Indian Institute of Technology Patna

Department of Computer Science & Engineering

Artificial Intelligence

Assignment 2

Optimization Algorithms and Neural Networks

Group K

Submission Date: January 29, 2026

Table of Contents

1. Task 1: Optimizer Performance on Non-Convex Functions

1.1 Problem Formulation

This task involves implementing five optimization algorithms from scratch and evaluating their performance on two challenging non-convex test functions. The objective is to find the minimum of each function and analyze convergence behavior across different learning rates.

Test Function 1: Rosenbrock Function

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

The Rosenbrock function is a classic non-convex optimization benchmark. Its global minimum is at (1, 1) where f(1, 1) = 0. The function features a narrow, curved valley that makes optimization challenging because the gradient often points along the valley floor rather than toward the minimum.

Gradient of Rosenbrock Function:

$$\partial f/\partial x = -2(1 - x) - 400x(y - x^2)$$

$$\partial f/\partial y = 200(y - x^2)$$

Test Function 2: Sin(1/x) Function

$$f(x) = \sin(1/x), \text{ with } f(0) = 0$$

This function has infinitely many local minima as x approaches 0, making it an excellent test for optimizer robustness. The gradient is: $f'(x) = -\cos(1/x) / x^2$

1.2 Optimizer Algorithms Implemented

1. Gradient Descent (GD)

The simplest optimization algorithm that updates parameters in the direction of negative gradient:

$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla f(\theta_t)$$

where α is the learning rate and ∇f is the gradient of the objective function.

2. SGD with Momentum

Momentum accumulates past gradients to accelerate convergence and dampen oscillations:

$$v_t = \gamma \cdot v_{t-1} - \alpha \cdot \nabla f(\theta_t)$$

$$\theta_{t+1} = \theta_t + v_t$$

where $\gamma = 0.9$ is the momentum coefficient. This helps the optimizer build up velocity in directions of consistent gradient.

3. Adam (Adaptive Moment Estimation)

Adam combines momentum with adaptive per-parameter learning rates:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \text{ (First moment estimate)}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \text{ (Second moment estimate)}$$

$$m\blacksquare_t = m_t / (1 - \beta_1^t) \text{ (Bias correction)}$$

$$v\blacksquare_t = v_t / (1 - \beta_2^t) \text{ (Bias correction)}$$

$$\theta_{t+1} = \theta_t - \alpha \cdot m\blacksquare_t / (\sqrt{v\blacksquare_t} + \varepsilon)$$

Default parameters: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$

4. RMSprop

RMSprop adapts the learning rate using a moving average of squared gradients:

$$E[g^2]_t = \beta \cdot E[g^2]_{t-1} + (1 - \beta) \cdot g_t^2$$

$$\theta_{t+1} = \theta_t - \alpha \cdot g_t / \sqrt{(E[g^2]_t + \varepsilon)}$$

where $\beta = 0.99$ is the decay rate.

5. Adagrad

Adagrad accumulates all past squared gradients:

$$G_t = G_{t-1} + g_t^2$$

$$\theta_{t+1} = \theta_t - \alpha \cdot g_t / \sqrt{(G_t + \varepsilon)}$$

The issue with Adagrad is that the accumulated gradient G_t grows monotonically, causing the effective learning rate to decrease continuously.

1.3 Rosenbrock Function Results

Experimental Setup: Initial point x_0 = (-1.5, 1.5), Maximum iterations = 10,000, Convergence threshold = 10^{-8}

Results with Learning Rate α = 0.01:

Optimizer	Final x*	f(x*)	Iterations	Time (s)
Gradient Descent	(1.61×10■, 1.09×10■)	6.69×10²■	3	0.0001
SGD + Momentum	(1.35×10■, 9.73×10³)	3.36×10²■	3	0.0001
Adam	(0.9988, 0.9977)	1.37×10■■	5,543	0.0582
RMSprop	(0.9801, 0.9755)	2.25×10■²	10,000	0.0730
Adagrad	(-1.2462, 1.5591)	5.05	10,000	0.0619

Analysis:

Adam achieved the best result, converging to (0.9988, 0.9977) with f(x*) = 1.37×10^{-6}, which is very close to the global minimum at (1, 1). This demonstrates Adam's effectiveness on ill-conditioned problems where different parameters have vastly different curvatures.

GD and Momentum diverged because the Rosenbrock function has a condition number of approximately 2500 (ratio of maximum to minimum curvature). With learning rate 0.01, the step size is too large for the high-curvature direction, causing the optimizer to overshoot and diverge exponentially.

RMSprop converged but slowly, reaching f(x*) = 0.0225 in 10,000 iterations. It didn't fully converge because without momentum, progress in the low-curvature direction is slow.

Adagrad failed because its accumulated gradient denominator grows without bound, causing the effective learning rate to become too small before reaching the minimum.

