Homework1 Report: Linear Regression Implementation

Course Name: AI and Machine Learning
Course Code: SDM274

Submitted By:

- **Name:** `Han Lubing`
- **Student ID:** `12210262`
- **Email:** `12210262@mail.sustech.edu.cn`

Date of Submission: `September 28, 2025`

Abstract

This report documents the design, implementation, and evaluation of a Linear Regression model built from scratch using Python and NumPy. The project aims to provide a deep understanding of the foundational mechanics of model training by implementing various gradient descent optimization algorithms and feature scaling techniques. The methodology involved creating a `LinearRegression` class encompassing Mean Squared Error (MSE) as the loss function, and three distinct weight update methods: Batch Gradient Descent (BGD), Stochastic Gradient Descent (SGD), and Mini-Batch Gradient Descent (MBGD). Furthermore, the class includes min-max and mean normalization for data preprocessing. Key outcomes include a functional linear regression model and a comparative analysis of the implemented optimization and normalization techniques on a synthetic dataset. The findings demonstrate the trade-offs between different gradient descent methods in terms of convergence stability and speed, and confirm that feature normalization contributes to more efficient training. This project serves as a practical exercise in understanding the core principles of iterative optimization in machine learning.

Keywords: `Linear Regression`, `Gradient Descent`, `SGD`, `BGD`, `MBGD`, `Normalization`

1. Introduction

1.1. Project Background & Motivation

Linear regression is a fundamental algorithm in machine learning and statistics used for modeling the relationship between a dependent variable and one or more independent variables. At its core, training a linear regression model involves finding the optimal parameters (weights) that minimize a loss function. The most common method for this optimization is gradient descent. This project addresses the need for a practical understanding of how this optimization works by implementing the model from the ground up, without relying on high-level machine learning libraries. Understanding the nuances of different gradient descent variants (Batch, Stochastic, and

Mini-Batch) and the importance of data preprocessing techniques like normalization is crucial for building and tuning more complex models in the future.

1.2. Summary of the Project

This project involves the implementation of a `LinearRegression` class in Python, using only the NumPy library for numerical operations. The class is equipped with the Mean Squared Error (MSE) loss function and three distinct optimization algorithms: Batch Gradient Descent (BGD), Stochastic Gradient Descent (SGD), and Mini-Batch Gradient Descent (MBGD). It also includes methods for min-max and mean normalization to preprocess the input data. The implemented model was then tested on a synthetically generated dataset to analyze and compare the performance, convergence behavior, and impact of the different optimizers and normalization techniques. The final results, including the learned regression line, were visualized using Matplotlib.

2. Problem Description and Project Objectives

2.1 Problem Description

The primary problem is to move beyond a theoretical understanding of linear regression and gain hands-on experience with its internal mechanics. This involves manually implementing the key components of the model training pipeline: the loss function, the gradient calculation, the parameter update rules for different optimization strategies, and common data preprocessing steps. The goal is to build a robust `LinearRegression` class that can be trained using various configurations to solve a regression problem.

2.2Primary Objective:

The specific aims of my project are as follows:

- To implement a `LinearRegression` class from scratch using Python and NumPy.
- To utilize the Mean Squared Error (MSE) as the cost function for evaluating the model's performance.
- To implement and integrate three distinct gradient descent update methods:
    - Batch Gradient Descent (BGD)
    - Stochastic Gradient Descent (SGD)
    - Mini-Batch Gradient Descent (MBGD)
- To add min-max normalization and mean normalization as preprocessing options within the class.
- To evaluate the effectiveness and behavior of each implemented method by training the model on a known, synthetic dataset.
- To analyze and visualize the training process and final results using Matplotlib.

3. Design & Methodology

3.1. System Architecture / Overall Design

The project is implemented as a single, self-contained Python class named `LinearRegression`. This object-oriented approach encapsulates all the necessary data, hyperparameters, and functionalities, including model training, prediction, and data preprocessing. The architecture is designed to be modular and flexible, allowing a user to easily instantiate a model and train it using different optimization strategies and normalization techniques.

