**Experiment No :** 01

**Name of the Experiment:** Write a Matlab or Python program using perception net for AND function with bipolar inputs and targets. The convergence curves and the decision boundary lines are also shown.

**Theory:**

Perceptron is one of the simplest[Artificial neural network architectures](https://www.geeksforgeeks.org/artificial-neural-networks-and-its-applications/). It is the simplest type of feedforward neural network, consisting of a single layer of input nodes that are fully connected to a layer of output nodes. It can learn the linearly separable patterns.

Types of Perceptron

* **Single-Layer Perceptron:** This type of perceptron is limited to learning linearly separable patterns. effective for tasks where the data can be divided into distinct categories through a straight line.
* **Multilayer Perceptron:** Multilayer perceptrons possess enhanced processing capabilities as they consist of two or more layers, adept at handling more complex patterns and relationships within the data.

A perceptron, the basic unit of a neural network, comprises essential components that collaborate in information processing.

* **Input Features:** The perceptron takes multiple input features, each input feature represents a characteristic or attribute of the input data.
* **Weights**: Each input feature is associated with a weight, determining the significance of each input feature in influencing the perceptron’s output. During training, these weights are adjusted to learn the optimal values.
* **Summation Function**: The perceptron calculates the weighted sum of its inputs using the summation function. The summation function combines the inputs with their respective weights to produce a weighted sum.
* **Activation Function**: The weighted sum is then passed through an activation function. Perceptron uses Heaviside step function functions. which take the summed values as input and compare with the threshold and provide the output as 0 or 1.
* **Output:** The final output of the perceptron, is determined by the activation function’s result. For example, in binary classification problems, the output might represent a predicted class (0 or 1).
* **Bias:** A bias term is often included in the perceptron model. The bias allows the model to make adjustments that are independent of the input. It is an additional parameter that is learned during training.
* **Learning Algorithm (Weight Update Rule):** During training, the perceptron learns by adjusting its weights and bias based on a learning algorithm. A common approach is the perceptron learning algorithm, which updates weights based on the difference between the predicted output and the true output.

**Convergence Curve**

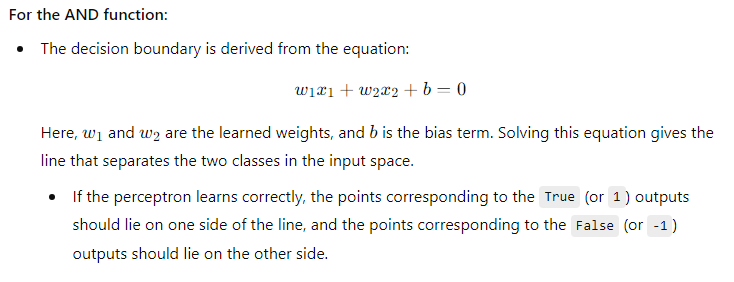
The **convergence curve** in the context of training a perceptron (or any learning model) is a plot that shows how the error rate changes over time (usually measured in epochs or iterations). It visualizes the learning process of the model and indicates whether and how quickly the perceptron is learning to classify inputs correctly.

* **X-axis**: The number of epochs (iterations over the entire dataset).
* **Y-axis**: The number of errors or some form of loss/error metric (in this case, how many misclassifications occurred in each epoch).

### Decision Boundary

The **decision boundary** is a line (or hyperplane, in higher dimensions) that separates the input space into different regions corresponding to different classes. In the case of a 2D input space (like the AND function with two inputs), the decision boundary is a straight line that divides the input space into two regions:

* One region where the perceptron will classify inputs as belonging to one class (e.g., 1 or True).
* The other region where the perceptron will classify inputs as belonging to the other class (e.g., -1 or False).



**Matlab Code:**

% Perceptron for AND function with bipolar inputs and targets

% Define bipolar inputs and targets for the AND function

inputs = [-1, -1;

-1, 1;

1, -1;

1, 1];

targets = [-1;

1;

1;

1];

% Initialize weights and bias

weights = rand(1, size(inputs, 2));

bias = rand();

N = size(inputs, 1);

% Set learning rate and maximum number of epochs

learning\_rate = 0.1;

max\_epochs = 2;

% Initialize variables for storing convergence data

convergence\_curve = zeros(max\_epochs, 1);

converged = false;

% Perceptron training

for epoch = 1:max\_epochs

errors = 0;

for i = 1:size(inputs, 1)

% Calculate the net input (weighted sum of inputs plus bias)

net\_input = dot(weights, inputs(i, :)) + bias

% Apply bipolar threshold activation function

output = 2 \* (net\_input > 0) - 1

% Calculate the error

error = targets(i) - output;

% Update weights and bias

weights = weights + learning\_rate \* error \* inputs(i, :);

bias = bias + learning\_rate \* error;

errors = errors + abs(error);

end

% Check for convergence

if errors == 0

converged = true;

break;

end

convergence\_curve(epoch) = errors/N;

end

% Plot convergence curve

figure;

plot(1:epoch, convergence\_curve(1:epoch));

xlabel('Epoch');

ylabel('Total Error');

title('Convergence Curve');

% Plot decision boundary line

figure;

scatter(inputs(targets==-1, 1), inputs(targets==-1, 2), 'o', 'MarkerFaceColor', 'b');

hold on;

scatter(inputs(targets==1, 1), inputs(targets==1, 2), 'o', 'MarkerFaceColor', 'r');

x\_line = -1:0.1:1;

y\_line = (-bias - weights(1)\*x\_line) / weights(2);

plot(x\_line, y\_line, 'g');

xlabel('Input 1');

ylabel('Input 2');

title('Decision Boundary Line');

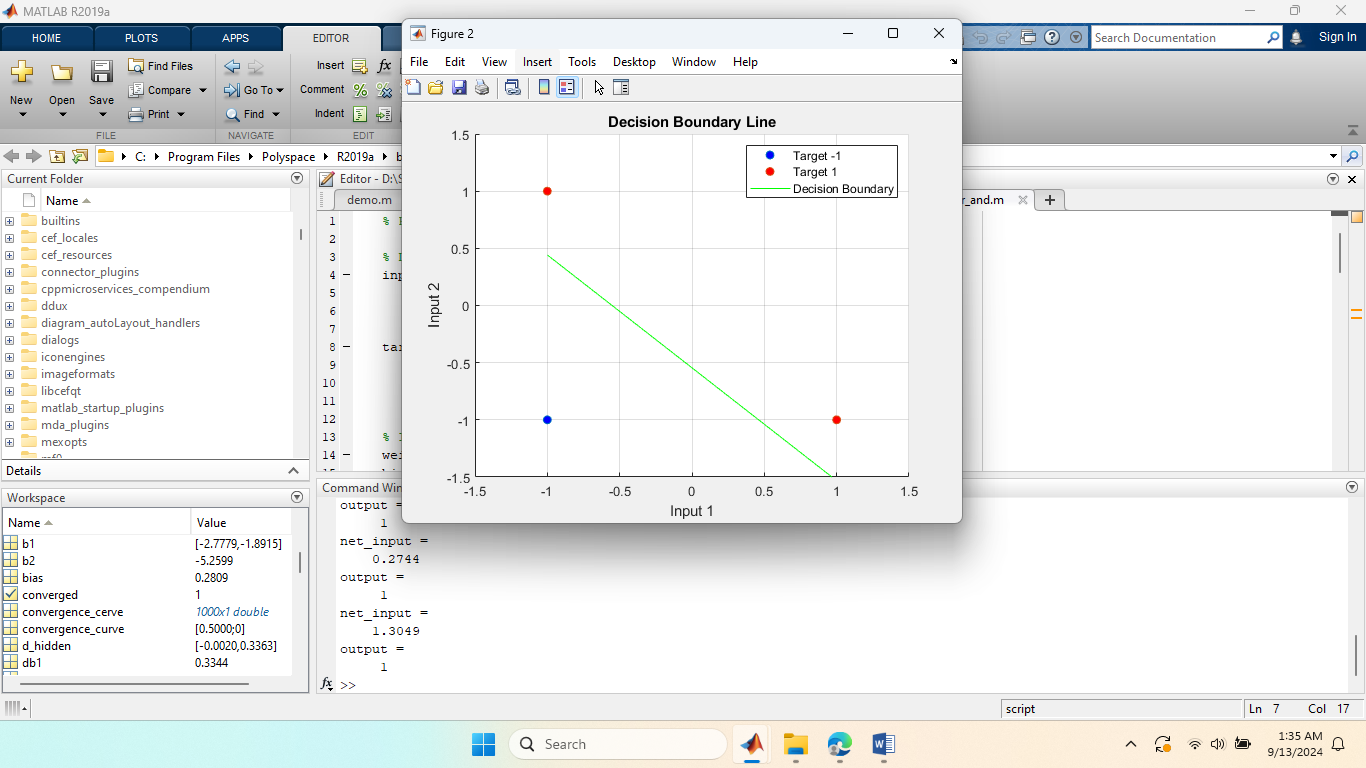
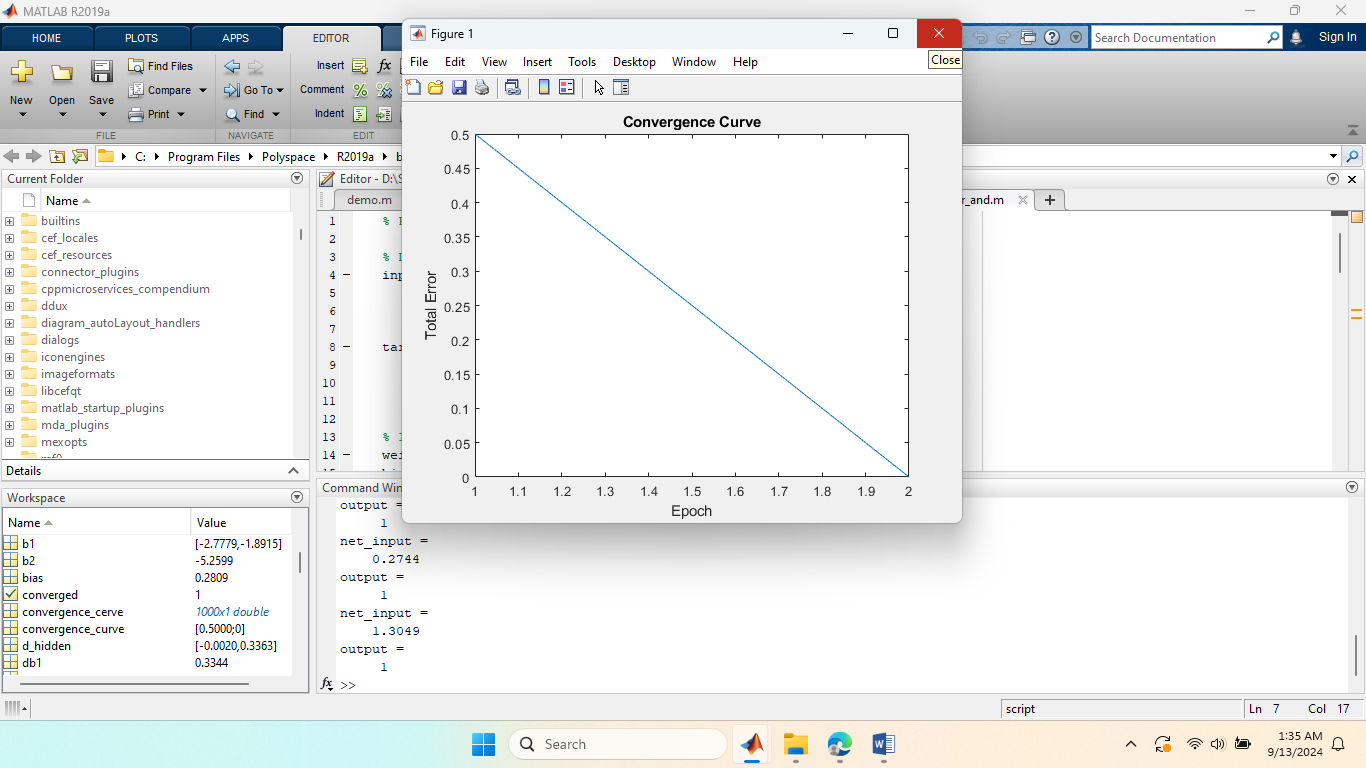
legend('Target -1', 'Target 1', 'Decision Boundary');

axis([-1.5 1.5 -1.5 1.5]);

grid on;

hold off;

**Output:**

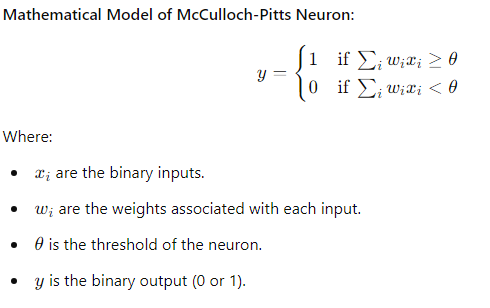


**Experiment No :** 02

**Name of the Experiment:** Generate the XOR function using the McCulloch-Pitts neuron by writing an M-file or .py file. The convergence curves and the decision boundary lines are also shown.

**Theory:**

The McCulloch-Pitts (M-P) neuron is one of the earliest models of a biological neuron, proposed in 1943 by Warren McCulloch and Walter Pitts. It is a simple threshold logic gate that processes binary inputs (either 0 or 1) and produces a binary output (0 or 1). The M-P neuron’s decision-making is based on a weighted sum of inputs and a threshold.



