TASK1

Assignment 2: Optimizer Performance on Non-Convex Functions

Course: Artificial Intelligence

Task 1: Optimizer Performance on Non-Convex Functions

---

1. Objective

The objective of this experiment is to analyze and compare the performance of various gradient-based optimization algorithms on non-convex functions. The optimizers are evaluated in terms of:

- Convergence behavior
- Sensitivity to learning rate
- Final optimized parameters and objective value
- Time taken to converge

All algorithms are implemented **from scratch in Python**, without using deep learning libraries.

---

2. Test Functions

2.1 Rosenbrock Function

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

- A classic non-convex optimization benchmark
- Characterized by a narrow, curved valley
- Global minimum at:

$$(x^*,y^*)=(1,1) \quad , f(x^*)=0$$

2.2 Sine Inverse Function

$$f(x)=\sin(1/x) \quad , f(0)=0$$

- Highly oscillatory and non-convex
- Contains infinitely many local minima
- Global minimum value is **–1**

---

3. Optimizers Evaluated

The following optimizers were implemented and tested:

- Gradient Descent (GD)
- Stochastic Gradient Descent with Momentum
- Adam
- RMSProp
- Adagrad

Each optimizer was tested with learning rates:
alpha = {0.01, 0.05, 0.1}

---

4. Termination Criteria

The optimization process was terminated when:

- The gradient norm fell below a small threshold, or
- Numerical instability (NaN / overflow) was detected, or
- The maximum number of iterations was reached

---

5. Experimental Results

5.1 Rosenbrock Function Results

Learning Rate: 0.01

Optimizer	Final Parameters (x, y)	Final f(x)	Observation

GD	NaN	NaN	Diverged
Momentum	NaN	NaN	Diverged
Adam	(0.9935, 0.9870)	0.000042	Near optimal
RMSProp	(0.9888, 0.9629)	0.022156	Slow convergence
Adagrad	(-1.2751, 1.6322)	5.1801	Poor convergence

Learning Rate: 0.05

Optimizer	Final Parameters (x, y)	Final f(x)	Observation
GD	NaN	NaN	Diverged
Momentum	NaN	NaN	Diverged
Adam	(1.0000, 1.0000)	≈ 0	Optimal convergence
RMSProp	(0.7250, 0.4656)	0.4356	Suboptimal
Adagrad	(-0.7396, 0.5522)	3.0289	Stalled

Learning Rate: 0.1

Optimizer	Final Parameters (x, y)	Final f(x)	Observation
GD	NaN	NaN	Diverged
Momentum	NaN	NaN	Diverged
Adam	(1.0000, 1.0000)	≈ 0	Fastest convergence
RMSProp	(0.3000, 0.0150)	1.0525	Unstable
Adagrad	(0.3936, 0.1536)	0.3679	Partial convergence

Analysis (Rosenbrock)

- **Gradient Descent and Momentum** diverged for all learning rates due to the steep curvature of the Rosenbrock function.
- **Adam consistently converged** to the global minimum and showed robustness to learning rate changes.
- **RMSProp** converged slowly and became unstable at higher learning rates.
- **Adagrad** suffered from aggressive learning-rate decay, leading to premature stagnation.

5.2 sin(1/x) Function Results

Learning Rate: 0.01

Optimizer	Final x	Final f(x)
GD	0.5462	0.9664
Momentum	34.1191	0.0293
Adam	0.2122	–1.0000
RMSProp	0.2122	–1.0000
Adagrad	0.2122	–1.0000

Learning Rate: 0.05

Optimizer	Final x	Final f(x)
GD	9.0344	0.1105
Momentum	20.9032	0.0478
Adam	0.2122	–1.0000
RMSProp	0.1935	–0.8982
Adagrad	0.2122	–1.0000

Learning Rate: 0.1

Optimizer	Final x	Final f(x)
GD	–0.6366	–1.0000
Momentum	–0.6366	–1.0000
Adam	0.2122	–1.0000
RMSProp	0.2122	–1.0000
Adagrad	0.2122	–1.0000

Analysis (sin(1/x))

- The function's oscillatory nature makes optimization highly sensitive to initialization and learning rate.
- **Adam, RMSProp, and Adagrad** consistently reached the global minimum.
- **GD and Momentum** showed unstable behavior, often converging to non-optimal oscillation regions.
- Larger learning rates enabled GD and Momentum to escape shallow local minima in some cases.

