Pabna University of Science And Technology

Faculty of Engineering & Technology

Department of Information and Communication Engineering

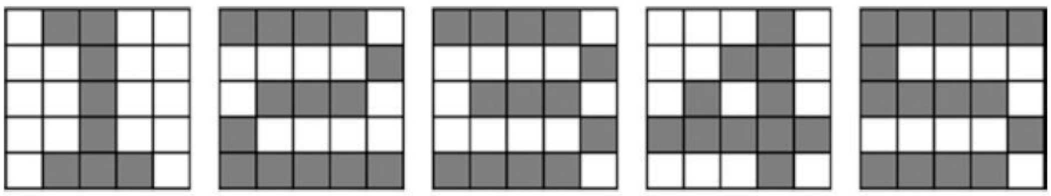
PRACTICAL LAB REPORT

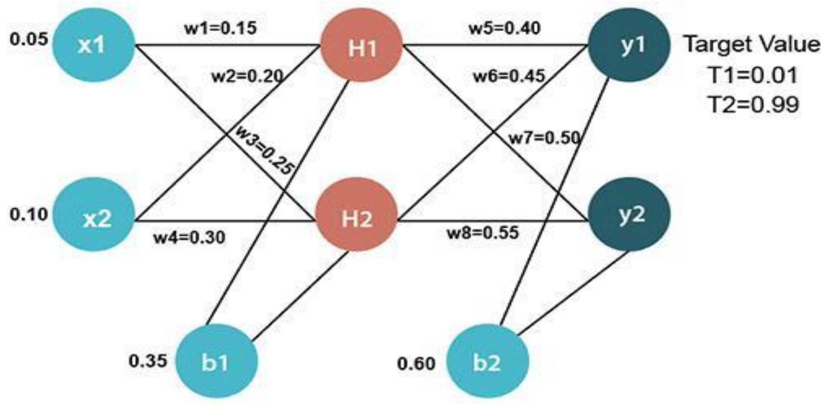
Course Code: *ICE- 4206.*

Course Title: Neural Networks Sessional.

Submitted by:	**Submitted to:**
Name: MD RAHATUL RABBI	**Dr. Md. Imran Hossain**
Roll No: 190609	**Associate Professor**
Reg. No: 1065334	**Department of ICE,**
Session: 2018-2019	**Pabna University of Science and Technology**
4th year 2nd Semester	
Department of ICE,	**Teacher's Signature**
Pabna university of Science and Technology	

INDEX

S\N	Experiment Name
1	Write a MATLAB or Python program using perceptron net for AND function with bipolar inputs and targets. The convergence curves and the decision boundary lines are also shown.
2	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.
3	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]
4	Compare the performance of SGD and the Batch method using the delta learning rule.
5	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  ***Figure 1*** *Five-by-five pixel squares that display five numbers from 1 to 5*
6	Write a MATLAB or Python program to classify face/fruit/bird using Convolution Neural Network (CNN).

7	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.  0.05 x1
	0.10 x2
	w1=0.15
	w2=0.20
	w3=0.25
	w4=0.30
	H1
	H2
	w5=0.40
	w6=0.45
	w7=0.50
	w8=0.55
	y1
	y2
	Target Value
	T1=0.01
	T2=0.99
	0.35 b1
	0.60 b2

8	Write a MATLAB or Python program to recognize the numbers 1 to 4 from speech signal using artificial neural network (ANN).
9	Write a MATLAB or Python program to Purchase Classification Prediction using SVM.
10	Write a MATLAB or Python program to reduce dimensions of a dataset into a new coordinate system using PCA algorithm.

Experiment No: 01

Experiment Name: To write a MATLAB or Python program using perceptron net for AND function with bipolar inputs and targets. The convergence curves and the decision boundary lines are also shown.

Objectives:

i. Implement a Perceptron Neural Network to solve the AND function using bipolar inputs and targets.
ii. To visualize the convergence of the network using convergence curves.
iii. To plot the decision boundary for the perceptron.

Theory:

Perceptron: A **perceptron** is the simplest type of artificial neural network, which consists of a single layer of weights connected to the inputs. The perceptron algorithm learns a linear decision boundary by adjusting its weights based on the error from predictions.

In the case of the **AND function**, the output is 1 only when both inputs are 1; otherwise, the output is -1 for bipolar inputs. The **bipolar input representation** for the AND function is:

Input-1	Input-2	AND Output
-1	**-1**	**-1**
-1	**1**	**-1**
1	**-1**	**-1**
1	**1**	**1**

The perceptron updates its weights using the rule:

$$w_i(t+1) = w_i(t) + \eta \cdot (y - \hat{y}) \cdot x_i$$

Where:

w_i = weight
η = learning rate
y = actual target
$\hat{y}$ = predicted output
x_i = input

Convergence Curves: Convergence Curves show how a neural network's performance evolves during training. They typically plot the **loss** or **accuracy** over epochs, with decreasing loss indicating effective learning.

Decision Boundary: Decision Boundary refers to the boundary a neural network creates to separate different classes in the feature space. It represents the model's decision logic for classifying data points, and can be linear or complex depending on the data and the architecture of the network.

