

Linear Regression Project

November 26, 2025

1 Task 1: Introduction

```
[1]: %matplotlib inline  
  
import numpy as np  
import matplotlib.pyplot as plt
```

2 Task 2: Dataset

Real estate agent table:

Area	Distance	Price
70	3	21200
50	1	22010
120	9	24305
100	2	31500

You can write the relationship with a 2-variable linear equation:

\$

$$y = b + w_1 \cdot x_1 + w_2 \cdot x_2 \quad (1)$$

\$

In a vector form:

\$

$$y = b + (w_1 w_2) \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad (2)$$

\$

Where \$

$$W = (w_1 w_2) \quad (3)$$

\$ and \$

$$X = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad (4)$$

\$

```
[2]: def generate_examples(num=1000):
    W = [1.0, -3.0]
    b = 1.0

    W = np.reshape(W, (2, 1))

    X = np.random.randn(num, 2)

    y = b + np.dot(X, W) + np.random.randn()

    y = np.reshape(y, (num, 1))

    return X, y
```

```
[3]: X, y = generate_examples()
```

```
[4]: print(X.shape, y.shape)
```

(1000, 2) (1000, 1)

```
[5]: print(X[0], y[0])
```

[-0.20177627 0.51101482] [-1.25514923]

3 Task 3: Initialize Parameters

The loss over \mathbf{m} examples:

\$

$$J = \frac{1}{2m} \sum_{i=1}^m (y - \hat{y})^2 \quad (5)$$

\$

The objective of the gradient descent algorithm is to minimize this loss value.

Gradient Descent Objective is to \$

$$\min(J) \quad (6)$$

\$

```
[6]: class Model:
    def __init__(self, num_features):
        self.num_features = num_features
        self.W = np.random.randn(num_features, 1)
        self.b = np.random.randn()
```

```
[7]: model = Model(2)
print(model.W)
print(model.b)
```

```
[[ 0.20343355]
 [-1.35725996]]
1.2512199075925117
```

4 Task 4: Forward Pass

The gradient descent algorithm can be simplified in 4 steps:

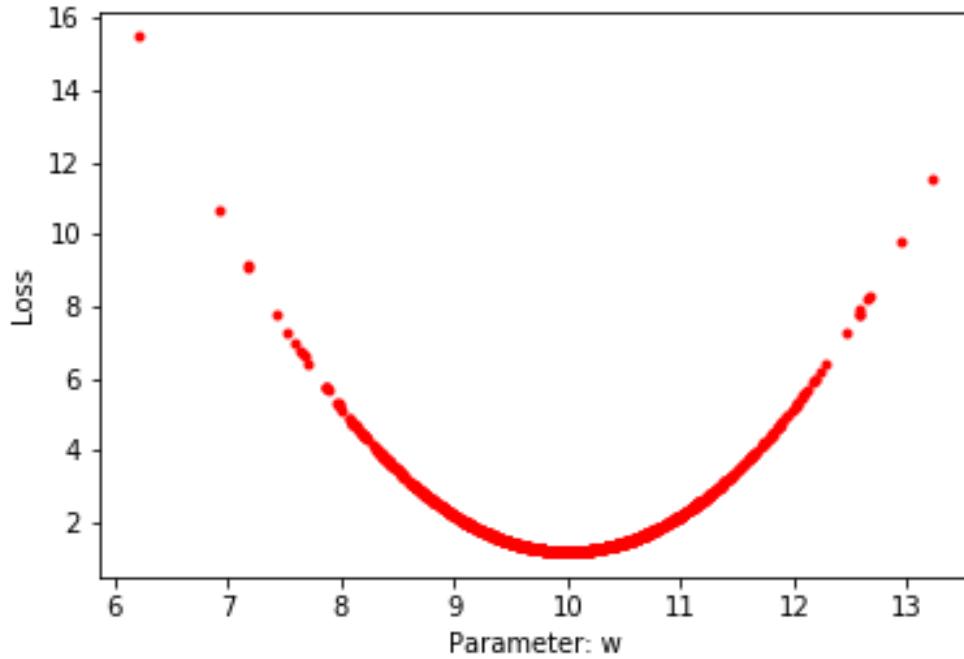
1. Get predictions y_{hat} for X with current values of W and b .
2. Compute the loss between y and y_{hat}
3. Find gradients of the loss with respect to parameters W and b
4. Update the values of W and b by subtracting the gradient values obtained in the previous step

Let's simplify our linear equation a bit more for an example: \$

$$y = wx \quad (7)$$

\$

Let's plot J as a function of w



The gradients of loss with respect to w :

$$\frac{dJ}{dw} = \frac{\delta J}{\delta w} = \lim_{\epsilon \rightarrow 0} \frac{J(w + \epsilon) - J(w)}{\epsilon} \quad (8)$$

```
[8]: class Model(Model):
    def forward_pass(self, X):
        y_hat = self.b + np.dot(X, self.W)
        return y_hat
```

```
[9]: y_hat = Model(2).forward_pass(X)
print(y_hat.shape)
```

(1000, 1)

```
[10]: #Taking a look at the first value
print(y_hat[0])
```

[0.69371359]

5 Task 5: Compute Loss

The loss over \mathbf{m} examples:

\$

$$J = \frac{1}{2m} \sum_{i=1}^m (y - \hat{y})^2 \quad (9)$$

\$

```
[11]: class Model(Model):
    def compute_loss(self, y_hat, y_true):
        return np.sum(np.square(y_hat - y_true))/(2 * y_hat.shape[0])
```

```
[12]: model = Model(2)
y_hat = model.forward_pass(X)
loss = model.compute_loss(y_hat, y)
```

```
[13]: loss
```

