

```

# Import necessary libraries
import numpy as np

# Define the activation function (step function for a simple perceptron)
def step_function(x):
    return 1 if x >= 0 else 0

# Perceptron class
class Perceptron:
    def __init__(self, learning_rate=0.1, n_iterations=10):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights = None
        self.bias = None

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

        for _ in range(self.n_iterations):
            for idx, x_i in enumerate(X):
                # Calculate the linear combination of inputs and weights
                linear_output = np.dot(x_i, self.weights) + self.bias
                # Get the predicted output after activation
                y_predicted = step_function(linear_output)

                # Update weights and bias based on the error
                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
        y_predicted = np.array([step_function(x) for x in linear_output])
        return y_predicted

# XOR dataset
X_xor = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
y_xor = np.array([0, 1, 1, 0])

# Train and test the single perceptron on XOR
perceptron_xor = Perceptron()
perceptron_xor.fit(X_xor, y_xor)
y_pred_xor = perceptron_xor.predict(X_xor)

print("Single Perceptron Predictions for XOR:")
for i in range(len(X_xor)):
    print(f"Input: {X_xor[i]}, Expected: {y_xor[i]}, Predicted: {y_pred_xor[i]}")

# Evaluate accuracy (will likely be low)
accuracy = np.mean(y_pred_xor == y_xor)
print(f"Accuracy: {accuracy*100:.2f}%")

```

```

Single Perceptron Predictions for XOR:
Input: [0 0], Expected: 0, Predicted: 1
Input: [0 1], Expected: 1, Predicted: 1
Input: [1 0], Expected: 1, Predicted: 0
Input: [1 1], Expected: 0, Predicted: 0
Accuracy: 50.00%

```

```

# Define the sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

# MLP class
class MultilayerPerceptron:
    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.1, n_iterations=10000):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

```

```

self.learning_rate = learning_rate
self.n_iterations = n_iterations

# Initialize weights and biases for hidden layer
self.weights_input_hidden = np.random.uniform(size=(input_size, hidden_size))
self.bias_hidden = np.random.uniform(size=(1, hidden_size))

# Initialize weights and biases for output layer
self.weights_hidden_output = np.random.uniform(size=(hidden_size, output_size))
self.bias_output = np.random.uniform(size=(1, output_size))

def fit(self, X, y):
    y = y.reshape(-1, 1) # Reshape y for consistent matrix operations

    for _ in range(self.n_iterations):
        # --- Forward Propagation ---
        # Hidden layer calculations
        hidden_layer_input = np.dot(X, self.weights_input_hidden) + self.bias_hidden
        hidden_layer_output = sigmoid(hidden_layer_input)

        # Output layer calculations
        output_layer_input = np.dot(hidden_layer_output, self.weights_hidden_output) + self.bias_output
        predicted_output = sigmoid(output_layer_input)

        # --- Backpropagation ---
        # Calculate error for the output layer
        error_output = y - predicted_output
        d_predicted_output = error_output * sigmoid_derivative(predicted_output)

        # Calculate error for the hidden layer
        error_hidden = np.dot(d_predicted_output, self.weights_hidden_output.T)
        d_hidden_layer = error_hidden * sigmoid_derivative(hidden_layer_output)

        # Update weights and biases
        self.weights_hidden_output += np.dot(hidden_layer_output.T, d_predicted_output) * self.learning_rate
        self.bias_output += np.sum(d_predicted_output, axis=0, keepdims=True) * self.learning_rate
        self.weights_input_hidden += np.dot(X.T, d_hidden_layer) * self.learning_rate
        self.bias_hidden += np.sum(d_hidden_layer, axis=0, keepdims=True) * self.learning_rate

    def predict(self, X):
        # Forward propagation through the trained network
        hidden_layer_input = np.dot(X, self.weights_input_hidden) + self.bias_hidden
        hidden_layer_output = sigmoid(hidden_layer_input)
        output_layer_input = np.dot(hidden_layer_output, self.weights_hidden_output) + self.bias_output
        predicted_output = sigmoid(output_layer_input)
        return (predicted_output > 0.5).astype(int).flatten() # Convert probabilities to binary output

# XOR dataset
X_xor = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
y_xor = np.array([0, 1, 1, 0])

# Train and test the MLP on XOR
# Input size = 2 (for x1, x2), Hidden size = 2 (can be adjusted), Output size = 1 (for XOR result)
mlp_xor = MultilayerPerceptron(input_size=2, hidden_size=2, output_size=1, learning_rate=0.1, n_iterations=10000)
mlp_xor.fit(X_xor, y_xor)
y_pred_mlp = mlp_xor.predict(X_xor)

print("\nMultilayer Perceptron Predictions for XOR:")
for i in range(len(X_xor)):
    print(f"Input: {X_xor[i]}, Expected: {y_xor[i]}, Predicted: {y_pred_mlp[i]}")

# Evaluate accuracy
accuracy_mlp = np.mean(y_pred_mlp == y_xor)
print(f"Accuracy: {accuracy_mlp*100:.2f}%")

```

```

Multilayer Perceptron Predictions for XOR:
Input: [0 0], Expected: 0, Predicted: 0
Input: [0 1], Expected: 1, Predicted: 1
Input: [1 0], Expected: 1, Predicted: 1
Input: [1 1], Expected: 0, Predicted: 1
Accuracy: 75.00%