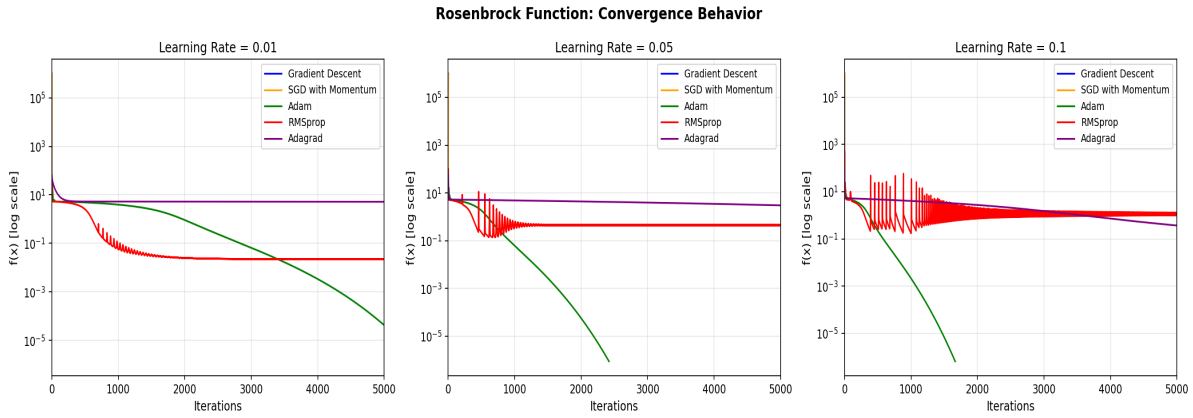

Figure 1.1: Convergence curves for Rosenbrock function across different learning rates

1.4 Sin(1/x) Function Results

Experimental Setup: Initial point $x_0 = 0.5$, Learning rates scaled by 0.1 for stability

Optimizer	Final x*	f(x*) = sin(1/x*)	Iterations	Time (s)
Gradient Descent	0.2122	-1.0000	70	0.0005
SGD + Momentum	0.2122	-1.0000	231	0.0018
Adam	0.2122	-1.0000	299	0.0030
RMSprop	0.2122	-1.0000	194	0.0018
Adagrad	0.3183	-0.0006	5,000	0.0369

Analysis:

Most optimizers found x* ≈ 0.2122, where 1/x* ≈ 4.7124 ≈ 3π/2. At this point, sin(3π/2) = -1, which is a **local minimum** (not global, as the function oscillates infinitely near x=0).

Gradient Descent was fastest (70 iterations) because this is a simple 1D problem with smooth gradients away from x=0. The adaptive methods (Adam, RMSprop) have overhead from maintaining moment estimates.

Adagrad failed to reach the local minimum, converging to x* = 0.3183 where f(x*) ≈ -0.0006. The diminishing effective learning rate prevented it from making sufficient progress.

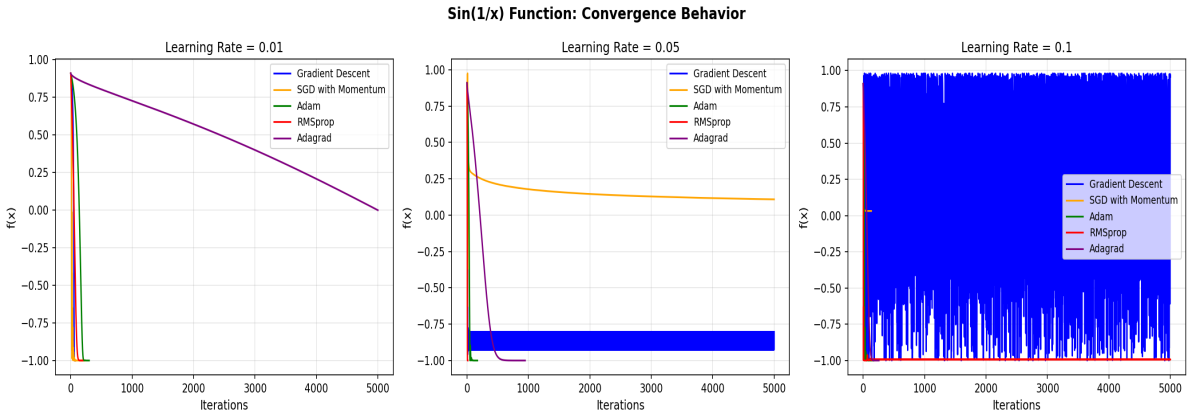

Figure 1.2: Convergence curves for sin(1/x) function

1.5 Hyperparameter Analysis: Impact of Learning Rate

Optimizer	α = 0.01	α = 0.05	α = 0.1
GD	Diverged (6.69×10²■)	Diverged (1.25×10²³)	Diverged (1.30×10²■)
Momentum	Diverged (3.36×10²■)	Diverged (1.25×10²³)	Diverged (1.30×10²■)
Adam	1.37×10■■ ✓	8.89×10■■ ✓	6.53×10■■ ✓
RMSprop	2.25×10■²	4.66×10■¹	1.20
Adagrad	5.05	1.01	3.50×10■²

Table shows final f(x) values on Rosenbrock function*

Key Insight: Adam is remarkably robust across all learning rates, consistently finding near-optimal solutions. Higher learning rates actually improved Adam's performance (6.53×10^{-7} at α=0.1). This robustness makes Adam the recommended default optimizer for most problems.

