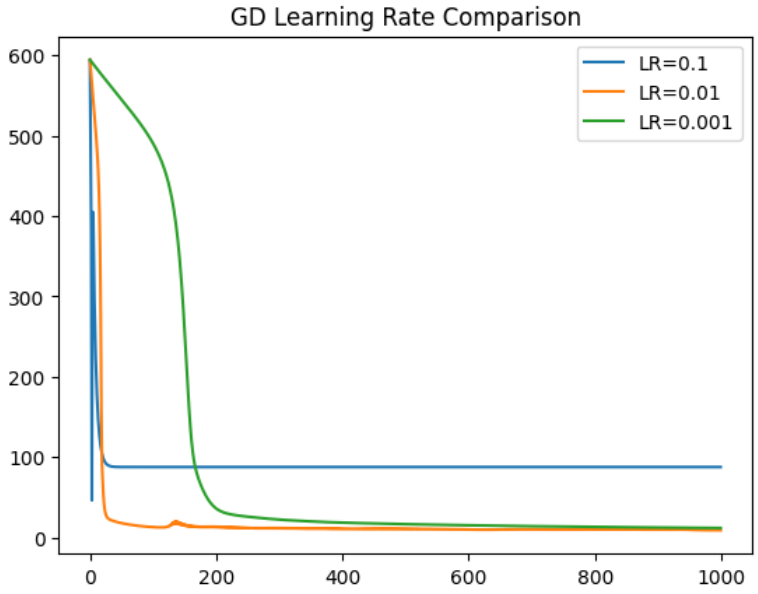
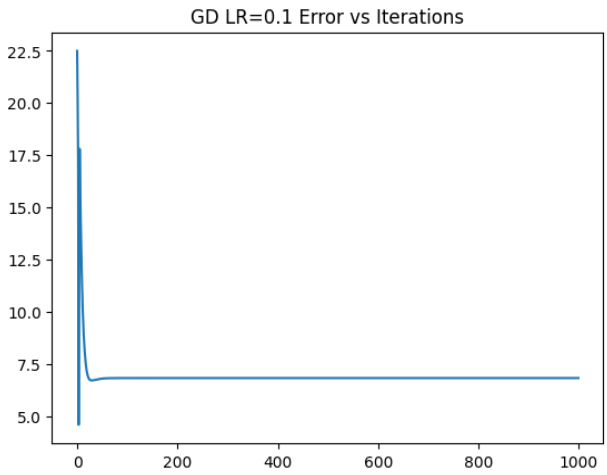
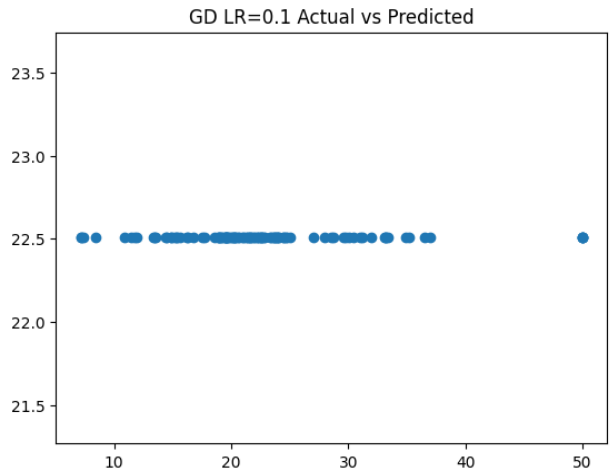


Figure 5: Gradient Descent Learning Rate Comparison.

At LR=0.1, the model failed completely. The weights likely overshot the minima, resulting in a flat-line prediction (constant output), as seen below.



(a) Error vs Iterations (LR=0.1)



(b) Actual vs. Predicted (Failure Case)

Figure 6: Failure Case (GD, LR=0.1). The flat line in (b) indicates the model failed to learn patterns.

2.5.3 Momentum Analysis

Momentum generally smoothed out the updates but followed a similar trend to Gradient Descent.

