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:

In Machine Learning, perceptron is an algorithm used for supervised learning of binary classifiers. It determines whether or not an input belong to a specific class. It is a type of liner classifier.

Perceptron algorithm:

1. Initialize input_vector, weight, bias, desired_output and learning rate.
2. Calculate predicted output, y(n) = sign(weight*input + bias)
3. If desired_output = predicted output then stop
   Else
   Update weight y(n+1) = w(n)+learning rate *(desried output – predicted output)*input
4. Go to step 2 until weight updated stop.

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:

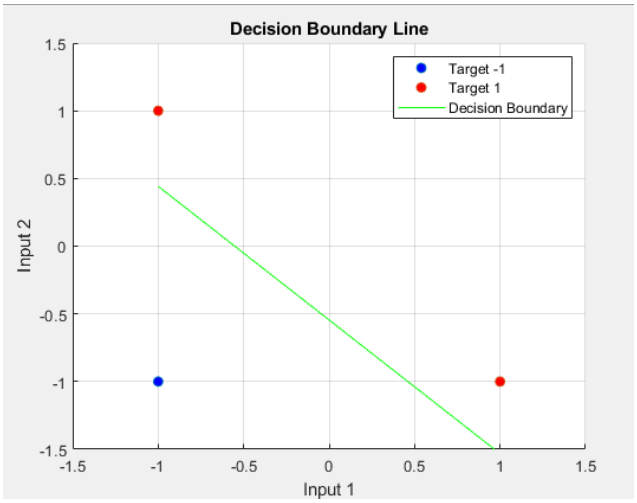
Matlab Code:

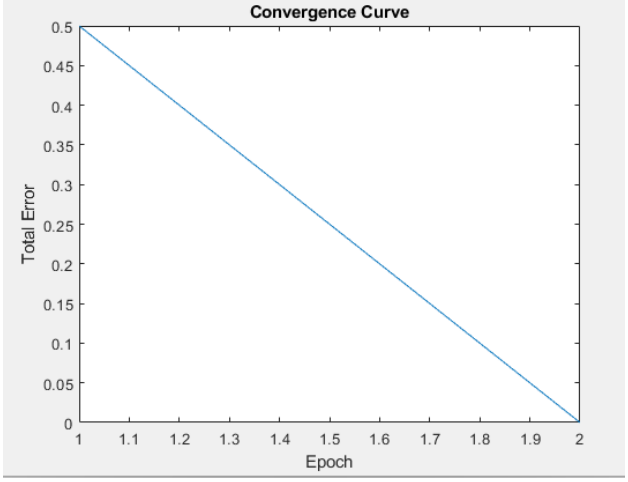
```
% Perceptron for AND function with bipolar inputs and targets
% Define bipolar inputs and targets for the AND function
inputs = [-1, -1;
         -1, 1;
          1, -1;
          1, 1];
targets = [-1;
           1;
           1;
           1];

% 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.

Mathematical Model of McCulloch-Pitts Neuron:

$$y = \begin{cases} 1 & \text{if } \sum_i w_i x_i \geq \theta \\ 0 & \text{if } \sum_i w_i x_i < \theta \end{cases}$$

Where:

- x_i are the binary inputs.
- w_i are the weights associated with each input.
- θ is the threshold of the neuron.
- y is the binary output (0 or 1).

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:

$$\text{XOR}(x_1, x_2) = (x_1 \text{ AND } \neg x_2) \text{ OR } (\neg x_1 \text{ AND } x_2)$$

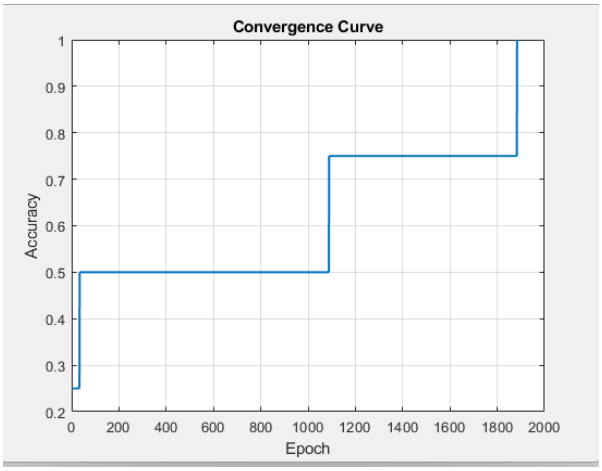
Matlab Code:

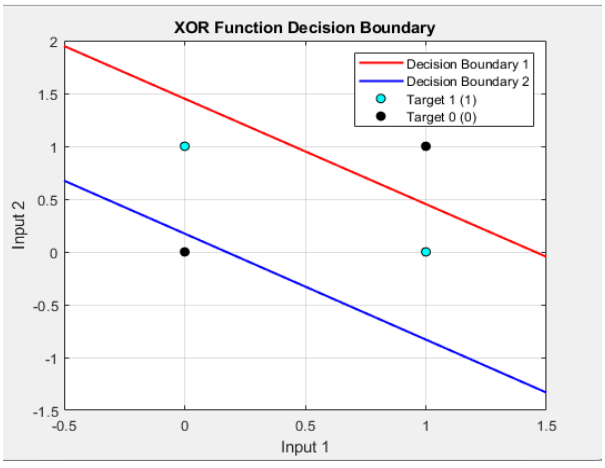
```
sigmoid = @(x) 1 ./ (1 + exp(-x));
sigmoid_derivative = @(x) x .* (1 - x);
inputs = [0 0; 0 1; 1 0; 1 1];
targets = [0; 1; 1; 0];
```

