```python
import numpy as np

# Sigmoid and derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

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

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

# Set random seed
np.random.seed(42)

# Network architecture
input_size = 2
hidden_size = 2
output_size = 1

# Initialize weights and biases
W1 = np.random.randn(input_size, hidden_size)
b1 = np.zeros((1, hidden_size))
W2 = np.random.randn(hidden_size, output_size)
b2 = np.zeros((1, output_size))

# Hyperparameters
lr = 0.1
epochs = 10000

# Training
for epoch in range(epochs + 1):
    # Forward pass
    Z1 = np.dot(X, W1) + b1
    A1 = sigmoid(Z1)

    Z2 = np.dot(A1, W2) + b2
    A2 = sigmoid(Z2)

    # Compute loss
    loss = np.mean((Y - A2) ** 2)

    # Print every 1000 epochs
    if epoch % 1000 == 0:
        print(f"Epoch {epoch}, Loss : {loss:.4f}")

    # Backpropagation
    dA2 = (A2 - Y)
    dZ2 = dA2 * sigmoid_derivative(A2)
    dW2 = np.dot(A1.T, dZ2)
    dB2 = np.sum(dZ2, axis=0, keepdims=True)

    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * sigmoid_derivative(A1)
    dW1 = np.dot(X.T, dZ1)
    dB1 = np.sum(dZ1, axis=0, keepdims=True)

    # Update weights
    W1 -= lr * dW1
    b1 -= lr * dB1
    W2 -= lr * dW2
    b2 -= lr * dB2

# Final predictions
print("\nFinal predictions:")
print(A2)
```

```
Epoch 0, Loss : 0.2558
Epoch 1000, Loss : 0.2494
Epoch 2000, Loss : 0.2454
Epoch 3000, Loss : 0.2047
Epoch 4000, Loss : 0.1532
Epoch 5000, Loss : 0.1387
Epoch 6000, Loss : 0.1336
Epoch 7000, Loss : 0.1312
Epoch 8000, Loss : 0.1297
Epoch 9000, Loss : 0.1288
Epoch 10000, Loss : 0.1282

Final predictions:
[[0.05300376]
 [0.49554286]
 [0.95091752]
 [0.50319846]]
```

9/08/25

Exp-6 Gradient Descent and Backpropagation

Aim:

To implement Gradient Descent and Backpropagation for training a simple feedforward neural network on the XOR problem.

Description:

* Gradient Descent is an optimization algorithm used to minimize the loss function by updating weights in the opposite direction of the gradient

* Backpropagation is the process of calculating the gradient of the loss function with respect to each weight using the chain rule, so we can apply gradient descent efficiently.

The steps are:

1.) Forward pass - Compute output from inputs

2.) Compute loss - Difference between predicted and actual values.

3.) Backward Pass - Compute gradients of Loss w.r.t weights.

4.) Update weights - Use Gradient Descent rule:

$$w = w - \eta \cdot \frac{dL}{dw}$$

where η is the learning rate.

Procedure :

1.) Initialize weights randomly

2.) Forward Pass

→ Compute the hidden layer

→ Compute the final output

3.) Compute loss : Mean Squared error between predicted output and actual output

4.) Backpropagation:

5.) Weight update :

update weights and biases using Gradient Descent

$$\omega = \omega + \eta \cdot \frac{\partial L}{\partial \omega}$$

6.) Repeat steps 2-5 for several epochs until the loss converges.

7.) Print trial predictions after training

CODE :

```
import numpy as np
def sigmoid (x):
    return 1/(1+ np.exp(-x))
def sigmoid-derivative (x):
    return x * (1-x)
X = np.array ([[0,0], [0,1], [1,0], [1,1]])
y = np. array ([[0], [1], [1], [0]])
np. random. seed (42)
input-size = 2
```

```
hidden-size = 2
output-size = 1

w1 = np.random.randn(input-size, hidden-size)
b1 = np.zeros((1, hidden-size))
w2 = np.random.randn(hidden-size, output-size)
b2 = np.zeros((1, output-size))

lr = 0.1
epochs = 1000
for epoch in range(epochs):
    z1 = np.dot(x, w1) + b1
    a1 = sigmoid(z1)
    z2 = np.dot(a1, w2) + b2
    a2 = sigmoid(z2)
loss = np.mean((y-a2)**2)
    d-a2 = (a2-y)
    d-z2 = d-a2 * sigmoid-derivative(a2)
    d-w2 = np.dot(a1.T, d-z2)
    d-b2 = np.sum(d-z2, axis=0, keepdims = True)
    d-a1 = np.dot(d-z2, w2.T)
    d-z1 = d-a1 * sigmoid-derivative(a1)
    dw1 = np.dot(x.T, d-z1)
    d-b1 = np.sum(d-z1, axis=0, keepdims = True)
    w1 = lr * d-w1            Next line    w2 = lr * dw2
    b2 -= lr * d-b1                        b2 = lr * d-b2
if epoch % 1000 == 0; N/L print(f"Epoch {epoch}, loss
: {Loss: %f}") N/L print("\n Final prediction) N/L p(a)
```

9/9/25

Result: The code has been successfully executed and shows the network learned the XOR logic output near 0 for [0,0] and [1,1] and near 1 for [0,1] and [1,0]

Output:

Epoch 0, Loss: 0.2558

Epoch 1000, Loss: 0.2494

Epoch 2000, Loss: 0.2454

Epoch 3000, loss: 0.2047

Epoch 4000, loss: 0.1532

Epoch 5000, Loss: 0.1387

Epoch 6000, Loss: 0.1336

Epoch 7000, Loss: 0.1312

Epoch 8000, Loss: 0.1297

Epoch 9000, Loss: 0.1288

Final Predictions:

[[0.05300868]
[0.49554213]
[0.95091319]
[0.5081988]]