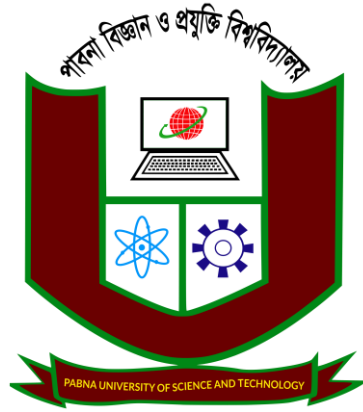


PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY



FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF INFORMATION AND COMMUNICATION ENGINEERING

Course Code: ICE-4204

Course Title: Neural Networks Sessional

Lab Report

SUBMITTED BY:

Md.Ashikur Rahman

Roll: 200607

Session: 2019-20

4th Year 2nd Semester

Department of Information and
Communication Engineering,

Pabna University of Science and
Technology, Pabna.

SUBMITTED TO:

Dr. Md. Imran Hossain

Associate Professor

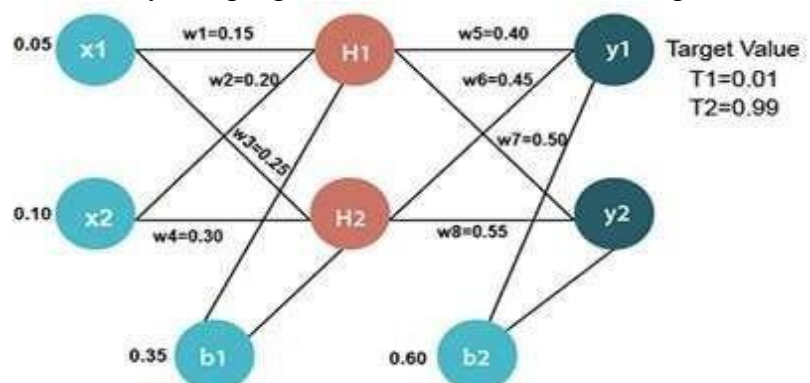
Department of Information and
Communication Engineering (ICE),

Pabna University of Science and
Technology, Pabna.

Submission Date:

SIGNATURE

INDEX

P. No.	Experiment Name
01	Write a program to evaluate using perceptron net for AND function with bipolar inputs and targets and also show the convergence curves and the decision boundary lines.
02	Implement the SGD Method using Delta learning rule for following input-target sets. $X_{Input} = [0\ 0\ 1; 0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 1]$, $D_{Target} = [0; 0; 1; 1]$
03	Write a program to evaluate XOR function and also show the convergence curves and the decision boundary.
04	Write a program to evaluate a simple feedforward neural network for classifying handwritten digits using the MNIST dataset.
05	Write a MATLAB or Python program to classify face/fruit/bird using Convolution Neural Network (CNN).
06	<p>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.</p> 
07	Write a program to evaluate a Recurrent Neural Network (RNN) for text Classification.
08	Write a MATLAB or Python program to Purchase Classification Prediction using SVM.

Problem No: 01

Problem Name: Write a program to evaluate using perceptron net for AND function with bipolar inputs and targets and also show the convergence curves and the decision boundary lines.

Theory:

Perceptron for AND Function: A **perceptron** is the simplest form of a neural network model. It consists of a single layer of output nodes connected to inputs. Each output node represents a single binary output.

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 linearly separable patterns.

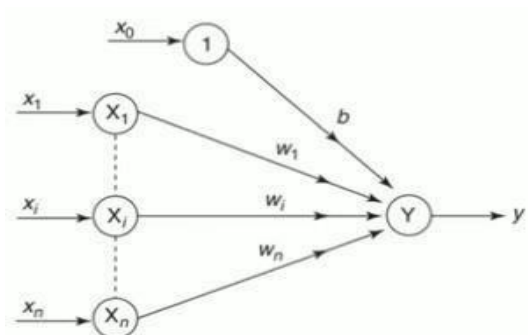
Types of Perceptron

- **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.

Bipolar Inputs and Targets: Bipolar values mean that inputs and outputs are either -1 or +1 instead of the standard 0 and 1 used in binary representation. For the AND function with two inputs, the input combinations and their corresponding outputs are:

Input x_1	Input x_2	Target t
+1	+1	+1
+1	-1	-1
-1	+1	-1
-1	-1	-1

Perceptron Learning Rule:



Calculated input y_{in} is:

$$y_{in} = b + \sum_{i=1}^n x_i \omega_i$$

Activation function:

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > \theta \\ 0 & \text{if } -\theta \leq y_{in} \leq \theta \\ -1 & \text{if } y_{in} < -\theta \end{cases}$$

The perceptron updates its weights w to minimize the error between the predicted output and the actual output. The weight update rule is given by:

$$\omega(t+1) = \omega(t) + \eta \cdot (t - y) \cdot x$$

where:

$\omega(t)$ is the weight vector at

time t η is the learning rate. t is

the target output.

y is the predicted output.

x is the input vector.

Convergence Curve: The convergence curve in the context of machine learning, particularly in training a perceptron, represents how the error (or loss) of the model changes over time (i.e., over the training epochs). It visually shows the process of learning and how quickly the model is converging to the correct solution.

Decision Boundary: A decision boundary is a concept in machine learning that represents the region of a problem space where the output label of a classifier changes. It is a line, curve, or surface (in higher dimensions) that separates different classes in a classification problem. The decision boundary for the perceptron is a line in the input space that separates classes where the output is +1 or -1.

Source Code:

```
import numpy as np
import matplotlib.pyplot as plt
# Bipolar activation function
def bipolar_activation(x):
    return 1 if x >= 0 else -1
# Perceptron training function
def perceptron_train(inputs, targets, learning_rate=0.1, max_epochs=100):
    num_inputs = inputs.shape[1]
    num_samples = inputs.shape[0]
    # Initialize weights and bias
    weights = np.random.randn(num_inputs)
    bias = np.random.randn()
    convergence_curve = []
    for epoch in range(max_epochs):
        misclassified = 0
        for i in range(num_samples):
            net_input = np.dot(inputs[i], weights) + bias
            predicted = bipolar_activation(net_input)

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

        accuracy = (num_samples - misclassified) / num_samples
        convergence_curve.append(accuracy)
        if misclassified == 0:
            print("Converged in {} epochs.".format(epoch + 1))
            break
    return weights, bias, convergence_curve
# Main function
if __name__ == "__main__":
    # Input and target data (bipolar representation)
    inputs = np.array([[-1, -1], [-1, 1], [1, -1], [1, 1]])
    targets = np.array([-1, -1, -1, 1])
    # Training the perceptron
    weights, bias, convergence_curve = perceptron_train(inputs, targets)
    # Decision boundary line
    x = np.linspace(-2, 2, 100)
    y = (-weights[0] * x - bias) / weights[1]
    # print(y)
    # Plot convergence curve
    plt.figure(figsize=(8, 4))
    plt.plot(range(1, len(convergence_curve) + 1), convergence_curve)
    plt.xlabel('Epoch')
```

