SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (DATA SCIENCE)

Name: Divyesh Khunt **SapId:60009210116** **Batch:D12**

COURSE CODE: DJ19DSC501 **DATE:**

COURSE NAME: Machine Learning - II **CLASS: AY 2021-22**

LAB EXPERIMENT NO.2

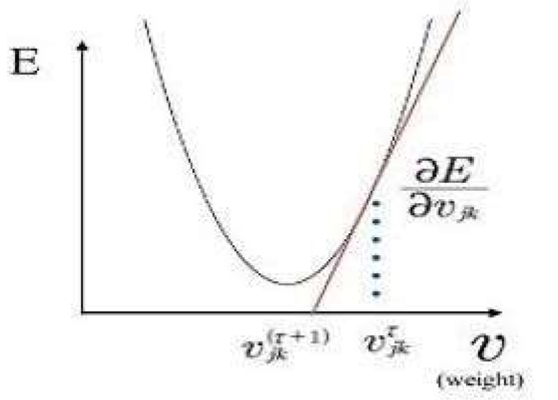
CO/LO:

AIM / OBJECTIVE:

Implement delta learning rule using Stochastic and Batch gradient descend algorithm from scratch.

DESCRIPTION OF EXPERIMENT:

Gradient descent is a way to find a minimum in a high-dimensional space in direction of the steepest descent. The delta rule is an update rule for single layer perceptron's. It makes use of gradient descent.



$$v_{jk}^{(\tau+1)} = v_{jk}^{\tau} + \Delta v_{jk}$$

$$\Delta v_{jk} = -\eta \frac{\partial E}{\partial v_{jk}}$$

$v_{jk}^{(\tau+1)}$	new weight
v_{jk}^{τ}	current weight
η	learning rate
E	Error Function

Weight updation in delta learning

The Delta Rule uses the difference between target activation (i.e., target output values) and obtained activation to drive learning. For reasons discussed below, the use of a threshold activation function (as used in both the McCulloch-Pitts network and the perceptron) is dropped & instead a linear sum of products is used to calculate the activation of the output neuron (alternative activation functions can also be applied). Thus, the activation function is called a Linear Activation function, in which the output node's activation is simply equal to the sum of the network's respective input/weight products. The strength of network connections (i.e., the values of the weights) are adjusted to reduce the difference between target and actual output activation (i.e., error). A graphical depiction of a simple two-layer network capable of deploying the Delta Rule is given in the figure above w (Such a network is not limited to having only one output node):

During forward propagation through a network, the output (activation) of a given node is a function of its inputs. The inputs to a node, which are simply the products of the output of preceding nodes with their associated weights, are summed and then passed through an activation function before being sent out from the node. Thus, we have the following:

This delta learning rule is also known as the Least Mean Squares (LMS) or widrow-Hoff rule. The basic delta rule is given by

$$\Delta w_{ij} = \alpha (t_j - y_j) x_i$$

Where

t_j = the teacher value (or desired value) for unit j .

y_j = the actual output for unit j .

α = learning rate.

in other words

$$\Delta w_{ij} = \alpha \delta x_i$$

Where $\delta = t_j - y_j$

i.e. the deference between the desired or target output and the actual output .The delta rule modifies the weights appropriately for both continuous and binary inputs and outputs .

The delta rule minimizes the squares of the differences between the actual and the desired O/P values.

the squared error for a particular training pattern

$$E = \sum_j (t_j - y_j)^2$$

Where E is a fn. of all the weights .The gradient of E is a vector consisting of the partial derivatives of E with respect to each of the weights. This vector gives the direction of most rapid increase in E , the opposite direction gives the direction of most rapid decrease in the error . The error can be reduced most rapidly by adjusting the weight w_{ij} in the direction of $-\partial E/\partial w_{ij}$.

Generalized delta rule:-

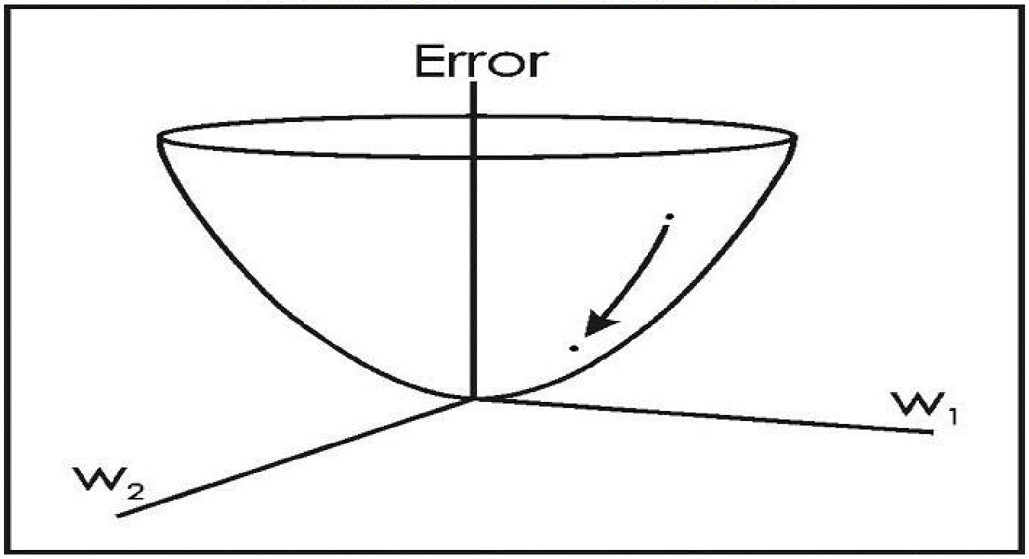
In a multilayer network, the determination of the error is a recursive process which starts with the O/P units and the error is back propagated to the input unit. Therefore the rule is called the error back propagation BP.