The typical workflow for utilizing the class follows these steps:

1. **Instantiation**: An instance of the `LinearRegression` class is created, optionally configuring hyperparameters like `learning_rate`, `epoch`, and `batch_size`.
2. **Data Normalization (Optional)**: The input features (`X`) can be scaled using the `min_max_normalization` or `mean_normalization` methods before training.
3. **Training**: The model is trained by calling one of the fitting methods (`BGD`, `SGD`, or `MBGD`). This process initializes and iteratively updates the model's internal weights to minimize the loss function.
4. **Prediction**: Once trained, the `predict` method can be used to generate output for new input data.

Besides, we can use `Ordinary Least Squares (OLS)` to get the analytical solution. By compared with the standard solution, we can analyze the method we use.

3.2. Implementation Details

3.2.1 Model Definition and Weight Initialization

The model assumes input features $X \in \mathbb{R}^{m \times n}$ and output $y \in \mathbb{R}^m$. The linear model is defined as:

$$\hat{y} = \mathbf{w}^T \mathbf{x} + b$$

In matrix form, adding a bias column:

$$\hat{y} = \mathbf{X}'\mathbf{W}, \quad \mathbf{X}' = [\mathbf{1}, X], \quad \mathbf{W} = [b, \mathbf{w}]^T$$

Weights are initialized using a standard normal distribution:

$$\mathbf{W} \sim \mathcal{N}(0, 1)$$

The code is implemented as below.

```
def preprocess(self, X):
    if len(X.shape) == 1:
        X = np.atleast_2d(X)
    m, n = X.shape
    X_ = np.zeros((m, n+1))
    X_[:, 0] = 1
    X_[:, 1:] = X
    return X_

def init_weights(self):
    self.weights = np.random.randn(self.n_features + 1, 1)
```

3.2.2. Loss Function and Gradient

To quantify the model's error, we use the Mean Squared Error (MSE) cost function, `J(w)`. The implementation uses half of the MSE, which simplifies the derivative for the gradient calculation without changing the location of the minimum.

$$J(\mathbf{W}) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2$$

This is implemented in the `MSE_loss` method:

```
def MSE_loss(self, y, y_pred):
    return np.mean((y - y_pred) ** 2)/2
```

The training process relies on gradient descent, which iteratively adjusts the weights to minimize this cost function. The core of this is the gradient, ∇J(w), which points in the direction of the steepest ascent of the cost function. The vectorized formula for the gradient is:

$$\nabla_{\mathbf{W}} J(\mathbf{W}) = \frac{1}{m} \mathbf{X'}^T (\hat{\mathbf{y}} - \mathbf{y})$$

The `gradient` method implements this formula efficiently:

```
def gradient(self, X, y, y_pred):
    X_ = self.preprocess(X)
    return - X_.T @ (y.reshape(y_pred.shape)- y_pred)/X.shape[0]
```

3.2.3. Gradient Descent Optimization Methods

The weights are updated according to the rule: **w** := **w** - η ∇J(w), where η is the learning rate. The key difference between the implemented optimizers is the amount of data used to compute the gradient in each step.

- **Stochastic Gradient Descent (SGD)**: computes the gradient on a single, randomly selected training example for each update. This is computationally faster per step but results in a noisier, more erratic convergence path.

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} J(\mathbf{W}_i)$$

```
def SGD(self, X, y):
    for n in range(self.epoch):
        index = np.random.choice(X.shape[0])
        X_choose = X[index]
        y_choose = y[index]
        y_pred = self.predict(X_choose)
        loss = self.MSE_loss(y_choose, y_pred)
        grad = self.gradient(X_choose, y_choose, y_pred)
        self.weights -= self.lr * grad
    return self.weights
```

- **Batch Gradient Descent (BGD)**: calculates the gradient using the entire training dataset at once. This leads to a stable, direct convergence path.

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} J(\mathbf{W})$$

```
def BGD(self, X, y):
    for n in range(self.epoch):
        y_pred = self.predict(X)
        loss = self.MSE_loss(y, y_pred)
        grad = self.gradient(X, y, y_pred)
        self.weights -= self.lr * grad
    return self.weights
```

- **Mini-Batch Gradient Descent (MBGD)**: offers a balance by calculating the gradient on small, random subsets of the data. This captures the stability of BGD and the efficiency of SGD.

