# Study of Optimization Algorithms and Neural Network Training

Artificial Intelligence Assignment 2

Course: Artificial Intelligence

**Abstract**

Optimization algorithms play a fundamental role in modern machine learning, particularly in scenarios involving non-convex loss landscapes. This report presents a detailed experimental and theoretical study of gradient-based optimization methods applied to non-convex mathematical functions and neural network training. In Task 1, multiple optimizers are analyzed on benchmark non-convex functions to study convergence behavior, robustness, and sensitivity to learning rates. In Task 2, a multi-layer neural network is implemented from scratch for linear regression on the Boston Housing dataset, and the effects of optimizer choice, learning rate, network depth, and regularization are examined. The results highlight the advantages of adaptive optimization techniques over classical gradient descent.

## 1 Introduction

Many problems in artificial intelligence and machine learning can be formulated as optimization problems, where the objective is to minimize a loss or cost function with respect to model parameters. While convex optimization problems admit unique global minima, practical machine learning models typically involve non-convex loss surfaces characterized by local minima, saddle points, and flat regions.

Gradient-based optimization algorithms are widely employed to solve such problems due to their scalability and simplicity. However, the performance of these algorithms depends heavily on the structure of the objective function and the choice of hyperparameters. This report investigates these aspects through two complementary tasks: optimization of non-convex functions and training of neural networks for regression.

## 2 Task 1: Optimization of Non-Convex Functions

### 2.1 Objective

The primary objective of Task 1 is to study and compare the convergence behavior of different gradient-based optimization algorithms when applied to non-convex functions. Particular emphasis is placed on convergence speed, stability, and the influence of learning rate on optimization performance.

### 2.2 Non-Convex Functions Considered

#### 2.2.1 Rosenbrock Function

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

The Rosenbrock function is a standard benchmark in optimization literature. Although it possesses a unique global minimum at $(1, 1)$, the function features a narrow, curved valley that poses significant challenges for gradient-based methods. Optimizers often oscillate across the valley walls, leading to slow convergence.

### 2.2.2 Highly Oscillatory Function

$$f(x) = \sin\left(\frac{1}{x}\right), \quad x \neq 0, \quad f(0) = 0$$

This function exhibits infinitely many oscillations near the origin, resulting in numerous local minima and rapid gradient variations. It is particularly useful for evaluating the robustness and stability of optimization algorithms in highly non-smooth regions.

## 2.3 Gradient Descent Framework

For a general objective function $f(\theta)$, gradient descent updates parameters iteratively according to:

$$\theta_{k+1} = \theta_k - \alpha \nabla f(\theta_k)$$

where $\alpha > 0$ is the learning rate. The learning rate controls the step size taken in the direction of the negative gradient and plays a crucial role in determining convergence behavior.

## 2.4 Optimizers Studied

### 2.4.1 Gradient Descent (GD)

Gradient Descent updates parameters solely based on the current gradient. While conceptually simple, GD is highly sensitive to the choice of learning rate and often converges slowly in ill-conditioned or non-convex settings.

### 2.4.2 Gradient Descent with Momentum

Momentum introduces a velocity term that accumulates gradients over time:

$$v_k = \beta v_{k-1} - \alpha \nabla f(\theta_k)$$

$$\theta_{k+1} = \theta_k + v_k$$

Here, $\beta \in (0, 1)$ is the momentum coefficient. Momentum accelerates convergence along consistent descent directions and reduces oscillations in narrow valleys.

### 2.4.3 Adagrad Optimizer

Adagrad adapts the learning rate individually for each parameter based on the accumulation of squared gradients:

$$G_k = \sum_{i=1}^{k} (\nabla f(\theta_i))^2$$

$$\theta_{k+1} = \theta_k - \frac{\alpha}{\sqrt{G_k + \epsilon}} \nabla f(\theta_k)$$

Adagrad is effective for sparse gradients but suffers from aggressive learning rate decay, which may lead to premature convergence.

### 2.4.4 RMSProp Optimizer

RMSProp addresses Adagrad's diminishing learning rate issue by using an exponential moving average of squared gradients:

$$G_k = \beta G_{k-1} + (1 - \beta)(\nabla f(\theta_k))^2$$

$$\theta_{k+1} = \theta_k - \frac{\alpha}{\sqrt{G_k + \epsilon}} \nabla f(\theta_k)$$

This allows RMSProp to maintain stable learning rates throughout training, making it suitable for non-convex optimization.

### 2.4.5 Adam Optimizer

Adam (Adaptive Moment Estimation) combines the advantages of momentum and RMSProp by maintaining first and second moment estimates of gradients:

$$m_k = \beta_1 m_{k-1} + (1 - \beta_1)\nabla f(\theta_k)$$

$$v_k = \beta_2 v_{k-1} + (1 - \beta_2)(\nabla f(\theta_k))^2$$

Bias-corrected estimates are given by:

$$\hat{m}_k = \frac{m_k}{1 - \beta_1^k}, \quad \hat{v}_k = \frac{v_k}{1 - \beta_2^k}$$

The parameter update rule is:

$$\theta_{k+1} = \theta_k - \alpha \frac{\hat{m}_k}{\sqrt{\hat{v}_k} + \epsilon}$$

Adam is widely used due to its fast convergence and robustness across a wide range of optimization problems.

## 2.5 Experimental Setup

All optimizers were evaluated using learning rates $\alpha = 0.01, 0.05, 0.1$. The stopping criterion was defined using either a maximum number of iterations or a threshold on the gradient norm. Convergence curves and execution times were recorded for comparison.

