Linear Regression from Scratch in Python Using Gradient Descent

Ruslan Yushvaev

January 5, 2025

Contents

1 Introduction

1.1 Why Linear Regression?

Linear Regression is one of the simplest and most fundamental algorithms in machine learning. It assumes a **linear** relationship between input features (X) and the target (y). Common use cases include predicting house prices, forecasting sales, and understanding how an output depends on input variables.

1.2 Why Gradient Descent?

Gradient Descent is a general optimization technique used in various ML algorithms (linear regression, logistic regression, neural networks, etc.). It updates parameters iteratively to **minimize** a chosen cost function (e.g., MSE). This approach is more **scalable** than solving closed-form equations for large datasets or many features.

2 Prerequisites

2.1 Basic Python Knowledge

You should be comfortable with Python syntax (variables, functions, loops) and familiar with lists or NumPy arrays.

2.2 Required Libraries

- **NumPy** for array and matrix operations
- **Matplotlib** (optional) for plotting

2.3 Project Setup

- Create a Python file (e.g., `linear_regression_scratch.py`) or use a Jupyter Notebook.
- Make sure you have NumPy (and Matplotlib if you want plots).

3 Theoretical Foundations

3.1 Linear Regression Equation

We assume a relationship:

$$\hat{y}_i = \beta_0 \ + \ \beta_1 x_{i,1} \ + \ \beta_2 x_{i,2} \ + \ \dots \ + \ \beta_m x_{i,m},$$

where $\hat{y}_i$ is the predicted value, and $\beta_0, \beta_1, \dots, \beta_m$ are parameters (weights).

3.2 Cost Function (Mean Squared Error)

A standard cost function for linear regression is the **Mean Squared Error** (MSE):

$$J(\beta) \;=\; \frac{1}{n}\sum_{i=1}^{n}\bigl(y_i - \hat{y}_i\bigr)^2.$$

Here, y_i is the actual target value and $\hat{y}_i$ is the model's prediction for the i-th data point.

3.3 Gradient Descent: Conceptual Overview

- Start with **initial** guesses for parameters β.
- Compute the **gradient** of the cost function w.r.t. each parameter.
- Update parameters in the *opposite* direction of that gradient:

  $$\beta_j \;:=\; \beta_j \;-\; \alpha\,\frac{\partial J}{\partial \beta_j},$$

  where α is the learning rate.
- Repeat until *convergence* (or for a fixed number of epochs).

4 Implementation Steps in Python

4.1 Data Preparation

- **Option A:** Synthetic data. For example, x in $[0, 10]$ and $y = 2x + 5 + \text{noise}$.
- **Option B:** Real-world data (CSV, public datasets, etc.).
- Ensure x and y are NumPy arrays of matching length.

4.2 Defining the Cost Function

A simple function for MSE in Python (for one feature):

```
def compute_cost(x, y, b0, b1):
    n = len(x)
    y_pred = b0 + b1*x
    errors = y - y_pred
    cost = (errors**2).mean() # MSE
    return cost
```

4.3 Computing the Gradient

For **simple** linear regression (one feature), the partial derivatives are:

$$\frac{\partial J}{\partial \beta_0} = -\frac{2}{n}\sum_{i=1}^{n}\Big[y_i - (\beta_0 + \beta_1 x_i)\Big], \quad \frac{\partial J}{\partial \beta_1} = -\frac{2}{n}\sum_{i=1}^{n}\Big[y_i - (\beta_0 + \beta_1 x_i)\Big]\, x_i.$$

4.4 Gradient Descent Loop

1. Initialize $\beta_0 = 0$, $\beta_1 = 0$ (or random small values).
2. For each epoch:
   - Compute predictions $y_{\text{pred}} = \beta_0 + \beta_1 x$.
   - Compute partial derivatives (gradient).
   - Update:
     $$\beta_0 \;:=\; \beta_0 - \alpha \cdot \frac{\partial J}{\partial \beta_0}, \quad \beta_1 \;:=\; \beta_1 - \alpha \cdot \frac{\partial J}{\partial \beta_1}.$$
   - Optionally store the current cost in a list for later plotting.

4.5 Putting It All Together (Code Snippet)

Below is a concise Python snippet:

```python
import numpy as np
import matplotlib.pyplot as plt

def compute_cost(x, y, b0, b1):
    n = len(x)
    y_pred = b0 + b1*x
    errors = y - y_pred
    return np.mean(errors**2) # MSE

def gradient_descent(x, y, alpha=0.01, epochs=1000):
    b0, b1 = 0.0, 0.0
    n = len(x)
    cost_history = []

    for _ in range(epochs):
        y_pred = b0 + b1*x
        # Partial derivatives
        db0 = -(2/n) * np.sum(y - y_pred)
        db1 = -(2/n) * np.sum((y - y_pred) * x)
        # Update
        b0 = b0 - alpha*db0
        b1 = b1 - alpha*db1
        # Track cost
```

```
        cost = compute_cost(x, y, b0, b1)
        cost_history.append(cost)

    return b0, b1, cost_history

# Example usage:
np.random.seed(42)
x_data = np.random.rand(50) * 10
noise = np.random.randn(50) * 2
y_data = 2.0 * x_data + 5.0 + noise

b0_final, b1_final, cost_hist = gradient_descent(x_data, y_data,
                                            alpha=0.01, epochs=1000)

print("Final parameters (beta0, beta1):", b0_final, b1_final)
print("Final cost (MSE):", cost_hist[-1])
```

5 Verifying and Visualizing Results

5.1 Plotting the Cost Over Iterations

```
plt.figure()
plt.plot(cost_hist, color='red')
plt.title("Cost over Iterations")
plt.xlabel("Epoch")
plt.ylabel("MSE")
plt.show()
```

A **decreasing** curve indicates gradient descent is working.

5.2 Plotting the Final Regression Line

```
plt.scatter(x_data, y_data, color='blue', label='Data')
y_line = b0_final + b1_final * x_data
plt.plot(x_data, y_line, color='green', label='Regression Line')
plt.legend()
plt.show()
```

Visually confirm how well it fits the data.

5.3 Evaluating MSE

The final MSE is `cost_hist[-1]`. If it's relatively small compared to the scale of y, the fit is decent. You can also compare to other metrics or advanced regression techniques.

6 Conclusion and Next Steps

6.1 Summary

We built a **simple** linear regression from scratch:

- Defined a **cost function** (MSE)
- Computed **gradients**
- Used **gradient descent** to find optimal parameters β_0 and β_1

6.2 Possible Extensions

- **Multiple Linear Regression**: Switch to vector form for many features.
- **Regularization**: Add L2 (Ridge) or L1 (Lasso) to prevent overfitting.
- **Adaptive Learning Rates**: E.g., Adam or RMSProp (common in deeper networks).
- **Feature Scaling**: Often speeds up convergence.

6.3 Further Reading

- Andrew Ng's Machine Learning Course
- Scikit-Learn Documentation (for production-ready linear models)
- Deep Learning frameworks like PyTorch, TensorFlow (for advanced optimization)

7 References

1. Andriy Burkov, *The Hundred-Page Machine Learning Book*.
2. Aurélien Géron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow*.
3. Numpy Documentation.
4. Matplotlib Documentation.
5. Andrew Ng's Machine Learning lectures (various sources).
6. Coursera Machine Learning course.
7. Scikit-Learn Documentation.