Aanvi Agarwal

22WU0104001

AI ML A

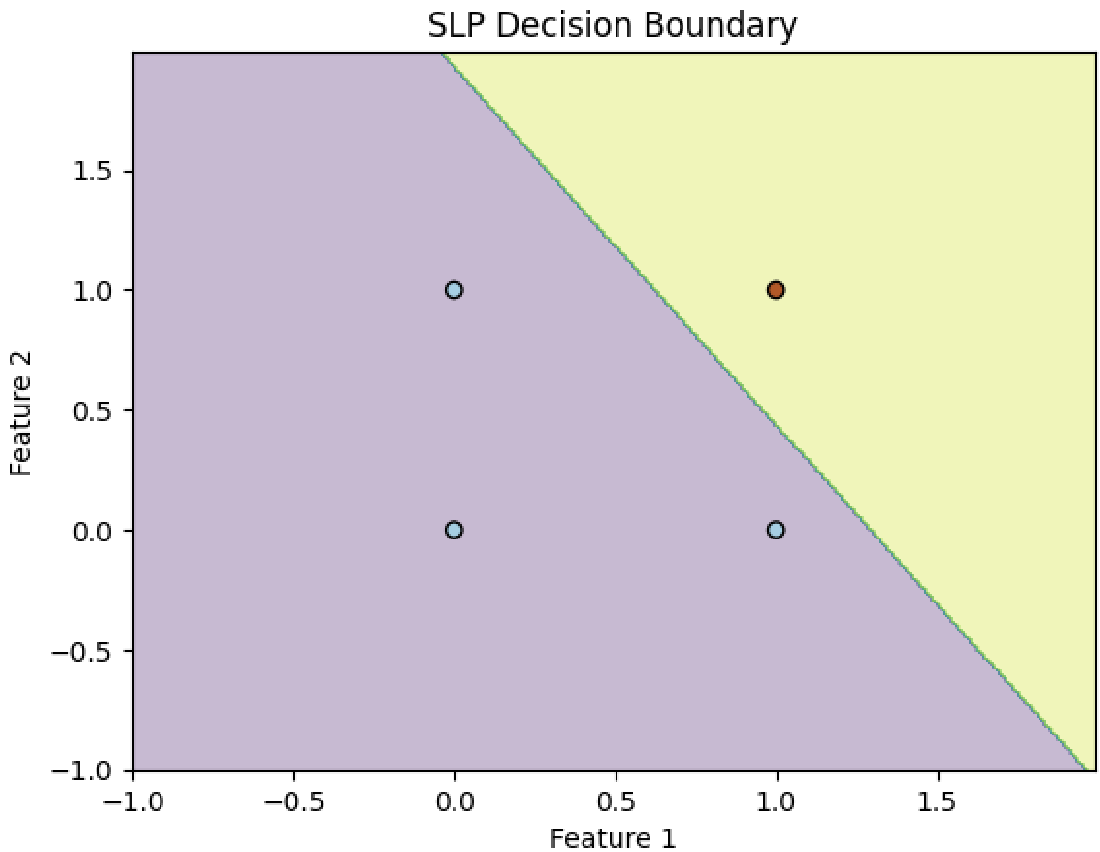

```python
#Lab 1 expiriment 1
import numpy as np

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

def predict(inputs, weights, bias):
    """Compute the output of a single artificial neuron."""
    # Weighted sum (z = w1*x1 + w2*x2 + ... + wn*xn + bias)
    z = np.dot(inputs, weights) + bias
    # Apply the activation function
    output = sigmoid(z)
    return output

def binary_classification(neuron_output):
    """Convert the neuron's output to a binary class label."""
    return 1 if neuron_output >= 0.5 else 0
# Example inputs
inputs = np.array([1.0, 2.0])  # Input values (x1, x2, ...)
weights = np.array([0.3, 0.7])  # Weights (w1, w2, ...)
bias = -0.5  # Bias term

# Compute the neuron's output
neuron_output = predict(inputs, weights, bias)

# Determine the binary classification
predicted_class = binary_classification(neuron_output)

# Display the results
print("Inputs:", inputs)
print("Weights:", weights)
print("Bias:", bias)
print("Neuron Output (after sigmoid):", neuron_output)
print("Predicted Class:", predicted_class)
```

```
Inputs: [1. 2.]
