Experiment: 2

AIM: XOR Prediction with Unit Step Function MLP Model

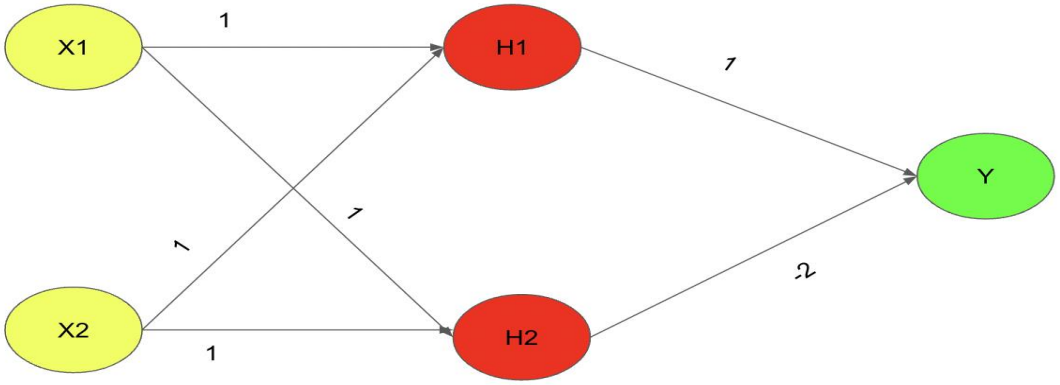
Theory:

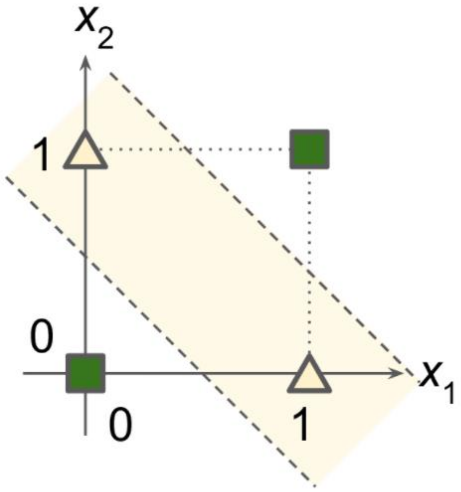
- The XOR (exclusive OR) operation is a classic problem in neural network learning. It involves predicting an output based on two binary inputs.
- A Multilayer Perceptron (MLP) with a unit step function can be employed to solve this problem.
- The unit step function is utilized as the activation function, helping in creating non-linear decision boundaries.
- The objective of this code is to implement an MLP model with a unit step function to predict the XOR operation's output accurately.

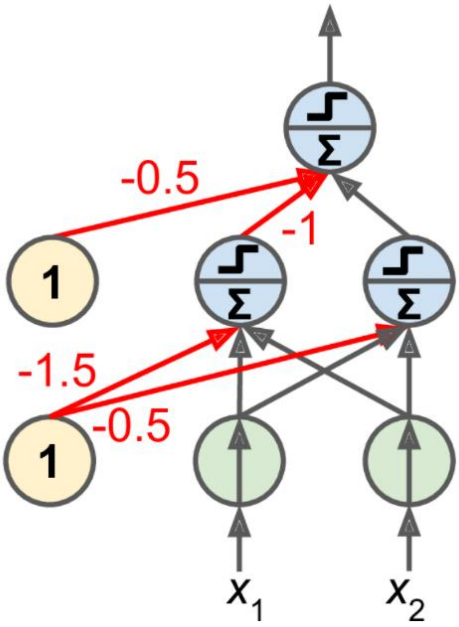
1. **Forward Pass:**
   - **Activation of Hidden Layer (a1):** $a1 = X \cdot w1$
   - **Output of Hidden Layer (z1):** $z1 = \text{unit\_step}(a1)$
   - **Add Bias:** $z1 = \text{concatenate}(1, z1)$
   - **Activation of Output Layer (a2):** $a2 = z1 \cdot w2$
   - **Output Prediction (z2):** $z2 = \text{unit\_step}(a2)$
2. **Backpropagation:**
   - **Output Layer Error (delta2):** $\delta_2 = z2 - y$
   - **Update for Output Layer Weights (w2):** $w2 = w2 - \text{lr} \times \frac{1}{m} \times \Delta_2$
   - **Hidden Layer Error (delta1):** $\delta_1 = (\delta_2 \cdot w2[1:, :].T) \times 1$
   - **Update for Hidden Layer Weights (w1):** $w1 = w1 - \text{lr} \times \frac{1}{m} \times \Delta_1$