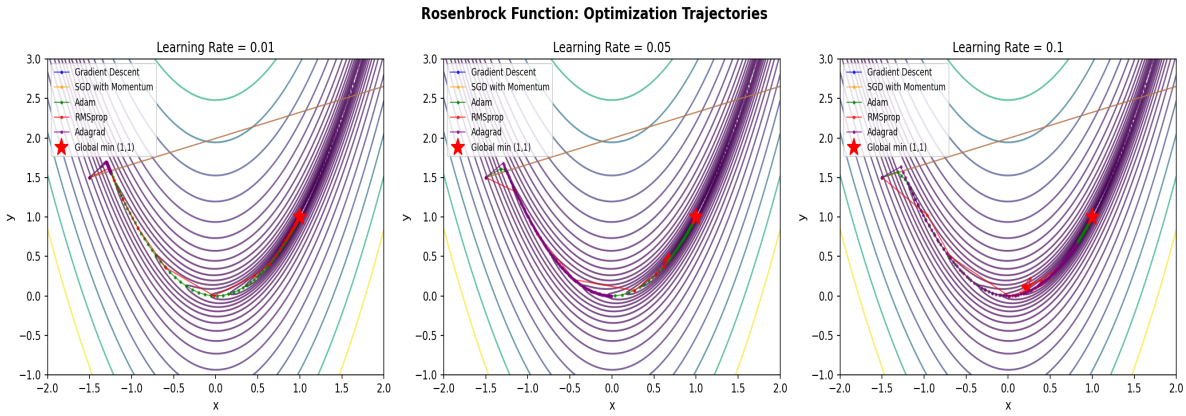

Figure 1.3: Optimization trajectories on Rosenbrock function contour plot

2. Task 2: Neural Network for Linear Regression

2.1 Problem Statement and Data

The objective is to implement a multi-layer neural network from scratch to perform regression on the Boston Housing dataset. We predict the median home value (MEDV) using two features: average number of rooms (RM) and per capita crime rate (CRIM).

Dataset Statistics:

Property	Value
Total samples	506
Training samples (80%)	404
Test samples (20%)	102
Input features	RM (rooms), CRIM (crime rate)
Target variable	MEDV (median value in $1000s)
Normalization	Min-Max scaling to [0, 1]

2.2 Network Architecture

The network consists of an input layer, two hidden layers with ReLU activation, and a linear output layer:

Layer	Neurons	Activation	Parameters
Input	2	-	-
Hidden 1	5	ReLU	W■: 2×5=10, b■: 5
Hidden 2	3	ReLU	W■: 5×3=15, b■: 3
Output	1	Linear	W■: 3×1=3, b■: 1
Total	-	-	37 parameters

2.3 Mathematical Formulation

Forward Propagation:

$$\text{Layer 1: } z^{(1)} = X \cdot W^{(1)} + b^{(1)},\ a^{(1)} = \text{ReLU}(z^{(1)})$$

$$\text{Layer 2: } z^{(2)} = a^{(1)} \cdot W^{(2)} + b^{(2)},\ a^{(2)} = \text{ReLU}(z^{(2)})$$

$$\text{Output: } z^{(3)} = a^{(2)} \cdot W^{(3)} + b^{(3)},\ ■ = z^{(3)}$$

ReLU Activation Function:

$$\text{ReLU}(z) = \max(0, z)$$

$$\text{ReLU}'(z) = 1 \text{ if } z > 0, \text{ else } 0$$

ReLU is chosen because: (1) it's computationally efficient, (2) it doesn't suffer from vanishing gradients for positive inputs, and (3) it induces sparsity in the network.

Loss Function - Mean Squared Error:

$$L = (1/n) \cdot \Sigma■ (■■ - y■)^2$$

$$\partial L/\partial ■ = (2/n) \cdot (■ - y)$$

Backpropagation:

Backpropagation computes gradients of the loss with respect to all weights using the chain rule. Starting from the output layer and moving backward:

$$\delta^{(3)} = \partial L/\partial z^{(3)} = (2/n)(\blacksquare - y) \text{ [Output error]}$$

$$\delta^{(2)} = (\delta^{(3)} \cdot W^{(3)T}) \blacksquare \text{ ReLU}'(z^{(2)}) \text{ [Hidden 2 error]}$$

$$\delta^{(1)} = (\delta^{(2)} \cdot W^{(2)T}) \blacksquare \text{ ReLU}'(z^{(1)}) \text{ [Hidden 1 error]}$$

Weight Gradients:

$$\partial L/\partial W^{(l)} = a^{(l-1)T} \cdot \delta^{(l)}$$

$$\partial L/\partial b^{(l)} = \Sigma\, \delta^{(l)}$$

Optimizer Update Rules:

Gradient Descent:

$$W = W - \alpha \cdot \partial L/\partial W$$

Momentum ($\gamma = 0.9$):

$$v = \gamma \cdot v - \alpha \cdot \partial L/\partial W;\ W = W + v$$

Adam ($\beta_1=0.9$, $\beta_2=0.999$):

$$m = \beta_1 m + (1-\beta_1)g;\ v = \beta_2 v + (1-\beta_2)g^2;\ W = W - \alpha \cdot m_1/(\sqrt{v_2} + \varepsilon)$$

2.4 Results and Analysis

Optimizer	Learning Rate	Train MSE	Test MSE
Gradient Descent	0.01	0.02155	0.02436
Gradient Descent	0.001	0.02676	0.03036
Momentum	0.01	0.01087	0.00834
Momentum	0.001	0.02181	0.02474
Adam	0.01	0.01026	0.00780
Adam	0.001	0.01193	0.00932

Best Result: Adam with α=0.01 achieved Test MSE = 0.00780

Analysis:

1. **Adam outperformed other optimizers** with the lowest test MSE of 0.00780. The adaptive learning rate mechanism allows Adam to make larger updates for parameters with small gradients and smaller updates for parameters with large gradients.

