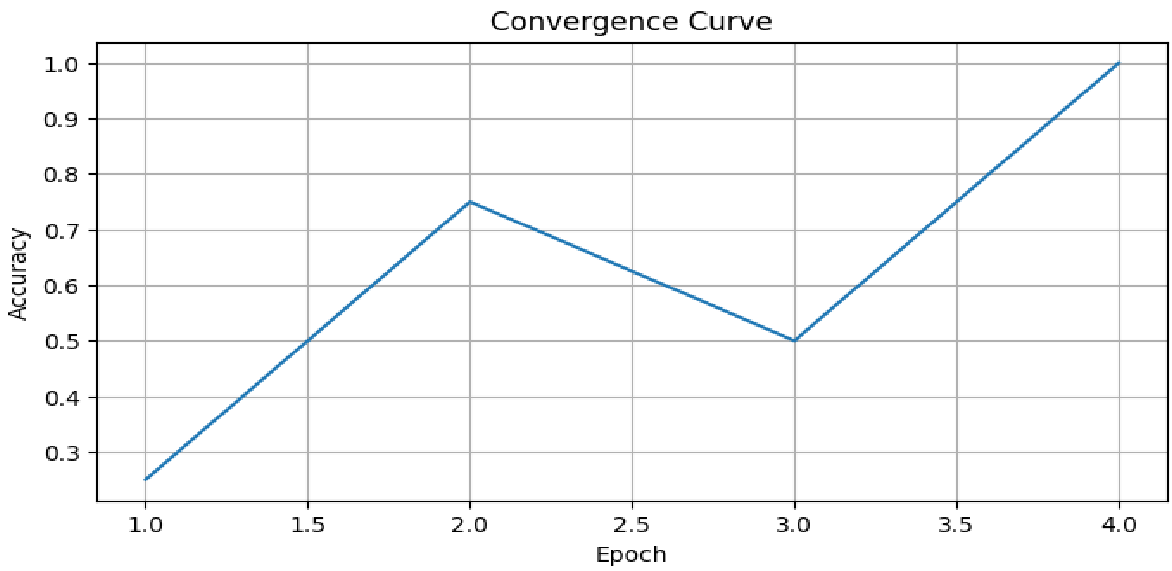
Source Code:

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

# Bipolar activation function
def bipolar_activation(x):
    return 1 if x >= 0 else -1

# Perceptron training function
def perceptron_train(inputs, targets, learning_rate=0.1, max_epochs=100):
    num_inputs = inputs.shape[1]
    num_samples = inputs.shape[0]

    # Initialize weights and bias
    weights = np.random.randn(num_inputs)
    bias = np.random.randn()

    convergence_curve = []

    for epoch in range(max_epochs):
        misclassified = 0

        for i in range(num_samples):
            net_input = np.dot(inputs[i], weights) + bias
            predicted = bipolar_activation(net_input)

            if predicted != targets[i]:
                misclassified += 1
                update = learning_rate * (targets[i] - predicted)
                weights += update * inputs[i]
                bias += update

        accuracy = (num_samples - misclassified) / num_samples
        convergence_curve.append(accuracy)

        if misclassified == 0:
            print("Converged in {} epochs.".format(epoch + 1))
            break

    return weights, bias, convergence_curve




# Main function
if __name__ == "__main__":
    # Input and target data (bipolar representation)
```

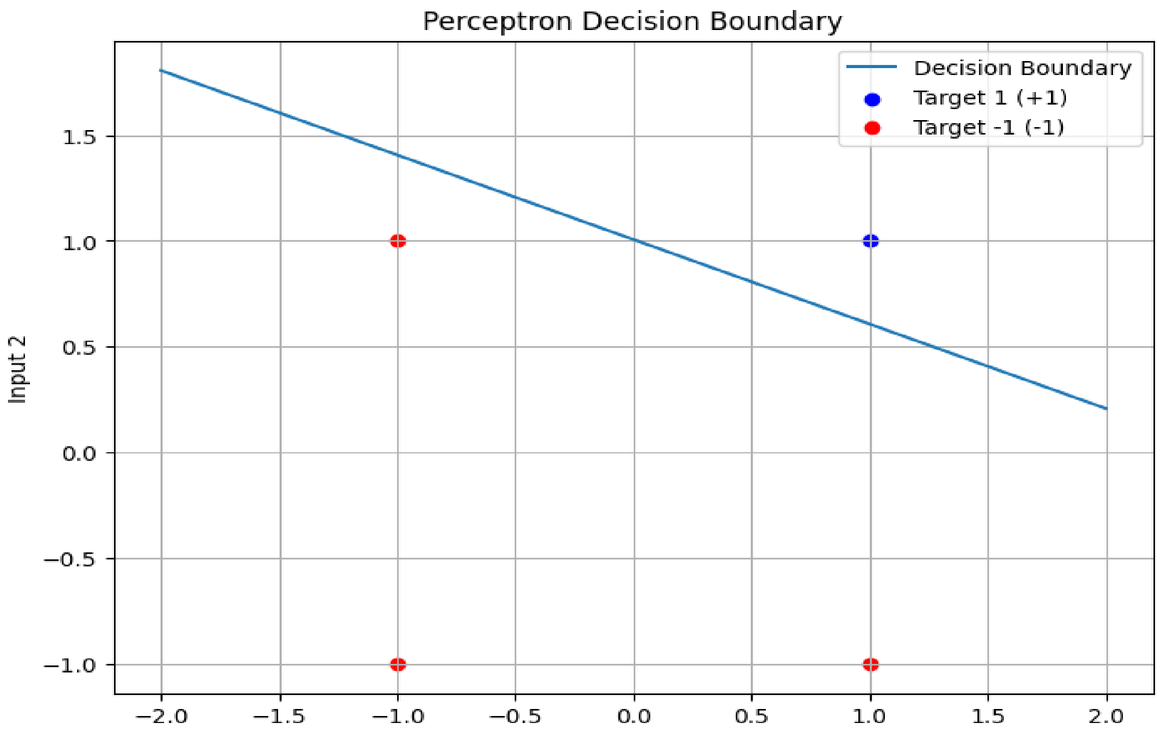
```
    inputs = np.array([[-1, -1], [-1, 1], [1, -1], [1, 1]])
    targets = np.array([-1, -1, -1, 1])