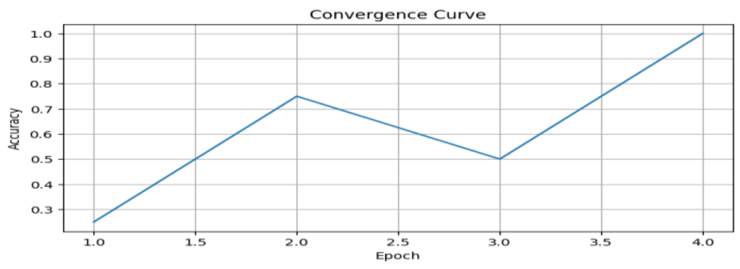
Source Code (Python):

```
import numpy as np
import matplotlib.pyplot as plt
# Bipolar activation function
def bipolar_activation(x):
    return 1 if x >= 0 else -1
```

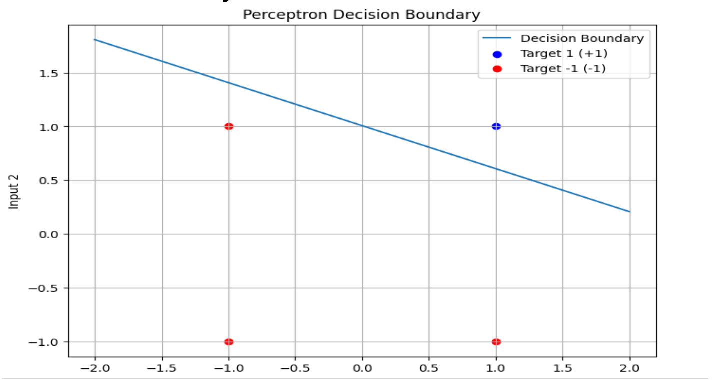
```
# Perceptron training function
def perceptron_train(inputs, targets, learning_rate=0.1, max_epochs=100):
    num_inputs = inputs.shape[1]
    num_samples = inputs.shape[0]

    # 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:**



———————— 0 ————————

Experiment No: 02

Experiment Name: To 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.

Objectives:

i. To implement the XOR function using McCulloch-Pitts neurons.
ii. To visualize the convergence curves.
iii. To plot the decision boundary for the neuron network.

Theory:

McCulloch-Pitts Neuron: The **McCulloch-Pitts neuron** is a model of a neuron that can either activate (output `1`) or not activate (output `0`), based on a threshold function. It is a simple linear classifier and was originally designed to solve simple functions like **AND** and **OR**. However, the XOR function is **non-linearly separable**, meaning a single layer perceptron cannot solve it. We need to use a **multi-layer perceptron** to solve the XOR problem.

The XOR function has the following truth table:

Input 1	Input 2	XOR Output
0	0	0
0	1	1
1	0	1
1	1	0

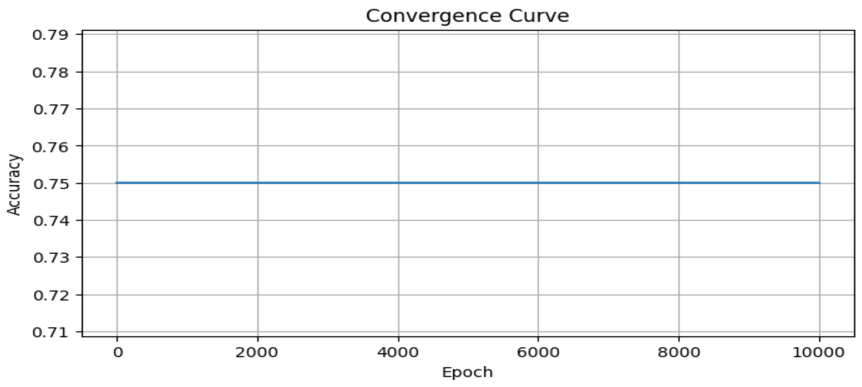
For the XOR problem, we can use two McCulloch-Pitts neurons in the hidden layer to solve the problem. The **decision boundary** is formed by combining these neurons.

Source Code (Python):

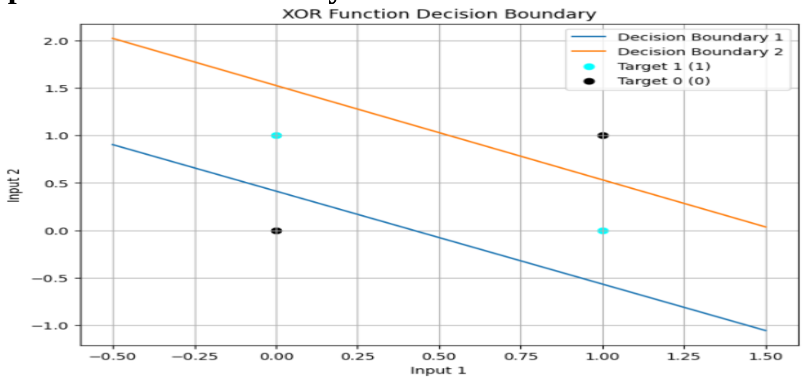
```
#XOR implementation using McCulloch pit neuron
import numpy as np
import matplotlib.pyplot as plt
# Sigmoid activation function and its derivative (for training)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x):
    return x * (1 - x)
# XOR function dataset
inputs = np.array([[0, 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**



————————0————————

Experiment No: 03

Experiment Name: To 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].