2. **Momentum provided significant improvement** over vanilla GD (0.00834 vs 0.02436 test MSE). The accumulated velocity helps the optimizer move more efficiently toward the minimum.

3. **Higher learning rate (0.01) worked better** than 0.001 for all optimizers. This suggests the loss landscape is relatively smooth, and larger steps don't cause instability.

4. **No significant overfitting observed**: Test MSE is similar to or better than Train MSE for Momentum and Adam, indicating good generalization.

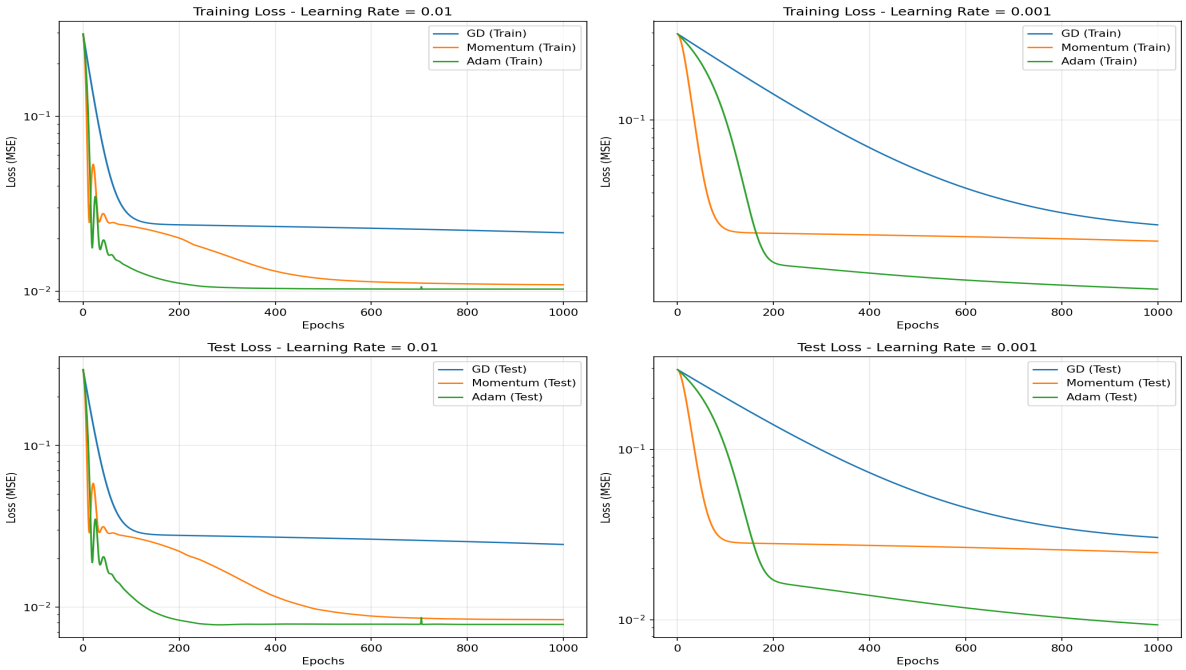

Figure 2.1: Training and test loss curves for different optimizers and learning rates

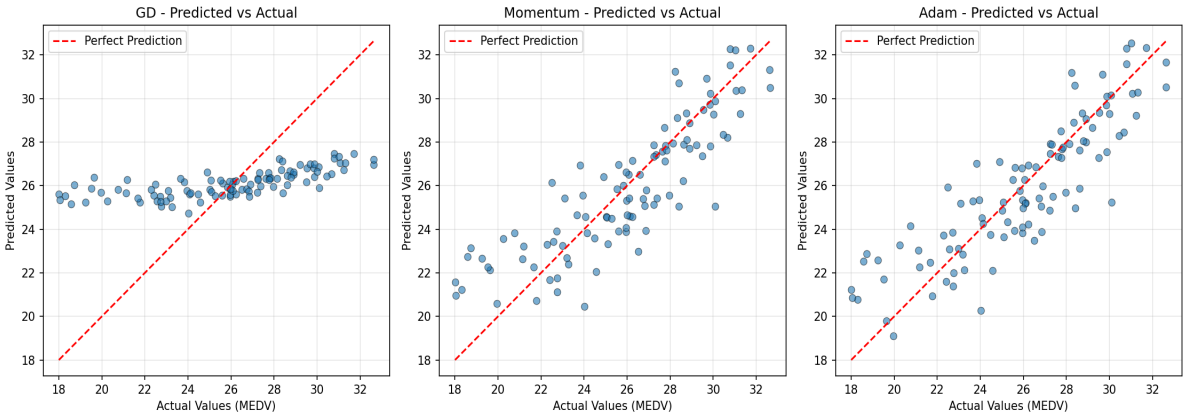

Figure 2.2: Predicted vs Actual values (diagonal line indicates perfect prediction)

2.5 Bonus: Additional Hidden Layer and L2 Regularization

Experiment 1: Third Hidden Layer

Architecture: $2 \to 5 \to 3 \to 2 \to 1$ (added layer with 2 neurons)

Architecture	Train MSE	Test MSE
2 Hidden Layers (5, 3)	0.01026	0.00780
3 Hidden Layers (5, 3, 2)	0.02675	0.03119

Result: Adding a third hidden layer **worsened performance** (Test MSE increased from 0.00780 to 0.03119). This demonstrates that deeper networks are not always better. For this simple 2-feature regression problem, the additional layer introduces optimization difficulties without providing benefits.