    # Training the perceptron
    weights, bias, convergence_curve = perceptron_train(inputs, targets)

    # Decision boundary line
    x = np.linspace(-2, 2, 100)
    y = (-weights[0] * x - bias) / weights[1]

    # Plot convergence curve
    plt.figure(figsize=(8, 4))
    plt.plot(range(1, len(convergence_curve) + 1), convergence_curve)
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.title('Convergence Curve')
    plt.grid()
    plt.show()

    # Plot the decision boundary line and data points
    plt.figure(figsize=(8, 6))
    plt.plot(x, y, label='Decision Boundary')
    plt.scatter(inputs[targets == 1][:, 0], inputs[targets == 1][:, 1], label='Target 1 (+1)',
color='blue')
    plt.scatter(inputs[targets == -1][:, 0], inputs[targets == -1][:, 1], label='Target -1 (-1)',
color='red')
    plt.xlabel('Input 1')
    plt.ylabel('Input 2')
    plt.title('Perceptron Decision Boundary')
    plt.legend()
    plt.grid()
    plt.show()
    print(inputs[targets == 1][:, 0])
    print(inputs[targets == 1][:, 1])
```

Output:

1. Convergence curve:



2. Perceptron decision boundary



Source Code:

```
import numpy as np

def softmax(x):
    ex = np.exp(x)
    return ex / np.sum(ex)

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

def multi_class(W1, W2, X, D):
    alpha = 0.9
    N = 4

    for k in range(N):
        x = X[:, :, k].reshape(25, 1)
        d = D[k, :].reshape(-1, 1)
        v1 = np.dot(W1, x)
        y1 = sigmoid(v1)
        v = np.dot(W2, y1)
        y = softmax(v)
        e = d - y
        delta = e
        e1 = np.dot(W2.T, delta)
        delta1 = y1 * (1 - y1) * e1
        dW1 = alpha * np.dot(delta1, x.T)
        W1 = W1 + dW1
        dW2 = alpha * np.dot(delta, y1.T)
        W2 = W2 + dW2

    return W1, W2

def main():
    np.random.seed(3)
    X = np.zeros((5, 5, 5))
    X[:, :, 0] = np.array([[0, 1, 1, 0, 0],
                           [0, 0, 1, 0, 0],
                           [0, 0, 1, 0, 0],
                           [0, 0, 1, 0, 0],
                           [0, 1, 1, 1, 0]])
    X[:, :, 1] = np.array([[1, 1, 1, 1, 0],
                           [0, 0, 0, 0, 1],
                           [0, 1, 1, 1, 0],
                           [1, 0, 0, 0, 0],
                           [1, 1, 1, 1, 1]])
    X[:, :, 2] = np.array([[1, 1, 1, 1, 0],
```

```
                    [0, 0, 0, 0, 1],
                    [0, 1, 1, 1, 0],
                    [0, 0, 0, 0, 1],
                    [1, 1, 1, 1, 0]])
    X[:, :, 3] = np.array([[0, 0, 0, 1, 0],
                    [0, 0, 1, 1, 0],
                    [0, 1, 0, 1, 0],
                    [1, 1, 1, 1, 1],
                    [0, 0, 0, 1, 0]])

    D = np.eye(5)

    W1 = 2 * np.random.rand(50, 25) - 1
    W2 = 2 * np.random.rand(5, 50) - 1

    for epoch in range(10000):
        W1, W2 = multi_class(W1, W2, X, D)

    N = 4
    for k in range(N):
        x = X[:, :, k].reshape(25, 1)
        v1 = np.dot(W1, x)
        y1 = sigmoid(v1)
        v = np.dot(W2, y1)
        y = softmax(v)
        print(f"\n\n Output for X[:,:,{k}]:\n\n")
        print(f"{y} \n\n This matrix from see that {k+1} position accuracy is higher that is
: {max(y)} So this number is correctly identified")

if __name__ == "__main__":
    main()
```

Output:

Output for X[:,:,0]:
[[9.99990560e-01]
[3.73975045e-06]
[7.29323123e-07]
[4.95516529e-06]
[1.56459758e-08]]
This matrix from see that 1 position accuracy is higher that is : [0.99999056] So this number is correctly identified

Output for X[:,:,1]:
[[3.81399150e-06]
[9.99984069e-01]
[1.07138749e-05]
[7.38201374e-07]
[6.65377695e-07]]
This matrix from see that 2 position accuracy is higher that is : [0.99998407] So this number is correctly identified

Output for X[:,:,2]:
[[2.10669179e-06]
[9.17015598e-06]
[9.99972467e-01]
[2.22084036e-06]
[1.40352894e-05]]
This matrix from see that 3 position accuracy is higher that is : [0.99997247] So this number is correctly identified

Output for X[:,:,3]:
[[4.72578106e-06]
[8.98916172e-07]
[9.07090140e-07]
[9.99990801e-01]
[2.66714208e-06]]
This matrix from see that 4 position accuracy is higher that is : [0.9999908] So this number is correctly identified

Source Code:

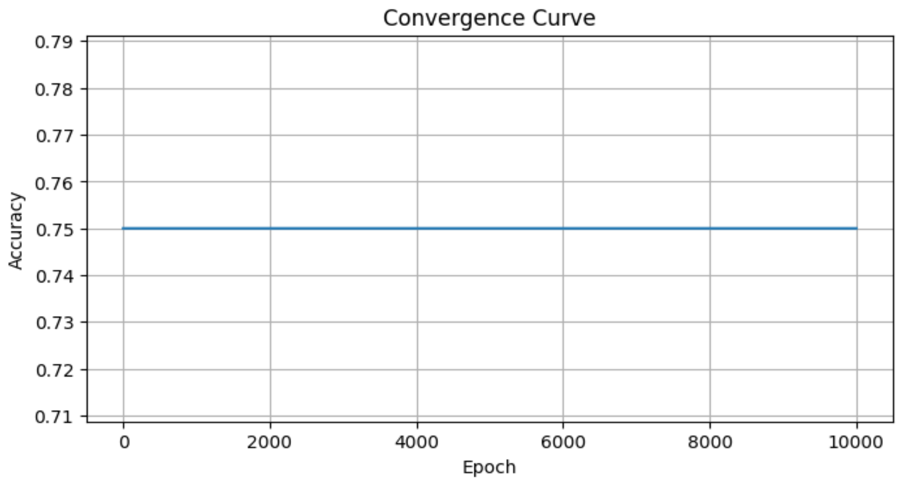
```
#XOR implementation using McCulloch pit neuron
import numpy as np
import matplotlib.pyplot as plt

