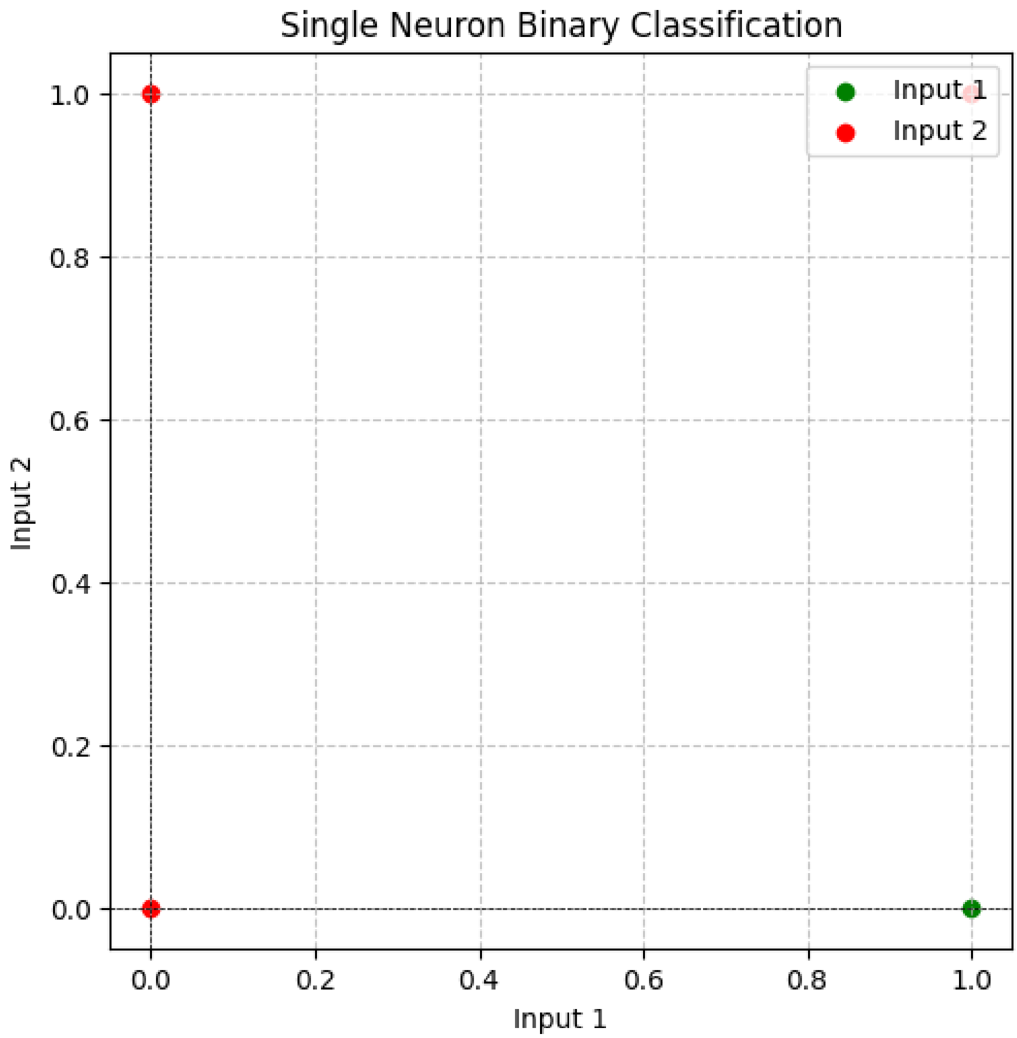
LAB 1

experiment 1

```
# Import required libraries
import numpy as np
import matplotlib.pyplot as plt

def step_function(x):
    """
    Step activation function.
    Returns 1 if x >= 0, else 0.
    """
    return 1 if x >= 0 else 0

class SingleNeuron:
    def __init__(self, weights, bias):
        """
        Initialize the neuron with given weights and bias.

        Args:
            weights (list or np.ndarray): Weights for the inputs.
            bias (float): Bias term.