```

```

import matplotlib.pyplot as plt

# Prepare data for plotting

```

```

X_xor = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
y_xor = np.array([0, 1, 1, 0])

# Single Perceptron Visualization
plt.figure(figsize=(6, 5))

# Plot actual XOR outputs
plt.scatter(X_xor[y_xor == 0, 0], X_xor[y_xor == 0, 1], color='red', marker='o', label='Expected 0')
plt.scatter(X_xor[y_xor == 1, 0], X_xor[y_xor == 1, 1], color='blue', marker='x', label='Expected 1')

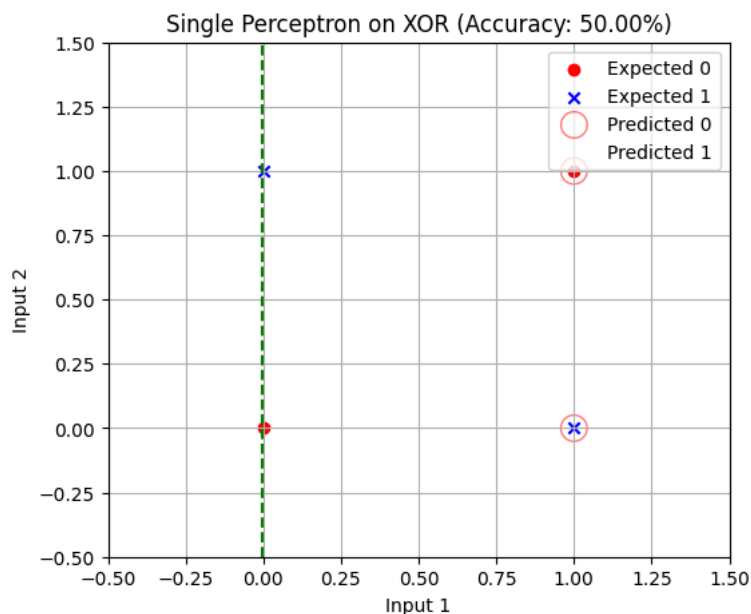
# Plot single perceptron predictions
y_pred_xor = perceptron_xor.predict(X_xor)
plt.scatter(X_xor[y_pred_xor == 0, 0], X_xor[y_pred_xor == 0, 1], color='red', marker='o', alpha=0.5, s=200, facecolors='none')
plt.scatter(X_xor[y_pred_xor == 1, 0], X_xor[y_pred_xor == 1, 1], color='blue', marker='x', alpha=0.5, s=200, edgecolors='black')

# Plot decision boundary for the single perceptron
# Only if weights are not all zeros
if np.any(perceptron_xor.weights):
    x_min, x_max = X_xor[:, 0].min() - 0.5, X_xor[:, 0].max() + 0.5
    y_min, y_max = X_xor[:, 1].min() - 0.5, X_xor[:, 1].max() + 0.5
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100), np.linspace(y_min, y_max, 100))
    Z = np.array([step_function(np.dot(np.array([xi, yi]), perceptron_xor.weights) + perceptron_xor.bias) for xi, yi in np.ndindex(xx.shape)])
    Z = Z.reshape(xx.shape)
    plt.contour(xx, yy, Z, levels=[0.5], colors='green', linestyle='--') # Removed label

plt.title(f'Single Perceptron on XOR (Accuracy: {accuracy*100:.2f}%)')
plt.xlabel('Input 1')
plt.ylabel('Input 2')
plt.legend()
plt.grid(True)
plt.show()

```

/tmp/ipython-input-2357768715.py:22: UserWarning: You passed a edgecolor/edgecolors ('blue') for an unfilled marker ('x').
 plt.scatter(X_xor[y_pred_xor == 1, 0], X_xor[y_pred_xor == 1, 1], color='blue', marker='x', alpha=0.5, s=200, facecolors='none')
/tmp/ipython-input-2357768715.py:32: UserWarning: The following kwargs were not used by contour: 'label'
 plt.contour(xx, yy, Z, levels=[0.5], colors='green', linestyle='--', label='Decision Boundary')



```

import matplotlib.pyplot as plt

# Multilayer Perceptron Visualization
plt.figure(figsize=(6, 5))

# Plot actual XOR outputs
plt.scatter(X_xor[y_xor == 0, 0], X_xor[y_xor == 0, 1], color='red', marker='o', label='Expected 0')
plt.scatter(X_xor[y_xor == 1, 0], X_xor[y_xor == 1, 1], color='blue', marker='x', label='Expected 1')

# Plot MLP predictions
y_pred_mlp = mlp_xor.predict(X_xor)
plt.scatter(X_xor[y_pred_mlp == 0, 0], X_xor[y_pred_mlp == 0, 1], color='red', marker='o', alpha=0.5, s=200, facecolors='none')
plt.scatter(X_xor[y_pred_mlp == 1, 0], X_xor[y_pred_mlp == 1, 1], color='blue', marker='x', alpha=0.5, s=200, edgecolors='black')

```

```
# For MLP, visualizing the decision boundary is more complex as it's non-linear
# This code provides a basic contour plot for the output layer's decision
# However, it might not perfectly represent the full non-linear boundary of the hidden layer
x_min, x_max = X_xor[:, 0].min() - 0.5, X_xor[:, 0].max() + 0.5
y_min, y_max = X_xor[:, 1].min() - 0.5, X_xor[:, 1].max() + 0.5
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100), np.linspace(y_min, y_max, 100))
Z = mlp_xor.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, levels=[-0.5, 0.5, 1.5], alpha=0.3, colors=['red', 'blue'])

plt.title(f'Multilayer Perceptron on XOR (Accuracy: {accuracy_mlp*100:.2f}%)')
plt.xlabel('Input 1')
plt.ylabel('Input 2')
plt.legend()
plt.grid(True)
plt.show()
```

/tmp/ipython-input-1375367668.py:13: UserWarning: You passed a edgecolor/edgecolors ('blue') for an unfilled marker ('x').
 plt.scatter(X_xor[y_pred_mlp == 1, 0], X_xor[y_pred_mlp == 1, 1], color='blue', marker='x', alpha=0.5, s=200, facecolors='')