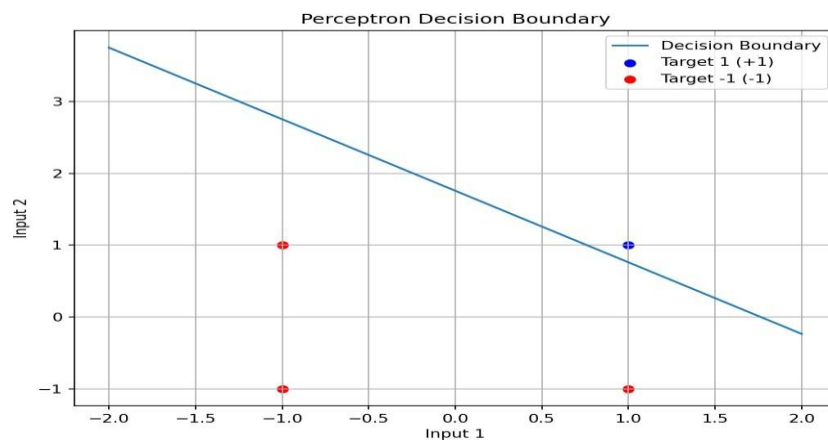
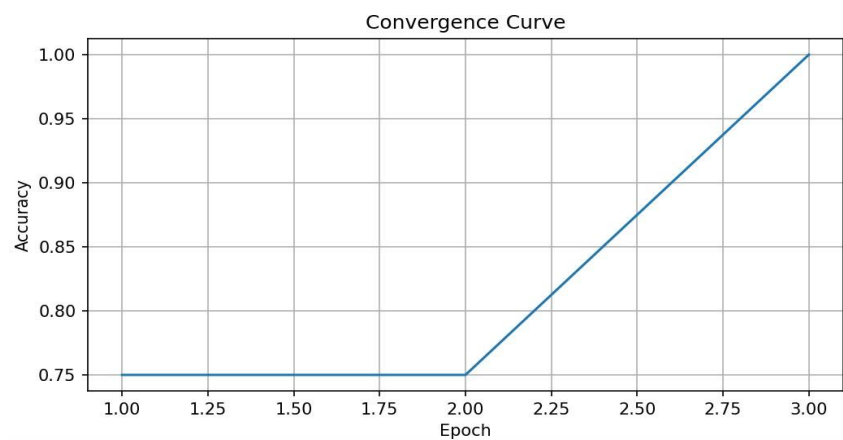
```

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.xlabel('Input 1')
plt.ylabel('Input 2')
plt.title('Perceptron Decision Boundary')
plt.legend()
plt.grid()
plt.show()

```

Output:

Converged in 3 epochs.



Problem No: 02

Problem Name: Implement the SGD Method using Delta learning rule for following input-target sets.

$X_{Input} = [0\ 0\ 1; 0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 1]$, $D_{Target} = [0; 0; 1; 1]$.

Theory: To implement the Stochastic Gradient Descent (SGD) method using the Delta learning rule, we need to build a simple perceptron model that updates its weights iteratively based on the error between the predicted output and the target output.

Delta Learning Rule: The Delta learning rule, also known as the Widrow-Hoff rule or Least Mean Squares (LMS) rule, is a supervised learning rule used for adjusting the weights of a single-layer neural network. The rule minimizes the difference between the desired output and the predicted output by making incremental adjustments to the weights.

The weight update formula using the Delta rule is given by:

$$\Delta\omega_i = \eta \cdot (D - Y) \cdot x_i$$

where:

- $\Delta\omega_i$ is the change in the weight w_i .
- η is the learning rate, a small positive constant that determines the step size.
- D is the desired target output.
- Y is the predicted output of the perceptron.
- x_i is the input feature corresponding to the weight w_i .

The weights are updated iteratively as:

$$\omega_i = \omega_i + \Delta\omega_i$$

Stochastic Gradient Descent (SGD): Stochastic Gradient Descent (SGD) is an optimization algorithm commonly used in training machine learning models, especially neural networks. Unlike Batch Gradient Descent, which computes the gradient of the entire dataset, SGD updates the model weights for each training example, making it faster and more efficient for large datasets.

Steps in SGD:

1. Initialize weights randomly.
2. For each training sample:
 - Compute the output of the perceptron using the current weights. ○
Calculate the error (difference between the predicted output and the actual target).

- Update the weights using the Delta rule.

3. Repeat the process for a specified number of epochs or until convergence.

SGD is particularly useful when the dataset is large, as it reduces computation time by updating weights based on a single sample or a small batch of samples.

Activation Functions: An activation function in a perceptron decides whether a neuron should be activated or not by calculating a weighted sum and applying a non-linear transformation. Common activation functions include:

- **Step Function:** Produces binary output based on a threshold. It is often used in simple perceptrons.

$$f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

- **Sigmoid Function:** Produces an output between 0 and 1, making it suitable for probabilities and differentiable, which is useful for backpropagation in more complex neural networks.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Source Code:

```
import numpy as np
import matplotlib.pyplot as plt
# Input and target values
Xinput = np.array([[0, 0, 1], # Bias included in the third column [0, 1,
                    1],
                  [1, 0, 1],
                  [1, 1, 1]])
Dtarget = np.array([0, 0, 1, 1]) # Target output
# Initialize weights randomly weights =
np.random.randn(3)
learning_rate = 0.1 epochs =
100
# Activation function (Step function) def
step_function(x):
    return np.where(x >= 0, 1, 0)
# Training using SGD and Delta learning rule
convergence_curve = []
converged = False for
epoch in range(epochs):
    total_error = 0

    # Shuffle the data for each epoch (stochastic gradient descent)
    indices = np.random.permutation(len(Xinput))
```



```

Xinput_shuffled = Xinput[indices]
Dtarget_shuffled = Dtarget[indices]
for i in range(len(Xinput)): x =
Xinput_shuffled[i]          d =
Dtarget_shuffled[i]
    # Forward pass (calculate output) y =
    step_function(np.dot(x, weights))

    # Calculate error error
    = d - y
    total_error += abs(error)
    # Delta rule: weight update weights
    += learning_rate * error * x
    convergence_curve.append(total_error)

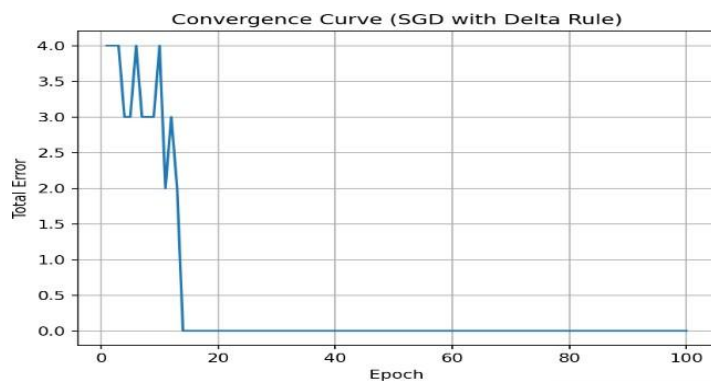
# Stop early if there is no error if total_error ==
0 and converged == False: print(f"Converged
in {epoch + 1} epochs.")
    converged = True #
Print final weights
print("Final weights:", weights) #
Plot convergence curve
plt.plot(range(1, len(convergence_curve) + 1), convergence_curve)
plt.xlabel('Epoch') plt.ylabel('Total Error')
plt.title('Convergence Curve
(SGD with Delta Rule)') plt.grid()
plt.show()

```

Output:

Converged in 14 epochs.

Final weights: [0.1209001 -0.04639043 -0.01045075]



Problem No: 03

Problem Name: Write a program to evaluate XOR function and also show the convergence curves and the decision boundary.