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} J(\mathbf{W}_{\text{batch}})$$

```python
def MBGD(self, X, y):
    for n in range(self.epoch):
        for X_batch, y_batch in self.iter_mini_batch(X, y):
            y_pred = self.predict(X_batch)
            loss = self.MSE_loss(y_batch, y_pred)
            grad = self.gradient(X_batch, y_batch, y_pred)
            self.weights -= self.lr * grad
    return self.weights
```

And we use `iter_mini_batch` method to produce random subsets of data.

```python
def iter_mini_batch(self, X, y):
    indices = list(range(X.shape[0]))
    np.random.shuffle(indices)
    for i in range(0, X.shape[0], self.batch_size):
        yield X[indices[i:i+self.batch_size]],
y[indices[i:i+self.batch_size]]
```

3.2.4. Data Normalization

To help the gradient descent algorithms converge faster and prevent gradient explosion or vanishing, two common normalization techniques were implemented.

- **Min-Max Normalization**:

$$X' = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$$

```python
  def min_max_normalization(self, X):
      return (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
```

- **Mean Normalization**:

$$X' = \frac{X - \bar{X}}{\sigma_X}$$

```python
def mean_normalization(self, X):
    return (X - X.mean(axis=0)) / X.std(axis=0)
```

When training on normalized data, the learned weights correspond to the normalized scale. To map them back to the original feature scale, we compute:

- **Min-Max Normalization Mapping**:

$$w_i^{\text{orig}} = \frac{w_i^{\text{norm}}}{X_{\max,i} - X_{\min,i}}, \quad b^{\text{orig}} = b^{\text{norm}} - \sum_i w_i^{\text{orig}} X_{\min,i}$$

- **Mean Normalization Mapping**:

$$w_i^{\text{orig}} = \frac{w_i^{\text{norm}}}{\sigma_i}, \quad b^{\text{orig}} = b^{\text{norm}} - \sum_i w_i^{\text{orig}} \bar{X}_i$$

Where w_i^{norm} and b^{norm} are the weights and bias learned on normalized data, σ_i is the standard deviation, and $\bar{X}_i$ is the mean of the i-th feature.

```
def mapping(self, X, method = None):
    if method == "min_max":
        w_norm = self.weights[1:] / (X.max(axis=0) - X.min(axis=0))
        b_norm = self.weights[0] - w_norm @ X.min(axis=0)
    elif method == "mean":
        w_norm = self.weights[1:] / X.std(axis=0)
        b_norm = self.weights[0] - w_norm @ X.mean(axis=0)
    return np.vstack([b_norm, w_norm])
```

3.2.5. Ordinary Least Squares (OLS)

For linear regression, an analytic solution can be computed using OLS:

$$\mathbf{W} = (\mathbf{X'}^T\mathbf{X'})^{-1}\mathbf{X'}^T\mathbf{y}$$

This method does not require iterative training and provides the optimal weights for small to medium datasets.

```
def OLS(self, X, y):
    X_ = self.preprocess(X)
    self.weights = np.linalg.inv(X_.T @ X_) @ X_.T @ y.reshape(-1, 1)
    return self.weights
```

4. Testing & Results

4.1. Testing

For testing, a synthetic dataset was generated to simulate a simple linear relationship with added noise. Specifically:

```
X_train = np.arange(100).reshape(100,1)
a, b = 1, 10
y_train = a * X_train + b + np.random.normal(0, 5, size=X_train.shape)
y_train = y_train.reshape(-1)
```

This produces training data following the underlying model

$$y = a \cdot x + b + \epsilon,$$

with $a = 1, b = 10$, and Gaussian noise $\epsilon \sim \mathcal{N}(0, 5))$

The testing procedure involved training the implemented **SGD**, **BGD**, and **MBGD** methods on this dataset and comparing their learned parameters with the **Ordinary Least Squares (OLS)** closed-form solution:

$$\hat{\mathbf{w}} = (X^T X)^{-1} X^T y$$

The OLS result was treated as the benchmark. To further validate correctness, the learned regression lines were visualized against the training points and the OLS line, ensuring that the gradient-based methods converged properly and that normalization improved numerical stability.