The XOR (exclusive OR) function is a two-input binary function that returns 1 only when the inputs are different.

The XOR function is not linearly separable, meaning that a single McCulloch-Pitts neuron cannot model the XOR function directly. This is because the output of the XOR is not a simple linear combination of the inputs. However, XOR can be modeled using a multi-layer network (at least 2 layers) of McCulloch-Pitts neurons.

**Constructing XOR with McCulloch-Pitts Neurons**

The XOR function can be expressed in terms of the basic logic functions AND, OR, and NOT, which can be implemented by McCulloch-Pitts neurons. The key observation is:

**Matlab Code:**

sigmoid = @(x) 1 ./ (1 + exp(-x));

sigmoid\_derivative = @(x) x .\* (1 - x);

inputs = [0 0; 0 1; 1 0; 1 1];

targets = [0; 1; 1; 0];

input\_layer\_size = 2;

hidden\_layer\_size = 2;

output\_layer\_size = 1;

learning\_rate = 0.1;

max\_epochs = 10000;

rng(42);

weights\_input\_hidden = randn(input\_layer\_size, hidden\_layer\_size);

bias\_hidden = randn(1, hidden\_layer\_size);

weights\_hidden\_output = randn(hidden\_layer\_size, output\_layer\_size);

bias\_output = randn(1, output\_layer\_size);

convergence\_curve = [];

for epoch = 1:max\_epochs

misclassified = 0;

for i = 1:size(inputs, 1)

hidden\_layer\_input = inputs(i, :) \* weights\_input\_hidden + bias\_hidden;

hidden\_layer\_output = sigmoid(hidden\_layer\_input);

output\_layer\_input = hidden\_layer\_output \* weights\_hidden\_output + bias\_output;

predicted\_output = sigmoid(output\_layer\_input);

error = targets(i) - predicted\_output;

if targets(i) ~= round(predicted\_output)

misclassified = misclassified + 1;

end

output\_delta = error \* sigmoid\_derivative(predicted\_output);

hidden\_delta = (output\_delta \* weights\_hidden\_output') .\* sigmoid\_derivative(hidden\_layer\_output);

weights\_hidden\_output = weights\_hidden\_output + hidden\_layer\_output' \* output\_delta \* learning\_rate;

bias\_output = bias\_output + output\_delta \* learning\_rate;

weights\_input\_hidden = weights\_input\_hidden + inputs(i, :)' \* hidden\_delta \* learning\_rate;

bias\_hidden = bias\_hidden + hidden\_delta \* learning\_rate;

end

accuracy = (size(inputs, 1) - misclassified) / size(inputs, 1);

convergence\_curve = [convergence\_curve; accuracy];

if misclassified == 0

fprintf('Converged in %d epochs.\n', epoch);

break;

end

end

x = linspace(-0.5, 1.5, 100);

y1 = (-weights\_input\_hidden(1, 1) \* x - bias\_hidden(1)) / weights\_input\_hidden(2, 1);

y2 = (-weights\_input\_hidden(1, 2) \* x - bias\_hidden(2)) / weights\_input\_hidden(2, 2);

figure;

plot(1:length(convergence\_curve), convergence\_curve, 'LineWidth', 1.5);

xlabel('Epoch');

ylabel('Accuracy');

title('Convergence Curve');

grid on;

figure;

plot(x, y1, 'r', 'DisplayName', 'Decision Boundary 1', 'LineWidth', 1.5); hold on;

plot(x, y2, 'b', 'DisplayName', 'Decision Boundary 2', 'LineWidth', 1.5);

scatter(inputs(targets == 1, 1), inputs(targets == 1, 2), 'filled', 'MarkerEdgeColor', 'k', 'MarkerFaceColor', 'cyan', 'DisplayName', 'Target 1 (1)');

scatter(inputs(targets == 0, 1), inputs(targets == 0, 2), 'filled', 'MarkerEdgeColor', 'k', 'MarkerFaceColor', 'black', 'DisplayName', 'Target 0 (0)');

xlabel('Input 1');

ylabel('Input 2');