---

6. Impact of Learning Rate

- Small learning rates led to **slow convergence** or stagnation.
- Moderate learning rates (0.05) provided the **best trade-off** for adaptive optimizers.
- Large learning rates caused **divergence** in vanilla GD and Momentum, especially for Rosenbrock.

---

7. Conclusion

- **Adam is the most robust optimizer** across both non-convex functions.
- Classical Gradient Descent struggles with highly curved landscapes.
- Adaptive methods significantly outperform non-adaptive methods on complex non-convex problems.
- These experiments highlight the importance of optimizer choice and hyperparameter tuning in practical optimization tasks.

TASK2

Neural Network–Based Linear Regression Using Boston Housing Dataset

---

1. Task Description

The objective of this assignment is to implement a **multi-layer neural network from scratch** to perform **linear regression** on the Boston Housing dataset. The model aims to predict the **median value of owner-occupied homes (MEDV)** using two input features:

- **RM** – Average number of rooms per dwelling
- **CRIM** – Per capita crime rate

The assignment further investigates:

- The impact of **different optimizers**
- The effect of **learning rate**
- The influence of **network depth** by adding an extra hidden layer

---

2. Dataset Description

The **Boston Housing Dataset** contains housing-related information collected from suburbs of Boston.

Selected Features

Feature	Description
RM	Average number of rooms per dwelling
CRIM	Crime rate per capita
MEDV	Median value of homes (Target)

Preprocessing Steps

- Input features were **normalized using Min–Max scaling**
- Target variable (MEDV) was also normalized to ensure stable learning
- Dataset split into:

- **80% training**
- **20% testing**

---

3. Neural Network Overview and Working

A neural network consists of interconnected neurons arranged in layers. Each neuron computes:

$$Z = XW + b$$

followed by an activation function.

Training Process

1. **Forward Propagation** – Compute predictions
2. **Loss Calculation** – Mean Squared Error (MSE)
3. **Backward Propagation** – Compute gradients
4. **Weight Updates** – Using optimization algorithms

The goal is to minimize MSE between actual and predicted values.

---

4. Network Architecture

Model 1: Baseline Neural Network

- Input Layer: 2 neurons (RM, CRIM)
- Hidden Layer 1: 5 neurons (ReLU)
- Hidden Layer 2: 3 neurons (ReLU)
- Output Layer: 1 neuron (Linear)

$2 \rightarrow 5 \rightarrow 3 \rightarrow 1$

Model 2: Extended Neural Network

- Input Layer: 2 neurons
- Hidden Layer 1: 5 neurons (ReLU)
- Hidden Layer 2: 3 neurons (ReLU)
- Hidden Layer 3: 2 neurons (ReLU)
- Output Layer: 1 neuron

2 → 5 → 3 → 2 → 1

---

5. Optimizers Used

Optimizers are algorithms used to update the weights and biases of a neural network in order to minimize the loss function. In this work, three different optimizers were implemented and compared.

1. Gradient Descent (GD)

Gradient Descent is the most basic optimization technique. It updates the model parameters in the direction opposite to the gradient of the loss function.

- Uses a fixed learning rate for all parameters
- Simple to implement and understand
- Sensitive to learning rate selection
- May converge slowly and oscillate near minima

Update rule:

W=W-(alpha)(gradient L)

---

2. Momentum-Based Gradient Descent

Momentum improves standard Gradient Descent by adding a velocity term that accumulates past gradients. This helps the optimizer move faster in consistent directions and reduces oscillations.