-
- X-OR Neural Network
- 1. Forward Pass:
- - Calculate the activation of the hidden layer (a1) using the dot product of input and weights.
- - Apply the unit step function to obtain the hidden layer's output (z1).
- - Add bias to the hidden layer output.
- - Calculate the activation of the output layer (a2) using the dot product of hidden layer output and weights.
- - Apply the unit step function to obtain the final output (z2).
-
- 2. Backpropagation:
- - Calculate the error in the output layer (delta2) by finding the difference between predicted output and actual output.
- - Update the weights of the output layer (w2) using the gradients obtained from the output layer error.

- - Calculate the error in the hidden layer (delta1) by backpropagating the error from the output layer.
- - Update the weights of the hidden layer (w1) using the gradients obtained from the hidden layer error.







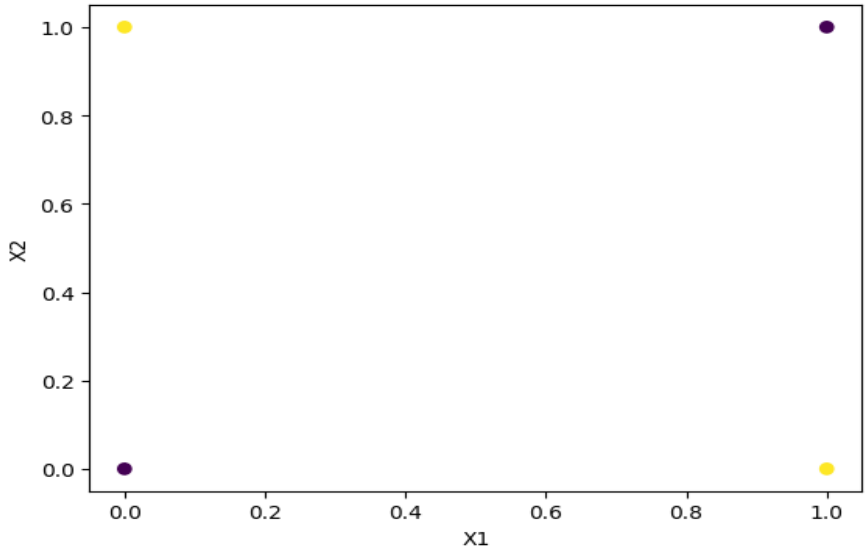
Code:

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

def unit_step(x):
    return np.where(x > 0, 1, 0)

def forward(x, w1, w2, predict=False):
    a1 = np.dot(x, w1)
    z1 = unit_step(a1)
    bias = np.ones((len(x), 1))
    z1 = np.concatenate((bias, z1), axis=1)
    a2 = np.dot(z1, w2)
    z2 = unit_step(a2)
    if predict:
        return z2
    return a1, z1, a2, z2

def backprop(a2, X, z1, z2, y, w2, a1):
    delta2 = z2 - y
    Delta2 = np.dot(z1.T, delta2)
    delta1 = (delta2.dot(w2[1:, :].T)) * 1
    Delta1 = np.dot(X.T, delta1)
    return delta2, delta1, Delta1, Delta2