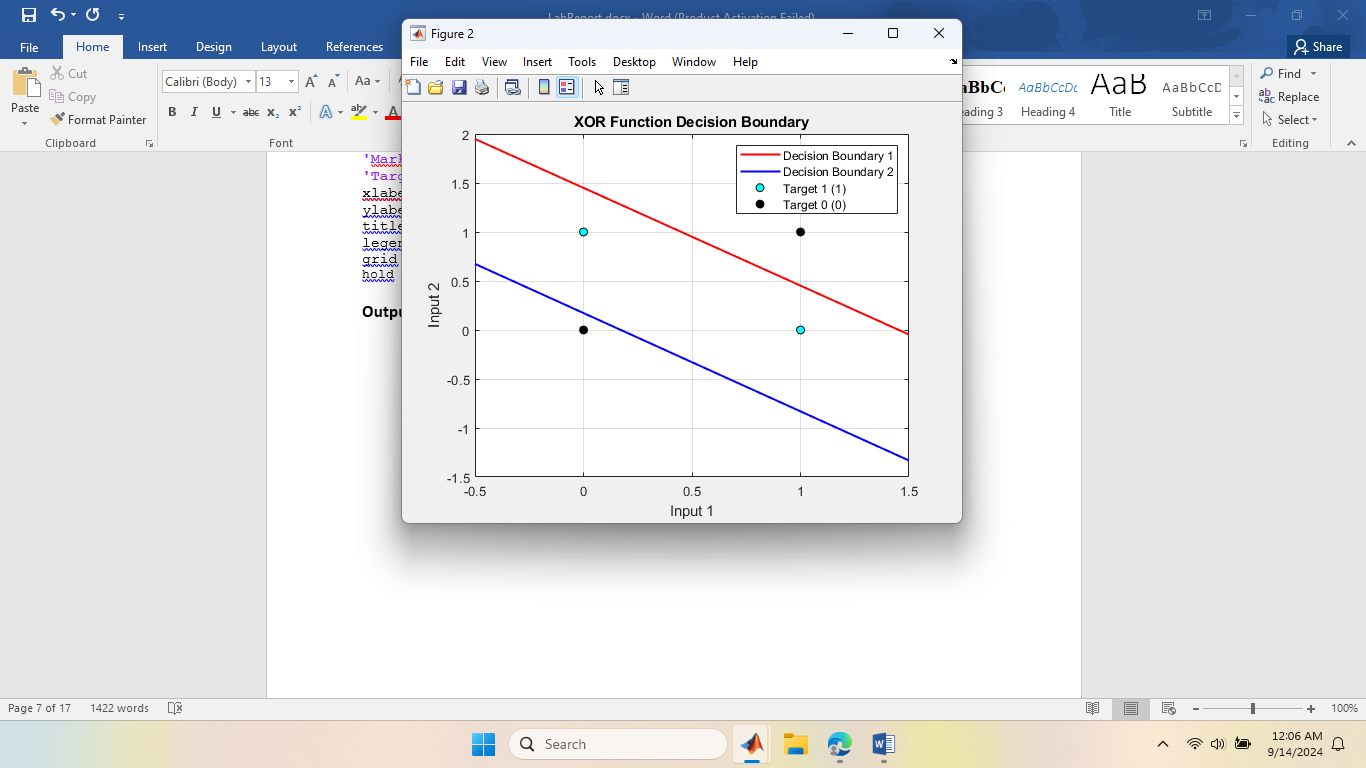
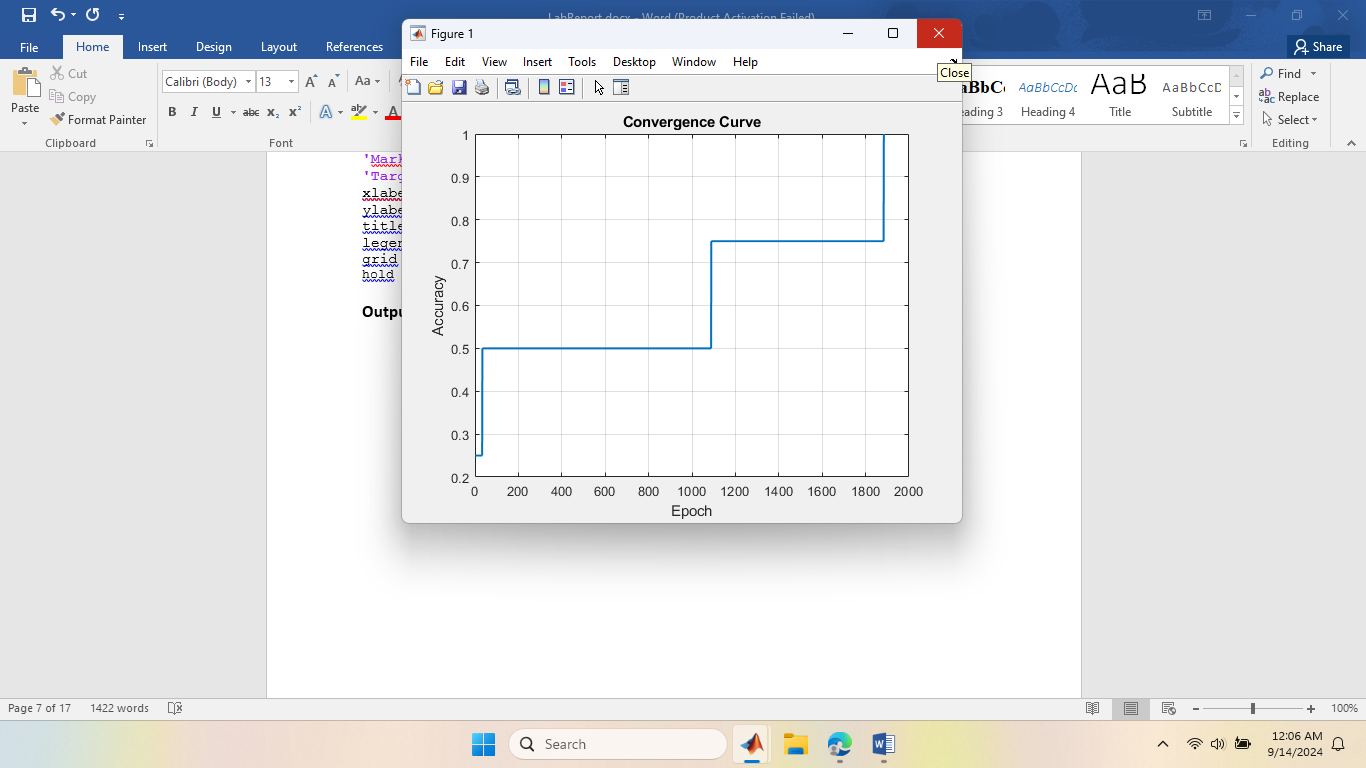
title('XOR Function Decision Boundary');

legend;

grid on;

hold off;

**Output:**



**Experiment No :** 03

**Name of the Experiment:** Implement the SGD Method using Delta learning rule for following input-target sets. = [ 0 0 1; 0 1 1;1 0 1; 1 1 1], = [ 0; 0; 1; 1]

**Theory:**

Gradient Descent is an iterative optimization process that searches for an objective function’s optimum value (Minimum/Maximum). It is one of the most used methods for changing a model’s parameters in order to reduce a cost function in machine learning projects. The primary goal of gradient descent is to identify the model parameters that provide the maximum accuracy on both training and test datasets.

Stochastic Gradient Descent (SGD) is a variant of the [Gradient Descent](https://www.geeksforgeeks.org/gradient-descent-algorithm-and-its-variants/)algorithm that is used for optimizing [machine learning](https://www.geeksforgeeks.org/machine-learning-algorithms/) models. In SGD, instead of using the entire dataset for each iteration, only a single random training example (or a small batch) is selected to calculate the gradient and update the model parameters. This random selection introduces randomness into the optimization process, hence the term “stochastic” in stochastic Gradient Descent

The advantage of using SGD is its computational efficiency, especially when dealing with large datasets. By using a single example or a small batch, the computational cost per iteration is significantly reduced compared to traditional Gradient Descent methods that require processing the entire dataset.

**Matlab Code :**

sigmoid = @(x) 1 ./ (1 + exp(-x));sigmoid\_derivative = @(x) x .\* (1 - x); X\_input = [0 0 1; 0 1 1; 1 0 1; 1 1 1];D\_target = [0; 0; 1; 1];

input\_layer\_size = 3;

output\_layer\_size = 1;

learning\_rate = 0.1;

max\_epochs = 10000;

rng(42); % For reproducibility

weights = randn(input\_layer\_size, output\_layer\_size);

for epoch = 1:max\_epochs

error\_sum = 0;

for i = 1:size(X\_input, 1)

% Forward pass

input\_data = X\_input(i, :);

target\_data = D\_target(i, :);

net\_input = input\_data \* weights;

predicted\_output = sigmoid(net\_input);

error = target\_data - predicted\_output;

error\_sum = error\_sum + abs(error);

weight\_update = learning\_rate \* error \* sigmoid\_derivative(predicted\_output) \* input\_data';

weights = weights + weight\_update;

end

if error\_sum < 0.01

fprintf('Converged in %d epochs.\n', epoch);

break;

end

end

% Test data

test\_data = X\_input;