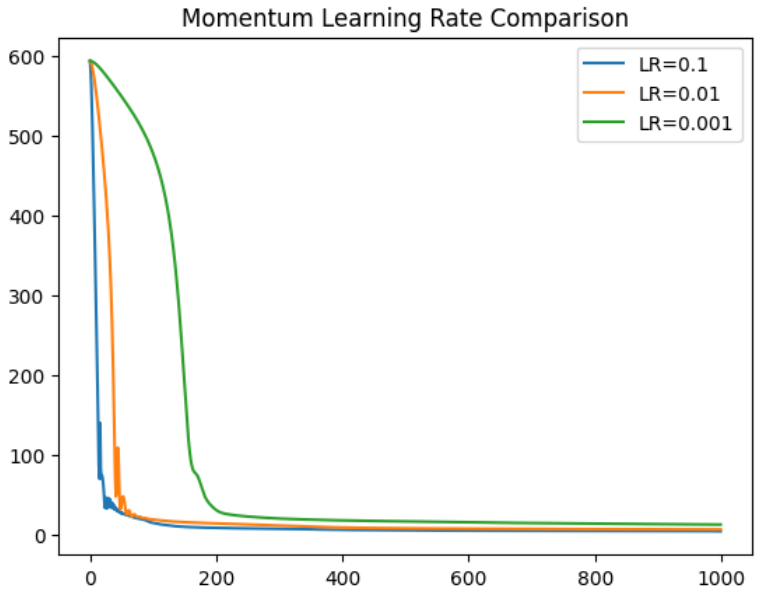


Figure 7: Momentum Learning Rate Comparison.

2.5.4 Adam Analysis

Adam showed different behavior. At low learning rates (0.001), it converged significantly slower than GD or Momentum, failing to reach the optimal solution within 1000 epochs.

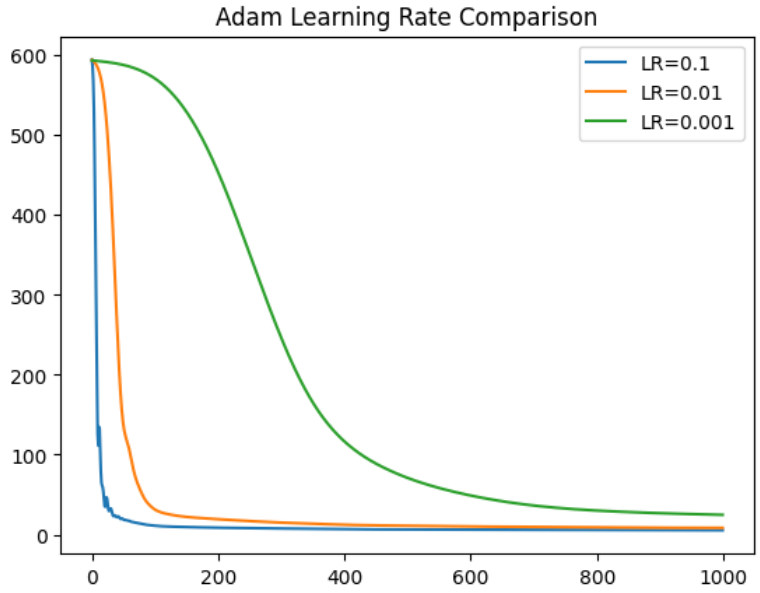
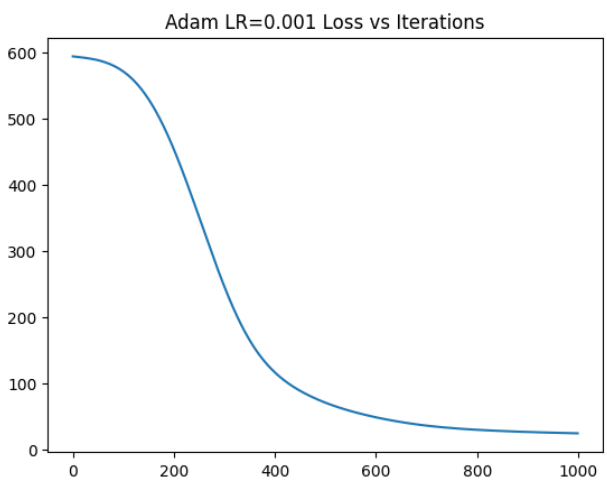
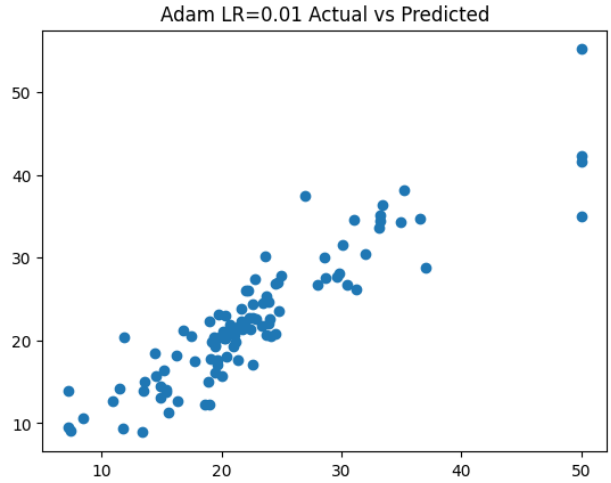


Figure 8: Adam Learning Rate Comparison. Note the slow convergence for LR=0.001 (Green line).