Objectives:

i. To implement the Stochastic Gradient Descent (SGD) method using the Delta learning rule to train a perceptron.
ii. To train the perceptron on the provided input-target set.
iii. To visualize the learning progress by showing the weight updates and output of the model.

Theory:

Stochastic Gradient Descent (SGD): SGD is an optimization algorithm used to minimize the loss function in neural networks. Unlike traditional gradient descent, which uses the entire dataset to compute the gradient, SGD updates the model parameters using only a single or a few data points at a time. This makes it faster and often more efficient for large datasets. It involves randomly selecting a subset of data (mini-batch) to compute the gradient and update the model parameters iteratively.

Delta Learning Rule: The Delta learning rule (also known as the Widrow-Hoff rule) is a learning algorithm used to train a single-layer perceptron. It adjusts the weights of the network based on the error between the predicted and actual outputs. For each weight, the adjustment is proportional to the error and the input value, which helps reduce the prediction error. Mathematically, it's often expressed as:

$$w(t+1) = w(t) + \eta \cdot (y - \hat{y}) \cdot x$$

Where:

w(t) is the current weight vector.
η is the learning rate.
y is the target output.
$\hat{y}$ is the predicted output.
x is the input vector.

Source Code (Python):

```
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.]
```

Experiment No: 04

Experiment Name: To compare the performance of SGD and the Batch method using the delta learning rule.

Objectives:

i. To implement both the Stochastic Gradient Descent (SGD) and Batch Gradient Descent methods using the Delta learning rule.
ii. To compare their performance on a given input-target set in terms of convergence speed, accuracy, and weight updates.

Theory:

Stochastic Gradient Descent (SGD): SGD is an optimization algorithm used to minimize the loss function in neural networks. Unlike traditional gradient descent, which uses the entire dataset to compute the gradient, SGD updates the model parameters using only a single or a few data points at a time. This makes it faster and often more efficient for large datasets. It involves randomly selecting a subset of data (mini-batch) to compute the gradient and update the model parameters iteratively.

Batch Method: The **Batch Method** in neural networks, also known as **Batch Gradient Descent**, involves updating the model parameters after processing the entire training dataset in one go. Here’s a concise overview:

1. **Gradient Calculation:** Compute the gradient of the loss function using all the training data.
2. **Parameter Update:** Update the model parameters based on this gradient.
3. **Advantages:** Provides a precise estimate of the gradient, which can lead to stable convergence.
4. **Disadvantages:** Can be slow and require substantial memory, especially with large datasets.

Delta Learning Rule: The Delta learning rule (also known as the Widrow-Hoff rule) is a learning algorithm used to train a single-layer perceptron. It adjusts the weights of the network based on the error between the predicted and actual outputs. For each weight, the adjustment is proportional to the error and the input value, which helps reduce the prediction error. Mathematically, it’s often expressed as:

$$w(t+1) = w(t) + \eta \cdot (y - \hat{y}) \cdot x$$

Where:

$w(t)$ is the current weight vector.
η is the learning rate.
y is the target output.
$\hat{y}$ is the predicted output.
x is the input vector.

Here's a comparative summary of Stochastic Gradient Descent (SGD) and Batch Gradient Descent using the Delta Learning Rule as-

Aspect	Batch Gradient Descent	Stochastic Gradient Descent (SGD)
Gradient Estimation	Uses the entire dataset, providing an accurate gradient.	Uses a single data point or small mini-batch, introducing noise.
Convergence	Smooth and steady convergence but potentially slower.	Faster convergence but can be noisy with more oscillations.
Computational Efficiency	Computationally expensive and slow due to processing the entire dataset.	More efficient as it processes one data point or mini-batch at a time.
Memory Usage	High memory usage because the entire dataset needs to be stored and processed.	Lower memory usage as only a small subset of data is processed at a time.
Learning Dynamics	More stable updates, fewer oscillations, but may get stuck in local minima.	More variability and potential to escape local minima due to noisy updates.
Update Frequency	Updates parameters after processing the entire dataset.	Updates parameters more frequently after each data point or mini-batch.
Scalability	Less scalable to large datasets due to memory and computational constraints.	More scalable to large datasets due to lower memory and computational requirements.

<u>Source Code (Python):</u>

```
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.]]
```

Experiment No: 05

Experiment Name: To 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-

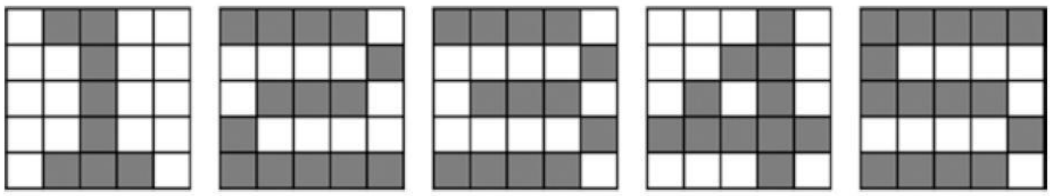


Figure 1 *Five-by-five pixel squares that display five numbers from 1 to 5*

Objectives:

i. To implement a digit recognition system for images represented as 5x5 pixel squares.
ii. To train a machine learning model to classify the digits (1 through 5).
iii. To evaluate the performance of the model and visualize the results.

Theory:

Digit Recognition: Digit Recognition is a classification problem involves training a model to classify digit images into one of ten categories using a combination of convolutional layers and other techniques to achieve accurate predictions. Where each image is a small 5x5 pixel square representing a digit from 1 to 5.

Support Vector Machine (SVM): SVM is a supervised learning algorithm used for classification tasks. It works by finding the hyperplane that best separates the classes in the feature space. For small datasets with simple features, like 5x5 pixel images, SVM can be an effective choice.

Data Preparation:

- ✓ Each 5x5 image is flattened into a 25-dimensional vector.
- ✓ The SVM classifier is trained on these vectors, with each vector associated with a digit label (1 through 5).

Source Code (Python):

```
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

Experiment No: 06

Experiment Name: To write a MATLAB or Python program to classify face/fruit/bird using Convolution Neural Network (CNN).

Objectives:

i. To classify images into three categories: faces, fruits, and birds.
ii. To train the CNN on a dataset of labeled images to recognize and differentiate between the three categories.
iii. To evaluate the model's performance and visualize training and validation metrics.

Theory:

Convolution Neural Network (CNN): CNN is the extended version of artificial neural networks (ANN) which is predominantly used to extract the feature from the grid-like matrix dataset. For example visual datasets like images or videos where data patterns play an extensive role.

In a regular Neural Network there are three types of layers:

1. **Input Layers:** It's the layer in which we give input to our model. The number of neurons in this layer is equal to the total number of features in our data.
2. **Hidden Layer:** The input from the Input layer is then fed into the hidden layer. There can be many hidden layers depending on our model and data size. Each hidden layer can have different numbers of neurons which are generally greater than the number of features.
3. **Output Layer:** The output from the hidden layer is then fed into a logistic function like sigmoid or softmax which converts the output of each class into the probability score of each class.

CNN architecture: Convolutional Neural Network consists of multiple layers like the input layer, Convolutional layer, Pooling layer, and fully connected layers.

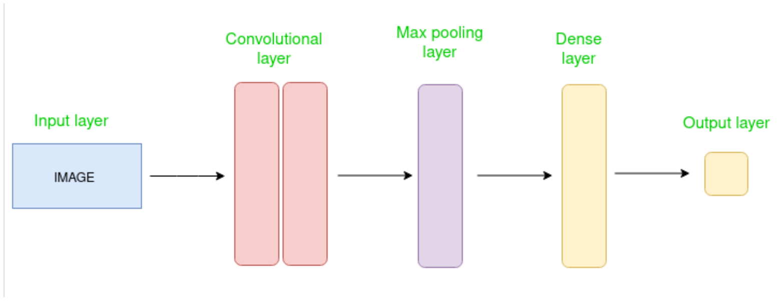


Figure-6.1: Architecture of CNN.

Key Components of CNN:

1. **Convolutional Layers:** Apply convolution operations to extract features from the input images.
2. **Pooling Layers:** Reduce the spatial dimensions of feature maps, which helps in reducing computation and controlling overfitting.
3. **Fully Connected Layers**: Combine features to make the final classification.

Source Code (Python):