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

Y = np.array([1, 1, 0, 0]).reshape(-1, 1)

w1 = np.random.randn(3, 5)
w2 = np.random.randn(6, 1)

lr = 0.1
costs = []
epochs = 200
m = len(X)

for i in range(epochs):
    a1, z1, a2, z2 = forward(X, w1, w2)
```

```
    delta2, delta1, Delta1, Delta2 = backprop(a2, X, z1, z2, Y, w2, a1)
    w1 = w1 - lr * (1/m) * Delta1
    w2 = w2 - lr * (1/m) * Delta2
    c = np.mean(np.abs(delta2))
    costs.append(c)

predictions = forward(X, w1, w2, predict=True)
print("Predicted Output:")
for i in range(len(X)):
    print(f"Input: {X[i, 1:]}, Actual: {Y[i][0]}, Predicted Output:
{predictions[i][0]}")

plt.plot(costs)
plt.title("Training Loss over Iterations")
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.show()

x_min, x_max = X[:, 1].min() - 0.1, X[:, 1].max() + 0.1
y_min, y_max = X[:, 2].min() - 0.1, X[:, 2].max() + 0.1
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100),
                     np.linspace(y_min, y_max, 100))

grid_data = np.c_[np.ones(xx.ravel().shape[0]), xx.ravel(), yy.ravel()]

Z = forward(grid_data, w1, w2, predict=True)
Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, cmap=plt.cm.RdBu, alpha=0.8)
plt.scatter(X[Y.ravel() == 0, 1], X[Y.ravel() == 0, 2], color='red',
label='Class 0')
plt.scatter(X[Y.ravel() == 1, 1], X[Y.ravel() == 1, 2], color='blue',
label='Class 1')

plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Classification with Neural Network (XOR Gate)')
plt.legend()
plt.show()
```

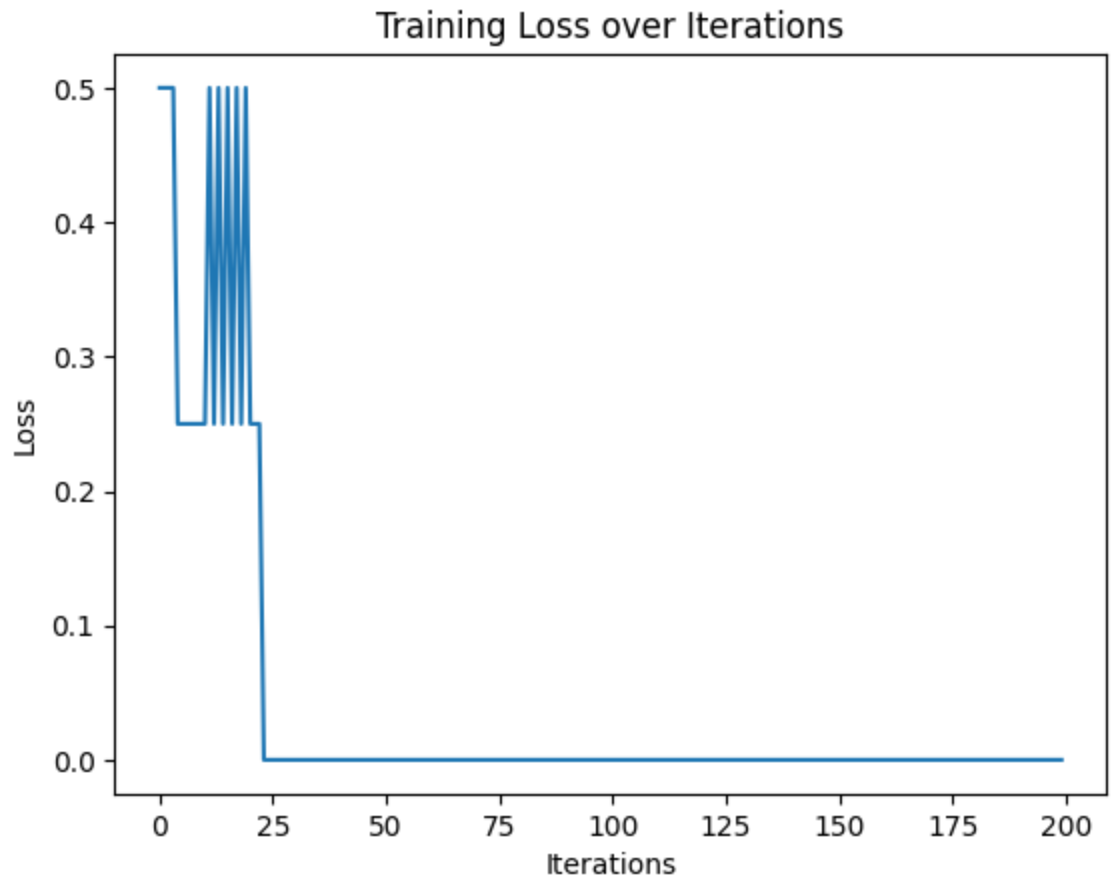
Output:

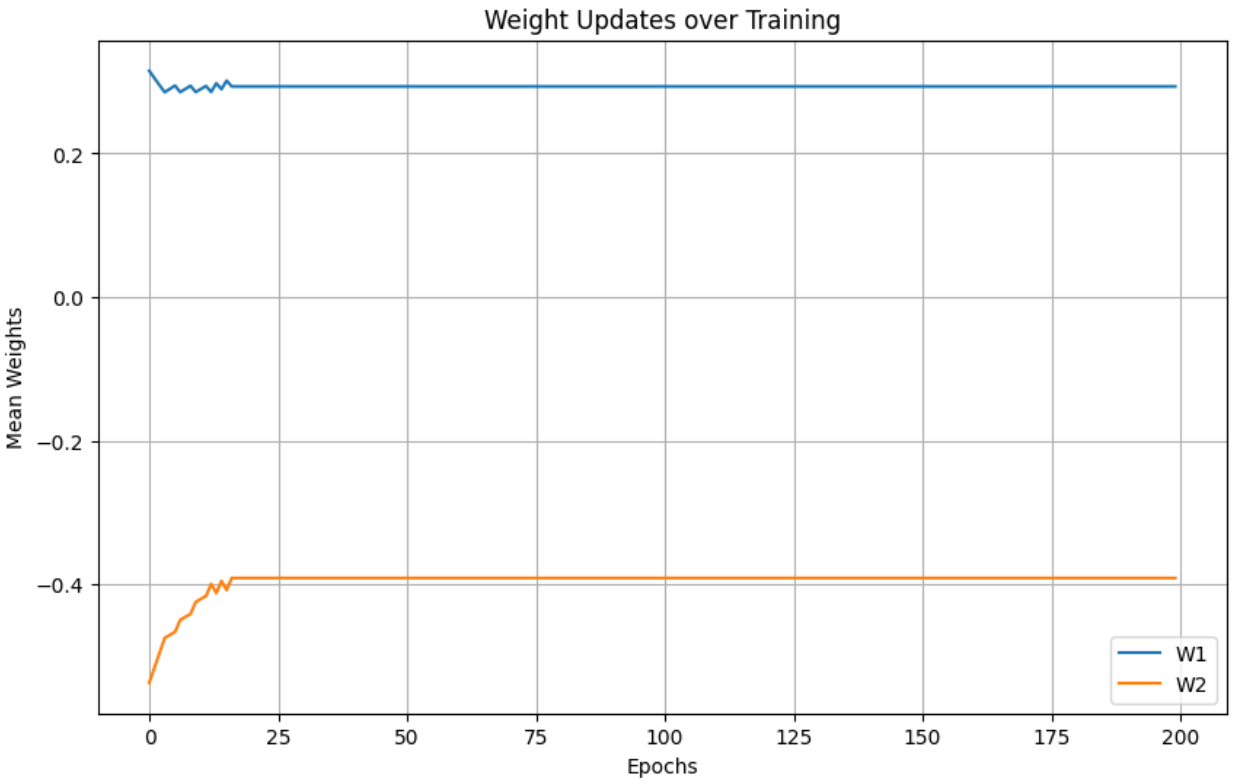
- **Plotting the input data:**



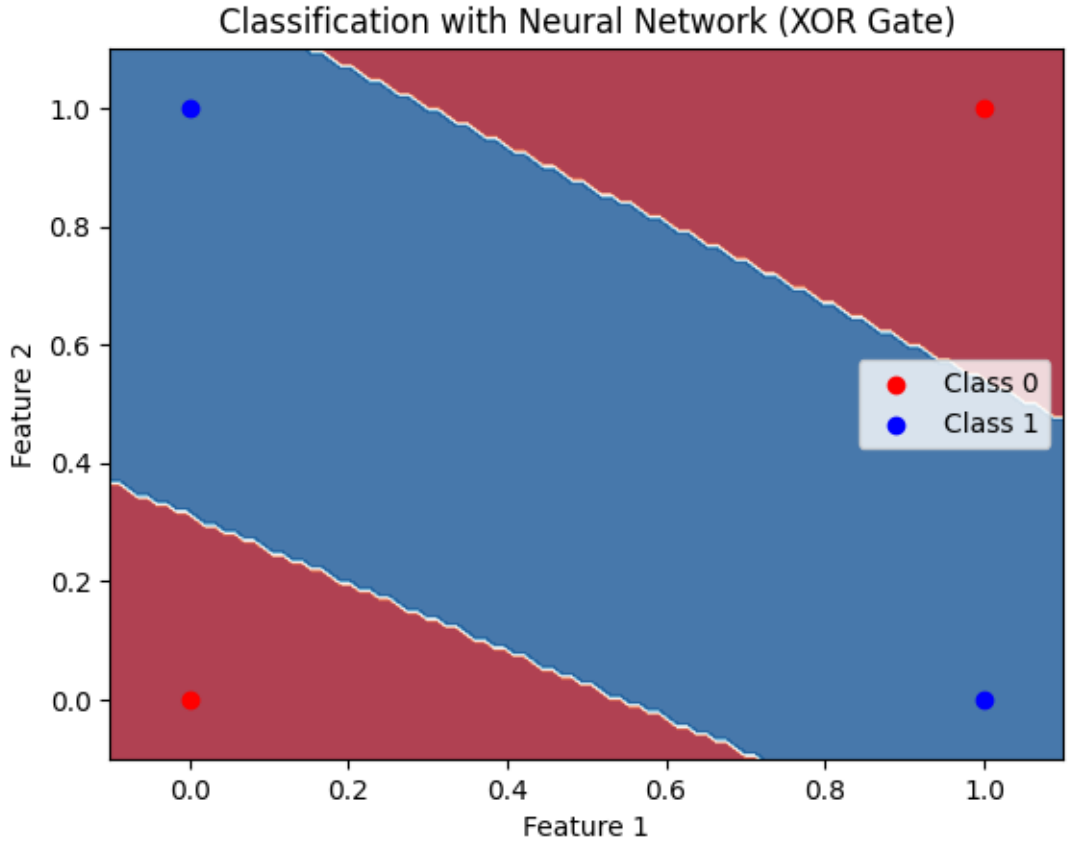
- **Making predictions on the trained model:**