        """
        self.weights = np.array(weights)
        self.bias = bias

    def predict(self, inputs):
        """
        Perform a forward pass of the neuron.

        Args:
            inputs (list or np.ndarray): Input values.

        Returns:
            int: Output of the neuron after applying the activation
function.
        """
        inputs = np.array(inputs)
        linear_output = np.dot(self.weights, inputs) + self.bias
        return step_function(linear_output)

# Example usage
if __name__ == "__main__":
    # Define weights and bias
    weights = [0.5, -0.6]
    bias = -0.2

    # Create an instance of the SingleNeuron class
    neuron = SingleNeuron(weights, bias)
```

```python
    # Define input samples
    inputs_list = [
        [1, 0],  # Example 1
        [0, 1],  # Example 2
        [1, 1],  # Example 3
        [0, 0]   # Example 4
    ]

    # Predict and display the outputs
    print("Input\tOutput")
    outputs = []
    for inputs in inputs_list:
        output = neuron.predict(inputs)
        outputs.append(output)
        print(f"{inputs}\t{output}")

    # Plot the results
    inputs_array = np.array(inputs_list)
    outputs_array = np.array(outputs)

    plt.figure(figsize=(6, 6))
    for i, inputs in enumerate(inputs_list):
        color = 'green' if outputs[i] == 1 else 'red'
        plt.scatter(inputs[0], inputs[1], color=color, label=f"Input
{i+1}" if i < 2 else None)

    plt.axhline(0, color='black', linewidth=0.5, linestyle='--')
    plt.axvline(0, color='black', linewidth=0.5, linestyle='--')
    plt.title("Single Neuron Binary Classification")
    plt.xlabel("Input 1")
    plt.ylabel("Input 2")
    plt.grid(True, linestyle='--', alpha=0.7)
    plt.legend(loc='upper right')
    plt.show()
```

```
Input	Output
[1, 0]	1
[0, 1]	0
[1, 1]	0
[0, 0]	0
```



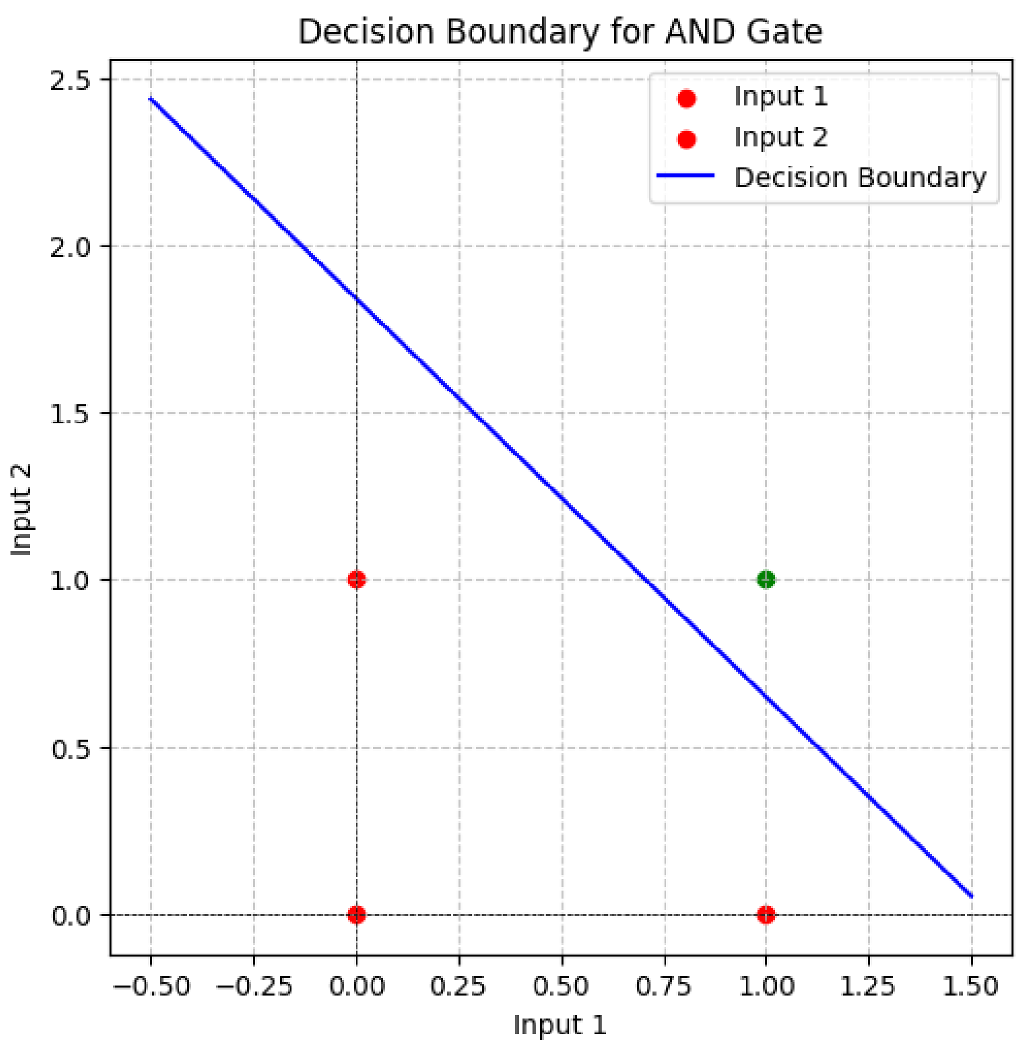
experiment 2

```
# Import required libraries
import numpy as np