[13]: 0.3580056392060336

6 Task 6: Backward Pass

The gradient of loss with respect to bias can be calculated with:

\$

$$\frac{dJ}{db} = \frac{1}{m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)}) \quad (10)$$

\$

\$

$$\frac{dJ}{dW_j} = \frac{1}{m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)}) \cdot x_j^{(i)} \quad (11)$$

```
$
```

```
[14]: class Model(Model):
    def backward_pass(self, X, y_true, y_hat):
        m = y_true.shape[0]
        db = (1/m) * np.sum(y_hat - y_true)
        dW = (1/m) * np.sum(np.dot(np.transpose(y_hat - y_true), X), axis=0)
        return dW, db
```

```
[15]: model = Model(2)

X, y = generate_examples()
y_hat = model.forward_pass(X)

dW, db = model.backward_pass(X, y, y_hat)
```

```
[16]: print(dW, db)
```

```
[-2.1747344  2.96694523] 1.9790009505364972
```

7 Task 7: Update Parameters

```
[17]: class Model(Model):
    def update_params(self, dW, db, lr):
        self.W = self.W - lr * np.reshape(dW, (self.num_features, 1))
        self.b = self.b - db
```

8 Task 8: Training Loop

```
[18]: class Model(Model):
    def train(self, x_train, y_train, iterations, lr):
        losses = []
        for i in range(0, iterations):
            y_hat = self.forward_pass(x_train)
            loss = self.compute_loss(y_hat, y_train)
            dW, db = self.backward_pass(x_train, y_train, y_hat)
            self.update_params(dW, db, lr)
            losses.append(loss)
            if i%int(iterations/10) == 0:
                print('Iter: {}, Loss: {:.4f}'.format(i, loss))
        return losses
```

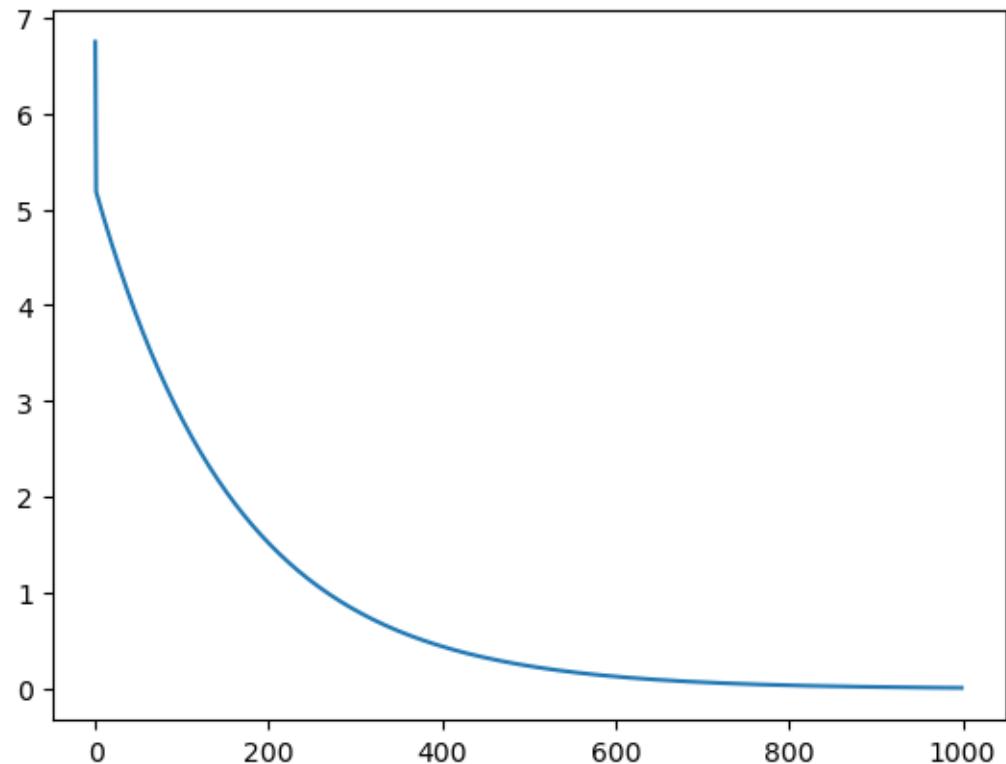
```
[19]: model = Model(2)
```

```
[20]: x_train, y_train = generate_examples()
```

```
[21]: losses = model.train(x_train, y_train, 1000, 3e-3)
```

```
Iter: 0, Loss: 6.7453
Iter: 100, Loss: 2.8146
Iter: 200, Loss: 1.5191
Iter: 300, Loss: 0.8199
Iter: 400, Loss: 0.4426
Iter: 500, Loss: 0.2389
Iter: 600, Loss: 0.1289
Iter: 700, Loss: 0.0696
Iter: 800, Loss: 0.0376
Iter: 900, Loss: 0.0203
```

```
[22]: plt.plot(losses);
```



9 Task 9: Predictions

```
[23]: model_untrained = Model(2)

x_test, y_test = generate_examples(500)
print(x_test.shape, y_test.shape)
```

(500, 2) (500, 1)

```
[24]: preds_untrained = model_untrained.forward_pass(x_test)
preds_trained = model.forward_pass(x_test)
```

```
[25]: plt.figure(figsize = (6, 6))
plt.plot(preds_untrained, y_test, 'rx', label = 'Untrained')
plt.plot(preds_trained, y_test, 'b.', label = 'Trained')
plt.legend()
plt.xlabel('Predictions')
plt.ylabel('Ground Truth')
plt.show()
```