```
   Predicted Output:

Input: [1 0], Actual: 1, Predicted Output: 1
Input: [0 1], Actual: 1, Predicted Output: 1
Input: [0 0], Actual: 0, Predicted Output: 0

Input: [1 1], Actual: 0, Predicted Output: 0
```


Training Loss over Iterations
Loss
Iterations
0.5
0.4
0.3
0.2
0.1
0.0
0
25
50
75
100
125
150
175
200


Weight Updates over Training
Mean Weights
Epochs
0.2
0.0
−0.2
−0.4
0
25
50
75
100
125
150
175
200
W1
W2

Conclusion:



- ⇨ The XOR prediction problem has been successfully solved using the implemented MLP model with a unit step function.
- ⇨ The model demonstrates accurate predictions for all possible input combinations.
- ⇨ The training loss decreases over iterations, indicating the model's learning process. Additionally, the decision boundary plotted visually confirms the model's effectiveness in separating the two classes corresponding to the XOR operation.
- ⇨ This solution showcases the capability of neural networks, particularly MLPs, in solving non-linear classification problems like XOR.

- ⇨ w1 is [-1.38615824 0.30280919 0.45994265 0.53626582 0.30153141]
- ⇨ w2 is [0.00763391 -0.78017054 -1.94433573 0.56364245 0.0914174]
- ⇨ bias is [1.23841439 -0.34564419 0.49192637 1.57695413 -0.45767063]
- ⇨ error loss is 0.0