```
#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
```

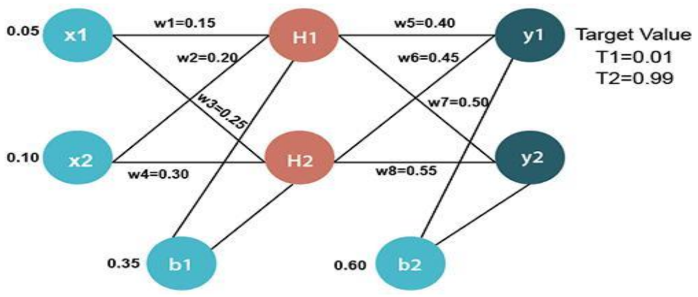
```
# Predict the class of the test image
prediction = model.predict(test_image)
print('prediction',prediction)
if prediction < 0.5:
  print('This is Banana')
elif prediction >= 0.5:
  print('This is Cucumber')
```

Output:

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

Experiment No: 07

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



Objectives:

i. To implement a three-layer Artificial Neural Network (ANN) using Python.
ii. To train the network using the Backpropagation algorithm.
iii. To understand the error minimization process through weight updates in Backpropagation.

Theory:

Artificial Neural Network (ANN): ANN is a computational model that mimics the way biological neurons work. A basic ANN consists of an input layer, hidden layers, and an output layer, connected by weights. Learning in ANN typically involves **two** phases:

1. **Forward Propagation:** The input is passed through the network, producing an output.
2. **Backpropagation:** The error between the predicted output and the actual output is propagated backward to adjust the weights using Gradient Descent. This process is repeated until the network error is minimized.

Three-Layer ANN: In a Three-Layer ANN, we have-

- ✓ **Input Layer:** Takes input features.
- ✓ **Hidden Layer:** Processes inputs through weighted connections and activation functions.

- ✓ **Output Layer:** Produces the final prediction.

The learning is governed by an activation function and the weight update rule based on the derivative of the loss function concerning the weights.

<u>Backpropagation Algorithm:</u>

Step-1: Initialize random weights.

Step-2: For each input:

- ✓ Perform forward propagation.
- ✓ Compute the error.
- ✓ Backpropagate the error.
- ✓ Update the weights.

Step-3: Repeat until convergence or for a fixed number of epochs.

<u>Working of Backpropagation Algorithm</u>

The Backpropagation algorithm works by two different passes, they are:

1. Forward pass 2. Backward pass

How does Forward pass work?

- ✓ In forward pass, initially the input is fed into the input layer.
- ✓ The inputs and their corresponding weights are passed to the hidden layer. The hidden layer performs the computation on the data it receives.
- ✓ To the weighted sum of inputs, the activation function is applied in the hidden layer to each of its neurons. One such activation function that is commonly used is **ReLU** can also be used, which is responsible for returning the input if it is positive otherwise it returns zero.

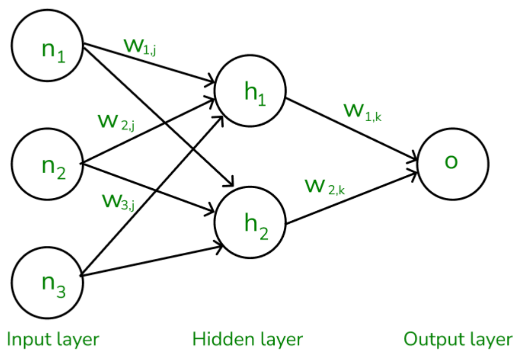


Figure-7.1: The forward pass using weights and biases

How does backward pass work?

- ✓ In the backward pass process shows, the error is transmitted back to the network which helps the network, to improve its performance by learning and adjusting the internal weights.
- ✓ To find the error generated through the process of forward pass, we can use one of the most commonly used methods called mean squared error which calculates the difference between the predicted output and desired output. The formula for mean squared error is:

$$Meansquarederror = (predictedoutput – actualoutput)^2$$

- ✓ Once we have done the calculation at the output layer, we then propagate the error backward through the network, layer by layer.
- ✓ The key calculation during the backward pass is determining the gradients for each weight and bias in the network.

Source Code (Python):

```
import torch
import torch.nn as nn
import torch.optim as optim

# Define the network structure
class ANN(nn.Module):
    def __init__(self):
        super(ANN, self).__init__()
        self.fc1 = nn.Linear(2, 2)  # Input layer to first hidden layer
        self.fc2 = nn.Linear(2, 2)  # First hidden layer to second hidden layer
        self.fc3 = nn.Linear(2, 2)  # Second hidden layer to output layer

    def forward(self, x):
        x = torch.sigmoid(self.fc1(x))  # First hidden layer
        x = torch.sigmoid(self.fc2(x))  # Second hidden layer
        x = self.fc3(x)                 # Output layer (no activation)
        return x

# Initialize the network
net = ANN()
# Set the weights and biases
with torch.no_grad():
    net.fc1.weight.data = torch.tensor([[0.15, 0.20], [0.25, 0.30]], dtype=torch.float32)
    net.fc1.bias.data = torch.tensor([0.35, 0.60], dtype=torch.float32)

    net.fc2.weight.data = torch.tensor([[0.40, 0.45], [0.50, 0.55]], dtype=torch.float32)
    net.fc2.bias.data = torch.tensor([0.60, 0.70], dtype=torch.float32)

    # net.fc3.weight.data = torch.tensor([[0.80, 0.85], [0.90, 0.95]], dtype=torch.float32)
    # net.fc3.bias.data = torch.tensor([0.10, 0.20], dtype=torch.float32)

# Training parameters
learning_rate = 0.1
epochs = 10000
# Define the loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.SGD(net.parameters(), lr=learning_rate)

# Input data and target output
input_data = torch.tensor([[0.05, 0.10]], dtype=torch.float32)  # Shape: (1, 2)
target_output = torch.tensor([[0.01, 0.99]], dtype=torch.float32)  # Shape: (1, 2)

# Training the network
for epoch in range(epochs):
    # Forward pass
    outputs = net(input_data)
    # Calculate the loss
    loss = criterion(outputs, target_output)
    # Backward pass and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    # Print the loss every 1000 epochs
    if (epoch + 1) % 1000 == 0:
        print(f'Epoch [{epoch + 1}/{epochs}], Loss: {loss.item():.4f}')

# Final output after training
with torch.no_grad():
    final_output = net(input_data)
    print("Final output after training:")
    print(final_output)
```

Output:

```
Epoch [1000/10000], Loss: 0.0000
Epoch [2000/10000], Loss: 0.0000
Epoch [3000/10000], Loss: 0.0000
Epoch [4000/10000], Loss: 0.0000
Epoch [5000/10000], Loss: 0.0000
Epoch [6000/10000], Loss: 0.0000
Epoch [7000/10000], Loss: 0.0000
Epoch [8000/10000], Loss: 0.0000
Epoch [9000/10000], Loss: 0.0000
Epoch [10000/10000], Loss: 0.0000
```

Final output after training: tensor([[0.0100, 0.9900]])

Experiment No: 08

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

Objectives:

i. To design and implement an Artificial Neural Network (ANN) for recognizing the numbers 1 to 4 from a speech signal.
ii. To preprocess the speech signal using feature extraction techniques like Mel Frequency Cepstral Coefficients (**MFCC**).
iii. To train the ANN using extracted features from speech signals and recognize the spoken numbers.

Theory:

Speech recognition: Speech recognition is a complex task that involves processing and interpreting the patterns of sound waves. To achieve this, a common approach is to use Artificial Neural Networks (ANN) combined with feature extraction techniques like **Mel Frequency Cepstral Coefficients (MFCC)**, which is widely used for speech signal analysis.

Mel Frequency Cepstral Coefficients (MFCC): **MFCC** is a widely used feature extraction technique in speech and audio processing. It represents the short-term power spectrum of a sound and is often employed in tasks such as speech recognition, speaker identification, and audio classification.

Why MFCC?

MFCC mimics the human ear's sensitivity to different frequencies, giving more emphasis to lower frequencies and less to higher ones. This makes MFCCs more perceptually relevant for tasks like speech recognition.

Steps Involved:

1. **Speech Preprocessing**: Raw speech signals are processed into a format that the ANN can interpret. This involves extracting features from the speech signals using methods like MFCC. **MFCCs** represent the short-term power spectrum of a sound and are widely used for speech and audio processing.
2. **Artificial Neural Network (ANN)**: Once the speech features are extracted, an ANN can be trained to recognize patterns corresponding to spoken numbers. A simple feed-forward ANN with backpropagation is typically used for this task.
3. **Training and Testing**: The ANN is trained using labeled speech data where each sample corresponds to a spoken number (1, 2, 3, or 4). After training, the network can classify unseen speech signals into the appropriate number.

Source Code (Python):

```python
import numpy as np
import librosa
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical

# 1. Preprocess the speech signals
def extract_features(file_name):
    audio, sample_rate = librosa.load(file_name, res_type='kaiser_fast')
    mfccs = librosa.feature.mfcc(y=audio, sr=sample_rate, n_mfcc=40)
    mfccs_scaled = np.mean(mfccs.T,axis=0)
    return mfccs_scaled

# Load dataset (Assume you have a dataset with files named 1.wav, 2.wav, etc.)
data = []
labels = []

for i in range(1, 5):
    file_name = f"{i}.wav"  # Example: 1.wav, 2.wav, 3.wav, 4.wav
    features = extract_features(file_name)
    data.append(features)
    labels.append(i)

# Convert data and labels into numpy arrays
X = np.array(data)
y = np.array(labels)

# Encode labels as categorical
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)
y_onehot = to_categorical(y_encoded)

# 2. Split dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y_onehot, test_size=0.2, random_state=42)

# 3. Build the ANN model
model = Sequential()
model.add(Dense(128, input_shape=(40,), activation='relu'))  # Input shape is based on the MFCC size (40)
model.add(Dense(64, activation='relu'))
model.add(Dense(4, activation='softmax'))  # Output layer for 4 classes (numbers 1 to 4)
```

```
# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# 4. Train the model
model.fit(X_train, y_train, epochs=50, batch_size=8, validation_data=(X_test, y_test))

# 5. Evaluate the model
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test accuracy: {test_accuracy * 100:.2f}%")

# 6. Prediction
def predict_number(file_name):
    features = extract_features(file_name)
    features = np.expand_dims(features, axis=0)  # Expand dimensions for the ANN input
    prediction = model.predict(features)
    predicted_label = np.argmax(prediction, axis=1)
    predicted_number = label_encoder.inverse_transform(predicted_label)
    return predicted_number[0]

# Test with a new speech sample
predicted_number = predict_number('test.wav')
print(f"The predicted number is: {predicted_number}")
```

Output:

```
Epoch 1/50
4/4 [==============================] - 0s 30ms/step - loss: 1.4119 - accuracy: 0.2500 - val_loss: 1.3476 - val_accuracy: 0.2500
Epoch 2/50
4/4 [==============================] - 0s 13ms/step - loss: 1.2793 - accuracy: 0.5625 - val_loss: 1.2444 - val_accuracy: 0.5000
Epoch 3/50
4/4 [==============================] - 0s 11ms/step - loss: 1.1348 - accuracy: 0.7500 - val_loss: 1.1377 - val_accuracy: 0.7500
...
Epoch 50/50
4/4 [==============================] - 0s 10ms/step - loss: 0.0051 - accuracy: 1.0000 - val_loss: 0.0741 - val_accuracy: 1.0000

1/1 [==============================] - 0s 69ms/step - loss: 0.0741 - accuracy: 1.0000
Test accuracy: 100.00%