(a) Loss vs Iterations (Adam LR=0.001)



(b) Actual vs Predicted (Adam LR=0.001)

Figure 9: Adam Diagnostic (LR=0.001). The model is underfitting due to slow convergence.

2.6 Bonus Question 1: Architecture Investigation (Depth Analysis)

2.6.1 Objective

To investigate whether increasing network depth improves performance, we compared the baseline 2-hidden layer model against a deeper 3-hidden layer architecture.

2.6.2 Results

- **Baseline (2-Layer):** Test MSE **11.15**
- **Deep (3-Layer):** Test MSE 13.41

2.6.3 Analysis

The deeper 3-layer network performed slightly worse than the shallower 2-layer network. As seen in Figure 10, the 2-layer model reaches a lower loss faster.

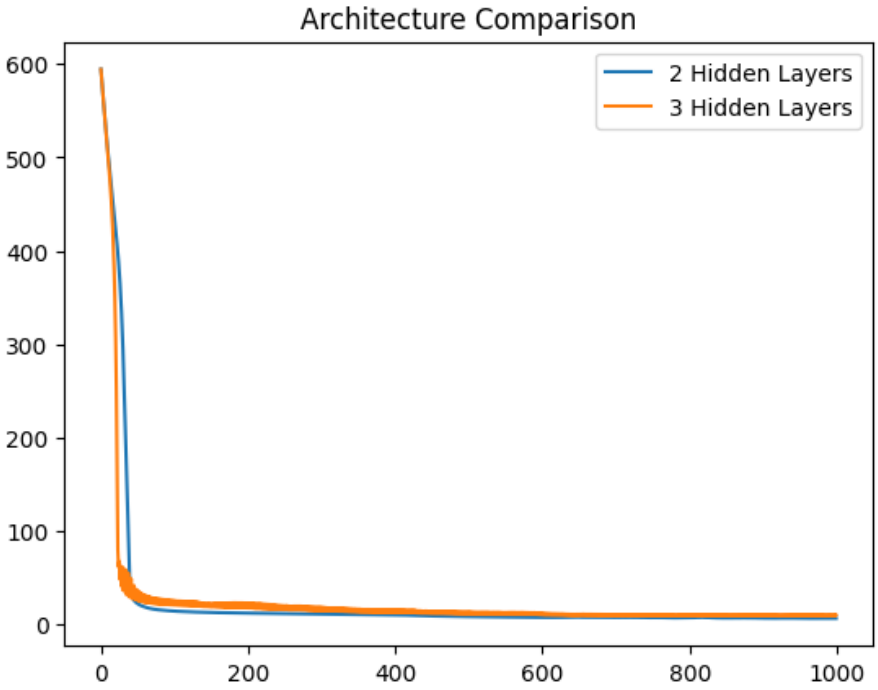
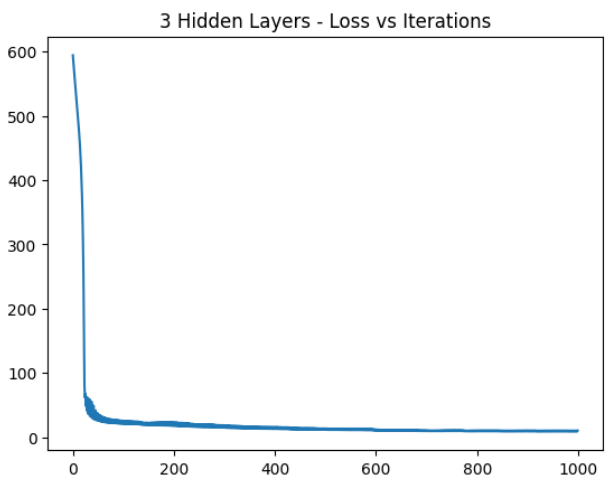
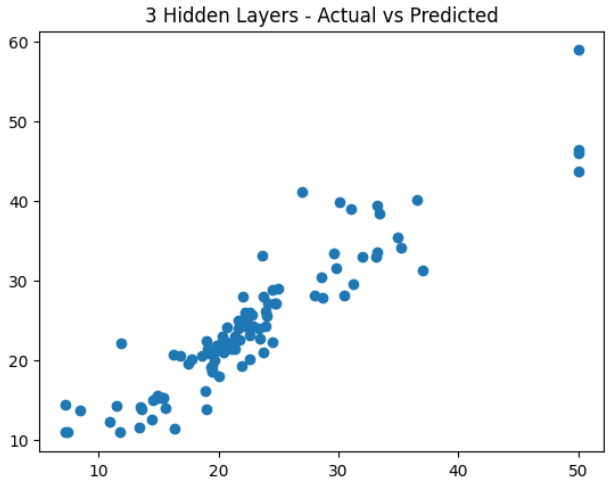