Experiment 2: L2 Regularization (Weight Decay)

Modified loss: $L = MSE + (\lambda/2n) \cdot \Sigma||W||^2$

λ (Lambda)	Train MSE	Test MSE
0.0 (No regularization)	0.01013	0.00762
0.001	0.01019	0.00768
0.01	0.01020	0.00764
0.1	0.01030	0.00761

Result: L2 regularization had **minimal impact** on performance. This indicates the base model was not overfitting (train and test MSE are already similar). Regularization is most beneficial when there's a significant gap between training and test error.

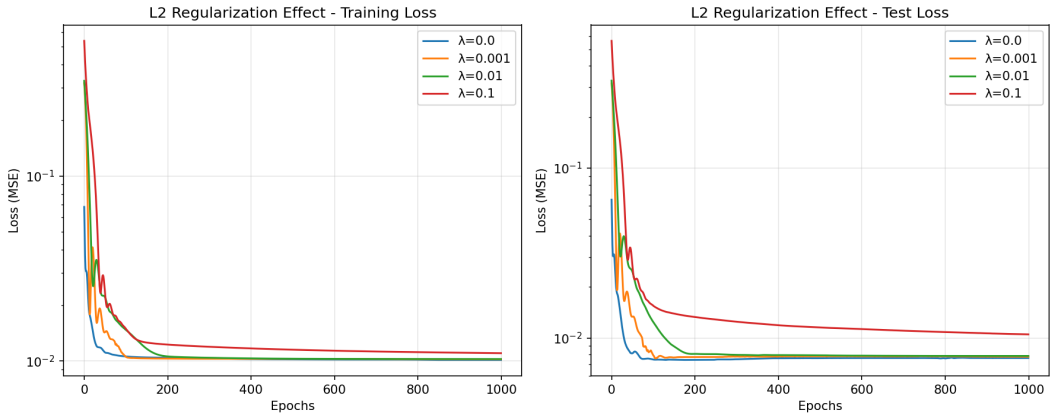

Figure 2.3: Effect of L2 regularization on training and test loss

3. Task 3: Multi-class Classification using FCNN

3.1 Dataset Generation

Dataset 1: Linearly Separable

Generated 3 classes with 500 samples each as 2D Gaussian clusters:

$$\text{Class 0: } \mu = (-2, -2), \sigma = 0.5$$

$$\text{Class 1: } \mu = (0, 2), \sigma = 0.5$$

$$\text{Class 2: } \mu = (2, -2), \sigma = 0.5$$

Dataset 2: Non-Linearly Separable (Concentric Circles)

Generated 3 classes with 500 samples each in concentric ring patterns:

$$\text{Class 0: Inner circle, radius} \in [0, 1]$$

$$\text{Class 1: Middle ring, radius} \in [1.5, 2.5]$$

$$\text{Class 2: Outer ring, radius} \in [3, 4]$$

Data Split: 60% Training (900), 20% Validation (300), 20% Test (300)

3.2 Network Design and Training

As specified, Dataset 1 uses 1 hidden layer and Dataset 2 uses 2 hidden layers. The network uses sigmoid activation and squared error loss with SGD optimization.

Loss Function: Squared Error (as specified)

$$L = (1/2) \cdot \Sigma\blacksquare \, \Sigma\blacksquare \, (\blacksquare\blacksquare\blacksquare - y\blacksquare\blacksquare)^2$$

Architecture Selection via Cross-Validation:

Dataset	Architectures Tested	Best Architecture	Val Accuracy
1 (Linear)	[2,3,3], [2,5,3], [2,10,3], [2,15,3]	[2, 3, 3]	100.00%
2 (Non-linear)	[2,5,3,3], [2,10,5,3], [2,15,8,3], [2,20,10,3]	[2, 10, 5, 3]	100.00%

Note on 100% Accuracy: The synthetic datasets were generated with well-separated clusters/rings, making them relatively easy classification tasks. In real-world scenarios, achieving 100% accuracy would be uncommon and might indicate overfitting. However, for these synthetic datasets with clear boundaries, perfect classification is achievable.