4.2. Results and Analysis

- **Table: Functional Test Results**

  The core functions of the linear regression class were tested successfully. Each training method (SGD, BGD, MBGD) produced parameters close to the OLS benchmark. The test outcomes are summarized in the following table:

Method	Normalization	Bias(w_0)	Weight(w_1)	Actual Result
OLS	\	**9.5143**	**0.9987**	
SGD	None	9.0940	1.0514	Close
SGD	Min_Max	9.6732	1.0002	Close
SGD	Mean	9.7421	0.9935	Close
BGD	None	9.0492	1.0057	Close
BGD	Min_Max	9.5143	0.9987	**Same**
BGD	Mean	9.5143	0.9987	**Same**
MBGD	None	9.5124	0.8753	Close
MBGD	Min_Max	9.5643	0.9982	Close
MBGD	Mean	9.6872	0.9952	Close

- **Figures: Visualization Results**

  The training data, regression lines learned by different methods, and the OLS benchmark line were plotted for visual comparison. Separate figures were also generated to illustrate the effect of normalization methods (none, min-max, mean) on the convergence behavior.

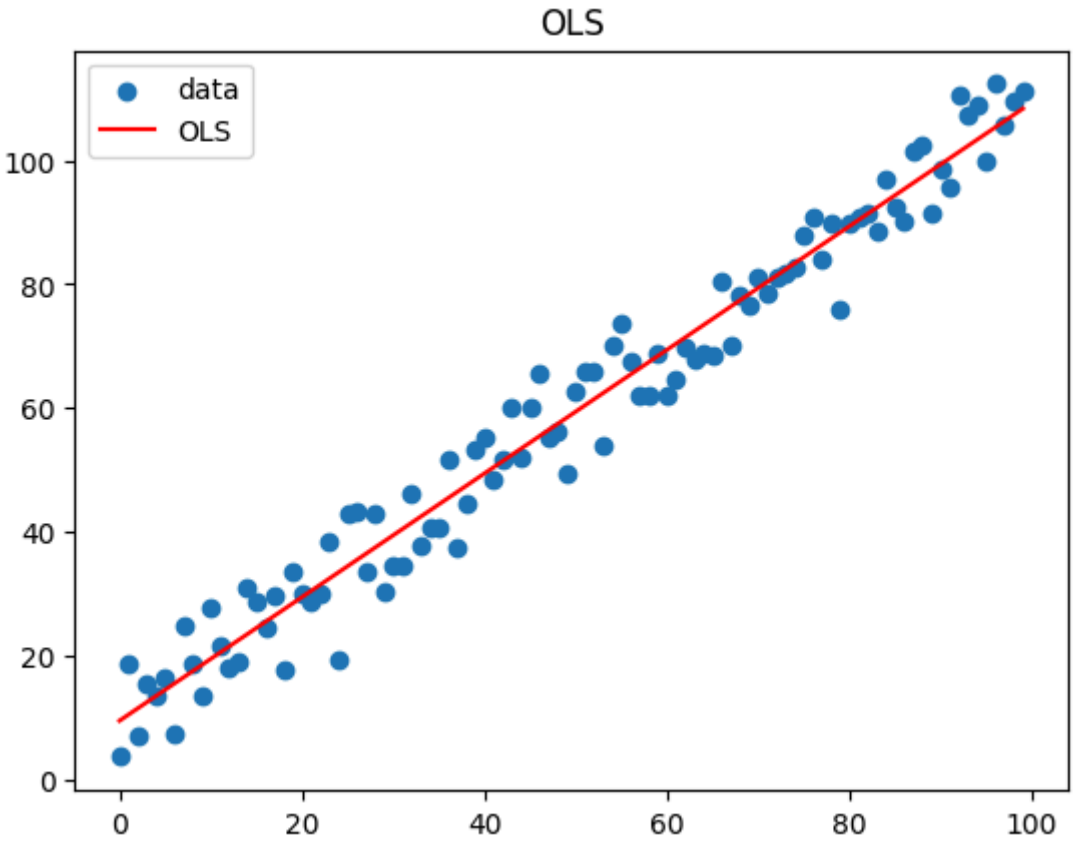

Fig.1 Regression results using OLS method

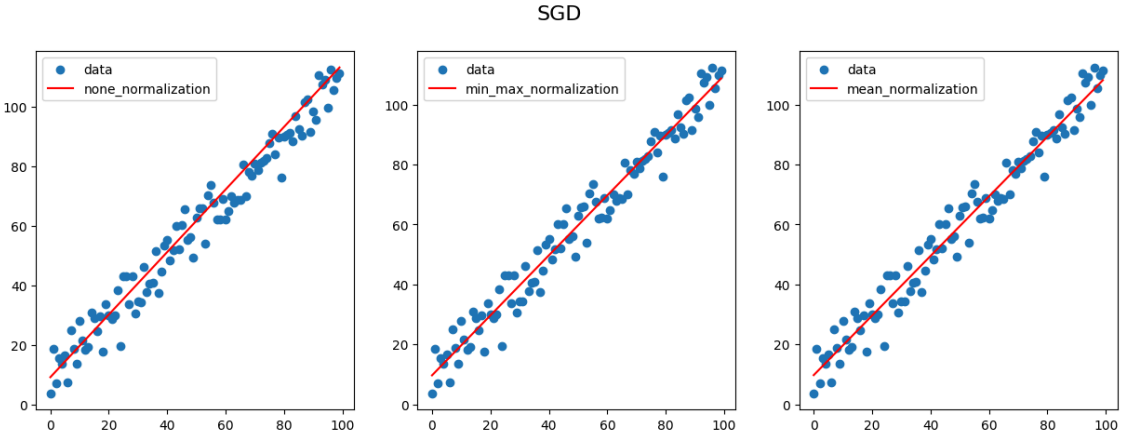

Fig.2 Regression results using SGD method

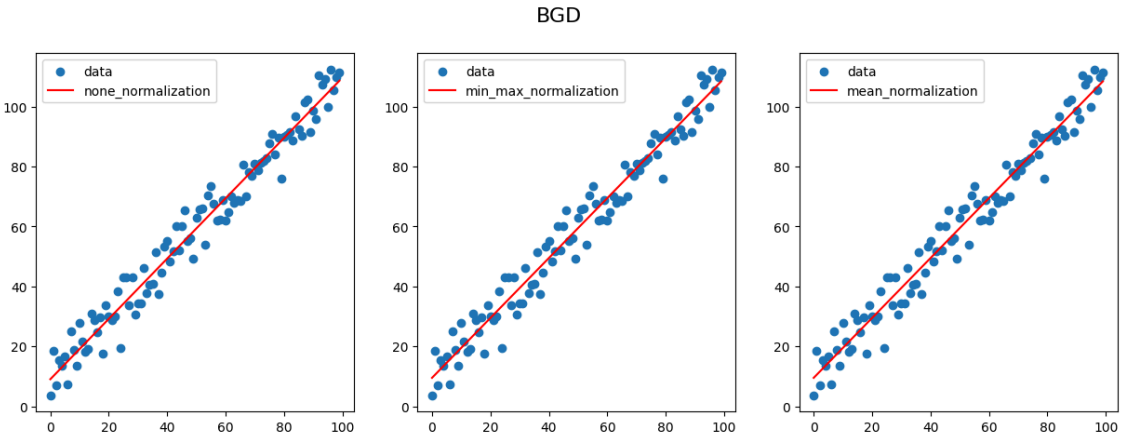

Fig.3 Regression results using BGD method

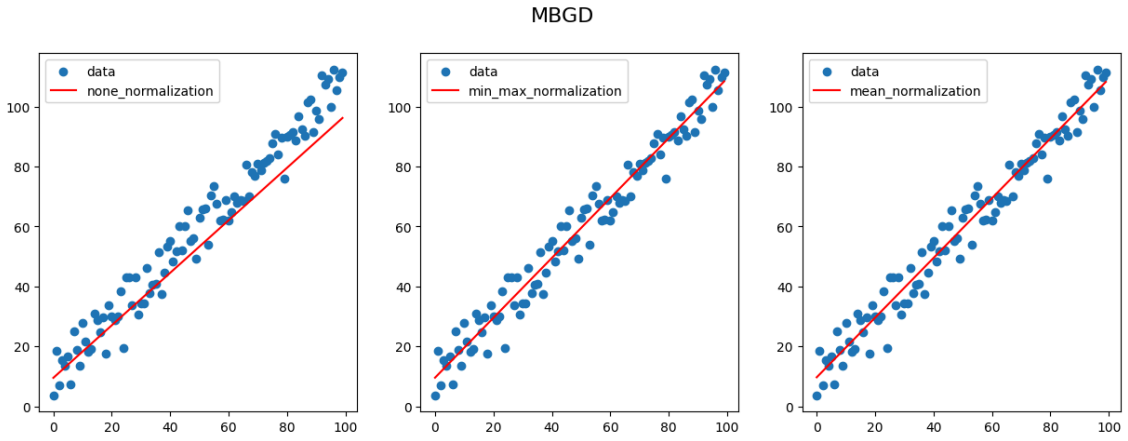

Fig.4 Regression results using MBGD method

- **Analysis:**

  1. Comparison of Gradient Descent Methods

  From the results, it is clear that the three gradient descent methods produced different levels of fitting accuracy compared to the OLS benchmark.

  - **Batch Gradient Descent (BGD):** Achieved the closest fit to the OLS solution. The regression line overlapped almost perfectly with the benchmark, reflecting its stable and precise updates. This is expected, since BGD uses the entire dataset to compute the exact gradient, ensuring convergence toward the global optimum of the convex MSE loss function.
