```
import numpy as np
```

```
# ---------------------------------------------------
# Sigmoid Activation Function
# ---------------------------------------------------
def sigmoid(x):
    """
    Sigmoid activation function.
    Converts input value into range (0, 1).
    Used in both hidden and output layers.
    """
    return 1 / (1 + np.exp(-x))
```

```
# ---------------------------------------------------
# Derivative of Sigmoid Function
# ---------------------------------------------------
def sigmoid_derivative(x):
    """
    Derivative of sigmoid function.
    Used during backpropagation to compute gradients.
    Formula: σ'(x) = σ(x) * (1 - σ(x))
    """
    return x * (1 - x)
```

```
# ---------------------------------------------------
# Neural Network Class Definition
# ---------------------------------------------------
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        """
        Constructor to initialize the neural network parameters.

        input_size  : Number of input neurons (x1, x2)
        hidden_size : Number of hidden neurons (h1, h2)
        output_size : Number of output neurons (y)
        """

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # ---------------------------------------------------
        # Weights from Input Layer to Hidden Layer
        # Taken EXACTLY from 4th example in the table:
        #
        # h1 weights = (-0.5, 0.9)
        # h2 weights = (0.4, 0.1)
        # ---------------------------------------------------
        self.weights_input_hidden = np.array([
            [-0.5,  0.4],   # weights from x1 → h1, h2
            [ 0.9,  0.1]    # weights from x2 → h1, h2
        ])

        # Bias for hidden layer
        # Bias values are not given in the table,
        # so we assume them as zero.
        self.bias_hidden = np.zeros((1, self.hidden_size))

        # ---------------------------------------------------
        # Weights from Hidden Layer to Output Layer
        # From 4th example:
        #
        # h1 → y = 0.2
```

```
        # h2 → y = -0.6
        # -------------------------------------------------
        self.weights_hidden_output = np.array([
            [ 0.2],
            [-0.6]
        ])

        # Bias for output layer (assumed zero)
        self.bias_output = np.zeros((1, self.output_size))

    # -------------------------------------------------
    # Forward Propagation
    # -------------------------------------------------
    def forward(self, X):
        """
        Performs forward propagation.

        Steps:
        1. Input → Hidden (weighted sum + bias)
        2. Apply sigmoid activation
        3. Hidden → Output (weighted sum + bias)
        4. Apply sigmoid activation
        """

        # Store input values
        self.input_layer = X

        # Calculate net input to hidden layer
        self.hidden_layer_input = np.dot(
            self.input_layer,
            self.weights_input_hidden
        ) + self.bias_hidden

        # Apply sigmoid activation to hidden layer
        self.hidden_layer_output = sigmoid(self.hidden_layer_input)

        # Calculate net input to output layer
        self.output_layer_input = np.dot(
            self.hidden_layer_output,
            self.weights_hidden_output
        ) + self.bias_output

        # Apply sigmoid activation to output layer
        self.output_layer_output = sigmoid(self.output_layer_input)

        return self.output_layer_output

    # -------------------------------------------------
    # Backpropagation
    # -------------------------------------------------
    def backward(self, X, y, learning_rate):
        """
        Performs backpropagation to update weights.

        Steps:
        1. Calculate output error
        2. Compute delta for output layer
        3. Backpropagate error to hidden layer
        4. Update all weights and biases
        """

        # Error at output layer
        error_output = y - self.output_layer_output

        # Delta at output layer
        output_layer_delta = error_output * sigmoid_derivative(
            self.output_layer_output
```

```
        )

        # Error propagated to hidden layer
        error_hidden = output_layer_delta.dot(
            self.weights_hidden_output.T
        )

        # Delta at hidden layer
        hidden_layer_delta = error_hidden * sigmoid_derivative(
            self.hidden_layer_output
        )

        # Update weights from hidden → output
        self.weights_hidden_output += self.hidden_layer_output.T.dot(
            output_layer_delta
        ) * learning_rate

        # Update output bias
        self.bias_output += np.sum(
            output_layer_delta,
            axis=0,
            keepdims=True
        ) * learning_rate

        # Update weights from input → hidden
        self.weights_input_hidden += X.T.dot(
            hidden_layer_delta
        ) * learning_rate

        # Update hidden bias
        self.bias_hidden += np.sum(
            hidden_layer_delta,
            axis=0,
            keepdims=True
        ) * learning_rate

    # ---------------------------------------------------
    # Training Function
    # ---------------------------------------------------
    def train(self, X, y, epochs, learning_rate):
        """
        Trains the network for a given number of epochs.
        """

        for epoch in range(epochs):
            self.forward(X)
            self.backward(X, y, learning_rate)

            # Mean Squared Error Loss
            if epoch % 1000 == 0:
                loss = np.mean(
                    np.square(y - self.output_layer_output)
                )
                print(f"Epoch {epoch} - Loss: {loss}")
```

```
# ---------------------------------------------------
# Main Program (4th Example)
# ---------------------------------------------------
if __name__ == "__main__":

    # Input from 4th table example: (0, 1)
    X = np.array([[0, 1]])

    # Target output from table
    y = np.array([[0]])
```

```
    # Learning rate from table
    learning_rate = 0.1

    # Create neural network
    nn = NeuralNetwork(input_size=2, hidden_size=2, output_size=1)

    # Forward pass before training
    print("Output before training:")
    print(nn.forward(X))

    # One backpropagation step (as in numerical problems)
    nn.backward(X, y, learning_rate)

    # Forward pass after training
    print("\nOutput after one training step:")
    print(nn.forward(X))
```

```
Output before training:
[[0.45690777]]

Output after one training step:
[[0.45176539]]
```

Start coding or generate with AI.