$$\Delta w_{ij} = \alpha \delta_j x_i$$

$$\delta_j = (t_j - y_j) f'(y\text{–}in_j).$$

where

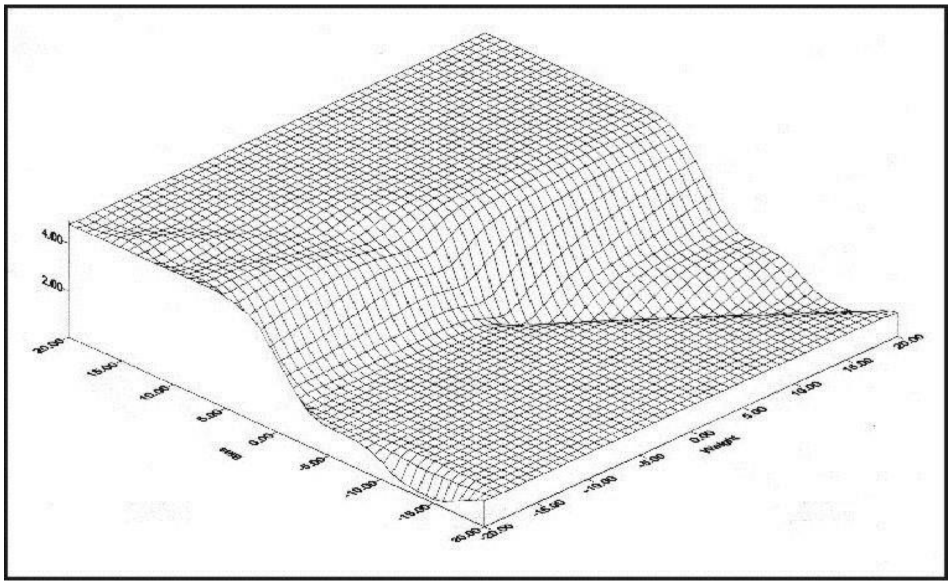
$$y\text{-}in_j = \sum w_j x_i + b$$



Error function with just 2 weights w1 and w2

For any given set of input data and weights, there will be an associated magnitude of error, which is measured by an error function (also known as a cost function). The Delta Rule employs the error function for what is known as Gradient Descent learning, which involves the ‘modification of weights along the most direct path in weight-space to minimize error’, so change applied to a given weight is proportional to the negative of the derivative of the error with respect to that weight. The Error/Cost function is commonly given as the sum of the squares of the differences between all target and actual node activation for the output layer. For a particular training pattern (i.e., training case), error is thus given by:

$$E = \sum_{j} (t_j - y_j)^2$$



Three-dimensional depiction of an Actual error surface

Algorithm Steps, which involves:

- **Selecting Input & Output:** The first step of the delta learning algorithm is to choose an input for the process and to set the desired output.
- **Setting Random Weights:** Once the input and output are set, random weights are allocated, as it will be needed to manipulate the input and output values. After this, the output of each neuron is calculated through the forward propagation, which goes through:
    - Input Layer
    - Output Layer
- **Error Calculation:** This is an important step that calculates the total error by determining how far and suitable the actual output is from the required output. This is done by calculating the errors at the output neuron.
- **Error Minimization:** Based on the observations made in the earlier step, here the focus is on minimizing the error rate to ensure accurate output is delivered.
- **Updating Weights & other Parameters:** If the error rate is high, then parameters (weights and biases) are changed and updated to reduce the rate of error using the delta rule or gradient descent. This is accomplished by assuming a suitable learning rate and propagating backward from the output layer to the previous layer. Acting as an example of dynamic programming, this helps avoid redundant calculations of repeated errors, neurons, and layers.
- **Modeling Prediction Readiness:** Finally, once the error is optimized, the output is tested with some testing inputs to get the desired result.
  This process is repeated until the error reduces to a minimum and the desired output is obtained.

Task(s) to be performed:

a) **Take the dataset with initial value of X = [0.5, 2.5],Y=[0.2, 0.9]**
b) **Initialize a neural network with random weights.**
c) **Calculate output of Neural Network:**
   i. **Calculate error**
   ii. **Note weight and bias changes**
   iii. **Calculate loss**

iv. Plot error surface using loss function verses weight, bias

d) Perform this cycle in step c for every input output pair
e) Perform multiple epochs of step d.
f) Update weights accordingly using stochastic and batch gradient descend.
g) Plot the mean squared error for each iteration 0 in stochastic and Batch Gradient Descent.
h) Similarly plot accuracy for iterations and note the results.

WRITEUP:

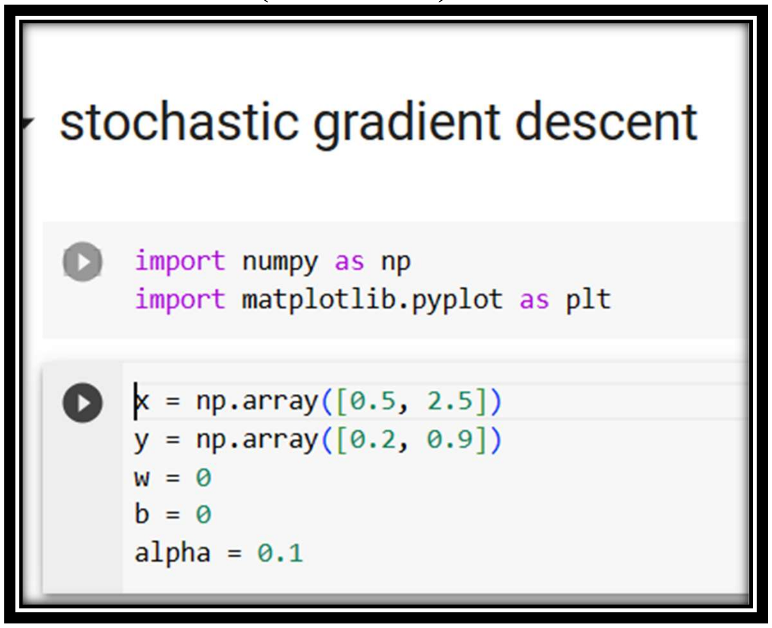
INPUT DATA:

Iris Dataset

PROCEDURE / ALGORITHM:

Describe the procedure that is used to carry out the experiment step-by-step. Describe the features of any programs you developed.