Weights: [0.3 0.7]
Bias: -0.5
Neuron Output (after sigmoid): 0.7685247834990175
Predicted Class: 1
```

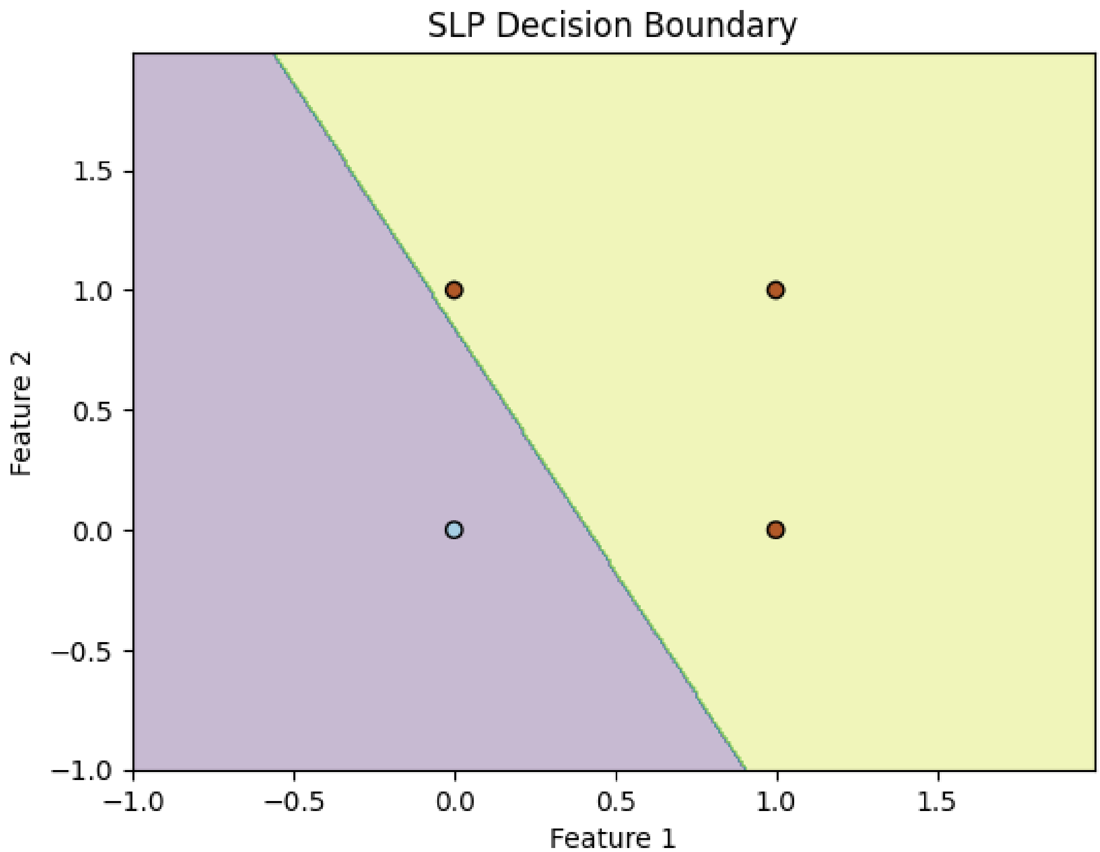
```
#Lab 1 expiriment 2
import numpy as np
import matplotlib.pyplot as plt

class SingleLayerPerceptron:
    def __init__(self, input_dim, learning_rate=0.1, epochs=100):

        self.weights = np.random.randn(input_dim)
        self.bias = np.random.randn()
        self.learning_rate = learning_rate
        self.epochs = epochs

    def activation(self, x):

        return 1 if x >= 0 else 0

    def fit(self, X, y):
        for epoch in range(self.epochs):
            for i in range(X.shape[0]):

                linear_output = np.dot(X[i], self.weights) + self.bias
                prediction = self.activation(linear_output)


                error = y[i] - prediction
                self.weights += self.learning_rate * error * X[i]
                self.bias += self.learning_rate * error

    def predict(self, X):
        predictions = []
        for i in range(X.shape[0]):
            linear_output = np.dot(X[i], self.weights) + self.bias
            prediction = self.activation(linear_output)
            predictions.append(prediction)
        return np.array(predictions)

    def plot_decision_boundary(self, X, y):

        x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
        y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
        xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                             np.arange(y_min, y_max, 0.01))

        # Predict on the grid points
        Z = self.predict(np.c_[xx.ravel(), yy.ravel()])
```

```
        Z = Z.reshape(xx.shape)

        # Plot the contour
        plt.contourf(xx, yy, Z, alpha=0.3)

        # Plot the data points
        plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', marker='o', 
cmap=plt.cm.Paired)
        plt.title('SLP Decision Boundary')
        plt.xlabel('Feature 1')
        plt.ylabel('Feature 2')
        plt.show()

X_and = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_and = np.array([0, 0, 0, 1])

X_or = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_or = np.array([0, 1, 1, 1])

