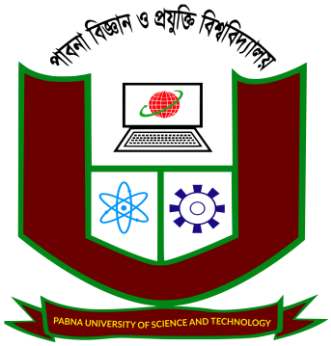
PABNA UNIVERSITY OF SCIENCE & TECHNOLOGY



DEPARTMENT OF

INFORMATION AND COMMUNICATION ENGINEERING

FACULTY OF ENGINEERING AND TECHNOLOGY

Lab Report

Course Title: Neural Networks Sessional

Course Code: ICE-4206

Submitted By:	*Submitted To:*
Name: Gulam Mustofa **Roll: 200615** **Session: 2019-20** **4th Year 2nd Semester,** **Department of ICE,** **Pabna University of Science and Technology.**	**Dr. Md. Imran Hossain** **Associate Professor,** **Department of ICE,** **Pabna University of Science and Technology.**

Submission Date: 25-05-2025 **Signature**

INDEX

Problem No.	Problem Name	Page no.
01	Write a program to evaluate AND function with bipolar inputs and target and show the convergence curves and the decision boundary lines	
02	Write a program to evaluate X-OR function with bipolar inputs and target and also show the convergence curves and the decision boundary lines	
03	Implement the SGD method using delta learning rules 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]	
04	Write a program to evaluate a simple feedforward neural network for classifying handwritten digits using the MNIST dataset.	
05	Write a program to evaluate a convolution neural network (CNN) for image classification.	
06	write a program to evaluate a recurrent neural network (RNN) for text classification.	
07	write a program to evaluate a Transformer model for text classification.	
08	Write a program to evaluate Generative adversarial Network (GAN) for image generation.	

Problem No: 01

Problem Name: Write a program to evaluate AND function with bipolar inputs and target and show the convergence curves and the decision boundary lines.

Theory:

The AND function is a fundamental logical operation in both classical computation and neural networks. In this lab, we aim to evaluate the AND function using bipolar inputs and targets and demonstrate how a neural network model can be trained to learn this behavior using the Perceptron learning algorithm. Additionally, we visualize the convergence curve (which shows how error reduces during training) and the decision boundary (which separates output classes in the input space).

Why Bipolar Inputs?

In neural networks, especially when using simple activation functions like the step function or sign function, we often represent Boolean logic using bipolar values instead of binary.

- In binary representation: 0 represents "False", and 1 represents "True".
- In bipolar representation: -1 represents "False", and +1 represents "True".

This form of representation improves the symmetry of data, helps in certain learning algorithms, and better suits gradient-based methods.

AND Function with Bipolar Inputs:

The bipolar version of the 2-input AND gate is shown in the table below:

Input x_1	**Input x_2**	**Target (AND)**
-1	-1	-1
-1	+1	-1
+1	-1	-1
+1	+1	+1

The target output is +1 only when both inputs are +1; otherwise, the output is -1.

Learning the AND Function with Perceptron:

To model this, we use the Perceptron learning rule, which is a simple algorithm for training a linear binary classifier. The Perceptron is a single-layer neural network that updates weights to minimize classification errors.

Working Principle:

1. The perception computes the weighted sum of inputs:

$$y = \text{sign}(w_0 + w_1 x_1 + w_2 x_2)$$

Here, w0 is the bias w1 and w2 are weights and the sign() function outputs +1 or -1.

2. During training, if the output is incorrect, the weight is updated using:

$$w_{\text{new}} = w_{\text{old}} + \eta(t - y)x$$

   Where:
   η = learning rate
   t = target
   y = predicted output
   x = input vector
3. The model iterates over the dataset multiple times (epochs), adjusting weights until all samples are correctly classified or until a maximum number of epochs is reached.

Convergence Curve:

During training, we track the error after each epoch. This allows us to plot a convergence curve, which shows how quickly the network is learning. A steep downward trend indicates rapid learning, while a flat curve indicates convergence or stagnation.

This curve is helpful for:

- Evaluating the learning speed
- Checking whether learning is happening
- Diagnosing if the learning rate is too high or too low

Decision Boundary:

In 2D problems like this, the learned model creates a linear decision boundary (a straight line) that separates the input space into two regions — one where the output is +1, and one where it is -1.