Theory:

McCulloch-Pitts Neuron Model: The McCulloch-Pitts neuron, proposed in 1943 by Warren McCulloch and Walter Pitts, is a simple model of a biological neuron. It is a binary threshold unit that outputs either 0 or 1 based on the weighted sum of its inputs and a bias term. The model is defined as follows:

- **Inputs:** $x_1, x_2, x_3, \dots, x_n$ (binary values, either 0 or 1)
- **Weights:** $w_1, w_2, w_3, \dots, w_n$ (real-valued weights associated with each input)
- **Bias:** θ (threshold value)

The neuron computes the weighted sum of its inputs:

$$y = \sum_{i=1}^n \omega_i x_i - \theta$$

The output Y of the neuron is determined by applying a step function (also called a threshold or activation function):

$$= f(y) = \begin{cases} 1 & \text{if } y \geq 0 \\ 0 & \text{if } y < 0 \end{cases}$$

Limitations of McCulloch-Pitts Neurons for XOR: The McCulloch-Pitts neuron is a **linear classifier**. It can represent logical operations like AND, OR, and NOT, which are linearly separable. However, the XOR (exclusive OR) function is **not linearly separable**. This means that a single-layer network of McCulloch-Pitts neurons cannot solve the XOR problem.

XOR Truth Table:

Input x_1	Input x_2	Target t
0	0	0
0	1	1
1	0	1
1	1	0

Solving XOR Using a Multi-Layer Neural Network: To solve the XOR problem, we need a **multi-layer neural network** that introduces non-linearity. In a multi-layer network, the hidden layer can transform the input space into a new space where the XOR function becomes linearly separable.

Architecture for XOR using McCulloch-Pitts Neurons:

The simplest neural network architecture to solve the XOR problem consists of:

- **Input Layer:** Two neurons, corresponding to the two inputs x_1 and x_2
- **Hidden Layer:** Two neurons to learn intermediate representations.
- **Output Layer:** One neuron to produce the final output.

Step-by-Step Logic:

1. Hidden Layer Neurons:

- One neuron computes the AND function of inputs.
- Another neuron computes the OR function of inputs.

2. Output Layer Neuron:

- The output neuron computes a logical function that combines the outputs of the hidden layer neurons. In the XOR case, this can be achieved by combining the outputs using a logical AND with negation or OR functions.

The hidden layer adds non-linearity to the model, enabling the XOR function to be learned.

Network Equations:

1. Hidden Layer Outputs:

$$h_1 = AND(x_1, x_2) = x_1 \wedge x_2$$

$$h_2 = OR(x_1, x_2) = x_1 \vee x_2$$

2. Output Layer Output:

$$o = AND(NOT(h_1), h_2) = \neg(x_1 \wedge x_2) \wedge (x_1 \vee x_2)$$

This logic creates a multi-layer neural network capable of computing the XOR function.

Convergence Curves and Decision Boundary:

- **Convergence Curves:** These show how the predicted output values approach the true XOR output values during training. For the McCulloch-Pitts model, weights are typically fixed, so convergence curves aren't as relevant unless you consider a dynamic learning scenario.
- **Decision Boundary:** The decision boundary separates the input space into regions where the output is either 0 or 1. For the XOR function, the decision boundary is non-linear. Visualizing it shows two distinct regions in the input space where the function outputs 1, separated by a region where it outputs 0.

Source Code:

```
import numpy as np
import matplotlib.pyplot as plt
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x):
    return x * (1 - x)

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)):
        # 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
    err = 1 - (len(inputs) - misclassified) / len(inputs)
    convergence_curve.append(err)
    if misclassified == 0:
        print("Converged in {} epochs.".format(epoch + 1)) break
# Plot convergence curve
plt.figure(figsize=(8, 4))
plt.plot(range(1, len(convergence_curve) + 1), convergence_curve)
plt.xlabel('Epoch')
plt.ylabel('Error')
```

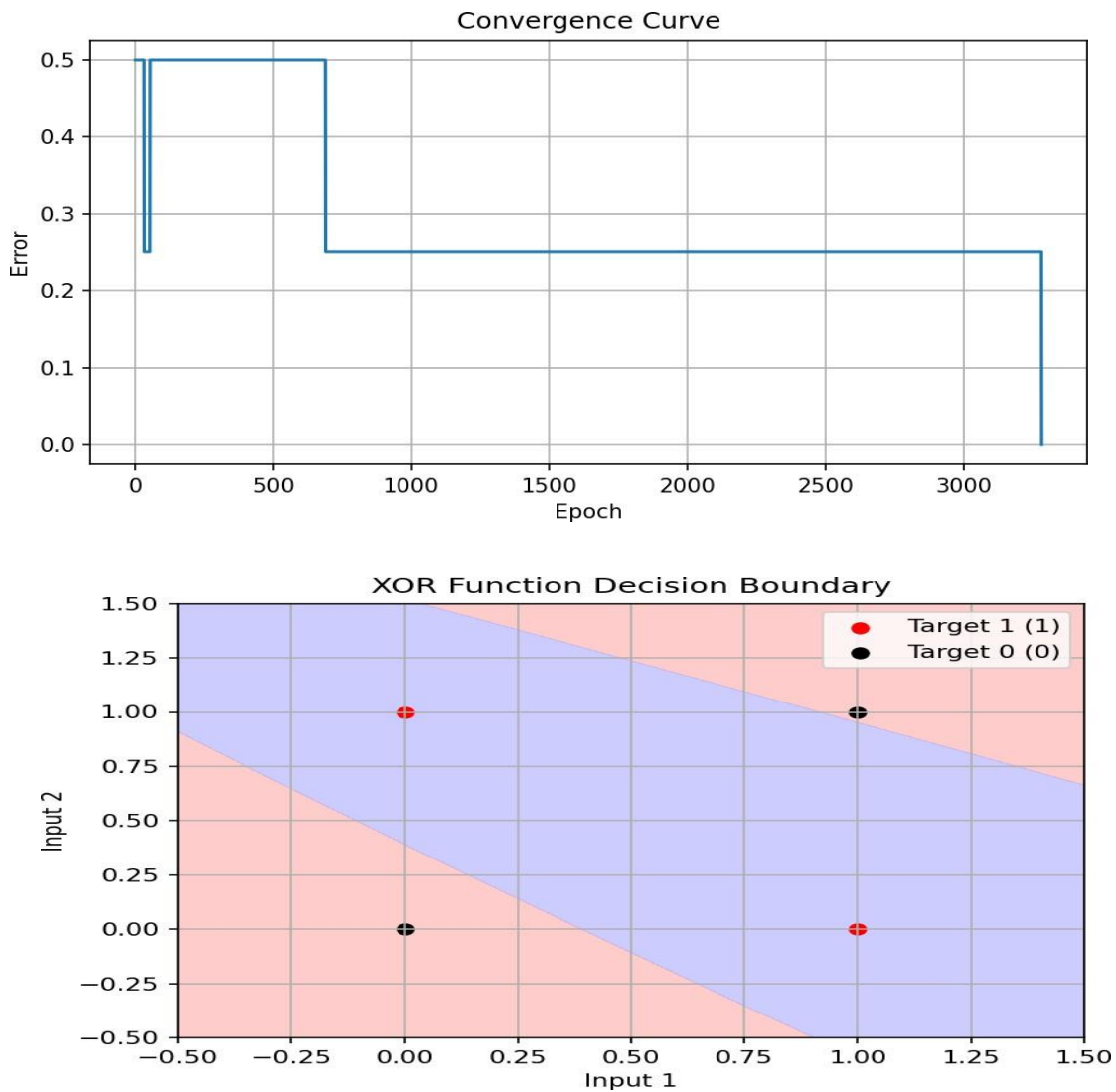
```

plt.title('Convergence Curve') plt.grid() plt.show()
# Create a grid of points to plot decision boundaries x1 = np.linspace(-0.5, 1.5, 200)
x2 = np.linspace(-0.5, 1.5, 200)
X1, X2 = np.meshgrid(x1, x2)
Z = predict(X1, X2)
# Plot decision boundaries plt.figure(figsize=(8, 6))
plt.xlabel('Input 1') plt.ylabel('Input 2')
plt.title('XOR Function Decision Boundary') plt.legend() plt.grid()
plt.show()

```

Output:

Generated XOR Result is: [0 1 1 0]



Experiment No: 04

Experiment Name: Write a program to evaluate a simple feedforward neural network for classifying handwritten digits using the MNIST dataset.