# Sigmoid activation function and its derivative (for training)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

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

targets = np.array([0, 1, 1, 0])
# Neural network parameters
input_layer_size = 2
hidden_layer_size = 2
output_layer_size = 1
learning_rate = 0.1
max_epochs = 10000

# Initialize weights and biases with random values
np.random.seed(42)
weights_input_hidden = np.random.randn(input_layer_size, hidden_layer_size)
bias_hidden = np.random.randn(hidden_layer_size)

weights_hidden_output = np.random.randn(hidden_layer_size, output_layer_size)
bias_output = np.random.randn(output_layer_size)

convergence_curve = []

# Training the neural network
for epoch in range(max_epochs):
    misclassified = 0
    for i in range(len(inputs)):
        # Forward pass
        hidden_layer_input = np.dot(inputs[i], weights_input_hidden) + bias_hidden
        hidden_layer_output = sigmoid(hidden_layer_input)

        output_layer_input = np.dot(hidden_layer_output, weights_hidden_output) + bias_output
        predicted_output = sigmoid(output_layer_input)
```

```
        # Backpropagation
        error = targets[i] - predicted_output
        #print(error)
        if targets[i] != predicted_output:
            misclassified += 1

        output_delta = error * sigmoid_derivative(predicted_output)
        hidden_delta = output_delta.dot(weights_hidden_output.T) * sigmoid_derivative(hidden_layer_output)

        # Update weights and biases
        weights_hidden_output += hidden_layer_output[:, np.newaxis] * output_delta * learning_rate
        bias_output += output_delta * learning_rate

        weights_input_hidden += inputs[i][:, np.newaxis] * hidden_delta * learning_rate
        bias_hidden += hidden_delta * learning_rate

    accuracy = (len(inputs) - misclassified) / len(inputs)
    #print((accuracy))
    convergence_curve.append(accuracy)

    if misclassified == 0:
        print("Converged in {} epochs.".format(epoch + 1))
        break

# Decision boundary line
x = np.linspace(-0.5, 1.5, 100)
y = (-weights_input_hidden[0, 0] * x - bias_hidden[0]) / weights_input_hidden[1, 0]
y2 = (-weights_input_hidden[0, 1] * x - bias_hidden[1]) / weights_input_hidden[1, 1]

# Plot convergence curve
plt.figure(figsize=(8, 4))
plt.plot(range(1, len(convergence_curve) + 1), convergence_curve)
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Convergence Curve')
plt.grid()
plt.show()

# Plot the decision boundary line and data points
plt.figure(figsize=(8, 6))
plt.plot(x, y, label='Decision Boundary 1')
plt.plot(x, y2, label='Decision Boundary 2')
plt.scatter(inputs[targets == 1][:, 0], inputs[targets == 1][:, 1], label='Target 1 (1)', color='blue')
plt.scatter(inputs[targets == 0][:, 0], inputs[targets == 0][:, 1], label='Target 0 (0)', color='red')
plt.xlabel('Input 1')
plt.ylabel('Input 2')
plt.title('XOR Function Decision Boundary')
plt.legend()
```

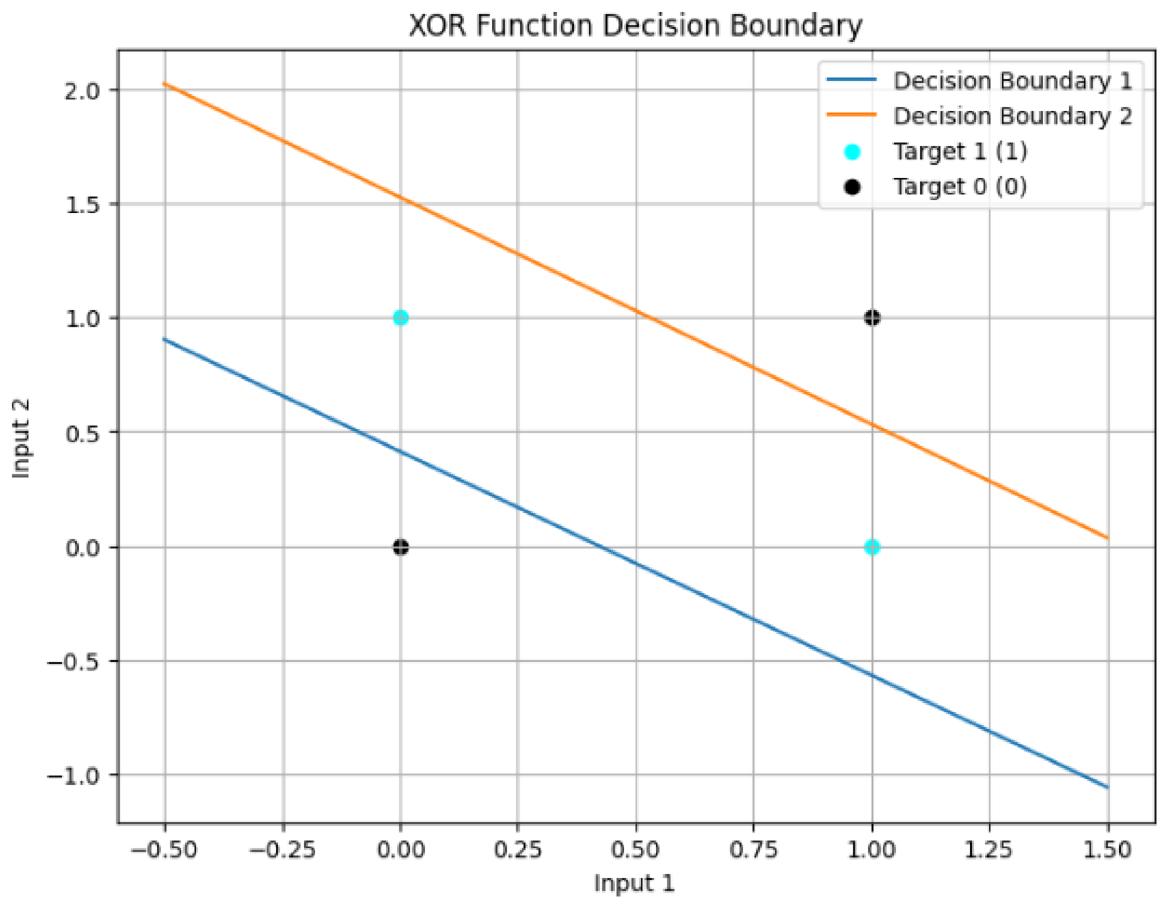
```
plt.grid()
plt.show()
```

Output:

1. Convergence Curve



2. Perceptron decision boundary



Source Code:

```
import numpy as np

# Sigmoid activation function and its derivative (for training)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

# XOR function dataset with binary inputs and outputs
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

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

# Neural network parameters
input_layer_size = 2
hidden_layer_size = 2
output_layer_size = 1
learning_rate = 0.1
max_epochs = 10000

# Initialize weights and biases with random values
np.random.seed(42)
weights_input_hidden = np.random.randn(input_layer_size, hidden_layer_size)
bias_hidden = np.random.randn(hidden_layer_size)

weights_hidden_output = np.random.randn(hidden_layer_size, output_layer_size)
bias_output = np.random.randn(output_layer_size)

# Training the neural network with backpropagation
for epoch in range(max_epochs):
    # Forward pass
    hidden_layer_input = np.dot(inputs, weights_input_hidden) + bias_hidden
    hidden_layer_output = sigmoid(hidden_layer_input)

    output_layer_input = np.dot(hidden_layer_output, weights_hidden_output) +
bias_output
    predicted_output = sigmoid(output_layer_input)

    # Calculate the error
    error = targets - predicted_output

    # Backpropagation
    output_delta = error * sigmoid_derivative(predicted_output)
    hidden_delta = output_delta.dot(weights_hidden_output.T) *
sigmoid_derivative(hidden_layer_output)
```

```
    # Update weights and biases
    weights_hidden_output += hidden_layer_output.T.dot(output_delta) * learning_rate
    bias_output += np.sum(output_delta, axis=0) * learning_rate  # Removed
keepdims=True here

    weights_input_hidden += inputs.T.dot(hidden_delta) * learning_rate
    bias_hidden += np.sum(hidden_delta, axis=0) * learning_rate

# Test the XOR function with the trained neural network
test_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

hidden_layer_input = np.dot(test_inputs, weights_input_hidden) + bias_hidden
hidden_layer_output = sigmoid(hidden_layer_input)

output_layer_input = np.dot(hidden_layer_output, weights_hidden_output) + bias_output
predicted_output = sigmoid(output_layer_input)

print("Predicted outputs:")
print(predicted_output)

# Round the predicted outputs to get binary values (0 or 1)
predicted_binary = np.round(predicted_output).astype(int)
print("Predicted binary outputs:")
print(predicted_binary)
```

Output:

```
Predicted outputs:
[[0.05395132]
 [0.9505447 ]
 [0.95009809]
 [0.05355567]]
Predicted binary outputs:
[[0]
 [1]
 [1]
 [0]]
```

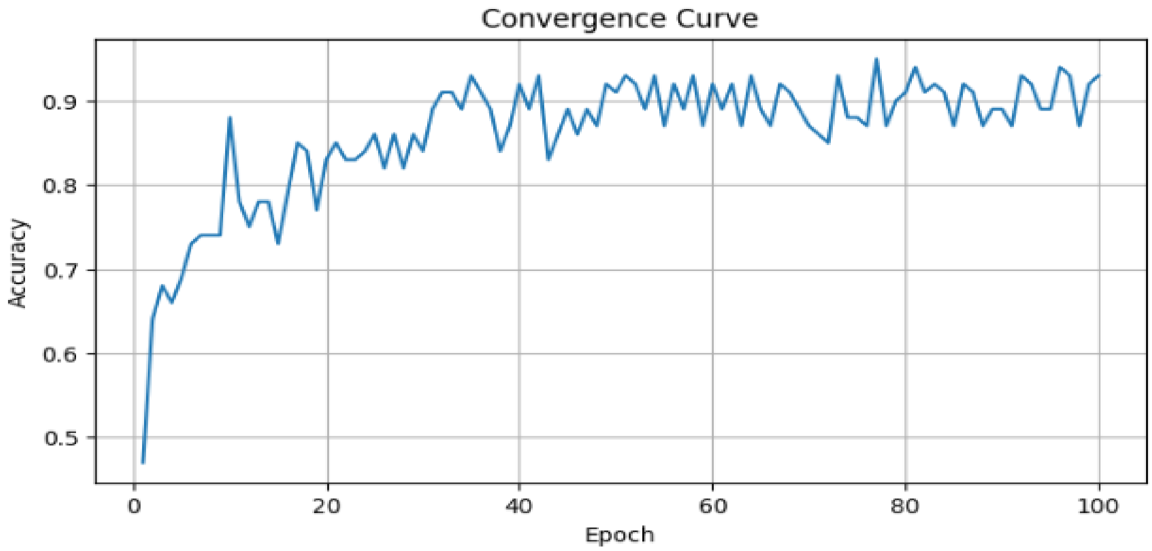
Source Code:

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

# Perceptron training function
def perceptron_train(inputs, targets, learning_rate=0.1, max_epochs=100):
    num_inputs = inputs.shape[1]
    num_samples = inputs.shape[0]

    # Initialize weights and bias
    weights = np.random.randn(num_inputs)
    bias = np.random.randn()

    convergence_curve = []

    for epoch in range(max_epochs):
        misclassified = 0

        for i in range(num_samples):
            net_input = np.dot(inputs[i], weights) + bias
            predicted = 1 if net_input >= 0 else 0

            if predicted != targets[i]:
                misclassified += 1
                update = learning_rate * (targets[i] - predicted)
                weights += update * inputs[i]
                bias += update

        accuracy = (num_samples - misclassified) / num_samples
        convergence_curve.append(accuracy)

        if misclassified == 0:
            print("Converged in {} epochs.".format(epoch + 1))
            break

    return weights, bias, convergence_curve

# Generate random linearly separable data points
def generate_data(n_samples):
    np.random.seed(42)
    inputs = np.random.rand(n_samples, 2) * 10
    targets = np.sum(inputs, axis=1) >= 10
    targets = targets.astype(int)
    return inputs, targets

# Main function
if __name__ == "__main__":
```