% Use the trained model to recognize target function

disp('Target Function Test:');

for i = 1:size(test\_data, 1)

input\_data = test\_data(i, :);

net\_input = input\_data \* weights;

predicted\_output = sigmoid(net\_input);

fprintf('Input: [%d %d %d] -> Output: %d\n', input\_data, round(predicted\_output));

end

**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

**Experiment No :** 04

**Name of the Experiment:** Compare the performance of SGD and the Batch method using the delta learning rule.

**Theory:**

In order to train a Linear Regression model, we have to learn some model parameters such as feature weights and bias terms. An approach to do the same is Gradient Descent which is an iterative optimization algorithm capable of tweaking the model parameters by minimizing the cost function over the train data. It is a complete algorithm i.e it is guaranteed to find the global minimum (optimal solution) given there is enough time and the learning rate is not very high. Two Important variants of Gradient Descent which are widely used in Linear Regression as well as Neural networks are Batch Gradient Descent and Stochastic Gradient Descent(SGD).

**Batch Gradient Descent:** Batch Gradient Descent involves calculations over the full training set at each step as a result of which it is very slow on very large training data. Thus, it becomes very computationally expensive to do Batch GD. However, this is great for convex or relatively smooth error manifolds.

Comparison between BGD and SGD.

|  |  |
| --- | --- |
| BGD | SGD |
| Computes gradient using the whole Training sample | Computes gradient using a single Training sample |
| Slow and computationally expensive algorithm | Faster and less computationally expensive than Batch GD |
| Not suggested for huge training samples. | Can be used for large training samples. |
| Convergence is slow. | Reaches the convergence much faster. |
| It typically converges to the global minimum for convex loss functions. | It may converge to a local minimum or saddle point. |
| It may suffer from overfitting if the model is too complex for the dataset. | It can help reduce overfitting by updating the model parameters more frequently. |

**Matlab Code:**

clear all;

X=[0 0 1; 0 1 1; 1 0 1; 1 1 1;];

D= [0; 0; 1; 1];

E1=zeros(1000,1);

E2=zeros(1000,1);

W1=2\*rand(1,3)-1;

W2=W1;

for epoch=1:1000

W1=DeltaSGD(W1,X,D);

W2=DeltaBatch(W2,X,D);

es1=0;

es2=0;

N=4;

for k=1:N

x=X(k,:)';

d=D(k);

v1=W1\*x;

y1=sgmd(v1);

es1=es1+(d-y1)^2;

v2=W2\*x;

y2=sgmd(v2);

es2=es2+(d-y2)^2;

end

E1(epoch)=es1/N;

E2(epoch)=es2/N;

end

plot(E1, 'r')

hold on

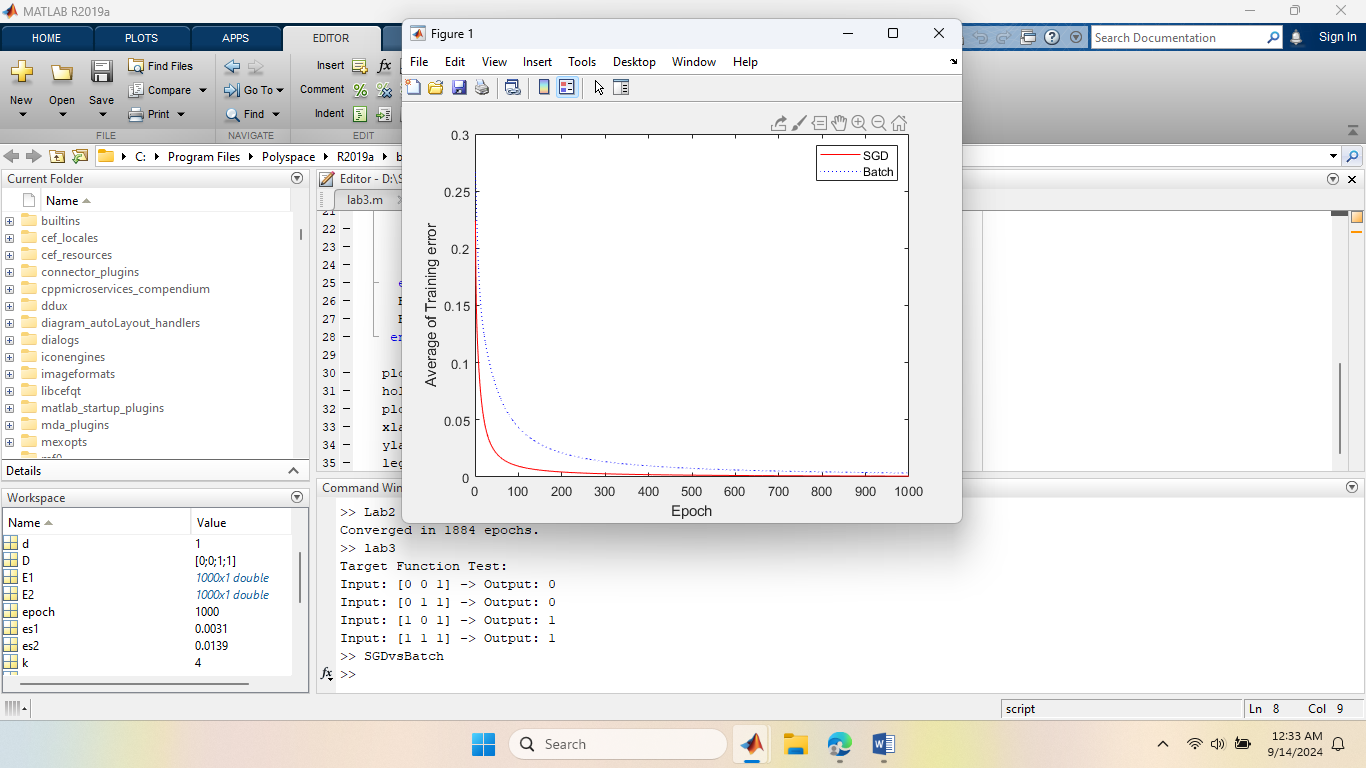
plot(E2, 'b:')

xlabel('Epoch');

ylabel('Average of Training error')

legend('SGD', 'Batch')

**Output:**



**Experiment No :** 05

**Name of the Experiment:** 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.

**Matlab Code:**

function lab5()

rng(3); % Set random seed

X = zeros(5, 5, 5);

X(:, :, 1) = [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(:, :, 2) = [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(:, :, 3) = [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(:, :, 4) = [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(:, :, 5) = [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 = eye(5);

W1 = 2 \* rand(50, 25) - 1;