slp_and = SingleLayerPerceptron(input_dim=2, learning_rate=0.1, 
epochs=100)

slp_and.fit(X_and, y_and)

predictions_and = slp_and.predict(X_and)
print("AND Gate Predictions:", predictions_and)

slp_and.plot_decision_boundary(X_and, y_and)

slp_or = SingleLayerPerceptron(input_dim=2, learning_rate=0.1, 
epochs=100)

slp_or.fit(X_or, y_or)


predictions_or = slp_or.predict(X_or)
print("OR Gate Predictions:", predictions_or)

slp_or.plot_decision_boundary(X_or, y_or)
```

```
AND Gate Predictions: [0 0 0 1]
```




```
OR Gate Predictions: [0 1 1 1]
```

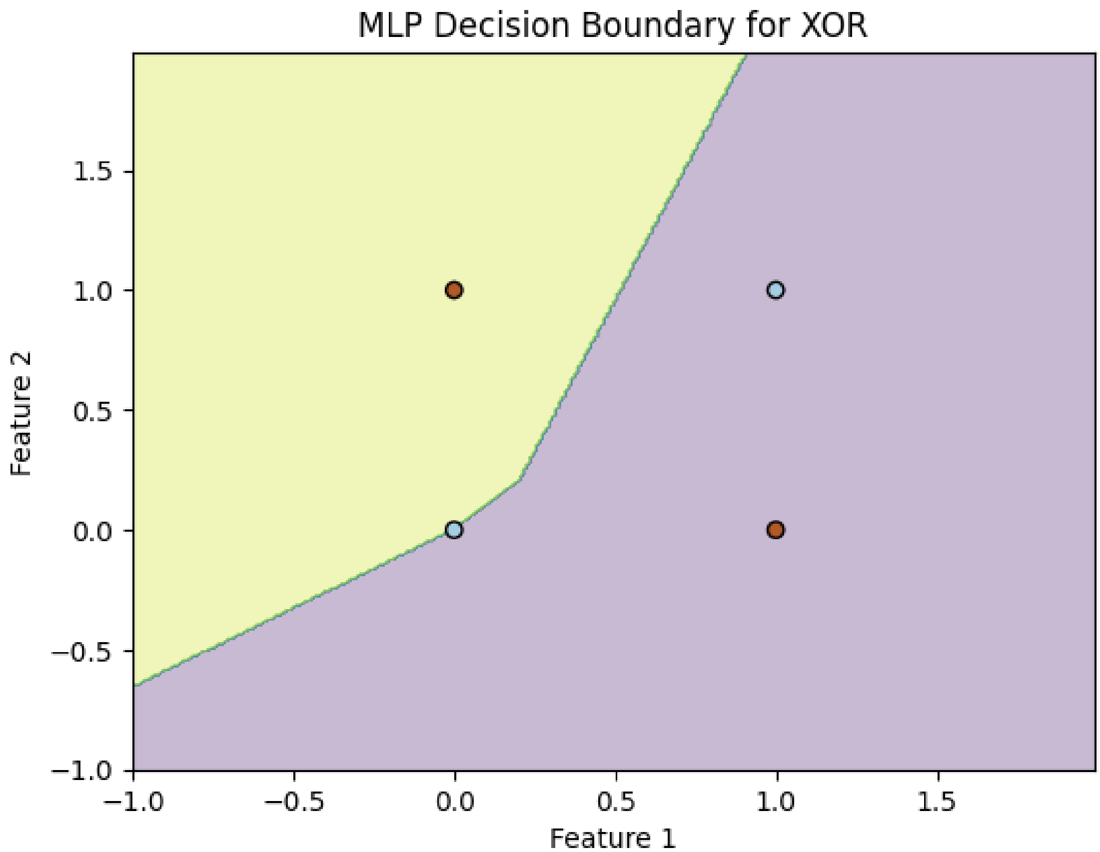



```
#Lab 1 expiriment 3
import numpy as np
import matplotlib.pyplot as plt


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


class MultiLayerPerceptron:
    def __init__(self, input_dim, hidden_dim, output_dim,
learning_rate=0.1, epochs=10000):

        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim
        self.learning_rate = learning_rate
        self.epochs = epochs

        self.W1 = np.random.randn(self.input_dim, self.hidden_dim)
        self.b1 = np.zeros((1, self.hidden_dim))

        self.W2 = np.random.randn(self.hidden_dim, self.output_dim)
```

```
        self.b2 = np.zeros((1, self.output_dim))

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

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

    def relu(self, x):
        return np.maximum(0, x)

    def relu_derivative(self, x):
        return (x > 0).astype(float)

    def forward(self, X):

        self.hidden_input = np.dot(X, self.W1) + self.b1
        self.hidden_output = self.relu(self.hidden_input)

        self.final_input = np.dot(self.hidden_output, self.W2) +
self.b2
        self.final_output = self.sigmoid(self.final_input)
        return self.final_output

    def backward(self, X, y, output):

        output_error = y - output
        output_delta = output_error * self.sigmoid_derivative(output)

        hidden_error = output_delta.dot(self.W2.T)
        hidden_delta = hidden_error *
self.relu_derivative(self.hidden_output)

        self.W2 += self.hidden_output.T.dot(output_delta) *
self.learning_rate
        self.b2 += np.sum(output_delta, axis=0, keepdims=True) *