import matplotlib.pyplot as plt

def step_function(x):
    """
    Step activation function.
    Returns 1 if x >= 0, else 0.
    """
    return 1 if x >= 0 else 0

class SingleLayerPerceptron:
    def __init__(self, input_dim, learning_rate=0.1):
        """
        Initialize the perceptron with random weights and bias.
```

```
        Args:
            input_dim (int): Number of input features.
            learning_rate (float): Learning rate for weight updates.
        """
        self.weights = np.random.rand(input_dim)
        self.bias = np.random.rand()
        self.learning_rate = learning_rate

    def predict(self, inputs):
        """
        Perform a forward pass of the perceptron.

        Args:
            inputs (list or np.ndarray): Input values.

        Returns:
            int: Output of the perceptron after applying the
activation function.
        """
        inputs = np.array(inputs)
        linear_output = np.dot(self.weights, inputs) + self.bias
        return step_function(linear_output)

    def train(self, X, y, epochs=10):
        """
        Train the perceptron using the provided dataset.

        Args:
            X (np.ndarray): Input dataset of shape (n_samples,
n_features).
            y (np.ndarray): Target labels of shape (n_samples,).
            epochs (int): Number of epochs for training.
        """
        for epoch in range(epochs):
            for inputs, target in zip(X, y):
                prediction = self.predict(inputs)
                error = target - prediction
                # Update weights and bias
                self.weights += self.learning_rate * error * inputs
                self.bias += self.learning_rate * error

# Example usage
if __name__ == "__main__":
    # Define training data for AND gate
    X = np.array([
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1]
    ])
```

```
    y_and = np.array([0, 0, 0, 1])  # AND gate targets