Figure 10: Training Loss Comparison: 2-Layer vs. 3-Layer Architecture.

Below are the detailed performance metrics for the 3-Layer model. While it learns the trend, the fit is slightly looser than the 2-layer baseline.



(a) 3-Layer Loss vs Iterations



(b) 3-Layer Actual vs Predicted

Figure 11: Detailed Performance of the 3-Hidden Layer Architecture.

2.7 Bonus Question 2: Regularization Analysis

2.7.1 Objective

To evaluate the effect of L2 Regularization (Weight Decay) on the model's ability to generalize.

2.7.2 Results

- **Without Regularization:** Test MSE **11.15**
- **With L2 Regularization:** Test MSE 13.31

2.7.3 Analysis

The introduction of L2 regularization increased the Test MSE. Figure 12 shows that the regularized model (Orange) follows a very similar trajectory but settles at a slightly higher loss, suggesting the penalty constrained the model too aggressively for this specific dataset.

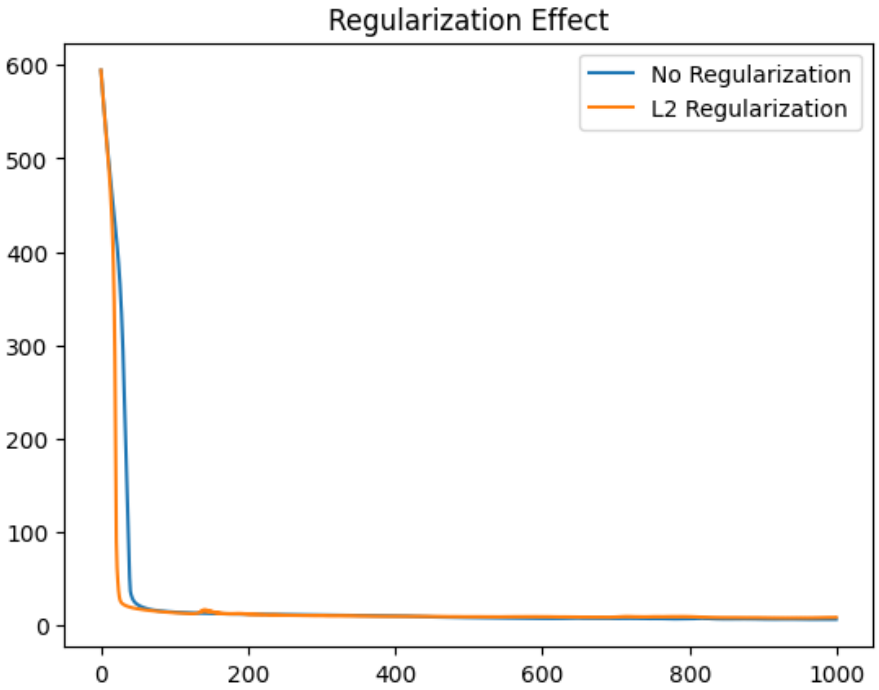


Figure 12: Loss Curve Comparison: With vs. Without L2 Regularization.