W2 = 2 \* rand(5, 50) - 1;

for epoch = 1:10000

[W1, W2] = multi\_class(W1, W2, X, D);

end

N = 5;

for k = 1:N

x = reshape(X(:, :, k), [25, 1]);

v1 = W1 \* x;

y1 = sigmoid(v1);

v = W2 \* y1;

y = softmax(v);

fprintf('\n\n Output for X(:,:, %d):\n\n', k);

disp(y);

fprintf('\n The highest value is %f, so this number is correctly identified.\n\n', max(y));

end

end

function [W1, W2] = multi\_class(W1, W2, X, D)

alpha = 0.9;

N = 5;

for k = 1:N

x = reshape(X(:, :, k), [25, 1]);

d = D(k, :)';

v1 = W1 \* x;

y1 = sigmoid(v1);

v = W2 \* y1;

y = softmax(v);

e = d - y;

delta = e;

e1 = W2' \* delta;

delta1 = y1 .\* (1 - y1) .\* e1;

dW1 = alpha \* delta1 \* x';

W1 = W1 + dW1;

dW2 = alpha \* delta \* y1';

W2 = W2 + dW2;

end

end

function y = sigmoid(x)

y = 1 ./ (1 + exp(-x));

end

function y = softmax(x)

ex = exp(x);

y = ex / sum(ex);

end

**Output:**

Output for X(:,:, 1):

1.0000

0.0000

0.0000

0.0000

0.0000

The highest value is 0.999989, so this number is correctly identified.

Output for X(:,:, 2):

0.0000

1.0000

0.0000

0.0000

0.0000

The highest value is 0.999986, so this number is correctly identified.

Output for X(:,:, 3):

0.0000

0.0000

1.0000

0.0000

0.0000

The highest value is 0.999978, so this number is correctly identified.

Output for X(:,:, 4):

0.0000

0.0000

0.0000

1.0000

0.0000

The highest value is 0.999993, so this number is correctly identified.

Output for X(:,:, 5):

0.0000

0.0000

0.0000

0.0000

1.0000

The highest value is 0.999983, so this number is correctly identified.

**Experiment No :** 06

**Name of the Experiment:** Write a MATLAB or Python program to classify face/fruit/bird using Convolution Neural Network (CNN).

**Theory :**

A [Convolutional Neural Network](https://www.geeksforgeeks.org/neural-networks-a-beginners-guide/)(CNN) is a type of [deep learning algorithm](https://www.geeksforgeeks.org/deep-learning-introduction-to-long-short-term-memory/) that is particularly well-suited for image recognition and processing tasks. It is made up of multiple layers, including convolutional layers, [pooling layers](https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/), and fully connected layers. The architecture of CNNs is inspired by the visual processing in the human brain, and they are well-suited for capturing hierarchical patterns and spatial dependencies within images.

Key components of a Convolutional Neural Network include:

1. Convolutional Layers: These layers apply convolutional operations to input images, using filters (also known as kernels) to detect features such as edges, textures, and more complex patterns. Convolutional operations help preserve the spatial relationships between pixels.
2. Pooling Layers: Pooling layers downsample the spatial dimensions of the input, reducing the computational complexity and the number of parameters in the network. Max pooling is a common pooling operation, selecting the maximum value from a group of neighboring pixels.
3. [Activation Functions](https://www.geeksforgeeks.org/activation-functions/): Non-linear activation functions, such as Rectified Linear Unit (ReLU), introduce non-linearity to the model, allowing it to learn more complex relationships in the data.
4. Fully Connected Layers: These layers are responsible for making predictions based on the high-level features learned by the previous layers. They connect every neuron in one layer to every neuron in the next layer.

**Matlab Code:**

fruitsPath = 'D:\Study\_Object\4\_2\_Course\NeuralNetworks\Neural network lab\path\_to\_sample\_fruit\_image\path\_to\_sample\_fruit\_image';

imdsTrain = imageDatastore(fruitsPath, ...

'IncludeSubfolders', true, 'LabelSource', 'foldernames');

numClasses = 3;

layers = [

imageInputLayer([224 224 3])

convolution2dLayer(3, 32, 'Padding', 'same')

reluLayer

maxPooling2dLayer(2, 'Stride', 2)

convolution2dLayer(3, 64, 'Padding', 'same')

reluLayer

maxPooling2dLayer(2, 'Stride', 2)

convolution2dLayer(3, 128, 'Padding', 'same')

reluLayer

fullyConnectedLayer(64)

reluLayer

fullyConnectedLayer(numClasses)

softmaxLayer

classificationLayer

];

miniBatchSize = 32;

numEpochs = 20;

initialLearnRate = 1e-4;

options = trainingOptions('adam', ...

'MiniBatchSize', miniBatchSize, ...

'MaxEpochs', numEpochs, ...

'InitialLearnRate', initialLearnRate, ...

'Shuffle', 'every-epoch', ...

'ValidationData', imdsTrain, ...

'ValidationFrequency', 50, ...

'Verbose', true, ...

'Plots', 'training-progress');

net = trainNetwork(imdsTrain, layers, options);

save('D:\Study\_Object\4\_2\_Course\NeuralNetworks\Neural network lab\path\_to\_sample\_fruit\_image\path\_to\_sample\_fruit\_image', 'net');

sampleImageFile = imdsTrain.Files{randi(length(imdsTrain.Files))};

sampleImage = imread(sampleImageFile);

inputImage = imresize(sampleImage, [224 224]);