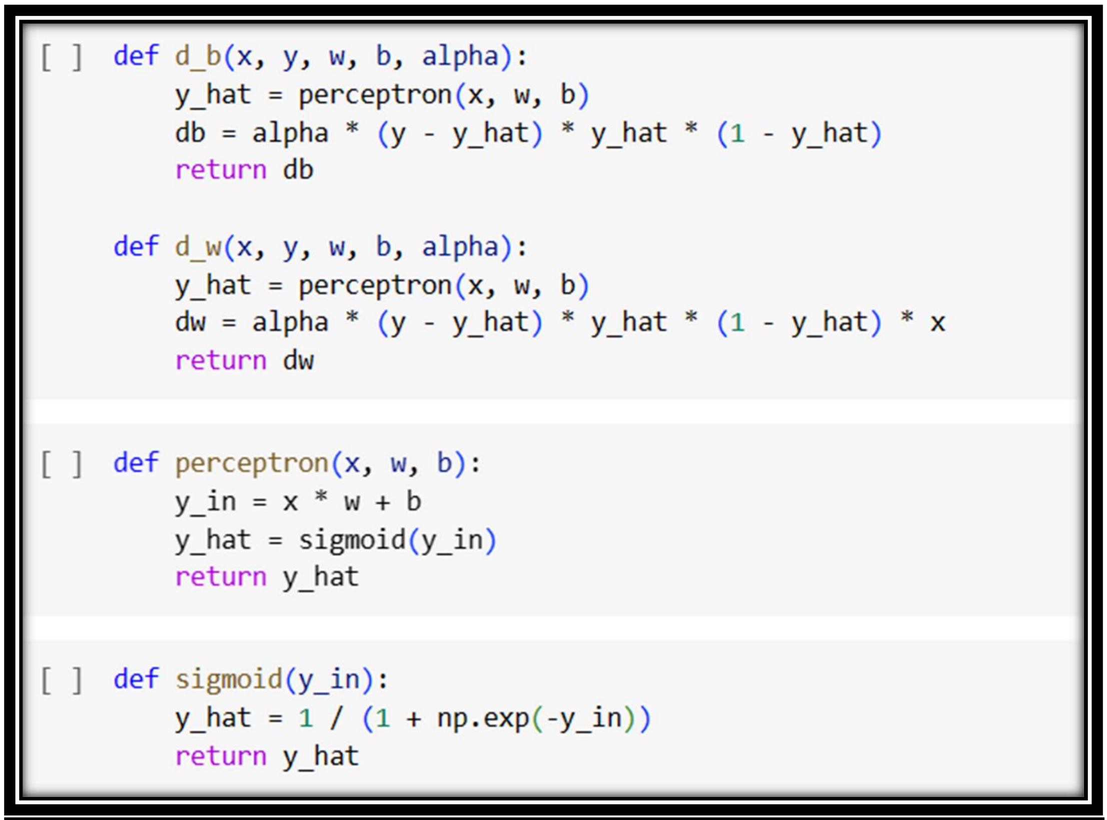
SOURCE CODE (OPTIONAL):



stochastic gradient descent

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

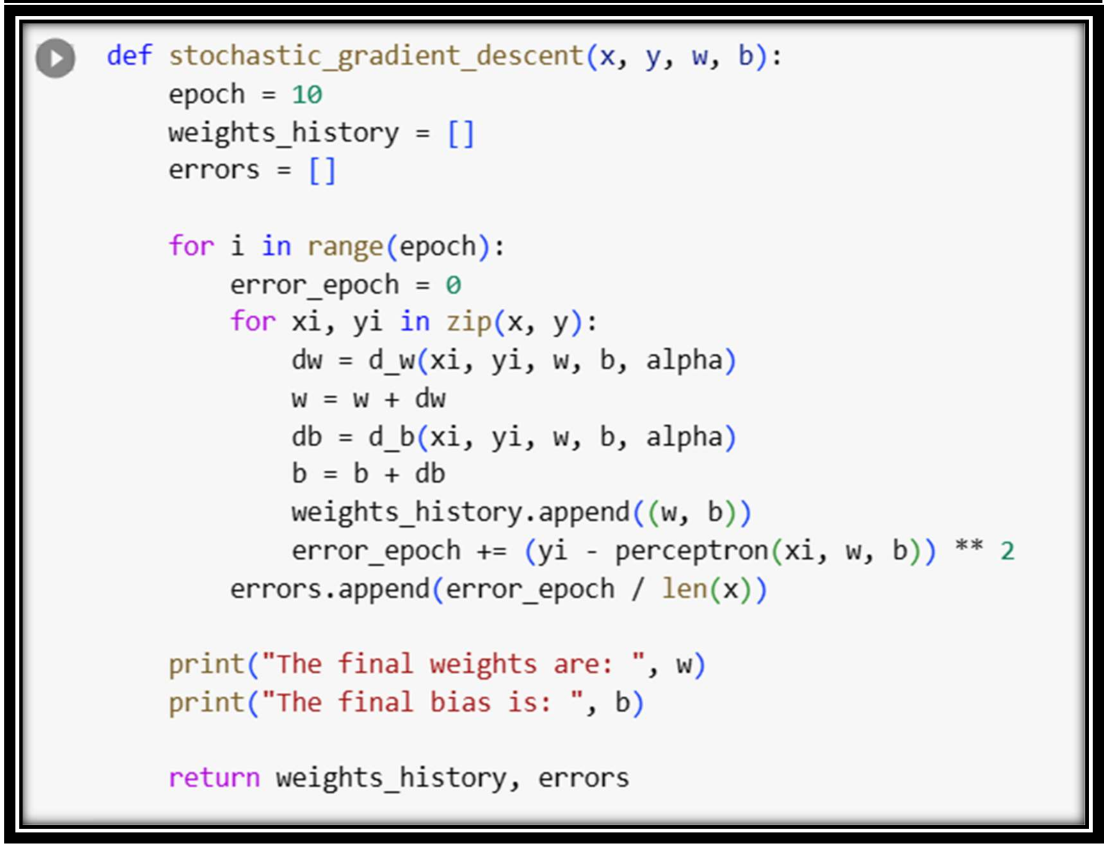
```
x = np.array([0.5, 2.5])
y = np.array([0.2, 0.9])
w = 0
b = 0
alpha = 0.1
```