```
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:

Delta rule is used for non-linearly separable data. Delta rule use the gradient decend rule and find out the best weights.

1. General Formula for Delta Rule:

The **weight update** formula using the Delta Rule is:

$$w_{new} = w_{old} + \Delta w$$

Where:

$$\Delta w = \eta \cdot (d - y) \cdot x$$

- w_{new} is the updated weight.
- w_{old} is the current weight.
- η is the **learning rate** (controls the step size of the weight update).
- d is the **desired output** or target value.
- y is the **actual output** (the predicted output by the model).
- x is the **input value** corresponding to the weight.

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:

There are two important variants of gradient descent that are widely used in linear regression and Neural Network. They are Batch Gradient Descent and Stochastic Gradient Descent.

BGD: It involves calculations over the full training set at each step. It is relatively slow on very large training data.

SGD: In SGD, instead of using the entire dataset for each iteration only a single random training example is selected to calculate the gradient and update the model parameters.

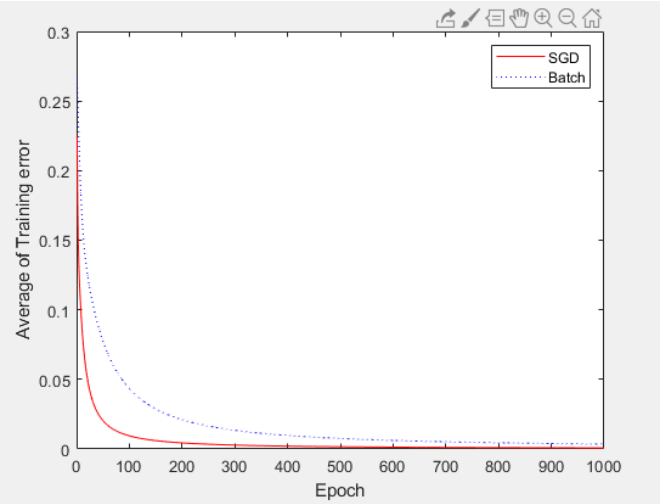
BGD takes longer to converge than SGD. On the other hand BGD is more accurate than SGD.