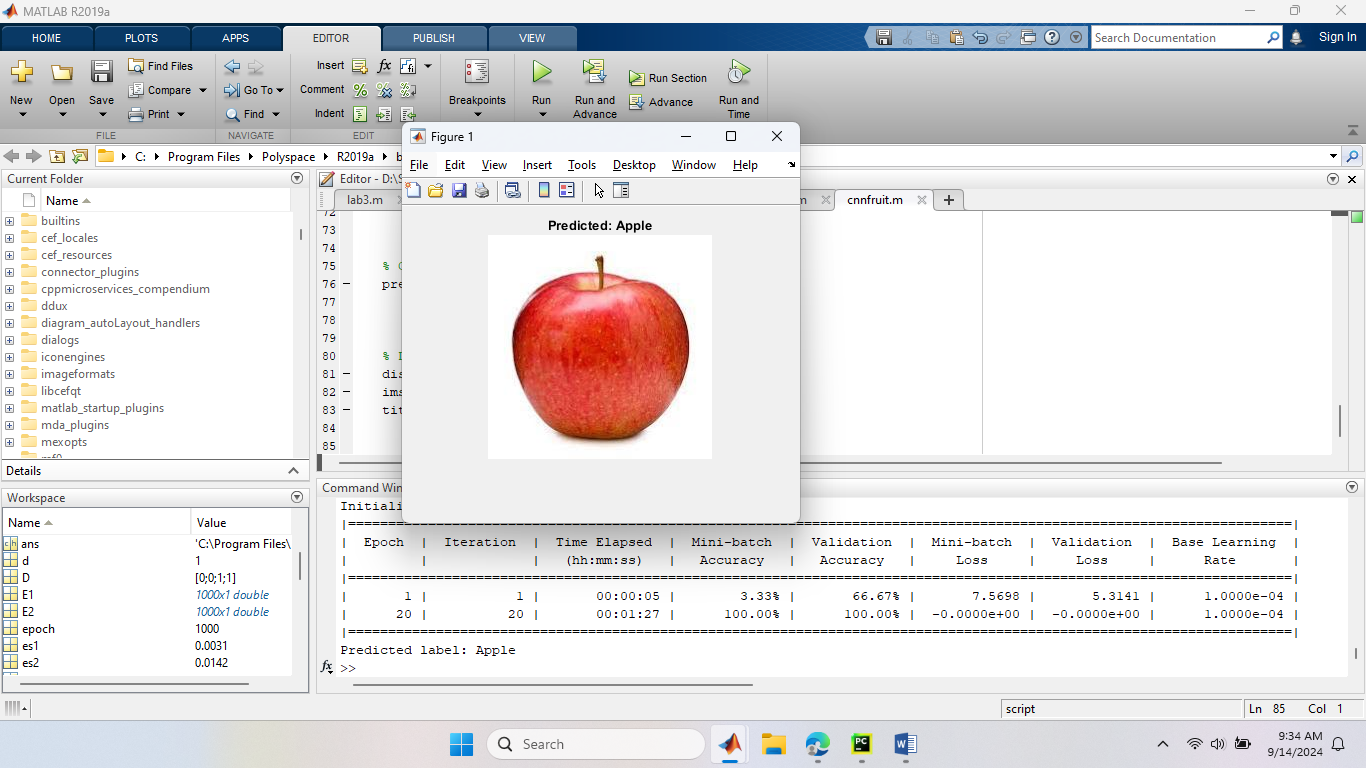
predictedLabel = classify(net, inputImage);

disp(['Predicted label: ' char(predictedLabel)]);

imshow(sampleImage);

title(['Predicted: ' char(predictedLabel)]);

**Output:**



**Experiment No :** 07

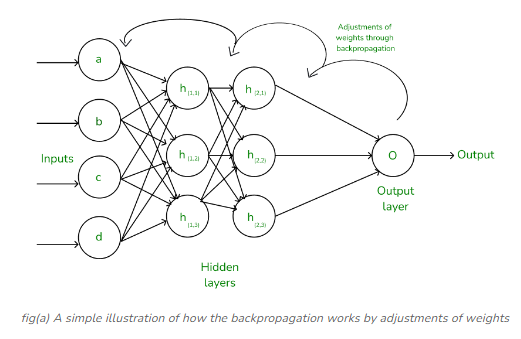
**Name of the Experiment:** Consider an artificial neural network (ANN) with three layers given below. Write a MATLAB or Python program to learn this network using Back Propagation Network.

**Theory:**

Artificial Neural Networks contain artificial neurons which are called **units**. These units are arranged in a series of layers that together constitute the whole Artificial Neural Network in a system. A layer can have only a dozen units or millions of units as this depends on how the complex neural networks will be required to learn the hidden patterns in the dataset.

In machine learning, backpropagation is an effective algorithm used to train artificial neural networks, especially in feed-forward neural networks.

Backpropagation is an iterative algorithm, that helps to minimize the cost function by determining which weights and biases should be adjusted. During every epoch, the model learns by adapting the weights and biases to minimize the loss by moving down toward the gradient of the error. Thus, it involves the two most popular optimization algorithms, such as gradient descent or stochastic gradient descent. Computing the gradient in the backpropagation algorithm helps to minimize the cost function and it can be implemented by using the mathematical rule called chain rule from calculus to navigate through complex layers of the neural network.



**Matlab Code:**

x1 = 0.05;

x2 = 0.10;

inputs = [x1; x2]; % Input vector

T1 = 0.01;

T2 = 0.99;

targets = [T1; T2]; % Target vector

w1 = 0.15; w2 = 0.20; % Weights for input to H1

w3 = 0.25; w4 = 0.30; % Weights for input to H2

w5 = 0.40; w6 = 0.45; % Weights from H1, H2 to y1

w7 = 0.50; w8 = 0.55; % Weights from H1, H2 to y2

b1 = 0.35; % Bias for hidden layer H1, H2

b2 = 0.60; % Bias for output layer y1, y2

learning\_rate = 0.5;

epochs = 10000; % Number of iterations

for epoch = 1:epochs

% Forward pass

h1\_input = w1 \* x1 + w2 \* x2 + b1;

h2\_input = w3 \* x1 + w4 \* x2 + b1;

h1\_output = sigmoid(h1\_input);

h2\_output = sigmoid(h2\_input);

y1\_input = w5 \* h1\_output + w6 \* h2\_output + b2;

y2\_input = w7 \* h1\_output + w8 \* h2\_output + b2;

y1\_output = sigmoid(y1\_input);

y2\_output = sigmoid(y2\_input);

% Calculate the error (Mean Squared Error)

error1 = (y1\_output - T1)^2;

error2 = (y2\_output - T2)^2;

total\_error = (error1 + error2) / 2;

% Backward pass (Backpropagation)

% Calculate gradients for output layer

delta\_y1 = (y1\_output - T1) \* sigmoid\_derivative(y1\_output);

delta\_y2 = (y2\_output - T2) \* sigmoid\_derivative(y2\_output);

% Calculate gradients for hidden layer

delta\_h1 = (delta\_y1 \* w5 + delta\_y2 \* w7) \* sigmoid\_derivative(h1\_output);

delta\_h2 = (delta\_y1 \* w6 + delta\_y2 \* w8) \* sigmoid\_derivative(h2\_output);

% Update weights and biases

w1 = w1 - learning\_rate \* delta\_h1 \* x1;

w2 = w2 - learning\_rate \* delta\_h1 \* x2;

w3 = w3 - learning\_rate \* delta\_h2 \* x1;

w4 = w4 - learning\_rate \* delta\_h2 \* x2;

w5 = w5 - learning\_rate \* delta\_y1 \* h1\_output;

w6 = w6 - learning\_rate \* delta\_y1 \* h2\_output;

w7 = w7 - learning\_rate \* delta\_y2 \* h1\_output;

w8 = w8 - learning\_rate \* delta\_y2 \* h2\_output;

b1 = b1 - learning\_rate \* (delta\_h1 + delta\_h2);

b2 = b2 - learning\_rate \* (delta\_y1 + delta\_y2);