    y_or = np.array([0, 1, 1, 1])   # OR gate targets

    # Initialize and train the perceptron for AND gate
    perceptron_and = SingleLayerPerceptron(input_dim=2,
learning_rate=0.1)
    perceptron_and.train(X, y_and, epochs=10)

    # Test the perceptron on AND gate
    print("AND Gate")
    print("Input\tOutput")
    for inputs in X:
        output = perceptron_and.predict(inputs)
        print(f"{inputs}\t{output}")

    # Initialize and train the perceptron for OR gate
    perceptron_or = SingleLayerPerceptron(input_dim=2,
learning_rate=0.1)
    perceptron_or.train(X, y_or, epochs=10)

    # Test the perceptron on OR gate
    print("\nOR Gate")
    print("Input\tOutput")
    for inputs in X:
        output = perceptron_or.predict(inputs)
        print(f"{inputs}\t{output}")

    # Plot the decision boundary for AND gate
    plt.figure(figsize=(6, 6))
    for i, inputs in enumerate(X):
        color = 'green' if y_and[i] == 1 else 'red'
        plt.scatter(inputs[0], inputs[1], color=color, label=f"Input
{i+1}" if i < 2 else None)

    x_values = np.linspace(-0.5, 1.5, 100)
    y_values = -(perceptron_and.weights[0] * x_values +
perceptron_and.bias) / perceptron_and.weights[1]
    plt.plot(x_values, y_values, label="Decision Boundary",
color="blue")

    plt.axhline(0, color='black', linewidth=0.5, linestyle='--')
    plt.axvline(0, color='black', linewidth=0.5, linestyle='--')
    plt.title("Decision Boundary for AND Gate")
    plt.xlabel("Input 1")
    plt.ylabel("Input 2")
    plt.grid(True, linestyle='--', alpha=0.7)
    plt.legend(loc='upper right')
    plt.show()
```

```
AND Gate
Input Output
[0 0] 0
[0 1] 0
[1 0] 0
[1 1] 1

OR Gate
Input Output
[0 0] 0
[0 1] 1
[1 0] 1
[1 1] 1
```



experiment 3

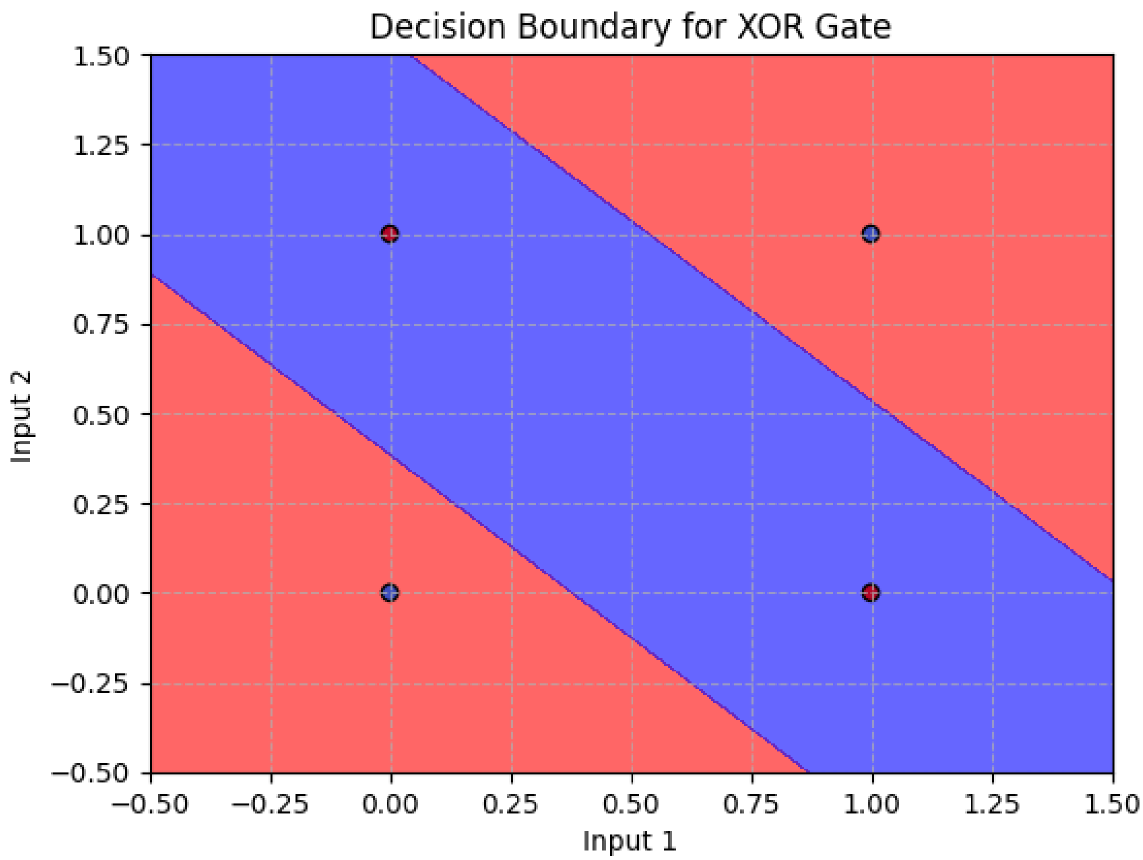
```python
# Import required libraries
import numpy as np
import matplotlib.pyplot as plt

def sigmoid(x):
    """
    Sigmoid activation function.