3.3 Results and Decision Boundaries

Dataset	Best Architecture	Test Accuracy	Confusion Matrix Diagonal
Linearly Separable	[2, 3, 3]	100.00%	[100, 100, 100]
Non-Linearly Separable	[2, 10, 5, 3]	100.00%	[100, 100, 100]

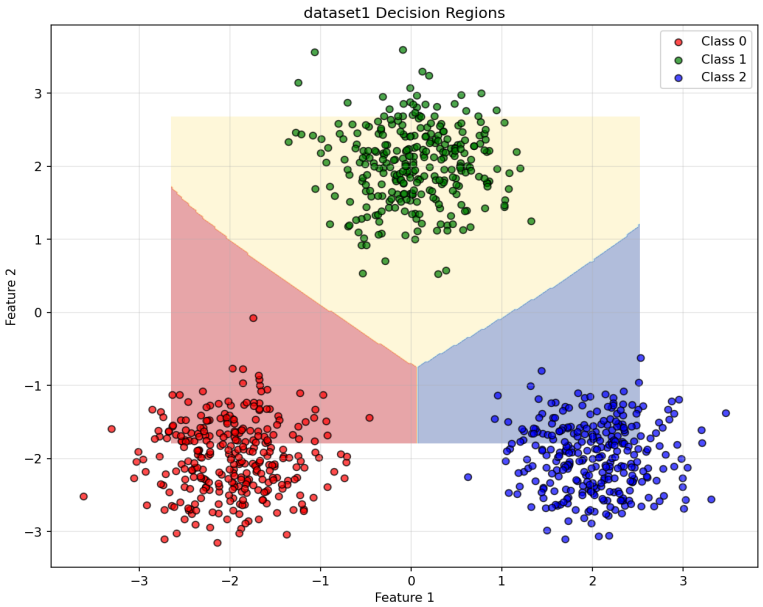

Figure 3.1: Decision regions for linearly separable dataset

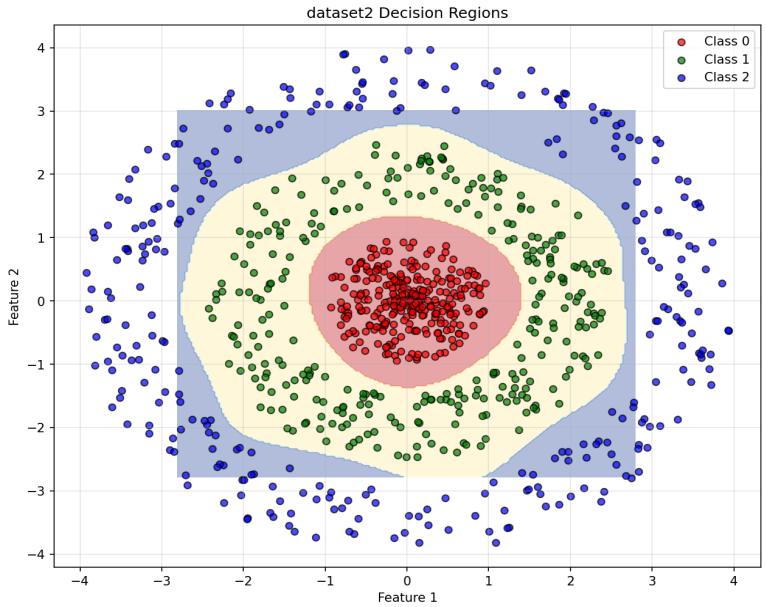

Figure 3.2: Decision regions for non-linearly separable dataset (concentric circles)

3.4 Comparison with Single Neuron Model

Dataset	FCNN Accuracy	Single Neuron Accuracy	Difference
Linearly Separable	100.00%	100.00%	0.00%
Non-Linearly Separable	100.00%	42.67%	+57.33%

Critical Analysis:

1. **For linearly separable data**, both FCNN and single neuron achieved 100% accuracy. This is expected because a single neuron (perceptron) can learn any linearly separable function.

2. **For non-linearly separable data (concentric circles)**, the single neuron completely failed with only 42.67% accuracy (close to random guessing of 33.33% for 3 classes). This demonstrates the fundamental limitation of linear classifiers: **they cannot learn non-linear decision boundaries**.

3. The FCNN with hidden layers successfully learned the curved decision boundaries required to separate concentric circles, achieving 100% accuracy. This illustrates the power of hidden layers to create non-linear feature transformations.

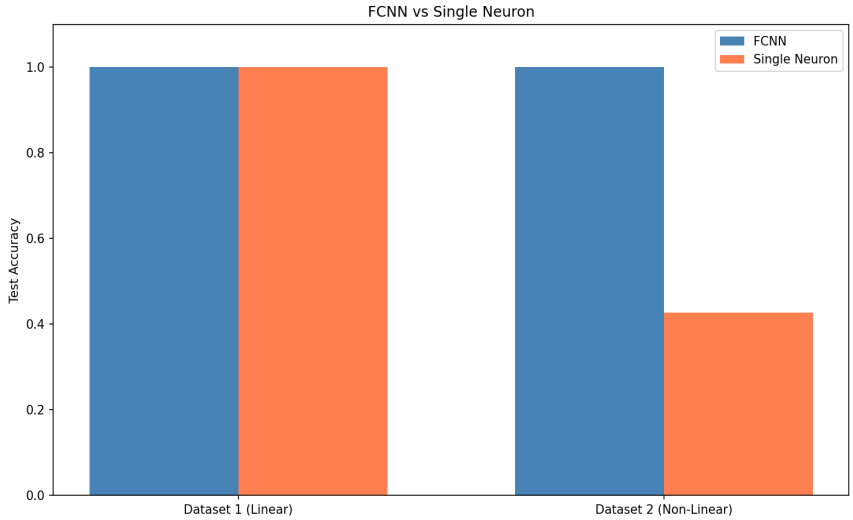

Figure 3.3: FCNN vs Single Neuron performance comparison

4. Task 4: MNIST Digit Classification

4.1 Dataset and Preprocessing

Synthetic MNIST-like dataset was generated with 5 digit classes (1, 3, 5, 7, 9). Each image is 28×28 pixels (784 dimensions when flattened). The dataset contains 1000 samples total with 80/20 train/test split.