  - **Mini-Batch Gradient Descent (MBGD):** Produced a fit slightly less accurate than BGD but still close to the OLS line. Its performance improved with moderate batch sizes, which reduced the variance of gradient estimates while keeping training efficient. MBGD strikes a balance between accuracy and efficiency, but when batch sizes are too small, the variance increases and fitting accuracy decreases.
  - **Stochastic Gradient Descent (SGD):** Showed the weakest fitting performance. The regression line fluctuated more around the benchmark and converged less smoothly, sometimes deviating from the optimal solution. This stems from the fact that SGD updates parameters based on a single sample at each step, introducing high variance in gradient estimates. While this can speed up training per iteration, it makes convergence noisy and less reliable within limited epochs.

  In summary, BGD provided the most accurate and stable results, MBGD achieved a reasonable trade-off between accuracy and efficiency, while SGD required more careful tuning and longer training to approach the same level of accuracy.

  2. Effect of Normalization

  - **Without normalization:** Input values were large, which forced the learning rate to be set very low. Otherwise, gradient explosion occurred. With a low learning rate, the bias term converged slowly, and more epochs were needed for proper training.
  - **With min-max normalization:** Features were scaled to ([0,1]), which allowed a larger learning rate and significantly improved convergence speed.
  - **With mean normalization:** Standardization stabilized the magnitude of updates and accelerated convergence, especially when features had very different scales.

  Both normalization techniques made the model easier to train and more numerically stable, confirming that normalization is crucial in practice.

3. Hyperparameter Tuning Insights

- In the **un-normalized case**, a very small learning rate was necessary to avoid divergence, but this slowed convergence, especially for the bias term.