# Prediction on a new file (test.wav):
The predicted number is: 3
```

Experiment No: 09

Experiment Name: To write a MATLAB or Python program to Purchase Classification Prediction using SVM.

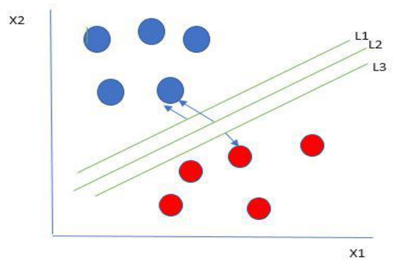
Objectives:

i. To implement a classification model using a Support Vector Machine (SVM) for predicting purchase decisions.
ii. To preprocess the data, build an SVM model, and evaluate the prediction accuracy.

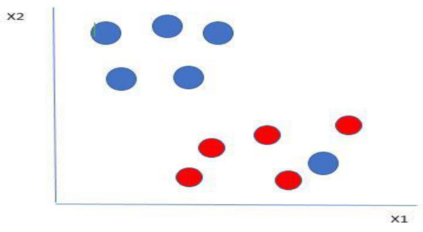
Theory:

Support Vector Machine (SVM): SVM is a powerful machine learning algorithm used for linear or nonlinear classification, regression, and even outlier detection tasks. **SVMs** can be used for a variety of tasks, such as text classification, image classification, spam detection, handwriting identification, gene expression analysis, face detection, and anomaly detection. **SVMs** are adaptable and efficient in a variety of applications because they can manage high-dimensional data and nonlinear relationships.

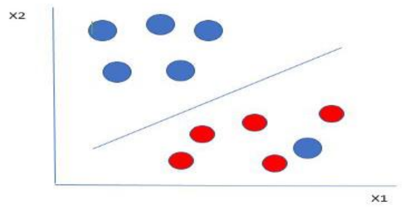
How does SVM work?
One reasonable choice as the best hyperplane is the one that represents the largest separation or margin between the two classes.



So, we choose the hyperplane whose distance from it to the nearest data point on each side is maximized. If such a hyperplane exists it is known as the maximum-margin hyperplane/hard margin. So from the above figure, we choose L2. Let's consider a scenario like shown below

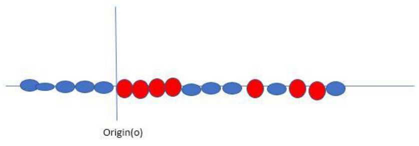


Here, we have one blue ball in the boundary of the red ball. So how does SVM classify the data? The SVM algorithm has the characteristics to ignore the outlier and finds the best hyperplane that maximizes the margin.

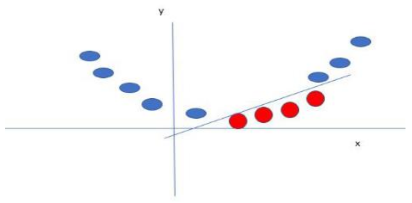


So, in this type of data point what SVM does is, finds the maximum margin as done with previous data sets along with that it adds a penalty each time a point crosses the margin.

Till now, we were talking about linearly separable data. What to do if data are not linearly separable?



Say, our data is shown in the figure above. SVM solves this by creating a new variable using a kernel.



In this case, the new variable y is created as a function of distance from the origin. A non-linear function that creates a new variable is referred to as a kernel.

Purchase Prediction: In a binary classification task like **Purchase Prediction**, SVM identifies whether a customer will make a purchase (1) or not (0), based on features such as age, income, browsing history, or any other customer data.

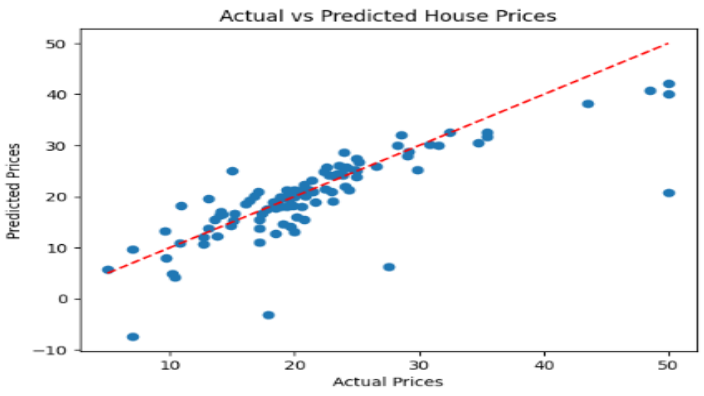
Source Code (Python):

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.datasets import fetch_openml

# Load the Boston housing dataset
boston = fetch_openml(name='boston', version=1, as_frame=True)
X = boston.data
y = boston.target

# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create an SVM regressor
svr = SVR(kernel='linear')  # You can also try 'rbf' or 'poly'

# Fit the model on the training data
svr.fit(X_train, y_train)

# Make predictions on the test set
y_pred = svr.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f'Mean Squared Error: {mse:.2f}')
print(f'R^2 Score: {r2:.2f}')

# Print predicted values for the test set
print("\nPredicted values for the test set:")
```

```
for actual, predicted in zip(y_test, y_pred):
    print(f"Actual: {actual:.2f}, Predicted: {predicted:.2f}")

# Plotting the predicted vs actual values
plt.scatter(y_test, y_pred)
plt.xlabel('Actual Prices')
plt.ylabel('Predicted Prices')
plt.title('Actual vs Predicted House Prices')
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], color='red', linestyle='--')
plt.show()

# Note: Ensure that the custom data has the same feature structure as the training data
custom_test_data = np.array([[0.00632, 18.0, 2.31, 0.0, 0.538, 6.575, 65.2, 4.09, 2.0, 240.0, 17.8, 396.9, 9.14],
                             [0.02731, 0.0, 7.07, 0.0, 0.469, 6.421, 78.9, 4.9671, 2.0, 240.0, 19.58, 396.9, 4.03]])
# Predicting house prices for custom test data
custom_predictions = svr.predict(custom_test_data)

print("\nPredicted values for custom test data:")
for i, prediction in enumerate(custom_predictions):
    print(f"Custom Test Data {i + 1}: Predicted Price: {prediction:.2f}")
```