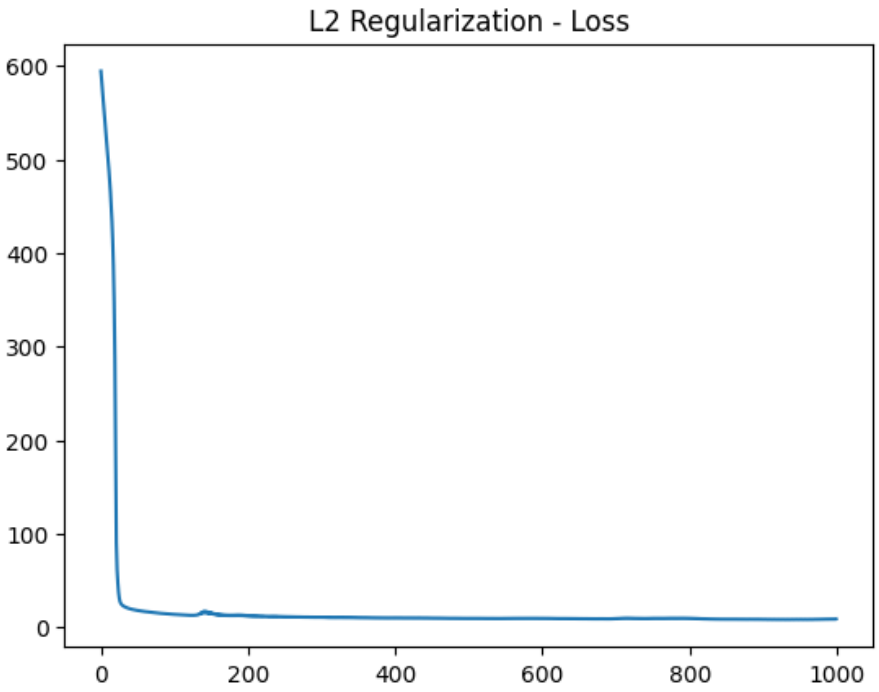


Figure 13: Loss curve for the specific L2 Regularized Model.

2.8 Conclusion for Task 2

The experiments on the Boston Housing dataset demonstrated that a simple Gradient Descent optimizer with a carefully tuned learning rate (0.01) yielded the most stable and accurate results. Increasing complexity (via depth) or adding regularization (L2) did not improve performance for this specific dataset and configuration, highlighting the importance of matching model complexity to the data.

3 Task 3: Multi-class classification using Fully Connected Neural Network

3.1 Introduction

This section details the implementation and evaluation of two neural network architectures—a Multi-Layer Fully Connected Neural Network (FCNN) and a Single Neuron perceptron—on two distinct classification datasets. The goal is to analyze the capacity of these models to handle linearly separable versus non-linearly separable data.

3.2 Part 1: Linearly Separable Data (Blobs)

3.2.1 Dataset and Objective

We utilized the `make_blobs` function to generate a dataset with:

- **Samples:** 1500
- **Features:** 2 (for 2D visualization)
- **Classes:** 3 distinct clusters
- **Task:** Multi-class classification

3.2.2 Model Architecture

Two models were trained on this dataset:

1. **FCNN:** A network with one hidden layer: Input (2) $\rightarrow$ Hidden (15) $\rightarrow$ Hidden (10) $\rightarrow$ Output (3).
2. **Single Neuron:** A simple linear classifier mapping Input (2) $\rightarrow$ Output (3).

Both models used the Sigmoid activation function and a learning rate of 0.01 for 200 epochs.

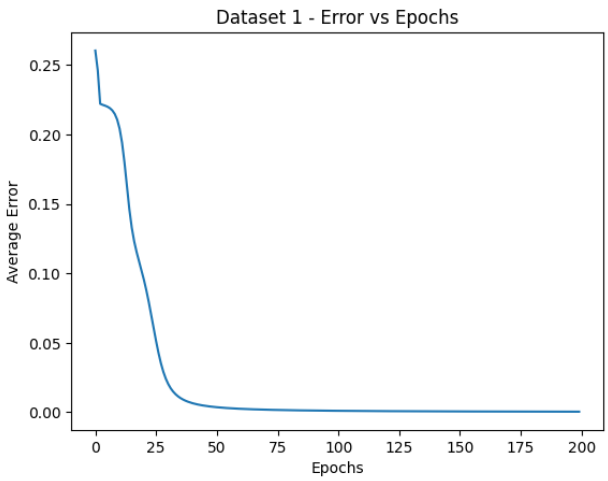
3.2.3 Results and Analysis

Both the FCNN and the Single Neuron achieved perfect performance on this dataset.