Mathematically, the boundary is given by:

$$w_0 + w_1 x_1 + w_2 x_2 = 0$$

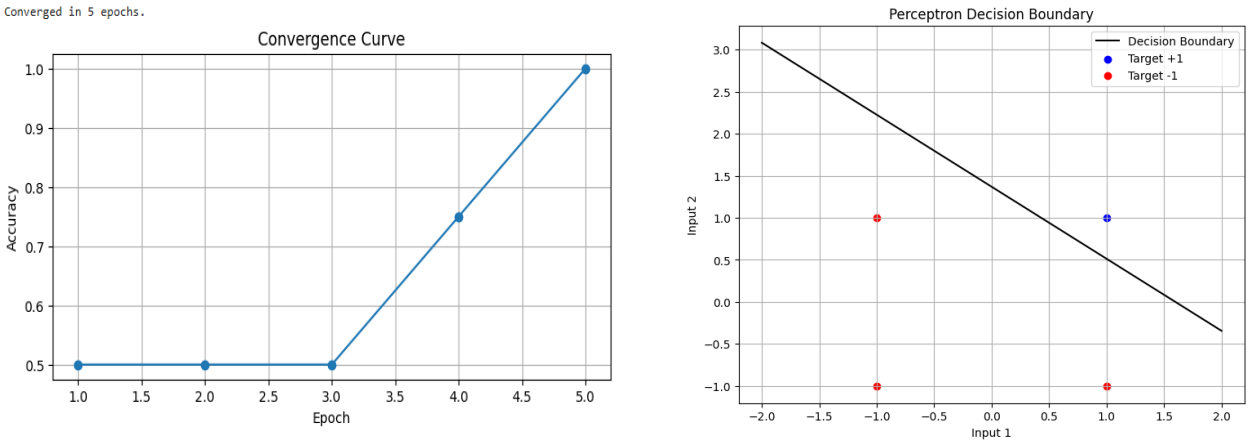
This line divides the plane into two halves. All points on one side of the line are classified as +1, and on the other side as -1. Visualizing this decision boundary helps us understand how the model is separating classes in the input space.

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 for AND function)
    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, marker='o')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.title('Convergence Curve')
    plt.grid(True)
    plt.show()

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

Output:



Problem No: 02

Problem Name: Write a program to evaluate X-OR function with bipolar inputs and target and also show the convergence curves and the decision boundary lines

Theory:

In this experiment, we evaluate the XOR (Exclusive OR) function using bipolar inputs and targets, and analyze its behavior when trained with a neural network model. The XOR function is very important in neural networks because it is a classic example of a problem that is not linearly separable. We also demonstrate this by plotting the convergence curve and decision boundary lines.

What is XOR?

The XOR function returns True (or +1 in bipolar) only when exactly one of the two inputs is True (+1). In all other cases, it returns False (or -1).

Bipolar Representation of XOR:

Input x_1	**Input x_2**	**Target (XOR)**
-1	-1	-1
-1	+1	+1
+1	-1	+1
+1	+1	-1

This table shows the bipolar version of the XOR truth table.

Why XOR is Special in Neural Networks?

XOR is not linearly separable, which means we cannot draw a straight line to separate its output classes. Because of this, a **single-layer perceptron** cannot learn the XOR function. This limitation was a major issue in early neural network research.

To solve the XOR problem, we need a multi-layer perceptron (MLP), which is a feedforward neural network with at least one hidden layer. This hidden layer introduces non-linearity into the model, allowing it to classify patterns like XOR.

Architecture of XOR Network:

In our experiment, we use:

- 2 input neurons (for x_1 and x_2)
- 1 hidden layer with 2 neurons
- 1 output neuron with a bipolar (sign) output
- A learning rule such as Backpropagation

The activation function used is typically a tanh or sigmoid function in the hidden layer and a sign function for the output.

Training Using Backpropagation:

The network learns the weights by minimizing the error between the predicted output and the target output using backpropagation, which works as follows:

1. **Forward Pass**: Inputs are passed through the network to get the output.
2. **Error Calculation**: Compare the output with the actual target to get the error.
3. **Backward Pass**: The error is propagated back, and weights are updated using the gradient descent rule.

This process continues for many epochs until the network learns the XOR pattern correctly.

Convergence Curve:

The convergence curve shows the reduction of error over time. In this case, it may take more epochs than linearly separable functions because of the added complexity of the XOR function. The curve helps us understand whether the network is learning and how quickly it is converging.

Decision Boundary:

After training, we can plot the decision boundary in the input space. For XOR, this boundary is non-linear and can only be formed by combining multiple linear functions (made possible using hidden layers). The decision boundary visualization clearly shows how the MLP has learned to separate the inputs correctly into two classes (+1 and -1).