    """
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    """
    Derivative of the sigmoid function.
    """
    return x * (1 - x)

class MultiLayerPerceptron:
    def __init__(self, input_dim, hidden_dim, output_dim,
learning_rate=0.1):
        """
        Initialize the MLP with random weights and biases.

        Args:
            input_dim (int): Number of input features.
            hidden_dim (int): Number of neurons in the hidden layer.
            output_dim (int): Number of output neurons.
            learning_rate (float): Learning rate for weight updates.
        """
        self.learning_rate = learning_rate

        # Initialize weights and biases
        self.weights_input_hidden = np.random.rand(input_dim,
hidden_dim)
        self.bias_hidden = np.random.rand(hidden_dim)
        self.weights_hidden_output = np.random.rand(hidden_dim,
output_dim)
        self.bias_output = np.random.rand(output_dim)

    def forward(self, inputs):
        """
        Perform a forward pass through the network.

        Args:
            inputs (np.ndarray): Input values.

        Returns:
            tuple: Outputs of hidden and output layers.
        """
        self.input_layer = inputs
        self.hidden_layer_input = np.dot(inputs,
```

```
self.weights_input_hidden) + self.bias_hidden
        self.hidden_layer_output = sigmoid(self.hidden_layer_input)
        self.output_layer_input = np.dot(self.hidden_layer_output,
self.weights_hidden_output) + self.bias_output
        self.output_layer_output = sigmoid(self.output_layer_input)
        return self.output_layer_output

    def backward(self, target_output):
        """
        Perform a backward pass and update weights and biases.

        Args:
            target_output (np.ndarray): Target output values.
        """
        # Compute error at output layer
        output_error = target_output - self.output_layer_output
        output_delta = output_error *
sigmoid_derivative(self.output_layer_output)

        # Compute error at hidden layer
        hidden_error = np.dot(output_delta,
self.weights_hidden_output.T)
        hidden_delta = hidden_error *
sigmoid_derivative(self.hidden_layer_output)

        # Update weights and biases
        self.weights_hidden_output += self.learning_rate *
np.dot(self.hidden_layer_output.T, output_delta)
        self.bias_output += self.learning_rate * np.sum(output_delta,
axis=0)
        self.weights_input_hidden += self.learning_rate *
np.dot(self.input_layer.T, hidden_delta)
        self.bias_hidden += self.learning_rate * np.sum(hidden_delta,
axis=0)

    def train(self, X, y, epochs=10000):
        """
        Train the MLP using the provided dataset.

        Args:
            X (np.ndarray): Input dataset of shape (n_samples,
n_features).
            y (np.ndarray): Target labels of shape (n_samples,
n_outputs).
            epochs (int): Number of epochs for training.
        """
        for epoch in range(epochs):
            outputs = self.forward(X)
            self.backward(y)
```

```python
# Example usage
if __name__ == "__main__":
    # Define training data for XOR gate
    X = np.array([
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1]
    ])
    y = np.array([
        [0],
        [1],
        [1],
        [0]
    ])

    # Initialize and train the MLP
    mlp = MultiLayerPerceptron(input_dim=2, hidden_dim=2,
output_dim=1, learning_rate=0.1)
    mlp.train(X, y, epochs=10000)

    # Test the MLP on XOR gate
    print("XOR Gate")
    print("Input Output")
    for inputs in X:
        output = mlp.forward(inputs)
        print(f"{inputs}    {output.round()}")

    # Plot decision boundary
    x_values = np.linspace(-0.5, 1.5, 100)
    y_values = np.linspace(-0.5, 1.5, 100)
    xv, yv = np.meshgrid(x_values, y_values)
    grid_points = np.c_[xv.ravel(), yv.ravel()]
    grid_predictions = np.array([mlp.forward(point) for point in
grid_points])
    grid_predictions = grid_predictions.reshape(xv.shape)

    plt.contourf(xv, yv, grid_predictions, levels=[-0.1, 0.5, 1.1],
colors=['red', 'blue'], alpha=0.6)
    plt.scatter(X[:, 0], X[:, 1], c=y.ravel(), cmap='coolwarm',
edgecolors='k')
    plt.title("Decision Boundary for XOR Gate")
    plt.xlabel("Input 1")
    plt.ylabel("Input 2")
    plt.grid(True, linestyle='--', alpha=0.7)
    plt.show()
```

```
XOR Gate
Input Output
[0 0] [0.]
```

```
[0 1] [1.]
[1 0] [1.]
[1 1] [0.]
```

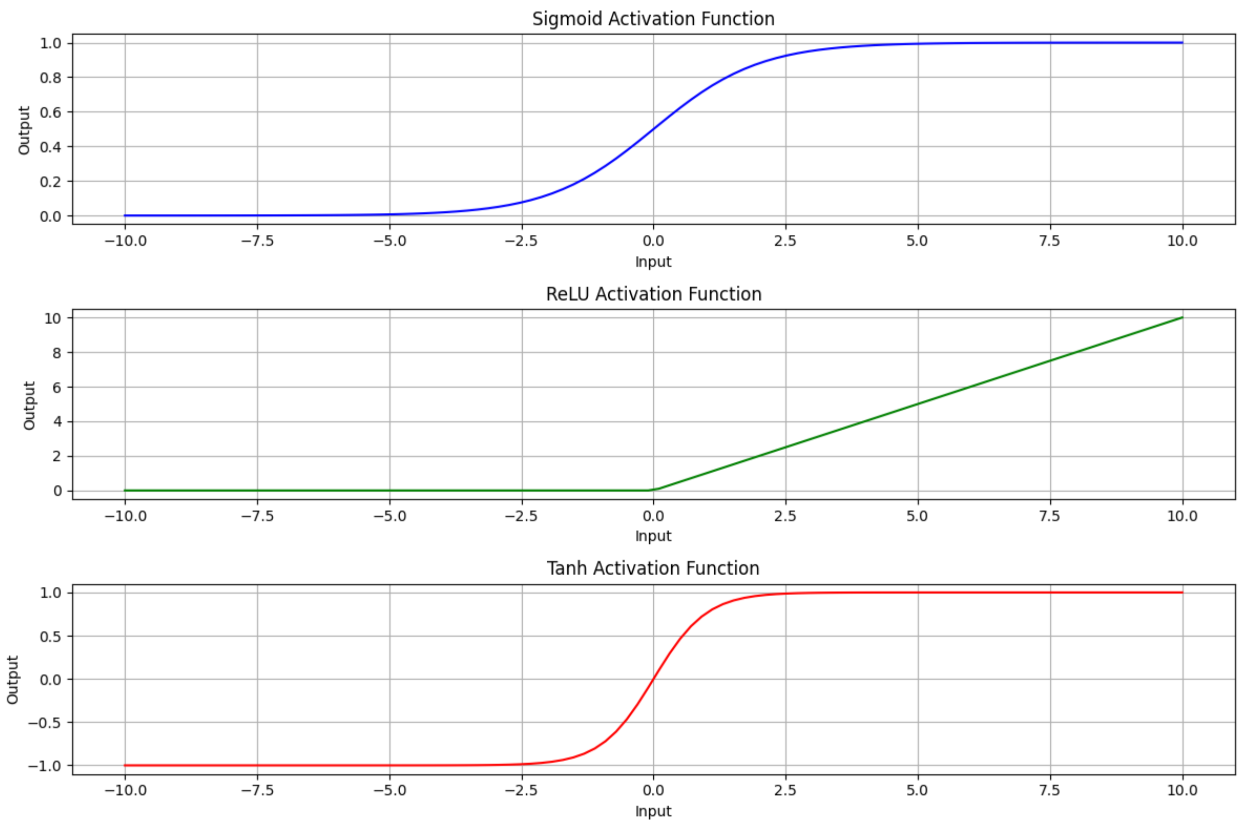


experiment 4

```python
# Import required libraries
import numpy as np
import matplotlib.pyplot as plt

def sigmoid(x):
    """
    Sigmoid activation function.