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 (CNN) is a type of deep learning algorithm that is particularly well-suited for image recognition and processing tasks. It is made up of multiple layers, including convolutional layers, pooling layers, 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: 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

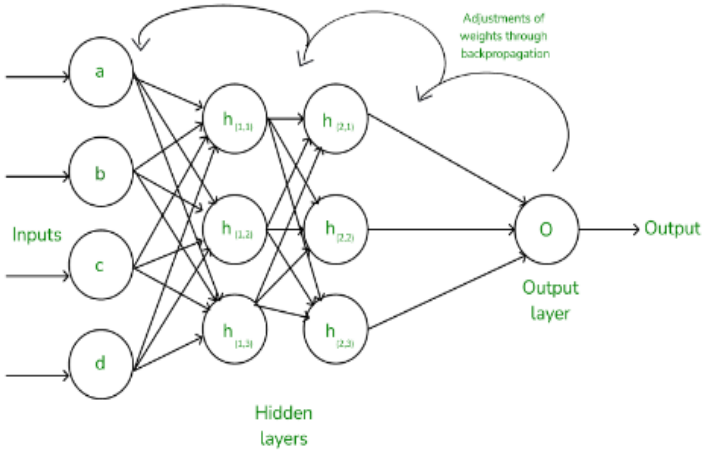
Name of the Experiment: Consider an artificial neural network (ANN) with three layers given below. Write a MATLAB or Python program to learn this network using Back Propagation Network.

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.



fig(a) A simple illustration of how the backpropagation works by adjustments of weights

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).

Matlab Code:

```
% Load a speech signal (Replace 'speech_signal.wav' with actual
speech files)
[x, Fs] = audioread('speech_signal.wav');

% Extract MFCCs from the audio signal
coeffs = mfcc(x, Fs);

% Use the first few coefficients as features (optional)
features = coeffs(:, 1:13);  % Use first 13 coefficients
% Define labels (for example, 1-4 corresponding to different files)
labels = [1; 2; 3; 4];  % Labels for numbers

% Create an ANN with one hidden layer
net = feedforwardnet(10);  % ANN with 10 hidden neurons

% Train the ANN with the features and labels
net = train(net, features', labels');

% Test the network with a new speech signal
new_audio = audioread('new_speech_signal.wav');
new_coeffs = mfcc(new_audio, Fs);
new_features = new_coeffs(:, 1:13);

% Predict the number
predicted_number = net(new_features');
disp(['Predicted Number: ', num2str(round(predicted_number))]);
```

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 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 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:

As the number of features or dimensions in a dataset increases, the amount of data required to obtain a statistically significant result increases exponentially. This can lead to issues such as overfitting, increased computation time, and reduced accuracy of machine learning models this is known as the curse of dimensionality problems that arise while working with high-dimensional data.

As the number of dimensions increases, the number of possible combinations of features increases exponentially, which makes it computationally difficult to obtain a representative sample of the data. It becomes expensive to perform tasks such as clustering or classification because the algorithms need to process a much larger feature space, which increases computation time and complexity. Additionally, some machine learning algorithms can be sensitive to the number of dimensions, requiring more data to achieve the same level of accuracy as lower-dimensional data.

To address the curse of dimensionality, Feature engineering techniques are used which include feature selection and feature extraction. Dimensionality reduction is a type of feature extraction technique that aims to reduce the number of input features while retaining as much of the original information as possible.

In this article, we will discuss one of the most popular dimensionality reduction techniques i.e. Principal Component Analysis(PCA).

What is Principal Component Analysis(PCA)?

Principal Component Analysis(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 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 of data.

Matlab Code:

```
load fisheriris
X = meas;  % Features
y = species;
X_scaled = (X - mean(X)) ./ std(X);
disp("Before:");
disp(size(X_scaled));
[coeff, X_pca, latent, ~, explained] = pca(X_scaled);
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: