Implementation of Linear Regression - Labwork 2

Do Thanh Dat - 2440059

May 3 2025

1 Introduction

Linear Regression is one of the core, simple yet powerful algorithm in the field of Machine learning. It is commonly used to model the relationship between one (or more) input variables (features) - x and a continuous output variable (target) - y. This labwork shows my implementation of Linear Regression from scratch. With the help of Gradient Descent method (already implemented in Labwork 1), I tried to optimize the loss of a trained model that fits a set of 2D points in the form (x_i, y_i).

2 Method

The goal of this task is to find the best fitting line $y = w_1x + w_0$, with y is the target variable, x is the input variable, $w1$ is the weight and $w0$ is the bias - for the provided data points (also called training data). To achieve this, I used Mean Squared Error (MSE) as the loss function:

$$J(w_0, w_1) = \frac{1}{2N}\sum_{i=1}^{N}(w_1x_i + w_0 - y_i)^2$$

To minimize J, I applied Gradient Descent update rules for both $w1$ and $w0$:

$$w_0 \leftarrow w_0 - r \cdot \frac{\partial J}{\partial w_0}, \quad w_1 \leftarrow w_1 - r \cdot \frac{\partial J}{\partial w_1}$$

where:

$$\frac{\partial J}{\partial w_0} = \frac{1}{N}\sum_{i=1}^{N}(w_1x_i + w_0 - y_i), \quad \frac{\partial J}{\partial w_1} = \frac{1}{N}\sum_{i=1}^{N}(w_1x_i + w_0 - y_i)x_i$$

- Initialize $w_0 = 0$, $w_1 = 1$
- Update both weights $w1$ and bias $w0$ using the gradient algorithm from above.
- Repeat after a countable iterations, or the loss is small engougn.

3 Code Implementation

```
import csv
from google.colab import drive
import matplotlib.pyplot as plt
drive.mount('/content/drive')

file_path = '/content/drive/MyDrive/Colab Notebooks/DL Implementation/price

def load_data_from_csv(file_path):
    x, y = [], []
    with open(file_path, 'r') as file:
        reader = csv.reader(file)
        for row in reader:
            x.append(float(row[0]))
            y.append(float(row[1]))
    return x, y

def linear_regression(x, y, r: float, iters: int):
  """
  This function takes 4 arguments: y-target value, x-input value, r- learni
  It initializes 2 values: weight w1 = 1 and bias w0 = 0
  Inside the loop, gradient descent is applied: partial derivative of w0 an
  """
  N = len(x)
  w0 = 0
  w1 = 1
  print(f"{'Step'} {'w0'} {'w1'} {'Loss'}")

  for i in range(iters):
        y_hat = [w1 * x[j] + w0 for j in range(N)]
        # update derivative
        dw0 = sum((y_hat[j] - y[j]) for j in range(N)) / N
        dw1 = sum((y_hat[j] - y[j]) * x[j] for j in range(N)) / N
        # update the weight, like in the formula
        w0 -= r * dw0
        w1 -= r * dw1

        loss = sum((y_hat[j] - y[j]) ** 2 for j in range(N)) / (2 * N)
        print(f"{i} {w0:} {w1} {loss}")

  return w0, w1

x, y = load_data_from_csv(file_path)

for lr in [0.1,0.001,0.0001]:
  w0, w1 = linear_regression(x, y, lr, iters=10)
```

```
print(f"LR model: y = {w1} * x + {w0}")
```

4 Results

The same as Labwork 1, I also tested with 3 different learning rates and observed interesting results:

Step	$\mathbf{w_0}$	$\mathbf{w_1}$	**Loss**
0	5.900	266.000	84337108.805
1	-1101.790	-63623.780	4.90133e+12
2	266238.365	15338465.498	2.84846e+17
3	-64181930.463	-3697687879.151	1.65542e+22
4	15472525365.117	891412343490.337	9.62065e+26
5	-3730006569820.708	-2.14895e+14	5.59115e+31
6	8.99203e+14	5.18054e+16	3.24936e+36
7	-2.16774e+17	-1.24889e+19	1.88840e+41
8	5.22582e+19	3.01073e+21	1.09747e+46
9	-1.25980e+22	-7.25806e+23	6.37805e+50