    """
    return 1 / (1 + np.exp(-x))

def relu(x):
    """
    ReLU activation function.
    """
    return np.maximum(0, x)

def tanh(x):
```

```
    """
    Tanh activation function.
    """
    return np.tanh(x)

def sigmoid_derivative(x):
    """
    Derivative of the sigmoid function.
    """
    return x * (1 - x)

class MultiLayerPerceptron:
    def __init__(self, input_dim, hidden_dim, output_dim,
learning_rate=0.1):
        """
        Initialize the MLP with random weights and biases.

        Args:
            input_dim (int): Number of input features.
            hidden_dim (int): Number of neurons in the hidden layer.
            output_dim (int): Number of output neurons.
            learning_rate (float): Learning rate for weight updates.
        """
        self.learning_rate = learning_rate

        # Initialize weights and biases
        self.weights_input_hidden = np.random.rand(input_dim,
hidden_dim)
        self.bias_hidden = np.random.rand(hidden_dim)
        self.weights_hidden_output = np.random.rand(hidden_dim,
output_dim)
        self.bias_output = np.random.rand(output_dim)

    def forward(self, inputs, activation_function=sigmoid):
        """
        Perform a forward pass through the network.

        Args:
            inputs (np.ndarray): Input values.
            activation_function (function): Activation function to
use.

        Returns:
            tuple: Outputs of hidden and output layers.
        """
        self.input_layer = inputs
        self.hidden_layer_input = np.dot(inputs,
self.weights_input_hidden) + self.bias_hidden
        self.hidden_layer_output =
activation_function(self.hidden_layer_input)
```

```
        self.output_layer_input = np.dot(self.hidden_layer_output,
self.weights_hidden_output) + self.bias_output
        self.output_layer_output =
activation_function(self.output_layer_input)
        return self.output_layer_output

# Example usage
if __name__ == "__main__":
    # Define a sample dataset
    x_values = np.linspace(-10, 10, 100)

    # Compute outputs for different activation functions
    sigmoid_outputs = sigmoid(x_values)
    relu_outputs = relu(x_values)
    tanh_outputs = tanh(x_values)

    # Plot the outputs
    plt.figure(figsize=(12, 8))

    # Sigmoid
    plt.subplot(3, 1, 1)
    plt.plot(x_values, sigmoid_outputs, label="Sigmoid", color="blue")
    plt.title("Sigmoid Activation Function")
    plt.xlabel("Input")
    plt.ylabel("Output")
    plt.grid(True)

    # ReLU
    plt.subplot(3, 1, 2)
    plt.plot(x_values, relu_outputs, label="ReLU", color="green")
    plt.title("ReLU Activation Function")
    plt.xlabel("Input")
    plt.ylabel("Output")
    plt.grid(True)

    # Tanh
    plt.subplot(3, 1, 3)
    plt.plot(x_values, tanh_outputs, label="Tanh", color="red")
    plt.title("Tanh Activation Function")
    plt.xlabel("Input")
    plt.ylabel("Output")
    plt.grid(True)

    plt.tight_layout()
    plt.show()
```



experiment 5

```
# Import required libraries
import numpy as np
import matplotlib.pyplot as plt

def sigmoid(x):
    """
    Sigmoid activation function.
    """
    return 1 / (1 + np.exp(-x))

def relu(x):
    """
    ReLU activation function.
    """
    return np.maximum(0, x)

def tanh(x):
    """
    Tanh activation function.
    """
    return np.tanh(x)

def sigmoid_derivative(x):
```

```
    """
    Derivative of the sigmoid function.
