



✓ The Perceptron

✓ Define the Perceptron

```

1  import numpy as np
2
3  class Perceptron:
4      def __init__(self, learning_rate=0.01, n_iterations=1000):
5          self.learning_rate = learning_rate
6          self.n_iterations = n_iterations
7          self.weights = None
8          self.bias = None
9
10     def fit(self, X, y):
11         # Initialize parameters
12         n_samples, n_features = X.shape
13         self.weights = np.zeros(n_features)
14         self.bias = 0
15
16         # Training loop
17         for _ in range(self.n_iterations):
18             for idx, x_i in enumerate(X):
19                 linear_output = np.dot(x_i, self.weights) + self.bias
20                 y_predicted = 1 if linear_output >= 0 else 0
21
22                 # Perceptron update rule
23                 update = self.learning_rate * (y[idx] - y_predicted)
24                 self.weights += update * x_i
25                 self.bias += update
26
27     def predict(self, X):
28         linear_output = np.dot(X, self.weights) + self.bias
29         return np.where(linear_output >= 0, 1, 0)
30

```

✓ Run The Perceptron

```

1
2  import numpy as np
3  from matplotlib import pyplot as plt
4
5  # Training data for AND gate
6  X = np.array([[0, 0],
7                [0, 1],
8                [1, 0],
9                [1, 1]])
10 y = np.array([0, 0, 0, 1])
11
12 # Initialize and train the perceptron
13 perceptron = Perceptron(learning_rate=0.1, n_iterations=100)
14 perceptron.fit(X, y)
15
16 # Display results
17 print("Weights:", perceptron.weights)
18 print("Bias:", perceptron.bias)
19 print("Predictions:", perceptron.predict(X))
20

```

```

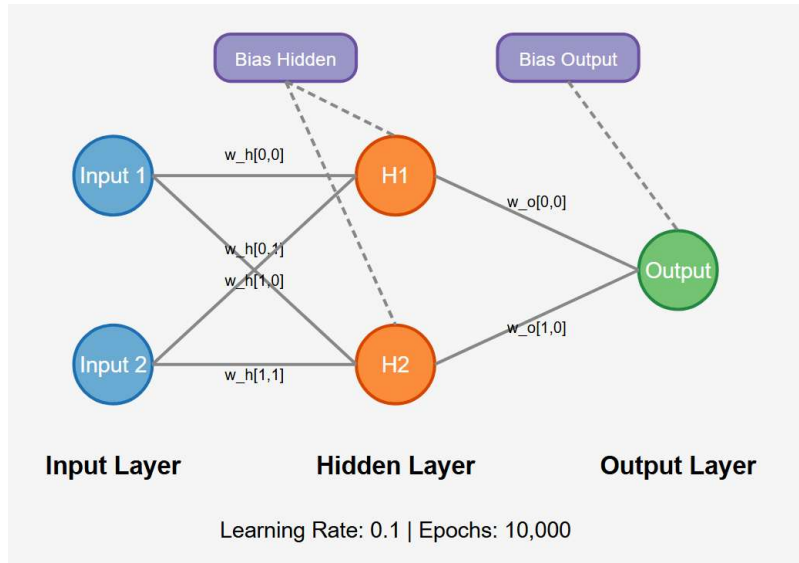
Weights: [0.2 0.1]
Bias: -0.20000000000000004
Predictions: [0 0 0 1]

```

✓ The Mult-Layer Perceptron - MLP

✓ Solving the XOR Problem with a Neural Network

This code demonstrates how to build and train a simple neural network from scratch using NumPy to learn the XOR logic gate.