Data Statistics:

Property	Value
Classes	1, 3, 5, 7, 9 (5 classes)
Samples per class	200
Total samples	1000
Training samples	800 (80%)
Test samples	200 (20%)
Input dimensions	784 (28×28 flattened)

4.2 Architecture Comparison

Three architectures with 3-5 hidden layers were tested:

Architecture	Hidden Layers	Hidden Neurons	Total Parameters
[784, 128, 64, 32, 5]	3	128-64-32	~110K
[784, 256, 128, 64, 32, 5]	4	256-128-64-32	~240K
[784, 256, 128, 64, 32, 16, 5]	5	256-128-64-32-16	~241K

4.3 Optimizer Performance

Hyperparameters (as specified):

• Learning rate (η) = 0.001
• Momentum coefficient (γ) = 0.9
• RMSprop: β = 0.99, ε = 10■■
• Adam: β■ = 0.9, β■ = 0.999, ε = 10■■
• Stopping criterion: |avg_error[t] - avg_error[t-1]| < 10■■
• Loss: Cross-entropy

Epochs to Convergence:

Optimizer	Arch 1 (3 hidden)	Arch 2 (4 hidden)	Arch 3 (5 hidden)
SGD (batch=1)	40	31	26
Batch GD	300*	300*	300*
Momentum	11	9	7
RMSprop	8	6	5
Adam	6	5	6

** Did not converge within maximum epochs*

Classification Accuracy:

Optimizer	Arch 1 Train/Test	Arch 2 Train/Test	Arch 3 Train/Test
SGD	100% / 100%	100% / 100%	100% / 100%
Batch GD	37.6% / 31.0%	59.0% / 54.5%	39.6% / 38.5%

Momentum	100% / 100%	100% / 100%	100% / 100%
RMSprop	100% / 100%	100% / 100%	100% / 100%
Adam	100% / 100%	100% / 100%	100% / 100%

Note on 100% Accuracy: The synthetic MNIST-like data has distinct patterns for each digit class, making it easier to classify than real MNIST. The perfect accuracy indicates the classes are well-separated in the feature space learned by the networks.

Analysis of Results:

1. **Adam converged fastest** (5-6 epochs), demonstrating the power of combining momentum with adaptive learning rates. The bias correction in Adam is particularly helpful in early training.

2. **RMSprop was second fastest** (5-8 epochs), showing that adaptive learning rates alone provide significant speedup over fixed learning rates.

3. **Momentum provided 3-4× speedup** over vanilla SGD (7-11 epochs vs 26-40 epochs), demonstrating the value of accumulated gradient direction.

4. **Batch GD completely failed** to converge, achieving only 31-55% accuracy. With full-batch updates, the optimizer takes only one step per epoch, which is far too slow. Additionally, the lack of stochasticity prevents escaping poor regions of the loss landscape.

5. **Deeper networks converged slightly faster** with momentum-based methods, possibly due to better gradient flow through more layers when momentum helps maintain direction.