Table 1: Learning rate $r = 0.1$

Step	$\mathbf{w_0}$	$\mathbf{w_1}$	**Loss**
0	0.059	3.650	3250.31
1	0.0066	-0.1155	6233.41
2	0.1125	5.2337	12254.87
3	-0.0064	-2.3666	24409.18
4	0.1940	8.4308	48942.48
5	-0.0593	-6.9099	98462.41
6	0.3319	14.8845	198417.11
7	-0.1925	-16.0800	400172.93
8	0.5840	27.9117	807411.37
9	-0.4879	-34.5891	1629410.49

Table 2: Learning rate $r = 0.001$

Step	w_0	w_1	Loss
0	0.0059	1.2650	1154.94
1	0.0107	1.4658	800.17
2	0.0146	1.6181	596.37
3	0.0179	1.7334	479.29
4	0.0207	1.8209	412.02
5	0.0232	1.8871	373.37
6	0.0254	1.9373	351.16
7	0.0273	1.9754	338.40
8	0.0291	2.0042	331.06
9	0.0308	2.0261	326.83

Table 3: Learning rate $r = 0.0001$

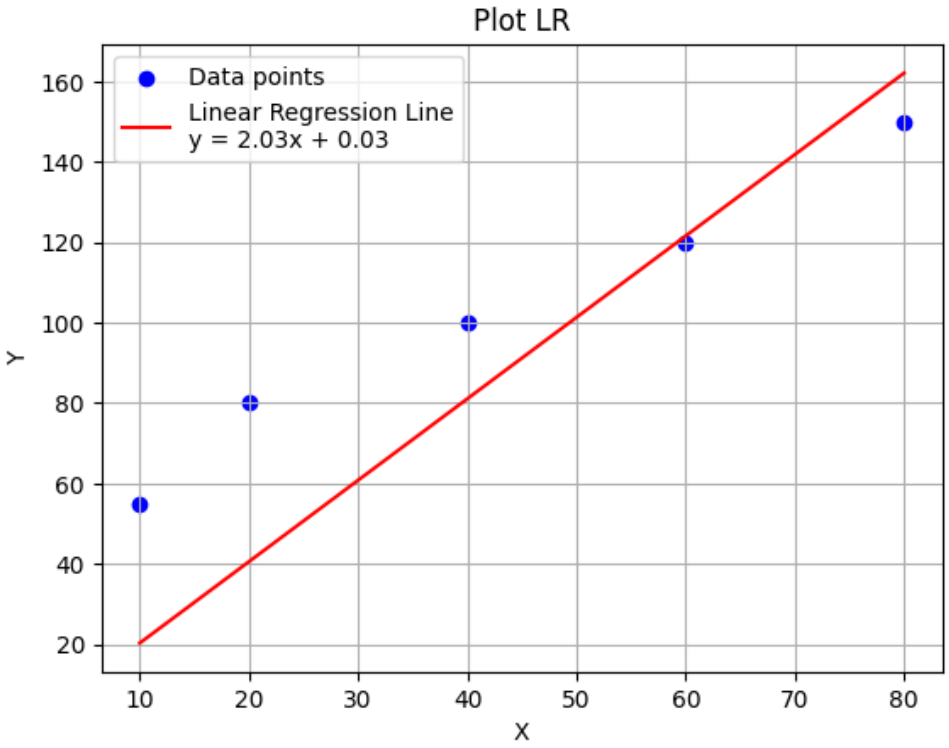

Figure 1: LR line with $r = 0.0001$

5 Comment

In the above experiment, I witnessed some interesting findings:

- The algorithm only converged at $r = 0.0001$. The loss decreased significantly after only a few iterations (from 1154 to 350), showing that gradient descent works perfectly.
- With learning rate $r = 0.1$ and $r = 0.001$, both $w1$, $w0$ and $loss$ grew exponentially, showing a term "gradient explosion". This is because of choosing a too big learning rate, making each step "jump" over the optimal minima, leading to unexpected growth.

6 Conclusion

I have demonstrated a simple implementation of Linear Regression. The same as Labwork 1, the algorithm is also affected by proper learning rates and number of iterations.