Objectives:

- To understand the architecture and principles of Generative Adversarial Networks (GANs).
- To implement a GAN model for image generation.
- To train the GAN effectively.
- To evaluate the quality of generated images.
- To investigate the impact of hyperparameters.
- To demonstrate the capability of GANs in unsupervised learning.

Theory:

A **generative adversarial network (GAN)** is a machine learning (ML) model in which two neural networks compete by using deep learning methods to become more accurate in their predictions. GANs typically run unsupervised and use a cooperative zero-sum game framework to learn.

A generative adversarial network (GAN) has two parts:

1. Generator
2. Discriminator

The **generator** learns to generate plausible data. The generated instances become negative training examples for the discriminator.

The **discriminator** learns to distinguish the generator's fake data from real data. The discriminator penalizes the generator for producing implausible results.

When training begins, the generator produces obviously fake data, and the discriminator quickly learns to tell that it's fake



As training progresses, the generator gets closer to producing output that can fool the discriminator:



Finally, if generator training goes well, the discriminator gets worse at telling the difference between real and fake. It starts to classify fake data as real, and its accuracy decreases.

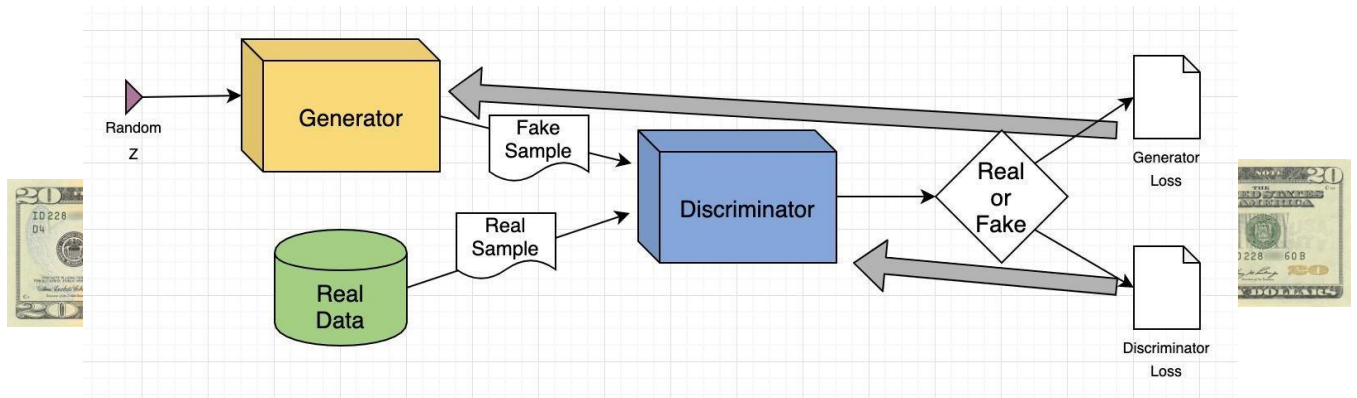


Fig. 1: General Block Diagram of Generative Adversarial Network (GAN)

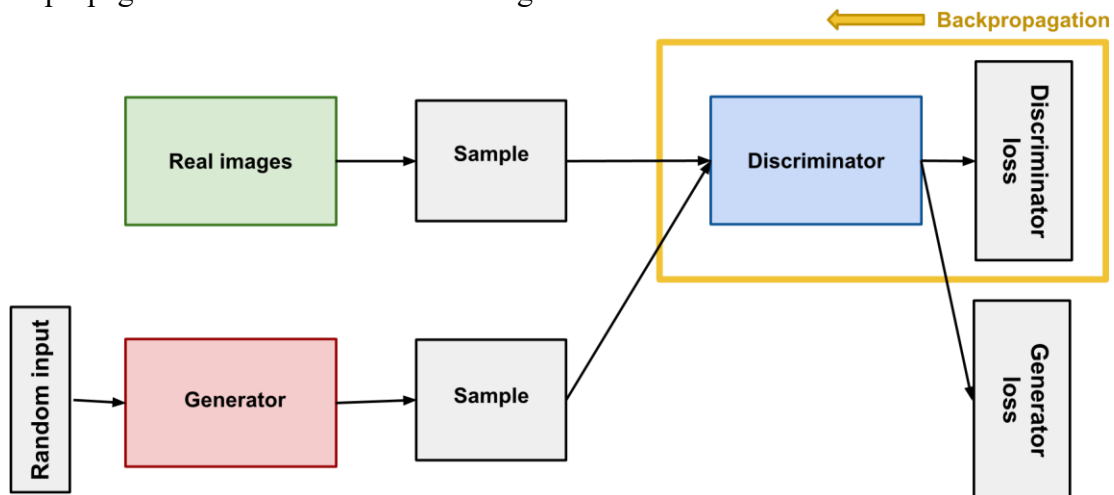
Both the generator and the discriminator are neural networks. The generator output is connected directly to the discriminator input. Through backpropagation, the discriminator's classification provides a signal that the generator uses to update its weights.

Let's explain the pieces of this system in greater detail:

The Discriminator:

The discriminator in a GAN is simply a classifier. It tries to distinguish real data from the data created by the generator. It could use any network architecture appropriate to the type of data it's classifying.

Fig.2: Backpropagation in Discriminator Training



The discriminator's training data comes from two sources:

Real data instances, such as real pictures of people. The discriminator uses these instances as positive examples during training.

Fake data instances created by the generator. The discriminator uses these instances as negative examples during training.

In Figure 2, the two "Sample" boxes represent these two data sources feeding into the discriminator. During discriminator training, the generator does not train. Its weights remain constant while it produces examples for the discriminator to train on.

Training the Discriminator

The discriminator connects to two loss functions. During discriminator training, the discriminator ignores the generator loss and just uses the discriminator loss. We use the generator loss during generator training, as described in the next section.

During discriminator training, the discriminator classifies both real data and fake data from the generator. The discriminator loss penalizes the discriminator for misclassifying a real instance as fake or a fake instance as real. The discriminator updates its weights through backpropagation from the discriminator loss through the discriminator network.

The Generator:

The generator part of a GAN learns to create fake data by incorporating feedback from the discriminator. It learns to make the discriminator classify its output as real.