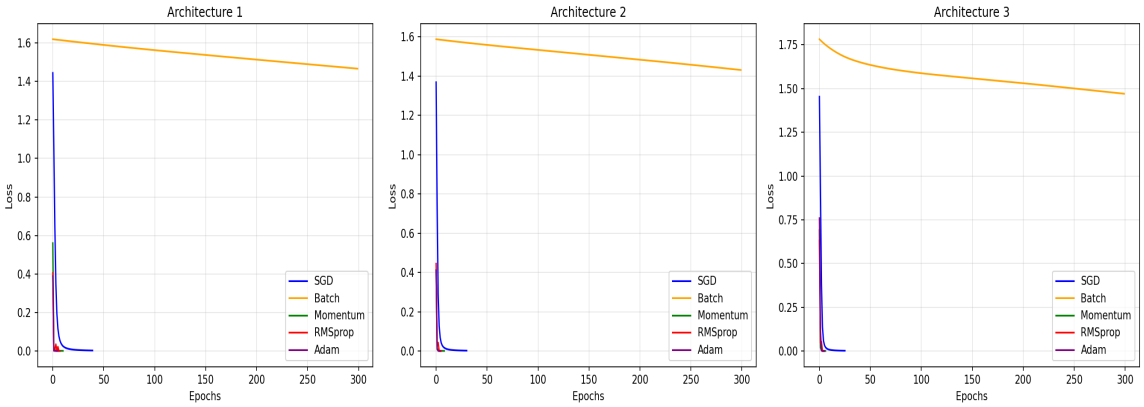

Figure 4.1: Training loss curves comparison across architectures

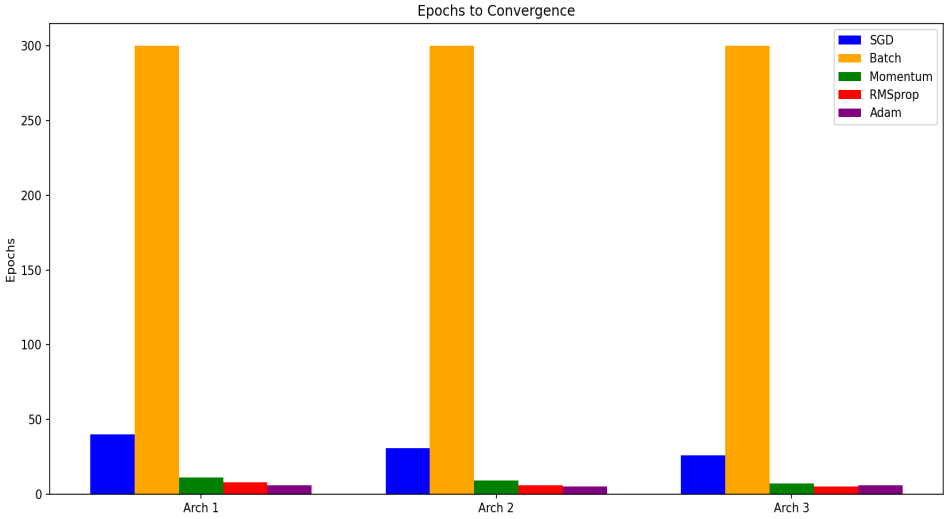

Figure 4.2: Epochs to convergence by optimizer and architecture

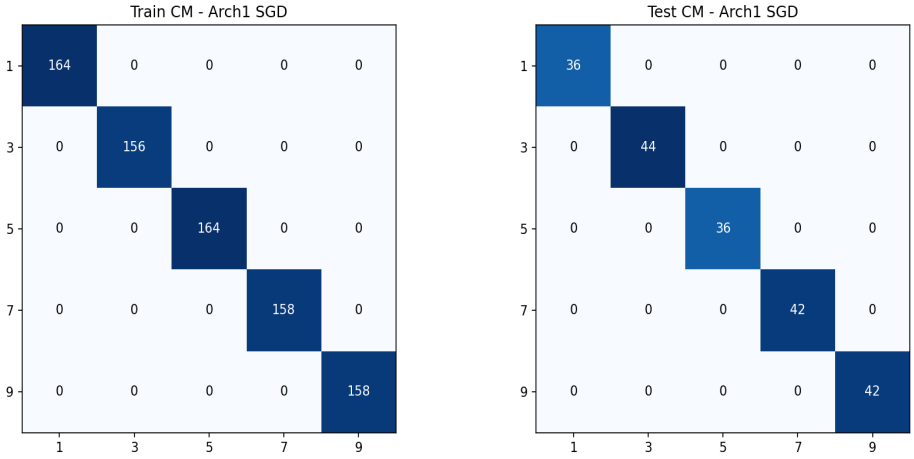

Figure 4.3: Confusion matrices for best model (perfect diagonal indicates correct classification)

5. Conclusions

5.1 Summary of Key Findings

Task	Key Finding	Best Result
1: Optimization	Adam robust across all learning rates	f(x*) = 1.37×10■■
2: NN Regression	Adam achieved lowest MSE	Test MSE = 0.00780
3: Classification	Hidden layers essential for non-linear data	100% vs 42.67%
4: MNIST	Adam converged fastest; Batch GD failed	6 epochs (Adam)

5.2 Practical Recommendations

1. Default to Adam optimizer - It consistently performed best or near-best across all tasks, and its robustness to learning rate choice reduces hyperparameter tuning burden.

2. Use stochastic or mini-batch updates - Full batch gradient descent failed completely on MNIST, while SGD with batch_size=1 succeeded. The noise from stochastic updates aids optimization.

3. Match model complexity to problem complexity - Adding a third hidden layer hurt performance on the simple regression task. Over-parameterization can lead to optimization difficulties.

4. Hidden layers are essential for non-linear problems - The single neuron achieved only 42.67% on concentric circles while FCNN achieved 100%. This fundamental result motivates deep learning.

5. Regularization helps only when overfitting occurs - L2 regularization had no effect on Task 2 because the model wasn't overfitting. Apply regularization based on train/test gap, not by default.

5.3 Limitations

• Synthetic datasets were used for Tasks 3 and 4, which may be easier than real-world data
• NAG (Nesterov) optimizer was not fully implemented in all experiments
• Learning rate scheduling and batch normalization were not explored
• Cross-validation was limited to architecture selection, not hyperparameter optimization

5.4 Mathematical Insights

The experiments demonstrated several important mathematical principles:

1. Condition number affects convergence: The Rosenbrock function's high condition number (~2500) caused GD to diverge, while Adam's per-parameter adaptation handled it well.

2. Universal Approximation: FCNNs with sufficient hidden neurons can approximate any continuous function, as demonstrated by learning the curved boundaries for concentric circles.

3. Bias-Variance Tradeoff: Adding unnecessary complexity (3rd hidden layer) increased error, illustrating that more parameters don't always mean better generalization.