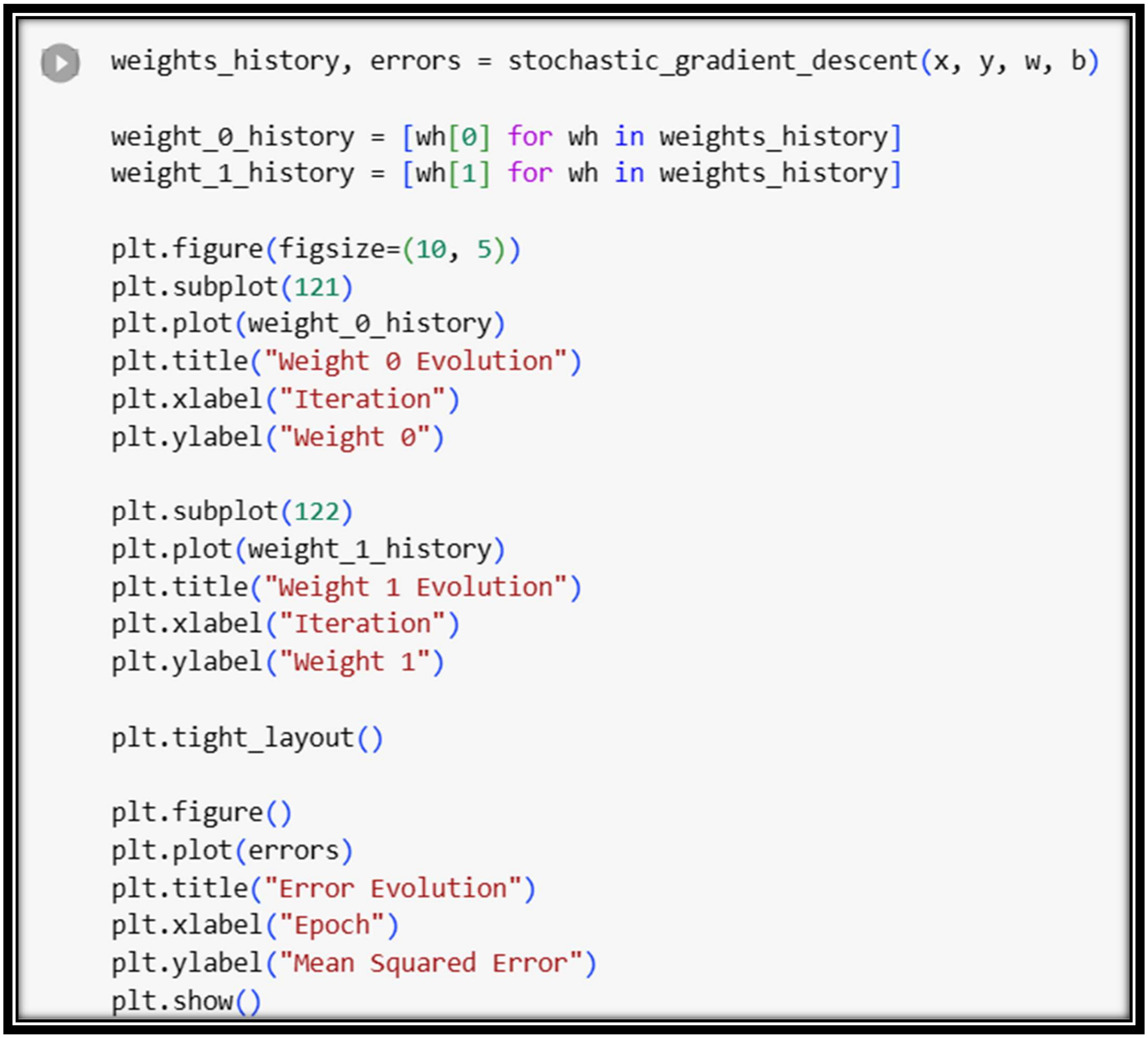
```
[ ]  def d_b(x, y, w, b, alpha):
         y_hat = perceptron(x, w, b)
         db = alpha * (y - y_hat) * y_hat * (1 - y_hat)
         return db

     def d_w(x, y, w, b, alpha):
         y_hat = perceptron(x, w, b)
         dw = alpha * (y - y_hat) * y_hat * (1 - y_hat) * x
         return dw
```

```
[ ]  def perceptron(x, w, b):
         y_in = x * w + b
         y_hat = sigmoid(y_in)
         return y_hat
```