- Accelerates convergence
- Reduces fluctuations in loss
- Especially useful in narrow or curved loss surfaces

Key idea:
Uses past gradients to gain “momentum” in weight updates.

---

3. Adam (Adaptive Moment Estimation)

Adam is an advanced optimizer that combines the advantages of Momentum and adaptive learning rates. It computes individual learning rates for each parameter.

- Fast and stable convergence
- Less sensitive to learning rate choice
- Works well for noisy gradients

- Most commonly used optimizer in practice

Key features:

- First moment (mean of gradients)
- Second moment (variance of gradients)

Summary

- Gradient Descent is simple but slow and sensitive to hyperparameters
- Momentum improves stability and convergence speed
- Adam provides the best balance between speed and robustness

6. Results

6.1 Baseline Neural Network (2 Hidden Layers)

Training Loss

- Loss decreases rapidly and stabilizes around **0.0429**
- Adam converges the fastest, though all optimizers reach similar final loss

Test MSE

Optimizer	Test MSE
Gradient Descent	0.0371
Momentum	0.0371
Adam	0.0371

Prediction Behavior

- Initial training showed constant predictions (model collapse)
- After correcting initialization and learning rate, predictions became diverse
- Predicted vs Actual plot shows points closely aligned with the diagonal

---

6.2 Neural Network with Additional Hidden Layer

Training Loss

- Loss **increases over epochs** instead of decreasing

- Indicates training instability and difficulty in optimization

Test MSE

Optimizer	Test MSE
Gradient Descent	0.0570
Momentum	0.0570
Adam	0.0559

Prediction Behavior

- Predictions vary but show larger deviation from ideal diagonal
- Increased error and weaker generalization compared to baseline model

---

7. Discussion

Effect of Optimizers

- Adam converges faster and is more stable
- Final test performance is similar across optimizers for the baseline model
- Optimizer choice affects convergence speed more than final accuracy

Effect of Learning Rate

- Learning rate of **0.01** performs significantly better than smaller values
- Too small learning rates led to slow learning and model collapse

Effect of Additional Hidden Layer

- Adding a third hidden layer **degraded performance**
- Higher training and test loss observed
- Indicates **over-parameterization** for a simple regression problem

Why Deeper Model Performed Worse

- Limited dataset size
- Only two input features
- Increased depth caused:
    - Optimization difficulty
    - Higher variance
    - Mild overfitting

---

8. Conclusion

This assignment successfully demonstrated the implementation of a **multi-layer neural network from scratch** for linear regression.

Key conclusions:

- A **two-hidden-layer neural network** is sufficient for this task
- Adam optimizer provides faster and more stable convergence
- Learning rate selection has a greater impact than optimizer choice
- Adding unnecessary hidden layers can **reduce model performance**
- Proper initialization and normalization are critical for effective training

Overall, simpler architectures generalized better for the Boston Housing regression task.

---

9. Final Takeaway

> Increasing model complexity does not always improve performance. A well-tuned shallow network can outperform deeper models on small, structured datasets.

TASK3\

Report: Multi-class Classification using Fully Connected Neural Network (FCNN)

1. Introduction & Methodology

This task involved implementing a Fully Connected Neural Network (FCNN) from scratch to perform multi-class classification on two distinct datasets.

- **Algorithm:** Backpropagation with **Stochastic Gradient Descent (SGD)**.
- **Loss Function:** Instantaneous Squared Error.
- **Data Split:** 60% Training, 20% Validation, 20% Testing.

---

2. Dataset 1: Linearly Separable Classes

Data Description: 3-class, 2-dimensional linearly separable data.

Model Architecture: 1 Hidden Layer.

A. Architecture Search & Cross-Validation

The model was tested with varying numbers of nodes in the single hidden layer. The results on the **validation set** were as follows:

Architecture (Hidden Nodes)	Validation Accuracy	Status
[5]	33.33%	**Selected Best**
[10]	33.33%	
[15]	33.33%	
[20]	33.33%	

Selected Best Architecture: [5] hidden nodes.