Generator training requires tighter integration between the generator and the discriminator than discriminator training requires. The portion of the GAN that trains the generator includes:

- random input
- generator network, which transforms the random input into a data instance
- discriminator network, which classifies the generated data
- discriminator output
- generator loss, which penalizes the generator for failing to fool the discriminator

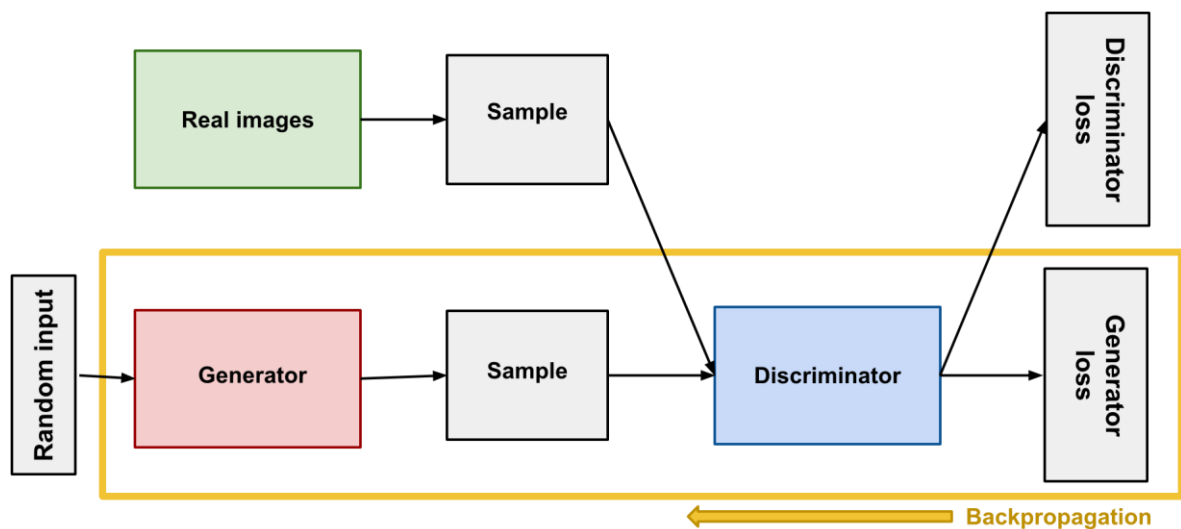


Fig.3: Backpropagation in Generator Training

Random Input

Neural networks need some form of input. Normally we input data that we want to do something with, like an instance that we want to classify or make a prediction about. But what do we use as input for a network that outputs entirely new data instances?

In its most basic form, a GAN takes random noise as its input. The generator then transforms this noise into a meaningful output. By introducing noise, we can get the GAN to produce a wide variety of data, sampling from different places in the target distribution.

Experiments suggest that the distribution of the noise doesn't matter much, so we can choose something that's easy to sample from, like a uniform distribution. For convenience the space from which the noise is sampled is usually of smaller dimension than the dimensionality of the output space.

Using the Discriminator to Train the Generator

To train a neural network, we alter the network's weights to reduce the error or loss of its output. In our GAN, however, the generator is not directly connected to the loss that we're trying to affect. The generator feeds into the discriminator net, and the discriminator produces the output we're trying to affect. The generator loss penalizes the generator for producing a sample that the discriminator network classifies as fake.

This extra chunk of the network must be included in backpropagation. Backpropagation adjusts each weight in the right direction by calculating the weight's impact on the output — how the output would change if you changed the weight. However the impact of a generator weight depends on the impact of the discriminator weights it feeds into. So backpropagation starts at the output and flows back through the discriminator into the generator. At the same time, we don't want the discriminator to change during generator training. Trying to hit a moving target would make a hard problem even harder for the generator.

So we train the generator with the following procedure:

- Sample random noise.
- Produce generator output from sampled random noise.
- Get discriminator "Real" or "Fake" classification for generator output.
- Calculate loss from discriminator classification.
- Backpropagate through both the discriminator and generator to obtain gradients.
- Use gradients to change only the generator weights.

GAN Training

GAN contains two separately trained networks, its training algorithm must address two complications:

- GANs must juggle two different kinds of training (generator and discriminator).
- GAN convergence is hard to identify.

Convergence : As the generator improves with training, the discriminator performance gets worse because the discriminator can't easily tell the difference between real and fake. If the generator succeeds perfectly, then the discriminator has a 50% accuracy. In effect, the discriminator flips a coin to make its prediction.

This progression poses a problem for convergence of the GAN as a whole: the discriminator feedback gets less meaningful over time. If the GAN continues training past the point when the discriminator is giving completely random feedback, then the generator starts to train on junk feedback, and its own quality may collapse. For a GAN, convergence is often a fleeting, rather than stable, state.

Loss Functions :GANs try to replicate a probability distribution. They should therefore use loss functions that reflect the distance between the distribution of the data generated by the GAN and the distribution of the real data.

One Loss Function or Two?

A GAN can have two loss functions: one for generator training and one for discriminator training. How can two loss functions work together to reflect a distance measure between probability distributions?

In the loss schemes we'll look at here, the generator and discriminator losses derive from a single measure of distance between probability distributions. In both of these schemes, however, the generator can only affect one term in the distance measure: the term that reflects the distribution of the fake data. So during generator training we drop the other term, which reflects the distribution of the real data.

The generator and discriminator losses look different in the end, even though they derive from a single formula.

Minimax Loss : In the paper that introduced GANs, the generator tries to minimize the following function while the discriminator tries to maximize it:

$$E_x[\log(D(x))] + E_z[\log(1 - D(G(z)))]$$

In this function:

- $D(x)$ is the discriminator's estimate of the probability that real data instance x is real.
- E_x is the expected value over all real data instances.
- $G(z)$ is the generator's output when given noise z .
- $D(G(z))$ is the discriminator's estimate of the probability that a fake instance is real.
- E_z is the expected value over all random inputs to the generator (in effect, the expected value over all generated fake instances $G(z)$).
- The formula derives from the cross-entropy between the real and generated distributions.
- The generator can't directly affect the $\log(D(x))$ term in the function, so, for the generator, minimizing the loss is equivalent to minimizing $\log(1 - D(G(z)))$.

Wasserstein Loss

This loss function depends on a modification of the GAN scheme (called "Wasserstein GAN" or "WGAN") in which the discriminator does not actually classify instances. For each instance it outputs a number. This number does not have to be less than one or greater than 0, so we can't use 0.5 as a threshold to decide whether an instance is real or fake. Discriminator training just tries to make the output bigger for real instances than for fake instances.

Because it can't really discriminate between real and fake, the WGAN discriminator is actually called a "critic" instead of a "discriminator". This distinction has theoretical importance, but for practical purposes we can treat it as an acknowledgement that the inputs to the loss functions don't have to be probabilities.

MNIST Dataset Description

The MNIST (Modified National Institute of Standards and Technology) handwritten digit dataset is one of the most widely used benchmark datasets in the field of machine learning and computer vision. It contains a total of 70,000 grayscale images of handwritten digits from 0 to 9, with 60,000 images designated for training and 10,000 for testing. Each image is a 28x28 pixel square, resulting in a total of 784 pixels per image. The pixel values range from 0 (white background) to 255 (black ink), and each image is labeled with a corresponding digit class (0 through 9).

The dataset was created by combining samples from American Census Bureau employees and high school students, offering a diverse representation of handwriting styles.