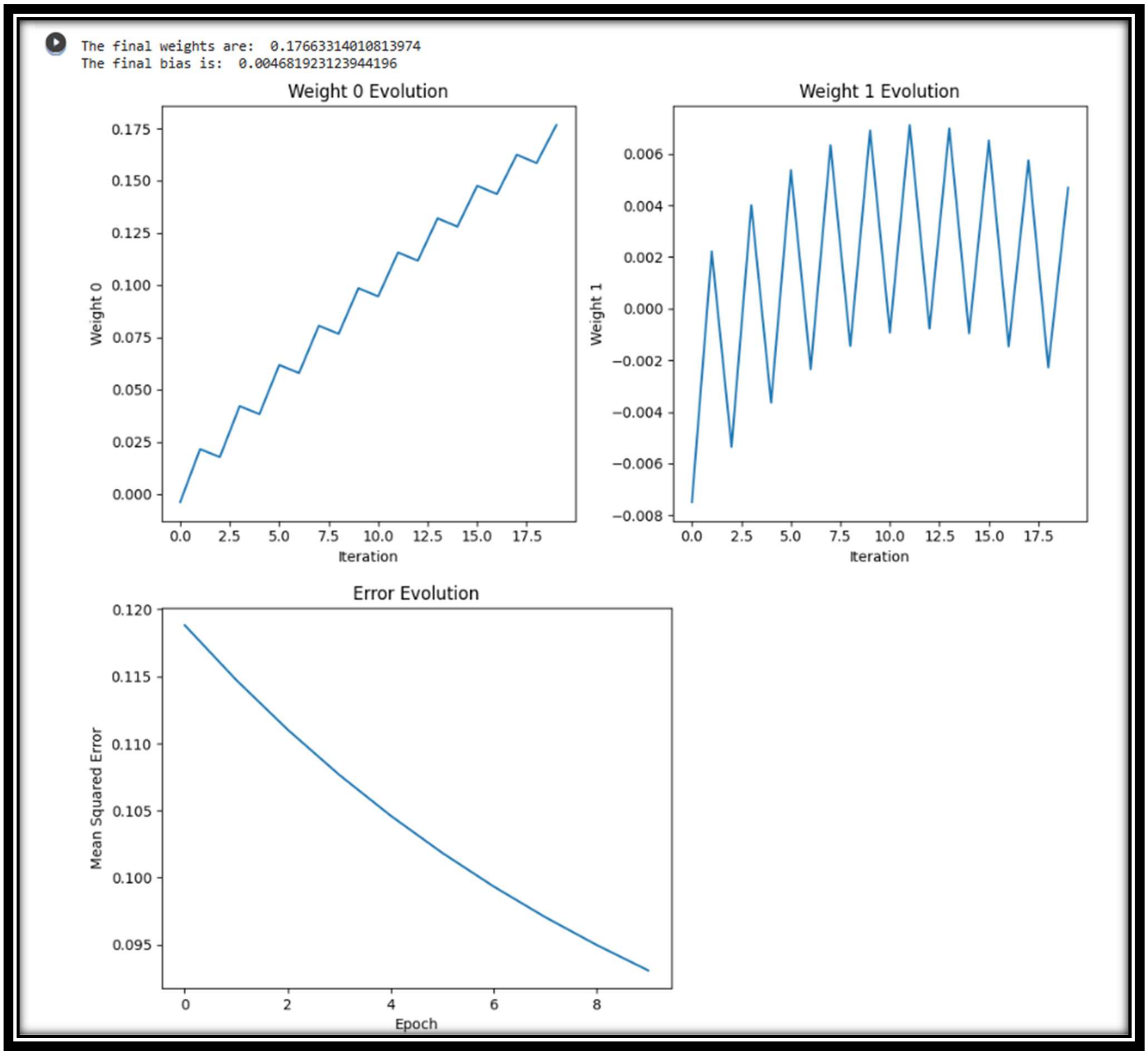
```
[ ]  def sigmoid(y_in):
         y_hat = 1 / (1 + np.exp(-y_in))
         return y_hat
```



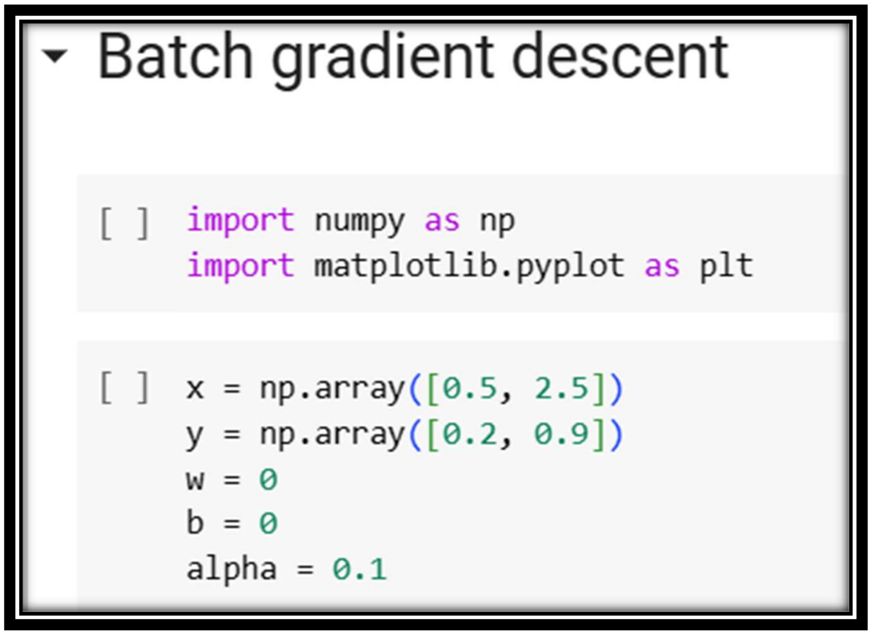
```
def stochastic_gradient_descent(x, y, w, b):
    epoch = 10
    weights_history = []
    errors = []

    for i in range(epoch):
        error_epoch = 0
        for xi, yi in zip(x, y):
            dw = d_w(xi, yi, w, b, alpha)
            w = w + dw
            db = d_b(xi, yi, w, b, alpha)
            b = b + db
            weights_history.append((w, b))
            error_epoch += (yi - perceptron(xi, w, b)) ** 2
        errors.append(error_epoch / len(x))

    print("The final weights are: ", w)
    print("The final bias is: ", b)

    return weights_history, errors
```



```
weights_history, errors = stochastic_gradient_descent(x, y, w, b)

weight_0_history = [wh[0] for wh in weights_history]
weight_1_history = [wh[1] for wh in weights_history]

plt.figure(figsize=(10, 5))
plt.subplot(121)
plt.plot(weight_0_history)
plt.title("Weight 0 Evolution")
plt.xlabel("Iteration")
plt.ylabel("Weight 0")

plt.subplot(122)
plt.plot(weight_1_history)
plt.title("Weight 1 Evolution")
plt.xlabel("Iteration")
plt.ylabel("Weight 1")

plt.tight_layout()

plt.figure()
plt.plot(errors)
plt.title("Error Evolution")
plt.xlabel("Epoch")
plt.ylabel("Mean Squared Error")
plt.show()
```