✓ 1. Import Required Libraries

```
1 import numpy as np
```

✓ 2. Define the Activation Function

```
1 def sigmoid(x):
2     return 1 / (1 + np.exp(-x))
3
4 def sigmoid_derivative(x):
5     return sigmoid(x) * (1 - sigmoid(x))
```

✓ 3. Define the XOR Input and Output

```
1 X = np.array([[0, 0],
2               [0, 1],
3               [1, 0],
4               [1, 1]])
5
6 y = np.array([[0],
7               [1],
8               [1],
9               [0]])
```

✓ 4. Initialize Network Parameters (The Weights and Biases)

```
1 np.random.seed(42)
2
3 # 2 input features, 2 hidden neurons
4 weights_hidden = np.random.uniform(size=(2, 2))
5
6 # 1 bias for each hidden neuron
7 bias_hidden = np.random.uniform(size=(1, 2))
8
9
```

```

10 # 2 hidden neurons, 1 output neuron
11 weights_output = np.random.uniform(size=(2, 1))
12
13 # 1 bias for output neuron
14 bias_output = np.random.uniform(size=(1, 1))
15
16 learning_rate = 0.1
17 epochs = 10000

```

5. Train the Network Using Backpropagation

```

1 for epoch in range(epochs):
2     # input to hidden layer
3     hidden_layer_input = np.dot(X, weights_hidden) + bias_hidden
4
5     # Activation of hidden layer
6     hidden_layer_output = sigmoid(hidden_layer_input)
7
8     # input to output layer
9     output_layer_input = np.dot(hidden_layer_output, weights_output) + bias_output
10
11     # final output
12     predicted_output = sigmoid(output_layer_input)
13
14     # Backpropagation
15     # calculate error, Mean Squared Error (MSE) loss function
16     error = y - predicted_output
17     # print error every 1000 epochs
18     if epoch % 1000 == 0:
19         print(f"Epoch {epoch+1}/{epochs}, Error: {np.mean(np.abs(error))}")
20
21     # derivative of sigmoid for output layer
22     d_predicted_output = error * sigmoid_derivative(output_layer_input)
23
24
25     # propagate error to hidden layer
26     error_hidden_layer = d_predicted_output.dot(weights_output.T)
27     # derivative of sigmoid for hidden layer
28     d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_input)
29
30     weights_output += hidden_layer_output.T.dot(d_predicted_output) * learning_rate # update weights
31     bias_output += np.sum(d_predicted_output, axis=0, keepdims=True) * learning_rate # update bias
32     weights_hidden += X.T.dot(d_hidden_layer) * learning_rate # update weights
33     bias_hidden += np.sum(d_hidden_layer, axis=0, keepdims=True) * learning_rate # update bias

```

```

Epoch 1/10000, Error: 0.4977550305860017
Epoch 1001/10000, Error: 0.48962844155619734
Epoch 2001/10000, Error: 0.43050559183023696
Epoch 3001/10000, Error: 0.3357263739761261
Epoch 4001/10000, Error: 0.17357496319517718
Epoch 5001/10000, Error: 0.11181272498560178
Epoch 6001/10000, Error: 0.08576413241547491
Epoch 7001/10000, Error: 0.07130866479694546
Epoch 8001/10000, Error: 0.06197519138577699
Epoch 9001/10000, Error: 0.055372184098791376

```

6. Evaluate the Final Output

Run a forward pass to get the final output after training

```

1 print("Final predicted output:")
2
3 # input to hidden layer
4 hidden_layer_output = sigmoid(np.dot(X, weights_hidden) + bias_hidden)
5
6 # final output
7 predicted_output = sigmoid(np.dot(hidden_layer_output, weights_output) + bias_output)
8 print(np.round(predicted_output, 3))

```

```

Final predicted output:
[[0.053]
 [0.952]
 [0.952]
 [0.052]]

```

7. Display the Learned Parameters

```
1 print("\nLearned weights and biases:")
2 print("\nHidden layer weights:\n", weights_hidden)
3 print("\nHidden layer bias:\n", bias_hidden)
4 print("\nOutput layer weights:\n", weights_output)
5 print("\nOutput layer bias:\n", bias_output)
```



Learned weights and biases:

Hidden layer weights:
[[3.79198478 5.81661184]
[3.80004873 5.8545897]]

Hidden layer bias:
[[-5.82020057 -2.46277158]]

Output layer weights:
[[-8.32186051]
[7.66063503]]

Output layer bias:
[[-3.45550373]]