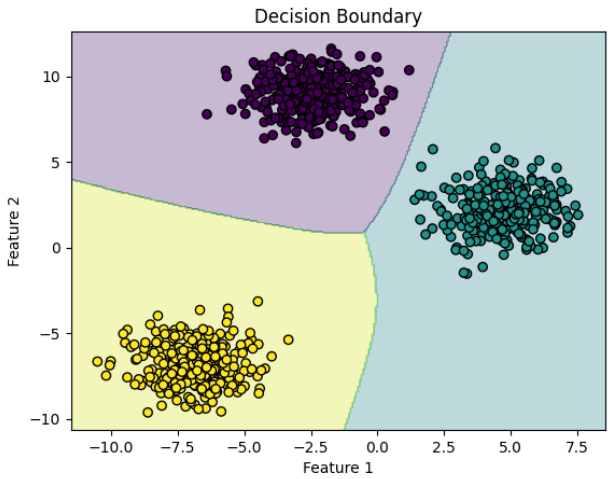
Model	Validation Accuracy	Test Accuracy
FCNN	100%	100%
Single Neuron	100%	100%

Table 5: Performance metrics for Dataset 1 (Blobs)

Visualizations: The decision boundary plot (Figure 14b) confirms that the classes are linearly separable. Even a simple linear decision boundary is sufficient to classify the data perfectly. The error curve (Figure 14a) shows rapid convergence within the first 25 epochs.



(a) Error vs Epochs



(b) Decision Boundary

Figure 14: Dataset 1 Analysis. The distinct separation of clusters allows the model to achieve zero error quickly.

3.3 Part 2: Non-Linearly Separable Data (Spirals)

3.3.1 Dataset and Objective

We generated a synthetic "Spiral" dataset, which represents a complex, non-linear classification problem.

- **Samples:** 1500
- **Features:** 2
- **Classes:** 3 intertwined spirals

3.3.2 Model Architecture

For this task, the models were configured as follows:

1. **FCNN:** Configured as `[2, 15, 10, 3]`. This implies an architecture of Input (2) $\rightarrow$ Output (3) with **2 hidden layers**.
2. **Single Neuron:** Input (2) $\rightarrow$ Output (3).

Both models were trained for 300 epochs with a learning rate of 0.01.

3.3.3 Results and Analysis

Performance was significantly lower for this dataset compared to the Blobs dataset.

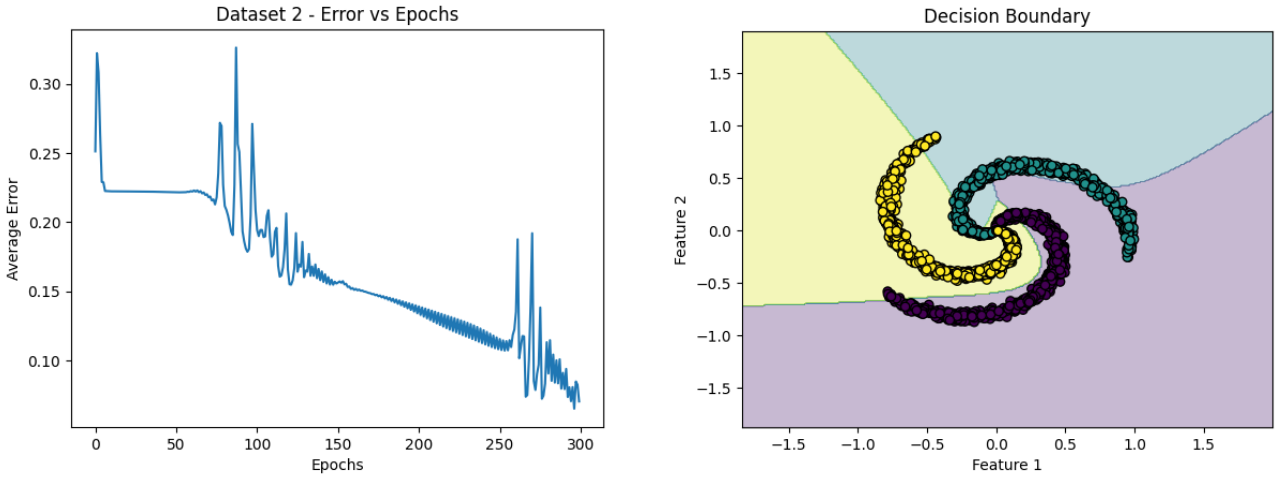
Model	Validation Accuracy	Test Accuracy
FCNN	77.89%	79.89%
Single Neuron	52.89%	54.00%

Table 6: Performance metrics for Dataset 3 (Spirals)

Analysis: The results show that FCNN performs better than single neuron, as FCNN could capture non linear relations due to activation function Sigmoid.