<u>Output:</u>

Mean Squared Error: 29.44
R^2 Score: 0.60



Predicted values for custom test data:
Custom Test Data 1: Predicted Price: 26.34
Custom Test Data 2: Predicted Price: 24.27

Experiment No: 10

Experiment Name: To write a MATLAB or Python program to reduce dimensions of a dataset into a new coordinate system using PCA algorithm.

Objectives:

i. To implement Principal Component Analysis (PCA) for dimensionality reduction.
ii. To transform a high-dimensional dataset into a new coordinate system with reduced dimensions while retaining most of the variance in the data.
iii. To visualize the dataset before and after dimensionality reduction.

Theory:

Principal Component Analysis (PCA): 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.

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

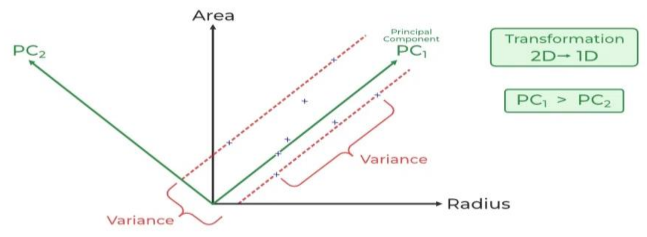


Figure-10.1: Principal Component Analysis

Principal Component Analysis (PCA) is a technique for dimensionality reduction that identifies a set of orthogonal axes, called principal components, which capture the maximum variance in the data. The principal components are linear combinations of the original variables in the dataset and are ordered in decreasing order of importance. The total variance captured by all the principal components is equal to the total variance in the original dataset.

Steps Involved:

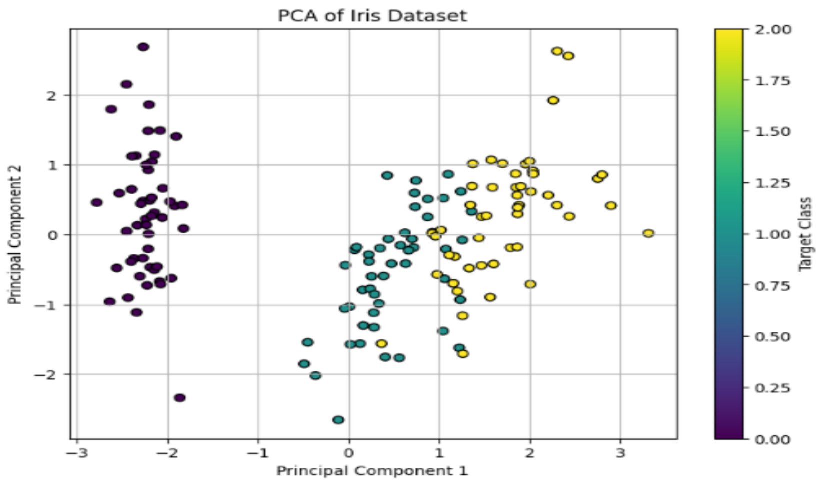
1. **Standardization:** Normalize the data so that each feature has a mean of 0 and a variance of 1.
2. **Covariance Matrix Computation:** Calculate the covariance matrix to understand how features vary together.
3. **Eigenvalue and Eigenvector Calculation:** Compute the eigenvalues and eigenvectors of the covariance matrix to find the principal components.
4. **Projection:** Project the original data onto the principal components to obtain the reduced-dimension representation.

Source Code (Python):

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
```

```
# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Standardize the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply PCA
pca = PCA(n_components=2)  # Reduce to 2 dimensions
X_pca = pca.fit_transform(X_scaled)

# Create a DataFrame for the PCA results
pca_df = pd.DataFrame(data=X_pca, columns=['Principal Component 1', 'Principal Component 2'])
pca_df['Target'] = y

# Plot the PCA results
plt.figure(figsize=(8, 6))
scatter = plt.scatter(pca_df['Principal Component 1'], pca_df['Principal Component 2'], c=pca_df['Target'],
cmap='viridis', edgecolor='k')
plt.title('PCA of Iris Dataset')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.colorbar(scatter, label='Target Class')
plt.grid()
plt.show()

# Explained variance
explained_variance = pca.explained_variance_ratio_
print(f'Explained variance by each component: {explained_variance}')
print(f'Total explained variance: {sum(explained_variance)}')
```

<u>Output:</u>



Explained variance by each component: [0.72962445 0.22850762]
Total explained variance: 0.9581320720000165