    """
    return x * (1 - x)

class MultiLayerPerceptron:
    def __init__(self, input_dim, hidden_dim, output_dim,
learning_rate=0.1):
        """
        Initialize the MLP with random weights and biases.

        Args:
            input_dim (int): Number of input features.
            hidden_dim (int): Number of neurons in the hidden layer.
            output_dim (int): Number of output neurons.
            learning_rate (float): Learning rate for weight updates.
        """
        self.learning_rate = learning_rate

        # Initialize weights and biases
        self.weights_input_hidden = np.random.rand(input_dim,
hidden_dim)
        self.bias_hidden = np.random.rand(hidden_dim)
        self.weights_hidden_output = np.random.rand(hidden_dim,
output_dim)
        self.bias_output = np.random.rand(output_dim)

    def forward(self, inputs, activation_function=sigmoid):
        """
        Perform a forward pass through the network.

        Args:
            inputs (np.ndarray): Input values.
            activation_function (function): Activation function to
use.

        Returns:
            tuple: Outputs of hidden and output layers.
        """
        self.input_layer = inputs
        self.hidden_layer_input = np.dot(inputs,
self.weights_input_hidden) + self.bias_hidden
        self.hidden_layer_output =
activation_function(self.hidden_layer_input)
        self.output_layer_input = np.dot(self.hidden_layer_output,
self.weights_hidden_output) + self.bias_output
        self.output_layer_output =
activation_function(self.output_layer_input)
        return self.output_layer_output
```

```
    def backward(self, inputs, targets, activation_function=sigmoid):
        """
        Perform backpropagation to update weights and biases.

        Args:
            inputs (np.ndarray): Input values.
            targets (np.ndarray): Target output values.
            activation_function (function): Activation function to
use.
        """
        # Forward pass
        outputs = self.forward(inputs, activation_function)

        # Compute output layer error
        output_error = targets - outputs
        output_delta = output_error * sigmoid_derivative(outputs)

        # Compute hidden layer error
        hidden_error = np.dot(output_delta,
self.weights_hidden_output.T)
        hidden_delta = hidden_error *
sigmoid_derivative(self.hidden_layer_output)

        # Update weights and biases
        self.weights_hidden_output += self.learning_rate *
np.dot(self.hidden_layer_output.T, output_delta)
        self.bias_output += self.learning_rate * np.sum(output_delta,
axis=0)
        self.weights_input_hidden += self.learning_rate *
np.dot(inputs.T, hidden_delta)
        self.bias_hidden += self.learning_rate * np.sum(hidden_delta,
axis=0)

# Example usage
if __name__ == "__main__":
    # Define a sample dataset (XOR problem)
    X = np.array([
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1]
    ])
    y = np.array([
        [0],
        [1],
        [1],
        [0]
    ])

    # Initialize the MLP
```

```python
    mlp = MultiLayerPerceptron(input_dim=2, hidden_dim=2,
output_dim=1, learning_rate=0.1)

    # Train the MLP
    epochs = 10000
    for epoch in range(epochs):
        mlp.backward(X, y, activation_function=sigmoid)
        if epoch % 1000 == 0:
            outputs = mlp.forward(X)
            loss = np.mean((y - outputs) ** 2)
            print(f"Epoch {epoch}, Loss: {loss}")

    # Test the MLP
    print("\nXOR Gate")
    print("Input Output")
    for inputs in X:
        output = mlp.forward(inputs)
        print(f"{inputs}    {output.round()}")
```

```
Epoch 0, Loss: 0.3593535401568201
Epoch 1000, Loss: 0.24973964952491828
Epoch 2000, Loss: 0.2482922417975174
Epoch 3000, Loss: 0.2379574327996263
Epoch 4000, Loss: 0.19368015817972706
Epoch 5000, Loss: 0.12580160583922387
Epoch 6000, Loss: 0.03196429971644519
Epoch 7000, Loss: 0.012826663458205188
Epoch 8000, Loss: 0.007462180254251521
Epoch 9000, Loss: 0.005133500107124002

XOR Gate
Input Output
[0 0] [0.]
[0 1] [1.]
[1 0] [1.]
[1 1] [0.]
```