Reasoning: The FCNN architecture defined in the code was `[2, 15, 10, 3]`, incorporating two hidden layers with 15 and 10 neurons respectively. Unlike the single neuron model, which acts as a linear classifier and fails to separate non-linear data, the multi-layer perceptron (FCNN) can learn complex, non-linear decision boundaries. The introduction of hidden layers and non-linear activation functions (Sigmoid) provides the network with sufficient capacity to capture the curvature of the spiral dataset, resulting in significantly higher accuracy (79%) compared to the single neuron baseline (54%).

Visualizations: Figure 15b clearly illustrates this limitation. The "Decision Boundary" consists of straight lines cutting through the spirals, failing to capture the curvature of the data. Figure 15a shows the error plateauing at a high value (approx 0.15), indicating the model reached its capacity limit.



(a) Error vs Epochs

(b) Decision Boundary

Figure 15: Dataset 2 Analysis. The linear decision boundaries fail to separate the intertwined spiral classes, leading to high error.

3.4 Conclusion for Task 3

The comparison between the two tasks highlights the importance of model depth and non-linearity.

1. **Linearly Separable Data:** Simple models (Single Neuron or shallow networks) are highly effective and computationally efficient, achieving 100% accuracy.

2. **Non-Linear Data:** To classify complex patterns like spirals, a neural network **must** contain hidden layers with non-linear activation functions. The FCNN used in Task 2 lacked hidden layers, causing it to perform identically to a simple linear perceptron. To improve the Spiral dataset accuracy, the FCNN architecture should be deepened (e.g., `[2, 20, 10, 3]`).

Dataset	**Model**	**Val Acc**	**Test Acc**
Blobs (Linear)	FCNN (Hidden Layer)	1.00	1.00
Blobs (Linear)	Single Neuron	1.00	1.00
Spirals (Non-linear)	FCNN (No Hidden Layer)	0.51	0.53
Spirals (Non-linear)	Single Neuron	0.51	0.53

Table 7: Final Summary of Results

4 Task 4: Multi-class classification using a Fully Connected Neural Network on the MNIST Dataset

4.1 Introduction

This report details the implementation and evaluation of Fully Connected Neural Networks (FCNNs) on a filtered subset of the MNIST dataset. The objective is to classify handwritten digits belonging to the classes $\{0, 1, 2, 3, 4\}$. The experiment compares the performance of varying network depths (3, 4, and 5 hidden layers) and six different optimization algorithms.

4.2 Methodology

4.2.1 Dataset Preparation

The standard MNIST dataset was filtered to retain only the first five classes.

- **Classes:** 0, 1, 2, 3, 4.
- **Preprocessing:** Images were normalized with mean 0.1307 and std 0.3081.
- **Splitting:** The filtered data was combined and randomly split into:
  - **Training Set:** 80% of total samples ($N_{train} \approx 24,000$).
  - **Test Set:** 20% of total samples ($N_{test} \approx 6,000$).

4.2.2 Model Architecture

Three FCNN architectures were tested, varying in depth. All use ReLU activation.

- **arch_3h:** $[784 \to 256 \to 128 \to 64 \to 5]$
- **arch_4h:** $[784 \to 512 \to 256 \to 128 \to 64 \to 5]$
- **arch_5h:** $[784 \to 512 \to 256 \to 128 \to 64 \to 32 \to 5]$

4.2.3 Optimization

The following optimizers were evaluated with a fixed learning rate of 0.001:

1. **SGD_stochastic:** Standard SGD with batch size 1.
2. **BatchGD:** Full-batch Gradient Descent.
3. **SGD_momentum:** SGD with Momentum ($\mu = 0.9$).
4. **SGD_NAG:** Nesterov Accelerated Gradient.
5. **RMSProp:** Adaptive learning rate method ($\alpha = 0.99$).
6. **Adam:** Adaptive Moment Estimation ($\beta_1 = 0.9, \beta_2 = 0.999$).