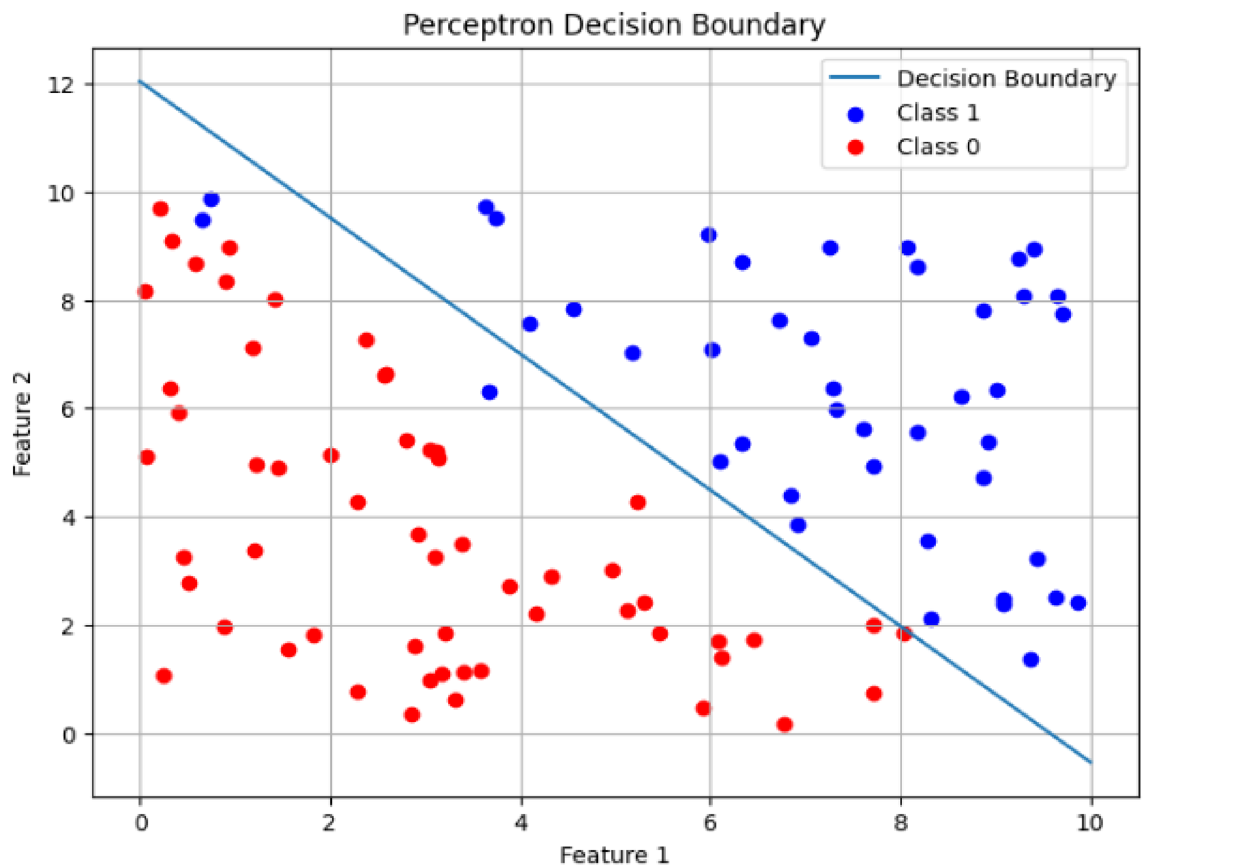
```
    # Generate linearly separable data
    n_samples = 100
    inputs, targets = generate_data(n_samples)

    # Training the perceptron
    weights, bias, convergence_curve = perceptron_train(inputs, targets)

    # Decision boundary line
    x = np.linspace(0, 10, 100)
    y = (-weights[0] * x - bias) / weights[1]

    # Plot convergence curve
    plt.figure(figsize=(8, 4))
    plt.plot(range(1, len(convergence_curve) + 1), convergence_curve)
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.title('Convergence Curve')
    plt.grid()
    plt.show()

    # Plot the decision boundary line and data points
    plt.figure(figsize=(8, 6))
    plt.plot(x, y, label='Decision Boundary')
    plt.scatter(inputs[targets == 1][:, 0], inputs[targets == 1][:, 1], label='Class 1',
color='blue')
    plt.scatter(inputs[targets == 0][:, 0], inputs[targets == 0][:, 1], label='Class 0',
color='red')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title('Perceptron Decision Boundary')
    plt.legend()
    plt.grid()
    plt.show()
```

Output:

1. Convergence curve



2. Perceptron decision boundary



Source Code:

```
import numpy as np

# Sigmoid activation function and its derivative (for training)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

# XOR function dataset with binary inputs and outputs
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

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

# Neural network parameters
input_layer_size = 2
hidden_layer_size = 2
output_layer_size = 1
learning_rate = 0.1
momentum_factor = 0.9
max_epochs = 10000

# Initialize weights and biases with random values
np.random.seed(42)
weights_input_hidden = np.random.randn(input_layer_size, hidden_layer_size)
bias_hidden = np.random.randn(hidden_layer_size)

weights_hidden_output = np.random.randn(hidden_layer_size, output_layer_size)
bias_output = np.random.randn(output_layer_size)

# Initialize previous weight updates with zeros for momentum
prev_weight_input_hidden_update = np.zeros((input_layer_size, hidden_layer_size))
prev_bias_hidden_update = np.zeros(hidden_layer_size)

prev_weight_hidden_output_update = np.zeros((hidden_layer_size, output_layer_size))
prev_bias_output_update = np.zeros(output_layer_size)

# Training the neural network with backpropagation and momentum
for epoch in range(max_epochs):
    # Forward pass
    hidden_layer_input = np.dot(inputs, weights_input_hidden) + bias_hidden
    hidden_layer_output = sigmoid(hidden_layer_input)

    output_layer_input = np.dot(hidden_layer_output, weights_hidden_output) +
bias_output
    predicted_output = sigmoid(output_layer_input)
```

```
    # Calculate the error
    error = targets - predicted_output

    # Backpropagation
    output_delta = error * sigmoid_derivative(predicted_output)
    hidden_delta = output_delta.dot(weights_hidden_output.T) *
sigmoid_derivative(hidden_layer_output)

    # Update weights and biases with momentum
    weight_input_hidden_update = inputs.T.dot(hidden_delta) * learning_rate
    bias_hidden_update = np.sum(hidden_delta, axis=0) * learning_rate

    weight_hidden_output_update = hidden_layer_output.T.dot(output_delta) *
learning_rate
    bias_output_update = np.sum(output_delta, axis=0) * learning_rate

    weights_input_hidden += weight_input_hidden_update + momentum_factor *
prev_weight_input_hidden_update
    bias_hidden += bias_hidden_update + momentum_factor * prev_bias_hidden_update

    weights_hidden_output += weight_hidden_output_update + momentum_factor *
prev_weight_hidden_output_update
    bias_output += bias_output_update + momentum_factor * prev_bias_output_update

    # Store previous updates for momentum
    prev_weight_input_hidden_update = weight_input_hidden_update
    prev_bias_hidden_update = bias_hidden_update

    prev_weight_hidden_output_update = weight_hidden_output_update
    prev_bias_output_update = bias_output_update

    # Calculate mean squared error for convergence check
    mse = np.mean(error ** 2)
    if mse < 1e-6:
        print("Converged in {} epochs.".format(epoch + 1))
        break

# Test the XOR function with the trained neural network
test_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

hidden_layer_input = np.dot(test_inputs, weights_input_hidden) + bias_hidden
hidden_layer_output = sigmoid(hidden_layer_input)

output_layer_input = np.dot(hidden_layer_output, weights_hidden_output) + bias_output
predicted_output = sigmoid(output_layer_input)

print("Predicted outputs:")
```

```
print(predicted_output)

# Round the predicted outputs to get binary values (0 or 1)
predicted_binary = np.round(predicted_output).astype(int)
print("Predicted binary outputs:")
print(predicted_binary)
```

Output:

```
Predicted outputs:
[[0.03383077]
 [0.97009142]
 [0.96988489]
 [0.03162844]]
Predicted binary outputs:
[[0]
 [1]
 [1]
 [0]]
```

Source Code:

```
import numpy as np

# Sigmoid activation function and its derivative (for training)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

# Input and target datasets
X_input = np.array([[0, 0, 1], [0, 1, 1], [1, 0, 1], [1, 1, 1]])

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

# Neural network parameters
input_layer_size = 3
output_layer_size = 1
learning_rate = 0.1
max_epochs = 10000

# Initialize weights with random values
np.random.seed(42)
weights = np.random.randn(input_layer_size, output_layer_size)

# Training the neural network with SGD
for epoch in range(max_epochs):
    error_sum = 0

    for i in range(len(X_input)):
        # Forward pass
        input_data = X_input[i]
        target_data = D_target[i]

        net_input = np.dot(input_data, weights)
        predicted_output = sigmoid(net_input)

        # Calculate error
        error = target_data - predicted_output
        error_sum += np.abs(error)

        # Update weights using the delta learning rule
        weight_update = learning_rate * error * sigmoid_derivative(predicted_output) *
input_data
        weights += weight_update[:, np.newaxis]  # Update weights for each input separately

    # Check for convergence
```

```
    if error_sum < 0.01:
        print("Converged in {} epochs.".format(epoch + 1))
        break

# Test data
test_data = X_input

# Use the trained model to recognize target function
print("Target Function Test:")
for i in range(len(test_data)):
    input_data = test_data[i]
    net_input = np.dot(input_data, weights)
    predicted_output = sigmoid(net_input)

    print(f"Input: {input_data} -> Output: {np.round(predicted_output)}")
```

Output:

```
Target Function Test:
Input: [0 0 1] -> Output: [0.]
Input: [0 1 1] -> Output: [0.]
Input: [1 0 1] -> Output: [1.]
Input: [1 1 1] -> Output: [1.]
```

Source Code:

```
import numpy as np

# Sigmoid activation function and its derivative (for training)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

# Input and target datasets
X_input = np.array([[0, 0, 1], [0, 1, 1], [1, 0, 1], [1, 1, 1]])

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

# Neural network parameters
input_layer_size = 3
output_layer_size = 1
learning_rate = 0.1
max_epochs = 10000

# Initialize weights with random values
np.random.seed(42)
weights = np.random.randn(input_layer_size, output_layer_size)

# Training the neural network with batch method
for epoch in range(max_epochs):
    # Forward pass
    net_input = np.dot(X_input, weights)
    predicted_output = sigmoid(net_input)

    # Calculate error
    error = D_target - predicted_output
    error_sum = np.sum(np.abs(error))

    # Update weights using the delta learning rule
    weight_update = learning_rate * np.dot(X_input.T, error *
sigmoid_derivative(predicted_output))
    weights += weight_update

    # Check for convergence
    if error_sum < 0.01:
        print("Converged in {} epochs.".format(epoch + 1))
        break

# Test data
test_data = X_input
```

```
# Use the trained model to recognize target function
print("Target Function Test:")
for i in range(len(test_data)):
    input_data = test_data[i]
    net_input = np.dot(input_data, weights)
    predicted_output = sigmoid(net_input)

    print(f"Input: {input_data} -> Output: {np.round(predicted_output)}")
```