Python Source Code:

```
import tensorflow as tf
from tensorflow.keras import layers, models import
numpy as np
import matplotlib.pyplot as plt

# Load and preprocess MNIST dataset
(train_images, _), (_, _) = tf.keras.datasets.mnist.load_data()
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).astype('float32')
train_images = (train_images - 127.5) / 127.5 # Normalize to [-1, 1]
BUFFER_SIZE = 60000
BATCH_SIZE = 256

# Create a tf.data.Dataset for training
train_dataset
tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)

# Define the Generator model def
make_generator_model():
model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
model.add(layers.BatchNormalization())    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))
    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
model.add(layers.BatchNormalization())    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
model.add(layers.BatchNormalization())    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same',
    use_bias=False, activation='tanh'))    return model

# Define the Discriminator model def
make_discriminator_model():
model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same', input_shape=[28, 28,
1]))    model.add(layers.LeakyReLU())    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2),
padding='same'))    model.add(layers.LeakyReLU())
model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
model.add(layers.Dense(1))
    return model
```

```

# Define loss functions
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

# Optimizers generator_optimizer =
tf.keras.optimizers.Adam(1e-4) discriminator_optimizer =
tf.keras.optimizers.Adam(1e-4)

# Initialize models
generator = make_generator_model() discriminator
= make_discriminator_model()

# Training step
@tf.function def
train_step(images):
    noise = tf.random.normal([BATCH_SIZE, 100])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

        gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
        gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)

        generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
        discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
        discriminator.trainable_variables))    return gen_loss, disc_loss

# Training loop def
train(dataset, epochs):
    for epoch in range(epochs):
        for image_batch in dataset:
            gen_loss, disc_loss = train_step(image_batch)

    # Print loss every epoch

```

```

print(f'Epoch {epoch + 1}, Generator Loss: {gen_loss:.4f}, Discriminator Loss: {disc_loss:.4f}')

# Generate and save images every 5 epochs      if
(epoch + 1) % 5 == 0:
    generate_and_save_images(generator, epoch + 1, seed)

# Generate and visualize test images      def
generate_and_save_images(model, epoch, test_input):
    predictions = model(test_input, training=False)    fig =
plt.figure(figsize=(4, 4))

    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i+1)
        plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')    plt.axis('off')

    plt.savefig(f'image_at_epoch_{epoch:04d}.png')    plt.show()

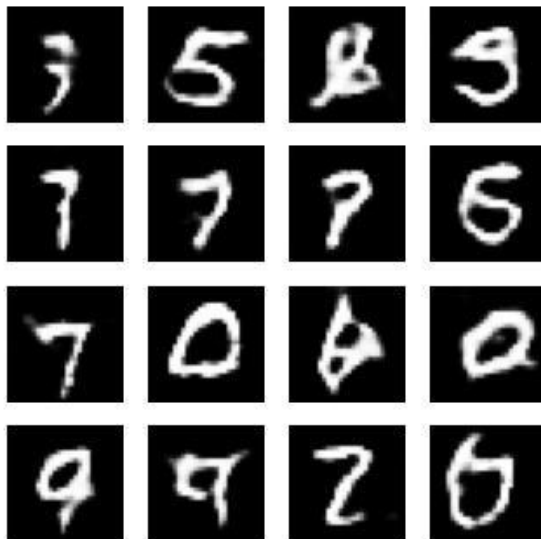
# Set random seed for reproducibility
seed = tf.random.normal([16, 100]) # 16 images for visualization

# Train the GAN EPOCHS = 50
train(train_dataset, EPOCHS)

# Generate final test images after training
generate_and_save_images(generator, EPOCHS, seed)

```

Output: Epoch 50, Generator Loss: 0.8876, Discriminator Loss: 1.2997



Problem No: 05

Problem Name: 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 neural network architecture commonly used in Computer Vision. Computer vision is a field of Artificial Intelligence that enables a computer to understand and interpret the image or visual data.

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

1. Input layer
2. Hidden layers
3. Output layer

Convolutional Neural Network (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.

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

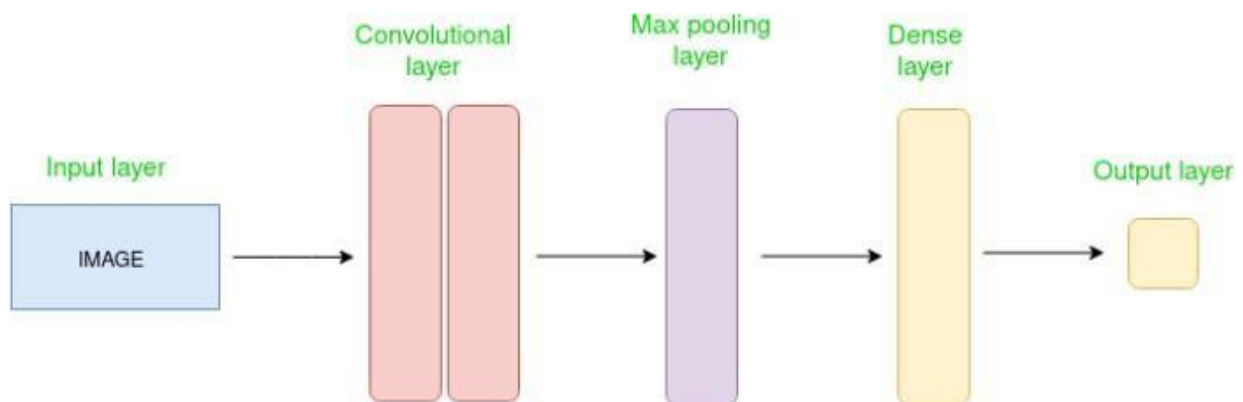


Figure-1: Simple CNN architecture

The Convolutional layer applies filters to the input image to extract features, the Pooling layer downsamples the image to reduce computation, and the fully connected layer makes the final prediction. The network learns the optimal filters through backpropagation and gradient descent.

Classification of images using Convolution Neural Network (CNN): In a classification task using a convolutional neural network (CNN), the goal is to categorize input data into predefined classes. CNNs are particularly effective for image classification due to their ability to capture spatial hierarchies in data. Here's a general outline of how you might approach a classification task with a CNN:

1. Data Preparation:
 - a. Collect Data
 - b. Preprocess Data
2. Define the CNN Architecture:
 - a. Input Layer
 - b. Convolutional Layers
 - c. Activation Function
 - d. Pooling Layers
 - e. Fully Connected Layers
 - f. Output Layer
3. Compile the Model:
 - a. Loss Function
 - b. Optimizer
 - c. Metrics
4. Train the Model
5. Evaluate the Model
6. Tune and Improve
7. Deploy the Model Source Code:

Source code:

```
#Training portion of the code
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 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 = "E:/BOOK/ICE-4-2/ICE-4206_Neural Networks Sessional/Neural Network
Sessional/lab_11/dataset/banana"
cucumber_folder = "E:/BOOK/ICE-4-2/ICE-4206_Neural Networks Sessional/Neural Network
Sessional/lab_11/dataset/cucumber" # 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
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.compile(optimizer='adam',
loss='binary_crossentropy', metrics=['accuracy'])
history = model.fit(X_train, y_train, epochs=100, 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)
#Testing portion of the code
#-----
from tensorflow.keras.preprocessing import image
import numpy as np
# Path to the test image
test_image_path = 'E:/BOOK/ICE-4-2/ICE-4206_Neural Networks
Sessional/Neural Network
Sessional/lab_11/pic1.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')

```

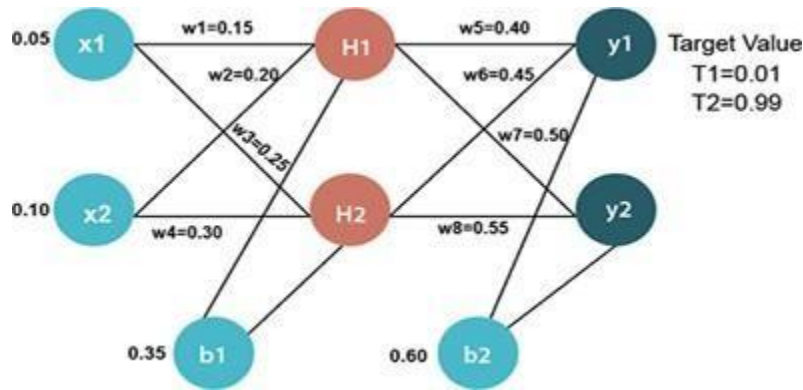
Output:

Prediction accuracy is: [[0.9634724]]

This is Cucumber

Problem No: 06

Problem Name: 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 Network (ANN): An Artificial Neural Network (ANN) is a computational model inspired by the way biological neural networks in the human brain process information. ANNs consist of interconnected layers of nodes (neurons), where each connection represents a synapse with an associated weight.