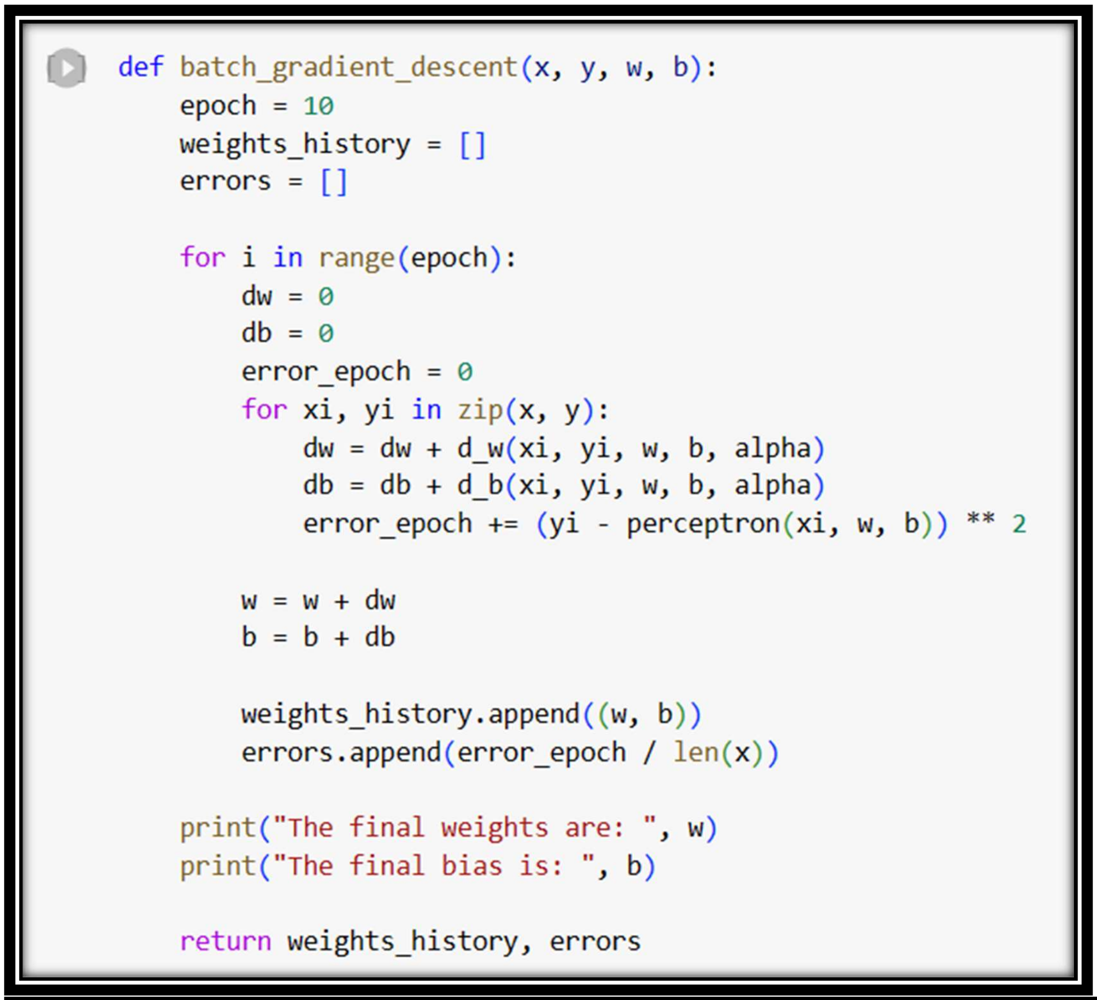
```
The final weights are:  0.17663314010813974
The final bias is:  0.004681923123944196
```