Output:

```
Target Function Test:
Input: [0 0 1] -> Output: [0.]
Input: [0 1 1] -> Output: [0.]
Input: [1 0 1] -> Output: [1.]
Input: [1 1 1] -> Output: [1.]
```

Source Code:

```
import numpy as np
import time

# Sigmoid activation function and its derivative (for training)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

# XOR function dataset with binary inputs and outputs
X_input = np.array([[0, 0, 1],[0, 1, 1],[1, 0, 1],[1, 1, 1]])

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

# Neural network parameters
input_layer_size = 3
output_layer_size = 1
learning_rate = 0.1
max_epochs = 10000

# Initialize weights with random values
np.random.seed(42)
weights_sgd = np.random.randn(input_layer_size, output_layer_size)
weights_batch = np.random.randn(input_layer_size, output_layer_size)

# Training the neural network with SGD
start_time_sgd = time.time()
for epoch in range(max_epochs):
    error_sum = 0

    for i in range(len(X_input)):
        # Forward pass
        input_data = X_input[i]
        target_data = D_target[i]

        net_input = np.dot(input_data, weights_sgd)
        predicted_output = sigmoid(net_input)

        # Calculate error
        error = target_data - predicted_output
        error_sum += np.abs(error)

        # Update weights using the delta learning rule
        weight_update = learning_rate * error * sigmoid_derivative(predicted_output) *
input_data
```

```
        weights_sgd += weight_update[:, np.newaxis]  # Update weights for each input
separately

    # Check for convergence
    if error_sum < 0.01:
        break
end_time_sgd = time.time()

# Training the neural network with the batch method
start_time_batch = time.time()
for epoch in range(max_epochs):
    # Forward pass
    net_input = np.dot(X_input, weights_batch)
    predicted_output = sigmoid(net_input)

    # Calculate error
    error = D_target - predicted_output
    error_sum = np.sum(np.abs(error))

    # Update weights using the delta learning rule
    weight_update = learning_rate * np.dot(X_input.T, error *
sigmoid_derivative(predicted_output))
    weights_batch += weight_update

    # Check for convergence
    if error_sum < 0.01:
        break
end_time_batch = time.time()

# Test data
test_data = X_input

# Use the trained models to recognize target function
def test_model(weights):
    predicted_output = sigmoid(np.dot(test_data, weights))
    return np.round(predicted_output)

print("SGD Results:")
print("Time taken: {:.6f} seconds".format(end_time_sgd - start_time_sgd))
print("Trained weights:")
print(weights_sgd)
print("Predicted binary outputs:")
print(test_model(weights_sgd))

print("\nBatch Method Results:")
print("Time taken: {:.6f} seconds".format(end_time_batch - start_time_batch))
print("Trained weights:")
print(weights_batch)
```

```
print("Predicted binary outputs:")
print(test_model(weights_batch))
```

Output:

```
SGD Results:
Time taken: 0.892055 seconds
Trained weights:
[[ 7.25950187]
 [-0.22431325]
 [-3.41036643]]
Predicted binary outputs:
[[0.]
 [0.]
 [1.]
 [1.]]

Batch Method Results:
Time taken: 0.263896 seconds
Trained weights:
[[ 7.26775966]
 [-0.22304058]
 [-3.41538639]]
Predicted binary outputs:
[[0.]
 [0.]
 [1.]
 [1.]]
```

Source Code:

```
import numpy as np

def softmax(x):
    ex = np.exp(x)
    return ex / np.sum(ex)

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

def multi_class(W1, W2, X, D):
    alpha = 0.9
    N = 5

    for k in range(N):
        x = X[:, :, k].reshape(25, 1)
        d = D[k, :].reshape(-1, 1)
        v1 = np.dot(W1, x)
        y1 = sigmoid(v1)
        v = np.dot(W2, y1)
        y = softmax(v)
        e = d - y
        delta = e
        e1 = np.dot(W2.T, delta)
        delta1 = y1 * (1 - y1) * e1
        dW1 = alpha * np.dot(delta1, x.T)
        W1 = W1 + dW1
        dW2 = alpha * np.dot(delta, y1.T)
        W2 = W2 + dW2

    return W1, W2

def main():
    np.random.seed(3)
    X = np.zeros((5, 5, 5))
    X[:, :, 0] = np.array([[0, 1, 1, 0, 0],
                           [0, 0, 1, 0, 0],
                           [0, 0, 1, 0, 0],
                           [0, 0, 1, 0, 0],
                           [0, 1, 1, 1, 0]])
    X[:, :, 1] = np.array([[1, 1, 1, 1, 0],
                           [0, 0, 0, 0, 1],
                           [0, 1, 1, 1, 0],
                           [1, 0, 0, 0, 0],
                           [1, 1, 1, 1, 1]])
    X[:, :, 2] = np.array([[1, 1, 1, 1, 0],
                           [0, 0, 0, 0, 1],
```

```
                 [0, 1, 1, 1, 0],
                 [0, 0, 0, 0, 1],
                 [1, 1, 1, 1, 0]])
    X[:, :, 3] = np.array([[0, 0, 0, 1, 0],
                 [0, 0, 1, 1, 0],
                 [0, 1, 0, 1, 0],
                 [1, 1, 1, 1, 1],
                 [0, 0, 0, 1, 0]])
    X[:, :, 4] = np.array([[1, 1, 1, 1, 1],
                 [1, 0, 0, 0, 0],
                 [1, 1, 1, 1, 0],
                 [0, 0, 0, 0, 1],
                 [1, 1, 1, 1, 0]])

    D = np.eye(5)

    W1 = 2 * np.random.rand(50, 25) - 1
    W2 = 2 * np.random.rand(5, 50) - 1

    for epoch in range(10000):
        W1, W2 = multi_class(W1, W2, X, D)

    N = 5
    for k in range(N):
        x = X[:, :, k].reshape(25, 1)
        v1 = np.dot(W1, x)
        y1 = sigmoid(v1)
        v = np.dot(W2, y1)
        y = softmax(v)
        print(f"\n\n Output for X[:,:,{k}]:\n\n")
        print(f"{y} \n\n This matrix from see that {k+1} position accuracy is higher that is :
{max(y)} So this number is correctly identified")

if __name__ == "__main__":
    main()
```

