Open in Colab

The Perceptron

Define the Perceptron

```
import numpy as np

class Perceptron:
    def __init__(self, learning_rate=0.01, n_iterations=1000):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights = None
        self.bias = None

    def fit(self, X, y):
        # Initialize parameters
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0

        # Training loop
        for _ in range(self.n_iterations):
            for idx, x_i in enumerate(X):
                linear_output = np.dot(x_i, self.weights) + self.bias
                y_predicted = 1 if linear_output >= 0 else 0

                # Perceptron update rule
                update = self.learning_rate * (y[idx] - y_predicted)
                self.weights += update * x_i
                self.bias += update

    def predict(self, X):
        linear_output = np.dot(X, self.weights) + self.bias
        return np.where(linear_output >= 0, 1, 0)

```

Run The Perceptron

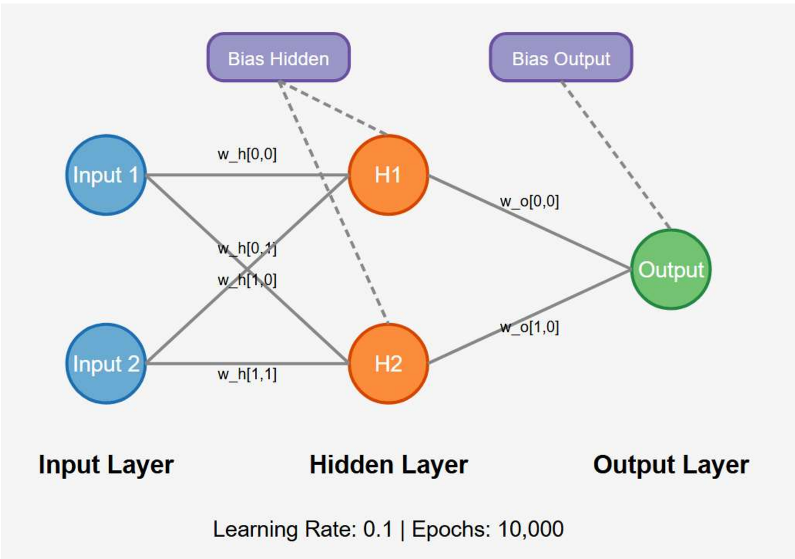
```

import numpy as np
from matplotlib import pyplot as plt

# Training data for AND gate
X = np.array([[0, 0],
              [0, 1],
              [1, 0],
              [1, 1]])
y = np.array([0, 0, 0, 1])

# Initialize and train the perceptron
perceptron = Perceptron(learning_rate=0.1, n_iterations=100)
perceptron.fit(X, y)

# Display results
print("Weights:", perceptron.weights)
print("Bias:", perceptron.bias)
print("Predictions:", perceptron.predict(X))

```

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

The Mulit-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

```
import numpy as np
```

2. Define the Activation Function

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))
```

3. Define the XOR Input and Output

```
X = np.array([[0, 0],
              [0, 1],
              [1, 0],
              [1, 1]])

y = np.array([[0],
              [1],
              [1],
              [0]])
```

4. Initialize Network Parameters (The Weights and Biases)

```
np.random.seed(42)

# 2 input features, 2 hidden neurons
weights_hidden = np.random.uniform(size=(2, 2))

# 1 bias for each hidden neuron
bias_hidden = np.random.uniform(size=(1, 2))


```

```
# 2 hidden neurons, 1 output neuron
weights_output = np.random.uniform(size=(2, 1))

# 1 bias for output neuron
bias_output = np.random.uniform(size=(1, 1))

learning_rate = 0.1
epochs = 10000
```

5. Train the Network Using Backpropagation

```
for epoch in range(epochs):
    # input to hidden layer
    hidden_layer_input = np.dot(X, weights_hidden) + bias_hidden

    # Activation of hidden layer
    hidden_layer_output = sigmoid(hidden_layer_input)

    # input to output layer
    output_layer_input = np.dot(hidden_layer_output, weights_output) + bias_output

    # final output
    predicted_output = sigmoid(output_layer_input)

    # Backpropagation
    # calculate error, Mean Squared Error (MSE) loss function
    error = y - predicted_output
    # print error every 1000 epochs
    if epoch % 1000 == 0:
        print(f"Epoch {epoch+1}/{epochs}, Error: {np.mean(np.abs(error))}")

    # derivative of sigmoid for output layer
    d_predicted_output = error * sigmoid_derivative(output_layer_input)


    # propagate error to hidden layer
    error_hidden_layer = d_predicted_output.dot(weights_output.T)
    # derivative of sigmoid for hidden layer
    d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_input)

    weights_output += hidden_layer_output.T.dot(d_predicted_output) * learning_rate # update weights
    bias_output += np.sum(d_predicted_output, axis=0, keepdims=True) * learning_rate # update bias
    weights_hidden += X.T.dot(d_hidden_layer) * learning_rate # update weights
    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

```
print("Final predicted output:")

# input to hidden layer
hidden_layer_output = sigmoid(np.dot(X, weights_hidden) + bias_hidden)

# final output
predicted_output = sigmoid(np.dot(hidden_layer_output, weights_output) + bias_output)
print(np.round(predicted_output, 3))
```

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

7. Display the Learned Parameters

```
print("\nLearned weights and biases:")
print("\nHidden layer weights:\n", weights_hidden)
print("\nHidden layer bias:\n", bias_hidden)
print("\nOutput layer weights:\n", weights_output)
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]]
```