self.learning_rate

        self.W1 += X.T.dot(hidden_delta) * self.learning_rate
        self.b1 += np.sum(hidden_delta, axis=0, keepdims=True) *
self.learning_rate

    def train(self, X, y):
        for epoch in range(self.epochs):
            output = self.forward(X)
            self.backward(X, y, output)

    def predict(self, X):
        return self.forward(X)

mlp = MultiLayerPerceptron(input_dim=2, hidden_dim=4, output_dim=1,
```

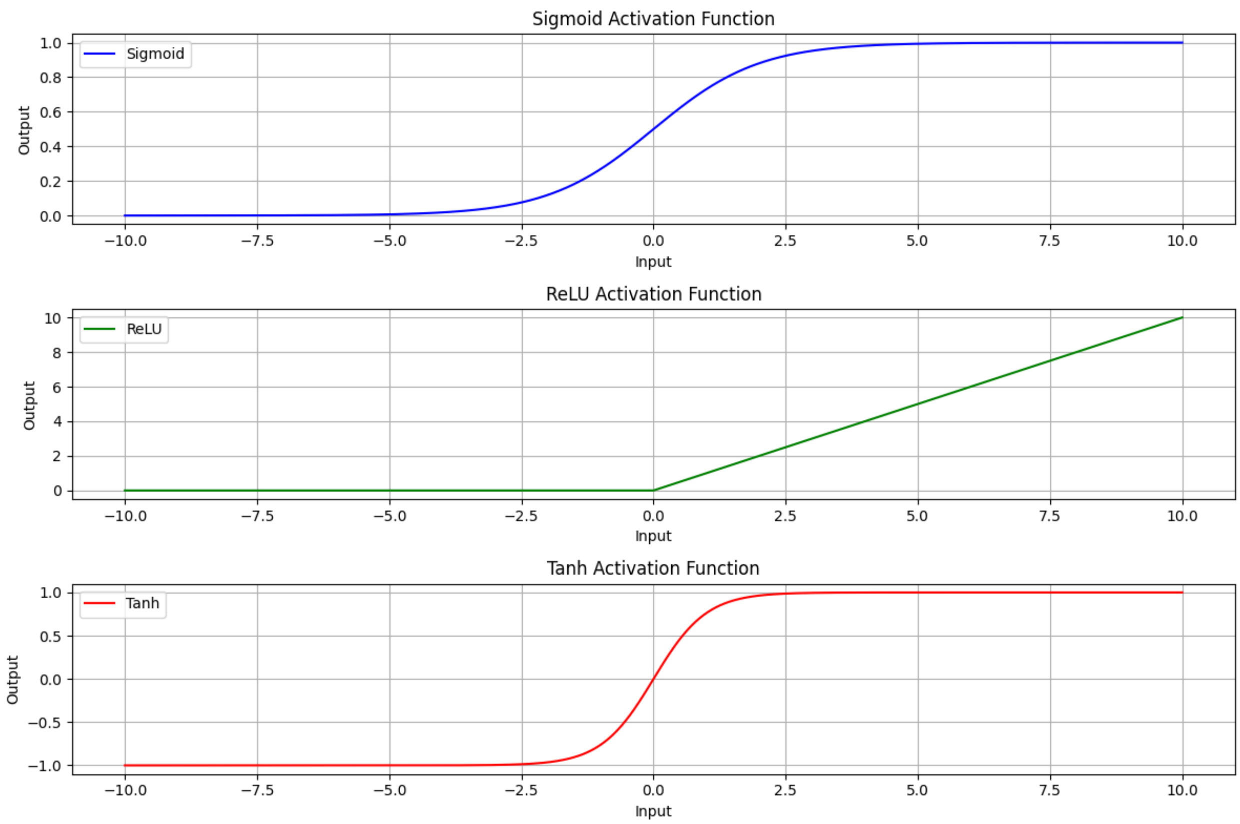
```
learning_rate=0.1, epochs=10000)
mlp.train(X_xor, y_xor)

predictions = mlp.predict(X_xor)
predictions = (predictions > 0.5).astype(int)

print("Predictions on XOR gate:")
print(predictions)

def plot_decision_boundary(X, y, model):
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                         np.arange(y_min, y_max, 0.01))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = (Z > 0.5).astype(int).reshape(xx.shape)

    plt.contourf(xx, yy, Z, alpha=0.3)
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', marker='o',
cmap=plt.cm.Paired)
    plt.title('MLP Decision Boundary for XOR')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.show()

plot_decision_boundary(X_xor, y_xor, mlp)
```

```
Predictions on XOR gate:
[[0]
 [1]
 [0]
 [0]]
```




```
#Lab 1 expiriment 4
import numpy as np
import matplotlib.pyplot as plt

# Activation functions
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def relu(x):
    return np.maximum(0, x)

def tanh(x):
    return np.tanh(x)

# Generate a sample dataset (random numbers)
x = np.linspace(-10, 10, 1000)  # Range of values from -10 to 10

# Apply activation functions
sigmoid_output = sigmoid(x)
relu_output = relu(x)
tanh_output = tanh(x)

# Plot the results
```

```python
plt.figure(figsize=(12, 8))

# Sigmoid
plt.subplot(3, 1, 1)
plt.plot(x, sigmoid_output, label='Sigmoid', color='blue')
plt.title('Sigmoid Activation Function')
plt.xlabel('Input')
plt.ylabel('Output')
plt.grid(True)
plt.legend()

# ReLU
plt.subplot(3, 1, 2)
plt.plot(x, relu_output, label='ReLU', color='green')
plt.title('ReLU Activation Function')
plt.xlabel('Input')
plt.ylabel('Output')
plt.grid(True)
plt.legend()

# Tanh
plt.subplot(3, 1, 3)