The basic components of an ANN include:

1. **Input Layer:** Receives input signals (features).
2. **Hidden Layer(s):** Intermediate layers that process the input data and extract patterns.
3. **Output Layer:** Produces the final output (predictions or classifications).

Structure of the Neural Network Given in the Diagram:

The given neural network is a **three-layer network**:

1. **Input Layer:** ○ Two input nodes: x_1 and x_2 with values 0.05 and 0.10.
2. **Hidden Layer:**
 - Two hidden neurons: H_1 and H_2 with biases b_1 and b_2 set to 0.35 each.
 - Weights between input and hidden layer:
 $w_1 = 0.15$, $w_2 = 0.20$, $w_3 = 0.25$, $w_4 = 0.30$.
3. **Output Layer:**
 - Two output neurons: y_1 and y_2 with biases 0.60.
 - Weights between hidden and output layer:
 $w_5 = 0.40$, $w_6 = 0.45$, $w_7 = 0.50$, $w_8 = 0.55$.
4. **Target Values:**

- The target values for the output nodes are $T1 = 0.01$ and $T2 = 0.99$.

Backpropagation Algorithm: Backpropagation is an algorithm used to train neural networks, by minimizing the error between the predicted output and the actual target values. It involves calculating the gradient of the loss function with respect to each weight by the chain rule, propagating backward through the network.

1. **Compute Output Layer Error:** ○ Calculate the error term for the output layer:

$$\delta_{output} = (T_i - O_i) \cdot \sigma'(O_i)$$

Where $\sigma'(O_i)$ is the derivatives of the hidden function (sigmoid in our case)

2. **Compute Hidden Layer Error:**

- Backpropagate the error to the hidden layer:

$$\delta_{hidden} = \delta_{output} \cdot w_{output-hidden} \sigma'(Output_H)$$

3. **Update Weights and Biases:**

- Adjust the weights and biases using the gradient descent algorithm:

$$w_{new} = w_{old} + \eta \cdot \delta \cdot Input$$

$$b_{new} = b_{old} + \eta \cdot \delta$$

where η is the learning rate.

Learning Rate (η): The learning rate (η) is a hyperparameter that determines the step size at each iteration while moving toward the minimum of the loss function. A smaller learning rate ensures more precise convergence but takes longer, whereas a larger learning rate might converge faster but can overshoot the minimum.

Source Code:

```
import torch
import torch.nn as nn
import torch.optim as optim
class SimpleANN(nn.Module):
    def __init__(self):
        super(SimpleANN, self).__init__()
        # Input to Hidden layer (2 inputs to 2 hidden nodes)
        self.hidden = nn.Linear(2, 2)
        # 2 input neurons, 2 hidden neurons
        # Hidden to Output layer (2 hidden nodes to 2 output nodes)
        self.output = nn.Linear(2, 2) # 2 hidden neurons, 2 output neurons
        # Sigmoid activation function self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # Forward pass through the network
        h = self.sigmoid(self.hidden(x)) # Hidden layer activation
        y = self.sigmoid(self.output(h)) # Output layer activation return
        y

# Create the network model
= SimpleANN()
# Set the weights and biases based on the diagram
with torch.no_grad():
    model.hidden.weight = torch.nn.Parameter(torch.tensor([[0.15, 0.20], [0.25, 0.30]])) # w1, w2, w3, w4
    model.hidden.bias = torch.nn.Parameter(torch.tensor([0.35, 0.35])) # Bias b1 for both hidden neurons
    model.output.weight = torch.nn.Parameter(torch.tensor([[0.40, 0.45], [0.50, 0.55]])) # w5, w6, w7, w8
    model.output.bias = torch.nn.Parameter(torch.tensor([0.60, 0.60])) # Bias b2 for both output neurons
# Define input (x1, x2) and target values (T1, T2)
inputs = torch.tensor([[0.05, 0.10]]) # Single input pair
targets = torch.tensor([[0.01, 0.99]]) # Target values for y1 and y2
# Define the loss function (Mean Squared Error Loss)
criterion = nn.MSELoss()
# Define the optimizer (Stochastic Gradient Descent)
optimizer = optim.SGD(model.parameters(), lr=0.5)
# Number of epochs (iterations)
epochs = 15000
# Training loop for epoch in
range(epochs):
    output = model(inputs)
    # Compute the loss (Mean Squared Error)
    loss = criterion(output, targets)
    # Print the loss every 1000 epochs if
    epoch % 1000 == 0:
        print(f'Epoch {epoch}, Loss: {loss.item()}')
# Print final weights and biases after training
print("\nFinal weights and biases:")
for name, param in
model.named_parameters():
    print(f'{name}: {param.data}')
# Final output after training
final_output = model(inputs)
print(f"\nFinal output (y1, y2): {final_output[0][0]:.2f}, {final_output[0][1]:.2f}")
```

Output:

Epoch 0, Loss: 0.2983711063861847
Epoch 1000, Loss: 0.0002707050589378923
Epoch 2000, Loss: 9.042368037626147e-05
Epoch 3000, Loss: 4.388535307953134e-05
Epoch 4000, Loss: 2.489008147676941e-05
Epoch 5000, Loss: 1.5388504834845662e-05
Epoch 6000, Loss: 1.0049888260255102e-05
Epoch 7000, Loss: 6.816366294515319e-06
Epoch 8000, Loss: 4.752399945573416e-06
Epoch 9000, Loss: 3.3828823688963894e-06
Epoch 10000, Loss: 2.4476007638440933e-06 Epoch 11000,
Loss: 1.7939109966391698e-06
Epoch 12000, Loss: 1.3286518196764519e-06
Epoch 13000, Loss: 9.925176982505945e-07 Epoch 14000,
Loss: 7.467624527635053e-07 Final weights and biases:
hidden.weight: tensor([[0.1833, 0.2667], [0.2826, 0.3652]])
hidden.bias: tensor([1.0170, 1.0025])
output.weight: tensor([[-1.4728, -1.4261], [1.5441, 1.5950]])
output.bias: tensor([-2.3714, 2.1961])
Final output (y1, y2): 0.01, 0.99

Problem No: 07

Problem Name: Write a program to evaluate a Recurrent Neural Network (RNN) for text Classification.

Theory: Recurrent Neural Networks (RNNs) are a class of neural networks specifically designed to handle sequential data such as time series, natural language, or audio. Unlike traditional feed forward neural networks, RNNs have loops in their architecture, allowing information to persist and be reused across different time steps. This makes them especially powerful for tasks where the current input is dependent on previous inputs.

Why RNN: Recurrent Neural Networks (RNNs) were introduced to overcome the limitations of traditional Artificial Neural - (ANNs) when dealing with sequence data. RNNs address the 3 main issues shown in figure 1.