% Display the error every 1000 epochs

if mod(epoch, 1000) == 0

fprintf('Epoch %d, Total Error: %f\n', epoch, total\_error);

end

end

fprintf('Final output y1: %f\n', y1\_output);

fprintf('Final output y2: %f\n', y2\_output);

% Sigmoid function definition

function y = sigmoid(x)

y = 1 ./ (1 + exp(-x));

end

% Derivative of sigmoid function

function y = sigmoid\_derivative(x)

y = x .\* (1 - x);

end

**Output:**

Epoch 1000, Total Error: 0.000347

Epoch 2000, Total Error: 0.000119

Epoch 3000, Total Error: 0.000060

Epoch 4000, Total Error: 0.000035

Epoch 5000, Total Error: 0.000023

Epoch 6000, Total Error: 0.000015

Epoch 7000, Total Error: 0.000011

Epoch 8000, Total Error: 0.000008

Epoch 9000, Total Error: 0.000006

Epoch 10000, Total Error: 0.000004

Final output y1: 0.012102

Final output y2: 0.987928

**Experiment No :** 08

**Name of the Experiment:** Write a MATLAB or Python program to recognize the numbers 1 to 4 from speech signal using artificial neural network (ANN).

**Experiment No :** 09

**Name of the Experiment:** Write a MATLAB or Python program to Purchase Classification Prediction using SVM.

**Theory :**

Support Vector Machine (SVM) is a [supervised machine learning](https://www.geeksforgeeks.org/supervised-unsupervised-learning/) algorithm used for both classification and regression. Though we say regression problems as well it’s best suited for classification. The main objective of the SVM algorithm is to find the optimal [hyperplane](https://www.geeksforgeeks.org/separating-hyperplanes-in-svm/) in an N-dimensional space that can separate the data points in different classes in the feature space.

**Matlab Code:**

num\_samples = 1000;

Age = randi([30, 80], num\_samples, 1); % Age between 30 and 80

BloodPressure = randi([90, 200], num\_samples, 1);

Cholesterol = randi([150, 300], num\_samples, 1);

RestingHeartRate = randi([50, 100], num\_samples, 1);

Smoking = randi([0, 1], num\_samples, 1);

FamilyHistory = randi([0, 1], num\_samples, 1);

ExerciseLevel = randi([0, 2], num\_samples, 1);

HeartDisease = (BloodPressure > 140 | Cholesterol > 240 | Age > 55 | Smoking == 1 | FamilyHistory == 1) & ExerciseLevel == 0;

data = table(Age, BloodPressure, Cholesterol, RestingHeartRate, Smoking, FamilyHistory, ExerciseLevel, HeartDisease);

disp(data(1:10, :));

writetable(data, 'D:\Study\_Object\4\_2\_Course\NeuralNetworks\LbMitu.csv');

data = readtable('D:\Study\_Object\4\_2\_Course\NeuralNetworks\LbMitu.csv');

% Display the first few rows of the dataset

disp(data(1:5, :));

Y = data.HeartDisease; % 'HeartDisease' column

cv = cvpartition(size(X, 1), 'HoldOut', 0.3); XTrain = X(training(cv), :);

YTrain = Y(training(cv), :);

XTest = X(test(cv), :);

YTest = Y(test(cv), :);

svmModel = fitcsvm(XTrain, YTrain, 'KernelFunction', 'linear', 'Standardize', true);

CVSVMModel = crossval(svmModel);

loss = kfoldLoss(CVSVMModel);

fprintf('Cross-validation loss: %.4f\n', loss);

YPred = predict(svmModel, XTest);

accuracy = sum(YPred == YTest) / length(YTest) \* 100;

fprintf('Test set accuracy: %.2f%%\n', accuracy);

% Display confusion matrix

confMat = confusionmat(YTest, YPred);

disp('Confusion Matrix:');

disp(confMat);

% Plot confusion matrix

figure;

confusionchart(YTest, YPred);

title('Confusion Matrix for Heart Disease Prediction');

**Output:**

Cross-validation loss: 0.0100

Test set accuracy: 98.67%

Confusion Matrix:

198 4

0 98

**Experiment No :** 10

**Name of the Experiment:** Write a MATLAB or Python program to reduce dimensions of a dataset into a new coordinate system using PCA algorithm

**Theory:**

[Principal Component Analysis](https://www.geeksforgeeks.org/principal-component-analysis-with-python/)(PCA) technique was introduced by the mathematician **Karl Pearson** in 1901**.** It works on the condition that while the data in a higher dimensional space is mapped to data in a lower dimension space, the variance of the data in the lower dimensional space should be maximum.

* Principal Component Analysis (PCA)is a statistical procedure that uses an orthogonal transformation that converts a set of correlated variables to a set of uncorrelated variables. PCA is the most widely used tool in exploratory data analysis and in machine learning for predictive models. Moreover,
* Principal Component Analysis (PCA) is an [unsupervised learning](https://www.geeksforgeeks.org/supervised-unsupervised-learning/) algorithm technique used to examine the interrelations among a set of variables. It is also known as a general factor analysis where regression determines a line of best fit.
* The main goal of Principal Component Analysis (PCA) is to reduce the dimensionality of a dataset while preserving the most important patterns or relationships between the variables without any prior knowledge of the target variables.

Principal Component Analysis (PCA) is used to reduce the dimensionality of a data set by finding a new set of variables, smaller than the original set of variables, retaining most of the sample’s information, and useful for the [regression and classification](https://www.geeksforgeeks.org/regression-classification-supervised-machine-learning/) of data.

**Matlab Code:**

% Load the Iris dataset

load fisheriris

X = meas; % Features

y = species; % Target variable

% Standardize the features

X\_scaled = (X - mean(X)) ./ std(X);

disp("Before:");

disp(size(X\_scaled));

% Perform PCA

[coeff, X\_pca, latent, ~, explained] = pca(X\_scaled); % Reduce to 2 dimensions

% Select the first two principal components

X\_pca = X\_pca(:, 1:2);

disp("After:");

disp(size(X\_pca));

% Create a table for the reduced data

pca\_table = array2table(X\_pca, 'VariableNames', {'PrincipalComponent1', 'PrincipalComponent2'});

pca\_table.Target = categorical(y);

% Plot the PCA results

figure;

gscatter(pca\_table.PrincipalComponent1, pca\_table.PrincipalComponent2, pca\_table.Target, [], 'osd', [], 'off');

xlabel('Principal Component 1');

ylabel('Principal Component 2');

title('PCA of Iris Dataset');

legend('Location', 'best');

grid on;

% Explained variance

explained\_variance = explained(1:2) / sum(explained);

total\_explained\_variance = sum(explained\_variance);

disp(['Explained variance by component: ', num2str(explained\_variance')]);

disp(['Total explained variance: ', num2str(total\_explained\_variance)]);

**Output:**