plt.plot(x, tanh_output, label='Tanh', color='red')
plt.title('Tanh Activation Function')
plt.xlabel('Input')
plt.ylabel('Output')
plt.grid(True)
plt.legend()

# Show all plots
plt.tight_layout()
plt.show()
```




```
#Lab 1 expiriment 5
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]])


input_dim = 2
hidden_dim = 4
output_dim = 1
learning_rate = 0.1
epochs = 10000

W1 = np.random.randn(input_dim, hidden_dim)
b1 = np.zeros((1, hidden_dim))
W2 = np.random.randn(hidden_dim, output_dim)
b2 = np.zeros((1, output_dim))
```

```python
def forward(X):

    hidden_input = np.dot(X, W1) + b1
    hidden_output = sigmoid(hidden_input)

    final_input = np.dot(hidden_output, W2) + b2
    final_output = sigmoid(final_input)

    return hidden_output, final_output

def backward(X, y, hidden_output, final_output):
    global W2, b2, W1, b1

    output_error = y - final_output
    output_delta = output_error * sigmoid_derivative(final_output)

    hidden_error = output_delta.dot(W2.T)
    hidden_delta = hidden_error * sigmoid_derivative(hidden_output)

    W2 += hidden_output.T.dot(output_delta) * learning_rate
    b2 += np.sum(output_delta, axis=0, keepdims=True) * learning_rate
    W1 += X.T.dot(hidden_delta) * learning_rate
    b1 += np.sum(hidden_delta, axis=0, keepdims=True) * learning_rate

for epoch in range(epochs):

    hidden_output, final_output = forward(X)

    backward(X, y, hidden_output, final_output)

    if epoch % 1000 == 0:
        loss = np.mean(np.square(y - final_output))
        print(f"Epoch {epoch}, Loss: {loss}")

hidden_output, predictions = forward(X)
predictions = (predictions > 0.5).astype(int)

print("\nFinal Predictions on XOR dataset:")
print(predictions)
```

```
Epoch 0, Loss: 0.26099650300567545
Epoch 1000, Loss: 0.2422638250202789
Epoch 2000, Loss: 0.1414255691499342
Epoch 3000, Loss: 0.026415976688866125
Epoch 4000, Loss: 0.010685701598118635
Epoch 5000, Loss: 0.006294831090380298
Epoch 6000, Loss: 0.004365685998889471
Epoch 7000, Loss: 0.0033074697772519828
Epoch 8000, Loss: 0.002646942853165808
Epoch 9000, Loss: 0.0021984661830945413
```

```
Final Predictions on XOR dataset:
[[0]
 [1]
 [1]
 [0]]
```