## 2.6 Observations and Analysis

Basic Gradient Descent exhibited slow convergence and high sensitivity to learning rate selection. Momentum significantly improved convergence on the Rosenbrock function by reducing oscillatory behavior. Adagrad converged rapidly initially but stalled due to diminishing learning rates. RMSProp and Adam demonstrated superior stability and faster convergence, particularly on highly non-convex and oscillatory functions.

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

## 3.1 Objective

The objective of Task 2 is to design, implement, and analyze a multi-layer neural network trained from scratch to perform a regression task. Unlike classical linear regression, which directly estimates a linear mapping between inputs and output, a neural network introduces multiple layers and non-linear transformations. The goal is to study how gradient-based optimization algorithms behave in a high-dimensional, non-convex loss landscape arising from neural network training.

## 3.2 Problem Formulation

Given a dataset consisting of input-output pairs

$$\{(\mathbf{x}_i, y_i)\}_{i=1}^N,$$

where $\mathbf{x}_i \in \mathbb{R}^2$ represents the input features (RM and CRIM) and $y_i \in \mathbb{R}$ represents the target value (MEDV), the task is to learn a function $f_\theta(\mathbf{x})$ parameterized by $\theta$ such that

$$f_\theta(\mathbf{x}_i) \approx y_i.$$

This is achieved by minimizing a regression loss function over the training dataset.

### 3.3 Dataset and Preprocessing

The Boston Housing dataset is used with two selected features:

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

Since gradient-based optimization is sensitive to feature scale, normalization is applied:

$$x_{\text{norm}}^{(j)} = \frac{x^{(j)} - \mu_j}{\sigma_j},$$

where $\mu_j$ and $\sigma_j$ denote the mean and standard deviation of feature $j$, respectively. The dataset is split into 80% training data and 20% testing data.

### 3.4 Neural Network Architecture

The neural network used in this task consists of the following layers:

- Input layer: 2 neurons
- First hidden layer: 5 neurons with ReLU activation
- Second hidden layer: 3 neurons with ReLU activation
- Output layer: 1 neuron with linear activation

Let $\mathbf{x} \in \mathbb{R}^2$ denote an input vector.

### 3.5 Forward Propagation

The forward pass computes the network output as follows:

#### 3.5.1 First Hidden Layer

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$
$$\mathbf{a}^{(1)} = \text{ReLU}(\mathbf{z}^{(1)})$$

#### 3.5.2 Second Hidden Layer

$$\mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)}$$
$$\mathbf{a}^{(2)} = \text{ReLU}(\mathbf{z}^{(2)})$$

#### 3.5.3 Output Layer

$$\hat{y} = \mathbf{W}^{(3)}\mathbf{a}^{(2)} + b^{(3)}$$

Since this is a regression task, no activation function is applied at the output layer.

### 3.6 Activation Function

The Rectified Linear Unit (ReLU) activation function is defined as:

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

Its derivative, required for backpropagation, is given by:

$$\text{ReLU}'(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

ReLU introduces non-linearity into the network and helps avoid the vanishing gradient problem.

### 3.7 Loss Function

The Mean Squared Error (MSE) loss function is used:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

This loss penalizes large prediction errors more strongly and is widely used in regression problems.

### 3.8 Backpropagation

Training the neural network involves computing gradients of the loss with respect to all parameters using the chain rule.

#### 3.8.1 Gradient at Output Layer

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = -2(y - \hat{y})$$

#### 3.8.2 Gradients for Output Layer Parameters

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(3)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot (\mathbf{a}^{(2)})^T$$

$$\frac{\partial \mathcal{L}}{\partial b^{(3)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}}$$

#### 3.8.3 Backpropagation to Hidden Layers

Gradients are propagated backward layer by layer:

$$\delta^{(2)} = \left(\mathbf{W}^{(3)}\right)^T \frac{\partial \mathcal{L}}{\partial \hat{y}} \odot \mathrm{ReLU}'(\mathbf{z}^{(2)})$$

$$\delta^{(1)} = \left(\mathbf{W}^{(2)}\right)^T \delta^{(2)} \odot \mathrm{ReLU}'(\mathbf{z}^{(1)})$$

Corresponding weight and bias gradients are computed accordingly.

### 3.9 Training Procedure

The neural network is trained using the following steps:

1. Initialize weights and biases randomly
2. Perform forward propagation
3. Compute the loss
4. Perform backpropagation
5. Update parameters using an optimizer
6. Repeat for multiple epochs

### 3.10 Evaluation Metrics

Model performance is evaluated using:

- Mean Squared Error (MSE) on test data
- Predicted vs. actual value plots

### 3.11 Bonus Experiments

#### 3.11.1 Effect of Network Depth

A third hidden layer with two neurons was added. While deeper architectures increased representational capacity, only marginal performance improvement was observed, suggesting that the original network was sufficient for this task.

#### 3.11.2 L2 Regularization

L2 regularization modifies the loss function as:

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda \sum_{l} \|\mathbf{W}^{(l)}\|_2^2$$

This discourages large weight values and improves generalization by reducing overfitting.

### 3.12 Discussion

The experiments demonstrate that neural network training results in a highly non-convex optimization problem. Adaptive optimizers such as Adam significantly improve convergence speed and stability compared to vanilla gradient descent. Proper data normalization and regularization were found to be crucial for stable and effective learning.

## 4 Conclusion

This report presents a comprehensive study of optimization algorithms applied to non-convex functions and neural network training. Adaptive optimizers such as RMSProp and Adam consistently outperformed classical gradient descent methods in terms of convergence speed and robustness. In neural network regression, optimizer selection, learning rate tuning, and regularization were shown to have a significant impact on training stability and predictive performance. These findings emphasize the importance of understanding optimization dynamics in machine learning applications.