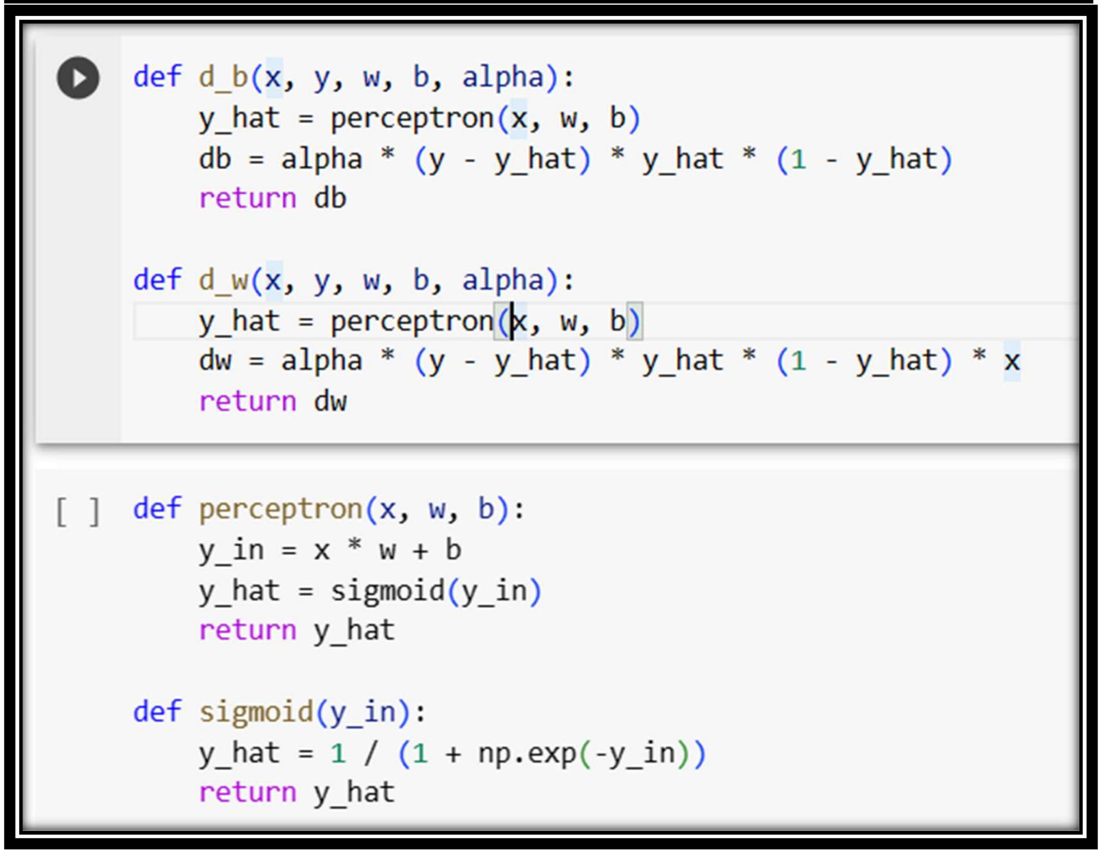
Batch gradient descent

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

```
x = np.array([0.5, 2.5])
y = np.array([0.2, 0.9])
w = 0
b = 0
alpha = 0.1
```



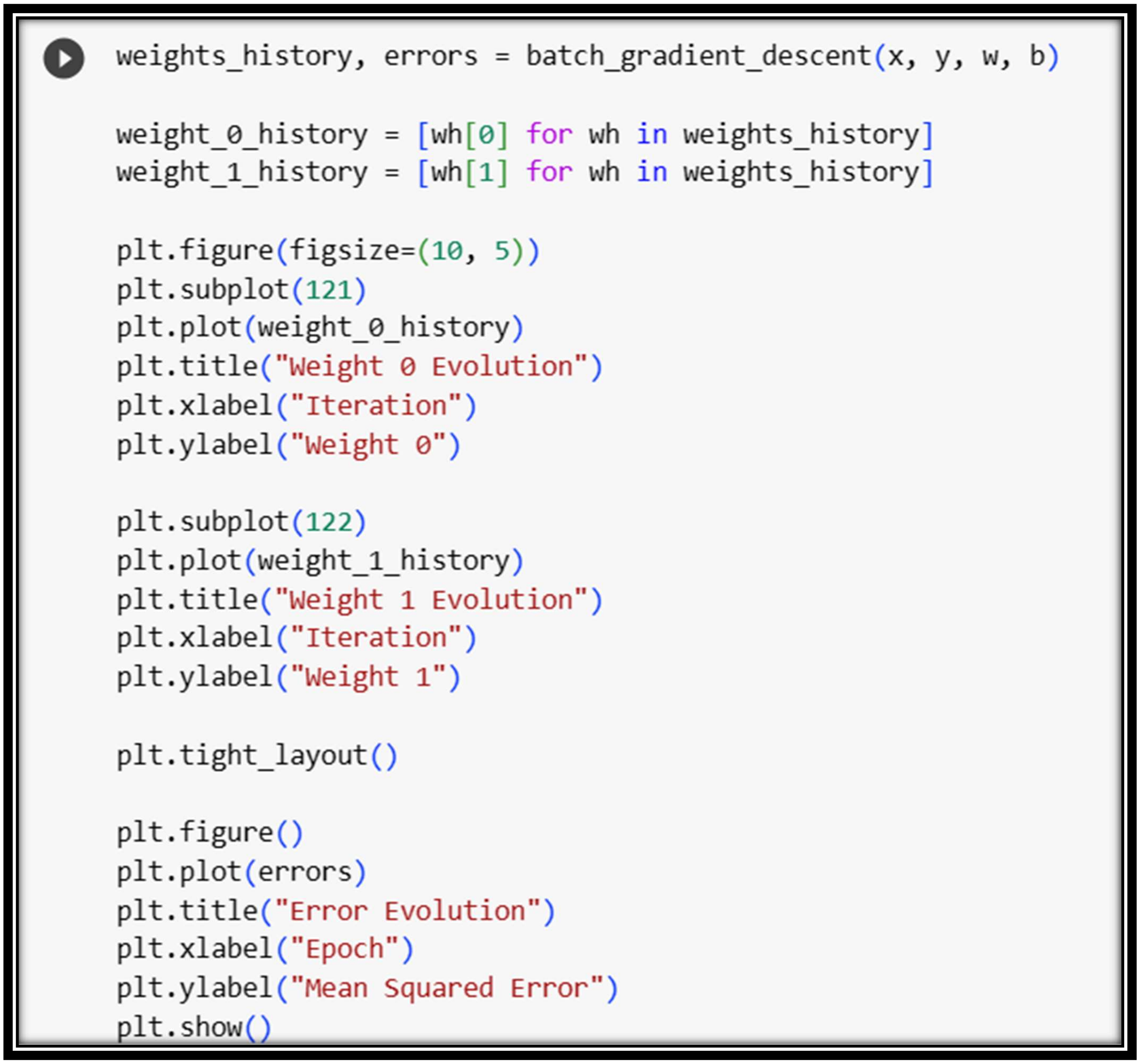
```python
def batch_gradient_descent(x, y, w, b):
    epoch = 10
    weights_history = []
    errors = []

    for i in range(epoch):
        dw = 0
        db = 0
        error_epoch = 0
        for xi, yi in zip(x, y):
            dw = dw + d_w(xi, yi, w, b, alpha)
            db = db + d_b(xi, yi, w, b, alpha)
            error_epoch += (yi - perceptron(xi, w, b)) ** 2

        w = w + dw
        b = b + db

        weights_history.append((w, b))
        errors.append(error_epoch / len(x))

    print("The final weights are: ", w)
    print("The final bias is: ", b)

    return weights_history, errors
```