- In the **normalized cases**, the learning rate could be safely increased, leading to faster convergence. However, setting it too low again caused the model to converge too slowly and risked underfitting within limited epochs.

These observations highlight that normalization not only accelerates training but also makes the choice of hyperparameters (learning rate, number of epochs) more flexible and effective.

5. Conclusion

In this project, a linear regression model was implemented in Python using NumPy, supporting multiple training strategies including Stochastic Gradient Descent (SGD), Batch Gradient Descent (BGD), and Mini-Batch Gradient Descent (MBGD). The model was extended with both min-max normalization and mean normalization, and was validated against the closed-form Ordinary Least Squares (OLS) solution.

Testing demonstrated that all three gradient-based methods could approximate the OLS benchmark, but with distinct convergence characteristics. BGD produced the most stable and accurate results, MBGD offered a practical trade-off between convergence speed and accuracy, while SGD showed higher variance and weaker fitting performance under limited epochs. Normalization proved essential, significantly improving convergence stability, accelerating training, and making hyperparameter tuning more flexible.

The experiments also highlighted the importance of hyperparameters such as learning rate and number of epochs. Without normalization, a very small learning rate was required to avoid divergence, slowing down bias convergence. With normalization, larger learning rates could be safely applied, yielding faster and more reliable learning.

Overall, the results confirmed the correctness of the implementation and emphasized two key lessons: **the choice of gradient descent method affects convergence accuracy and stability, and normalization fundamentally enhances the trainability of linear regression models.**