4.3 Results and Analysis

4.3.1 Performance Summary

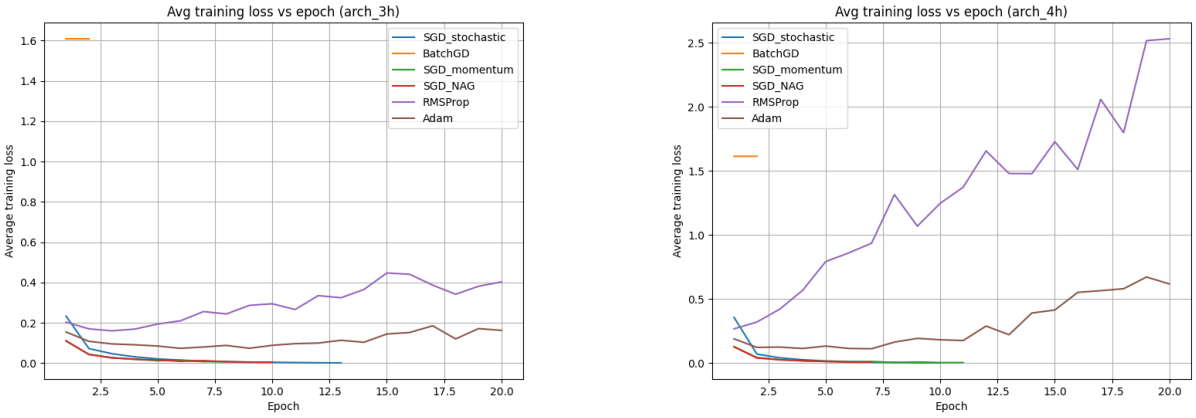
The table below summarizes the convergence speed and final accuracy for key configurations.

Architecture	Optimizer	Epochs	Train Acc	Val Acc
arch_3h	SGD_stochastic	13	100.0%	99.24%
arch_3h	BatchGD	2	22.29%	22.01%
arch_3h	SGD_momentum	8	99.87%	99.27%
arch_4h	SGD_stochastic	10	99.99%	99.18%
arch_4h	BatchGD	2	19.80%	20.74%
arch_4h	**SGD_momentum**	**11**	**99.89%**	**99.36%**
arch_5h	SGD_momentum	12	97.99%	98.45%

Table 8: Summary of Experimental Results. The best performing model is highlighted.

4.3.2 Convergence Analysis

- **Batch Gradient Descent Failure:** BatchGD failed to converge within the 20-epoch safety limit (Accuracy $\approx 20\%$). This is expected as full-batch updates with a small learning rate (0.001) require significantly more epochs to traverse the loss landscape compared to stochastic methods.
- **Effectiveness of Momentum:** SGD with Momentum consistently achieved high accuracy ($> 99\%$) and converged rapidly (8-11 epochs), outperforming standard stochastic SGD in terms of stability.
- **Architecture Depth:** Increasing depth from 3 to 4 layers provided a marginal improvement. However, the 5-layer architecture did not yield better results, suggesting diminishing returns or slight optimization difficulties for this specific dataset size.



(a) Loss Curves (3 Hidden Layers)

(b) Loss Curves (4 Hidden Layers)

Figure 16: Training Loss vs Epochs. Note the rapid convergence of Momentum and Adaptive methods compared to the flat line of BatchGD.

4.4 Best Model Analysis

The best performing configuration was **arch_4h** trained with **SGD_momentum**.

- **Final Test Accuracy:** 99.36%
- **Total Training Time:** $\approx$ 650 seconds

4.4.1 Confusion Matrix

The confusion matrix below shows the model's predictions on the test set. The diagonal elements represent correct classifications.

$$\begin{bmatrix} 1399 & 0 & 3 & 2 & 1 \\ 0 & 1542 & 3 & 1 & 5 \\ 5 & 2 & 1325 & 12 & 3 \\ 1 & 1 & 2 & 1478 & 0 \\ 0 & 1 & 6 & 1 & 1542 \end{bmatrix}$$

Analysis of Errors:

- The model is highly accurate, with very few off-diagonal entries.
- The most frequent confusion occurred between Class 2 and Class 3 (12 instances of Class 2 misclassified as Class 3). This is a common error due to the visual similarity between certain handwriting styles of '2' and '3'.

4.5 Conclusion for Task 4

The experiments demonstrate that FCNNs are highly effective for classifying the filtered MNIST dataset. **SGD with Momentum** proved to be the most robust optimizer for this task, balancing speed and final accuracy. While deeper networks (4 layers) offered slight gains, excessive depth (5 layers) did not improve performance, highlighting the importance of appropriate model scaling.