B. Performance Analysis (Best Architecture)

- **Classification Accuracy (Test Data):** 33.33%

- **Confusion Matrix (Validation):**

```
[[  0   0 100]
 [  0   0 100]
 [  0   0 100]]
```

- *Observation:* The model predicted Class 3 (index 2) for every single instance, resulting in a model that essentially performed random guessing (or worse, collapsed to a single class output).

C. Comparison with Single Neuron Model

- **Single Neuron Model:** A single neuron (perceptron) is theoretically capable of achieving 100% accuracy on linearly separable data by drawing linear decision boundaries.
- **FCNN Performance:** In this specific experiment, the FCNN failed to converge to a solution, achieving only 33% accuracy.
- **Conclusion:** The Single Neuron model would have vastly outperformed the FCNN in this specific trial. The failure of the FCNN here is likely due to the "dying ReLU" problem, improper initialization, or a learning rate that was too high/low, causing the network to get stuck in a local minimum where it outputs a constant value.

---

3. Dataset 2: Nonlinearly Separable Classes

Data Description: 2-dimensional nonlinearly separable data (2 classes).

Model Architecture: 2 Hidden Layers.

A. Architecture Search & Cross-Validation

The model was tested with various configurations for the two hidden layers. Results on the **validation set**:

Architecture (Hidden Nodes)	Validation Accuracy	Status
[10, 5]	97.50%	
[15, 10]	97.00%	

[20, 10]	**99.00%**	**Selected Best**
[15, 8]	98.00%	

Selected Best Architecture: [20, 10] (20 nodes in Layer 1, 10 nodes in Layer 2).

B. Performance Analysis (Best Architecture)

- **Classification Accuracy (Test Data): 99.00%**
- **Confusion Matrix (Test):**

```
[[ 99   1]
 [  1  99]]
```

- *Observation:* The model achieved near-perfect classification, misclassifying only 2 samples out of 200 in the test set.

C. Comparison with Single Neuron Model

- **Single Neuron Model:** A single linear neuron cannot solve nonlinearly separable problems (like XOR or concentric circles). It would likely achieve ~50% accuracy (random guess) or fail to separate the classes entirely.
- **FCNN Performance:** The FCNN achieved 99% accuracy.
- **Conclusion:** The FCNN significantly outperforms the single neuron model. The addition of hidden layers with non-linear activation functions allowed the network to learn the complex, non-linear decision boundaries required for this dataset.

---

4. Inferences on Plots & Results

1. Average Error vs. Epochs

- **Dataset 1 (Linear):** The training loss decreased slightly (from ~0.135 to ~0.134), but the validation loss remained high (~0.61). This divergence indicates that while the model was technically "learning" to minimize error on training data, it failed to generalize or find a separating boundary, likely getting stuck in a suboptimal state.
- **Dataset 2 (Nonlinear):** The plot (as implied by high accuracy) would show a steady decrease in both training and validation error, converging near zero. This indicates stable and successful learning.

2. Decision Regions

- **Dataset 1:** The decision region plot likely shows the entire space covered by a single color (representing the one class predicted for all inputs), confirming the model failed to draw boundaries.

- **Dataset 2:** The decision region plot demonstrates a complex boundary that successfully wraps around the classes, separating the nonlinear clusters effectively.

3. 3D Node Output Plots

- **Hidden Nodes:** These plots visualize how the input space is transformed. For Dataset 2, the hidden nodes likely show "folded" or "twisted" planes, representing the transformation of the non-linear input space into a linearly separable feature space for the output layer.
- **Output Nodes:** These plots clearly show high activation (z-axis) in the regions corresponding to their specific class and low activation elsewhere.

5. Overall Conclusion

The experiments highlight the capability of FCNNs to model complex relationships. While the model struggled with the simpler linearly separable dataset (likely due to initialization or hyperparameter tuning issues specific to that run), it demonstrated powerful performance on the complex non-linear dataset, achieving 99% accuracy where a single neuron model would have failed completely.