- Traditional ANNs expect fixed-size input and output, which is a problem in tasks like text generation or speech recognition. Where RNNs can process sequences of arbitrary length, making them suitable for dynamic input/output sizes.
- In ANNs, modeling temporal dependencies over long sequences often requires deep, complex architecture. But RNNs reuse the same weights across time steps, reducing model size and computation.
- ANNs treat each input independently, which leads to a lack of context and large numbers of parameters. On the other hand, RNNs share parameters across all time steps, allowing the network to generalize better and learn temporal patterns efficiently.

For example, when training an RNN for a task like name/entity identification (also called Named Entity Recognition, or NER), each word in the input sequence is passed through the RNN one step at a time.

In this example (shown in figure 2), the sentence is: "Ironman punched on hulk's face"

➔ The task is to identify names in the sequence (like "Ironman" and "hulk").

The expected output is: 1 0 0 1 0 (1 if the word is a name, 0 otherwise)

Here's how training works:

- Each word is input sequentially into the RNN.
- At each time step, the model produces a prediction y (whether the current word is a name or not).
- This prediction is compared with the actual label (0 or 1), and a loss is calculated.
- The total loss is the sum of all individual losses across the sequence.
- The model uses this total loss to update its weights using backpropagation through time (BPTT).

This structure allows the model to learn from context, so it can distinguish whether a word is a name based on surrounding words — for example, recognizing that “hulk’s” refers to a person because of the possessive form and prior context.

The working principle of an RNN revolves around the concept of a hidden state, which captures the memory of the network. At each time step t , the RNN takes the input x_t and the previous hidden state h_{t-1} to compute the new hidden state h_t . This hidden state acts as a summary of all previous inputs. The recurrence relation is typically defined by the equation:

$$h_t = \tanh(W_{xh} * x_t + W_{hh} * h_{t-1} + b_h) \quad 1$$

here, W_{xh} and W_{hh} are weight matrices for the input and hidden state, and b_h is the bias term. This equation updates the memory of the network. It combines the current input and past memory (via h_{t-1}), processes them through a weighted sum, and then squashes the result with the tanh function to get the new memory.

The output y_t at each step is often computed as:

$$y_t = W_{hy} * h_t + b_y \quad 2$$

Here, W_{hy} is weight matrix from hidden to output layer

To improve learning and avoid vanishing gradients, the error is backpropagated through time (BPTT) using:

$$\frac{\partial L}{\partial W} = \sum \frac{\partial L}{\partial h_t} * \frac{\partial h_t}{\partial W} \quad 3$$

Here, L is the loss function, $\frac{\partial L}{\partial h_t}$ is gradient of loss w.r.t. hidden state and $\frac{\partial h_t}{\partial W}$ is gradient of hidden state w.r.t. weight. To train an RNN, we need to compute gradients of the loss with respect to all weights. This equation shows that we accumulate gradients over all time steps to properly update the weights.

$$\frac{\partial h_t}{\partial W} = \frac{\partial \tanh(.)}{\partial W} = (1 - h_t^2) * input \quad 4$$

This is the derivative of the tanh activation function, used during backpropagation to compute how the hidden state changes with respect to the weights. The term $(1 - h_t^2)$ comes from the derivative of tanh, and it is multiplied by the input to complete the gradient.

However, RNNs suffer from problems like vanishing and exploding gradients when dealing with long sequences. To address this, advanced variants like Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) were introduced. These use gating mechanisms to control the flow of information and preserve long-term dependencies more effectively.

RNNs find widespread applications in natural language processing (NLP) tasks such as sentiment analysis, machine translation, and text generation. They're also used in speech recognition, music composition, video captioning, and time-series forecasting.

Despite their strength in modeling sequences, vanilla RNNs are being gradually replaced by more efficient and scalable architectures like Transformers. Still, RNNs remain foundational in understanding the evolution of deep learning for sequential data.

Source Code:

```
import tensorflow as tf
from tensorflow import keras
import numpy as np

# Load the IMDB dataset
vocab_size = 10000
max_length = 200

(x_train, y_train), (x_test, y_test) = keras.datasets.imdb.load_data(num_words=vocab_size)

# Pad sequences
x_train = keras.preprocessing.sequence.pad_sequences(x_train, maxlen=max_length)
x_test = keras.preprocessing.sequence.pad_sequences(x_test, maxlen=max_length)

# Define the RNN model using LSTM
model = keras.Sequential([
    keras.layers.Embedding(vocab_size, 32, input_length=max_length),
    keras.layers.LSTM(64, return_sequences=True),
    keras.layers.LSTM(32),
    keras.layers.Dense(1, activation='sigmoid')
])

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

# Train the model
model.fit(x_train, y_train, epochs=5, validation_data=(x_test, y_test))

# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f'\nTest accuracy: {test_acc}')
```

Output: Test accuracy 0.8562

Problem No: 08

Problem Name: Write a MATLAB or Python program to Purchase Classification Prediction using SVM.

Theory: Classification is a supervised learning task where the goal is to predict the categorical label of new instances based on a training dataset with known labels. In the context of purchase classification, the objective might be to predict whether a customer will purchase a product (e.g., yes/no) based on features such as demographic information, past purchase behavior, etc.

Support Vector Machines (SVM): SVM is a popular classification technique used in machine learning. It works by finding a hyperplane that best separates the data into different classes. The main concepts involved are:

- **Hyperplane:** In an n -dimensional space, a hyperplane is an $(n-1)$ -dimensional plane that separates the space into two halves. For instance, in a 2D space, a hyperplane is a line; in 3D space, it is a plane.
- **Support Vectors:** These are the data points that are closest to the hyperplane and are critical in defining the position and orientation of the hyperplane.
- **Margin:** The margin is the distance between the hyperplane and the nearest support vectors. SVM aims to maximize this margin to create a robust classifier.

SVM Types:

- **Linear SVM:** Used when the data is linearly separable. The algorithm finds a linear hyperplane that separates the classes.
- **Non-Linear SVM:** Used when the data is not linearly separable. The algorithm uses kernel functions to transform the data into a higher-dimensional space where a linear separation is possible.

Application in Purchase Classification: In the context of purchase classification, the SVM model can be used to predict whether a customer will make a purchase based on various features. For example, features might include:

- **Demographics:** Age, income, etc.
- **Behavioral Data:** Past purchase history, browsing behavior, etc.

Conclusion: Using SVM for purchase classification is effective for many practical scenarios where you need to classify instances based on feature vectors. SVM's ability to handle both linear and non-linear data through different kernels makes it a versatile tool in machine learning. Proper preprocessing, model training, and evaluation are crucial for developing an accurate and reliable classifier.

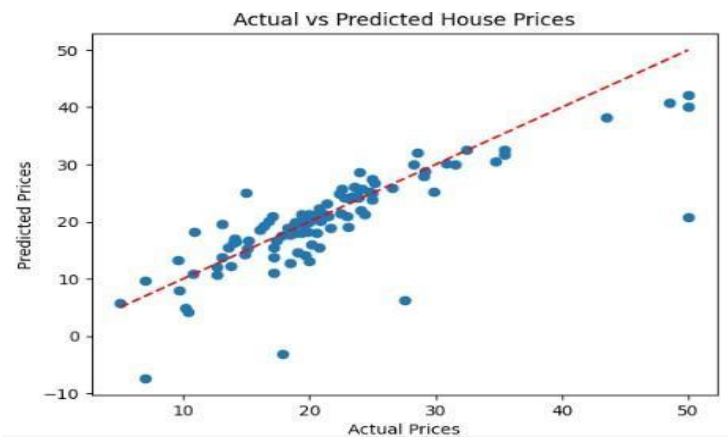
Source Code:

```
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("\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}')
```

Output:

Mean Squared Error: 29.44

R² 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