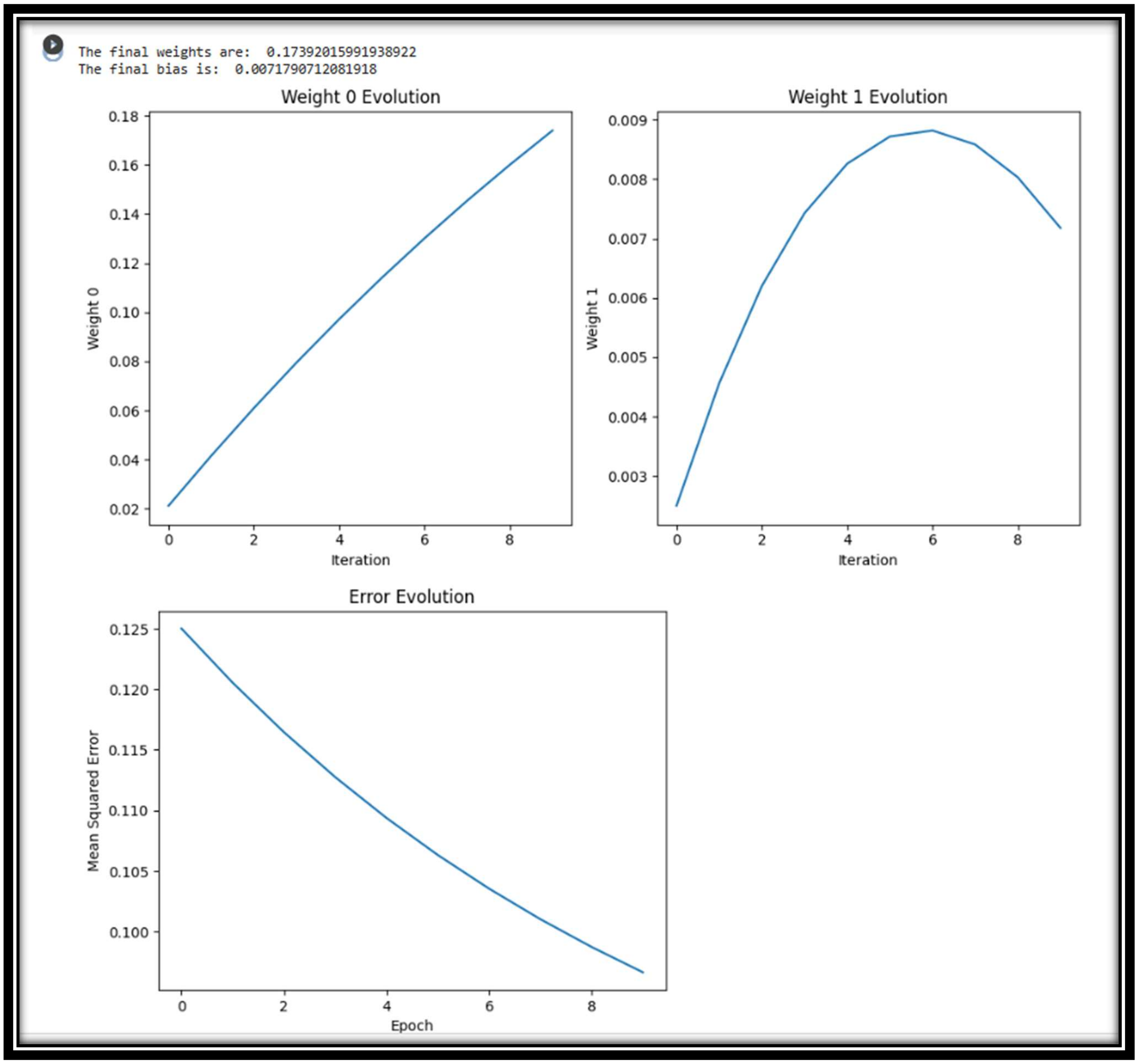
```python
def d_b(x, y, w, b, alpha):
    y_hat = perceptron(x, w, b)
    db = alpha * (y - y_hat) * y_hat * (1 - y_hat)
    return db

def d_w(x, y, w, b, alpha):
    y_hat = perceptron(x, w, b)
    dw = alpha * (y - y_hat) * y_hat * (1 - y_hat) * x
    return dw
```

```
[ ]
```

```python
def perceptron(x, w, b):
    y_in = x * w + b
    y_hat = sigmoid(y_in)
    return y_hat

def sigmoid(y_in):
    y_hat = 1 / (1 + np.exp(-y_in))
    return y_hat
```



```python
weights_history, errors = batch_gradient_descent(x, y, w, b)

weight_0_history = [wh[0] for wh in weights_history]
weight_1_history = [wh[1] for wh in weights_history]

plt.figure(figsize=(10, 5))
plt.subplot(121)
plt.plot(weight_0_history)
plt.title("Weight 0 Evolution")
plt.xlabel("Iteration")
plt.ylabel("Weight 0")

plt.subplot(122)
plt.plot(weight_1_history)
plt.title("Weight 1 Evolution")
plt.xlabel("Iteration")
plt.ylabel("Weight 1")

plt.tight_layout()

plt.figure()
plt.plot(errors)
plt.title("Error Evolution")
plt.xlabel("Epoch")
plt.ylabel("Mean Squared Error")
plt.show()
```



```
The final weights are:  0.17392015991938922
The final bias is:  0.0071790712081918
```

OBSERVATIONS / DISCUSSION OF RESULT:

This section should interpret the outcome of the experiment. The observations can be visually represented using images, tables, graphs, etc. This section should answer the question "What do the result tell us?" Compare and interpret your results with expected behavior. Explain unexpected behavior, if any.

CONCLUSION:

Base all conclusions on your actual results; describe the meaning of the experiment and the implications of your results.