Output:

Output for X[:,:,0]:
[[9.99990560e-01]
 [3.73975045e-06]
 [7.29323123e-07]
 [4.95516529e-06]

[1.56459758e-08]]
This matrix from see that 1 position accuracy is higher that is : [0.99999056] So this number is correctly identified

Output for X[:,:,1]:
[[3.81399150e-06]
[9.99984069e-01]
[1.07138749e-05]
[7.38201374e-07]
[6.65377695e-07]]
This matrix from see that 2 position accuracy is higher that is : [0.99998407] So this number is correctly identified

Output for X[:,:,2]:
[[2.10669179e-06]
[9.17015598e-06]
[9.99972467e-01]
[2.22084036e-06]
[1.40352894e-05]]
This matrix from see that 3 position accuracy is higher that is : [0.99997247] So this number is correctly identified

Output for X[:,:,3]:
[[4.72578106e-06]
[8.98916172e-07]
[9.07090140e-07]
[9.99990801e-01]
[2.66714208e-06]]
This matrix from see that 4 position accuracy is higher that is : [0.9999908] So this number is correctly identified

Output for X[:,:,4]:
[[6.12205780e-07]
[2.29663674e-06]
[1.16748707e-05]
[1.01696314e-06]
[9.99984399e-01]]
This matrix from see that 5 position accuracy is higher that is : [0.9999844] So this number is correctly identified

Source Code:

```
#import
import os
import cv2
import numpy as np
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

#dataset input || training
def load_data(folder):
    images = []
    labels = []
    for filename in os.listdir(folder):
        label = folder.split('/')[-1]
        img = cv2.imread(os.path.join(folder, filename))
        img = cv2.resize(img, (150, 150))  # Resize the image to a consistent size
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)  # Convert to RGB format
        images.append(img)
        labels.append(label)
    return images, labels



banana_folder = 'dataset/banana'
cucumber_folder = 'dataset/cucumber'

banana_images, banana_labels = load_data(banana_folder)
cucumber_images, cucumber_labels = load_data(cucumber_folder)

# Combine the data
images = np.array(banana_images + cucumber_images)
labels = np.array(banana_labels + cucumber_labels)
print(labels)

# Encode labels to numerical values
label_dict = {'banana': 0, 'cucumber': 1}
encoded_labels = np.array([label_dict[label] for label in labels])
print(encoded_labels)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(images, encoded_labels,
test_size=0.15,random_state=42)

# Normalize the pixel values between 0 and 1
X_train = X_train.astype('float32') / 255
X_test = X_test.astype('float32') / 255
```

```
#Adding CNN layer and epoch running
import matplotlib.pyplot as plt


model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(512, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))

model.add(Flatten())

model.add(Dense(512, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
history = model.fit(X_train, y_train, epochs=30, batch_size=32)

# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)

# Plotting loss
plt.plot(history.history['loss'], label='Training Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Plotting accuracy
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
print('Test accuracy:', accuracy*100)
```

#Import Image

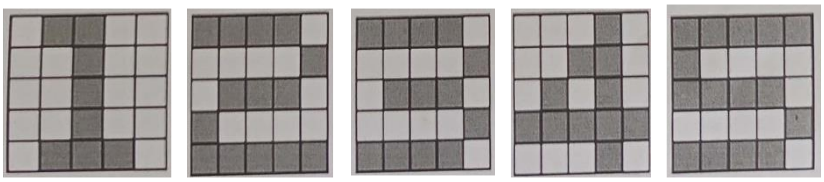
#Test image

```
from tensorflow.keras.preprocessing import image
import numpy as np

# Path to the test image
test_image_path = 'pic2.jpg'  # Replace with the actual path of your test image

# Load and preprocess the test image
test_image = image.load_img(test_image_path, target_size=(150, 150))
test_image = image.img_to_array(test_image)
test_image = np.expand_dims(test_image, axis=0)
test_image = test_image / 255.0  # Normalize the image

# Predict the class of the test image
prediction = model.predict(test_image)
print('prediction',prediction)
if prediction < 0.5:
  print('This is Banana')
elif prediction >= 0.5:
  print('This is Cucumber')
```

Output:

```
1/1 [==============================] - 0s 61ms/step
prediction [[0.9634724]]
This is Cucumber
```

INDEX

Exp No.	Experiment Name
01	Write a MATLAB or program using perception net for AND function with bipolar inputs and targets. The convergence curves and the decision boundary lines are also shown.
02	Write a MATLAB or Python program to recognize the numbers 1 to 4 from the matrix form of number. The net has to be trained to recognize all the numbers, and when the test data is given, the network has to recognize the particular number.
03	Generate the XOR function using the McCulloch-Pitts neuron by writing an M-file or .py file.
04	Write a MATLAB or Python program to show Back Propagation Network for XOR function with Binary Input and Output.
05	Write a MATLAB or Python program for solving linearly separable problem using Perceptron Model. The convergence curves and the decision boundary lines are also shown.
06	Write a MATLAB or Python program for XOR function (binary input and output) with momentum factor using back propagation algorithm.
07	Implement the SGD Method using Delta learning rule for following input-target sets. X_{input} = [0 0 1; 0 1 1; 1 0 1; 1 1 1], D_{Target} = [0; 0; 1; 1]
08	Implement the Batch Method using Delta learning rule for following input-target sets. X_{input} = [0 0 1; 0 1 1; 1 0 1; 1 1 1], D_{Target} = [0; 0; 1; 1]
09	Compare the performance of SGD and the Batch method using the delta learning rule.
10	Write a MATLAB or Python program to recognize the image of digits. The input images are five-by-five-pixel squares, which display five numbers from 1 to 5, as shown in Figure 1. 
11	Write a MATLAB or Python program to classify face/fruit/bird using Convolutional Neural Network(CNN).