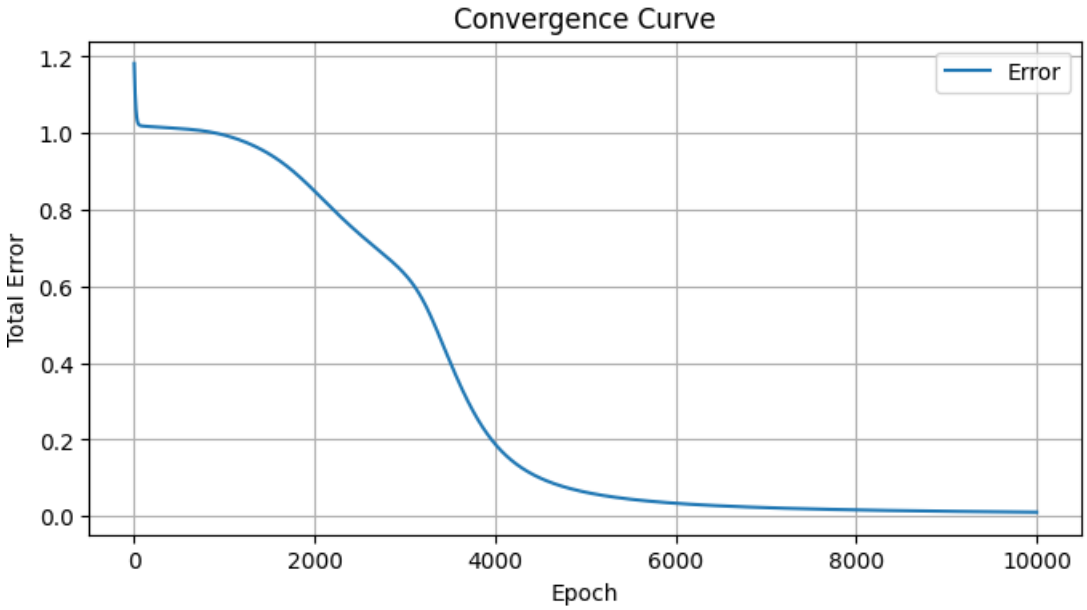
Code:

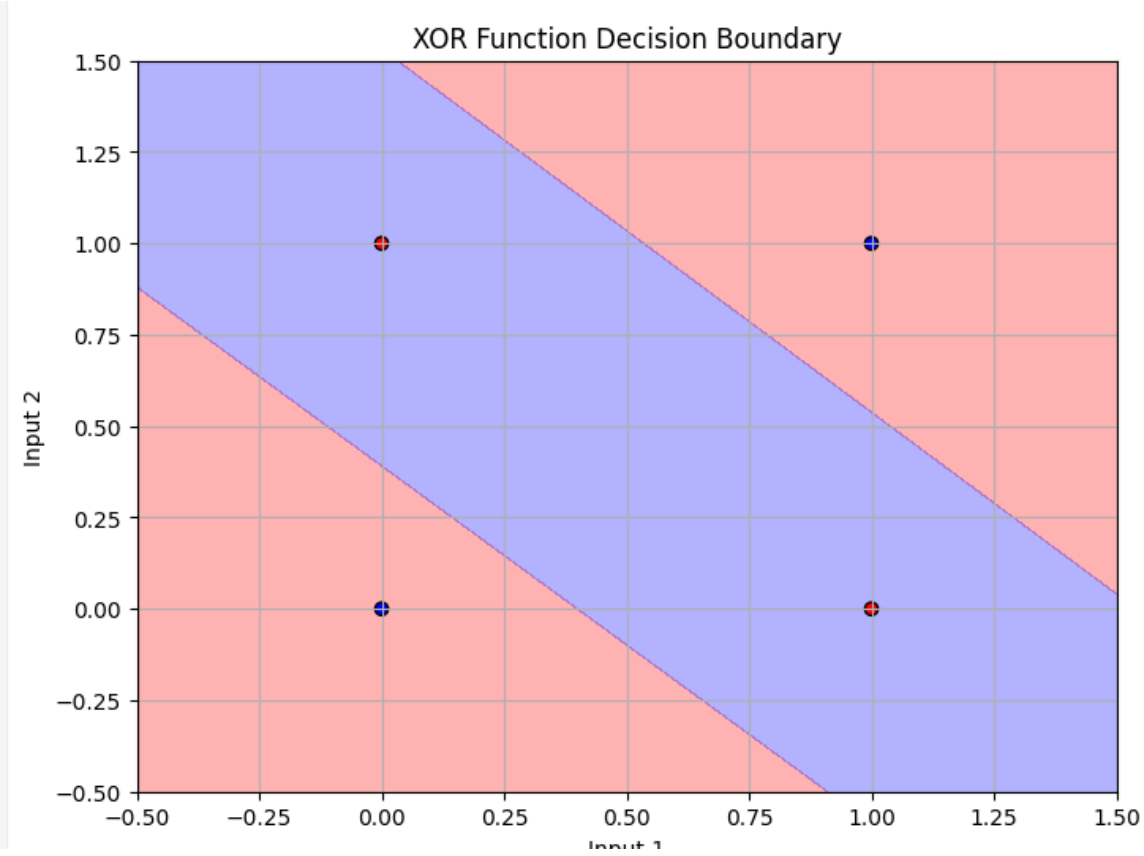
```
import numpy as np
import matplotlib.pyplot as plt

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

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

# XOR input and target
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
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):
    total_error = 0

    for i in range(len(inputs)):
        # Forward pass
        input_layer = inputs[i]
        hidden_input = np.dot(input_layer, weights_input_hidden) + bias_hidden
        hidden_output = sigmoid(hidden_input)

        final_input = np.dot(hidden_output, weights_hidden_output) + bias_output
        final_output = sigmoid(final_input)

        # Error
        error = targets[i] - final_output
        total_error += np.sum(error ** 2)

        # Backpropagation
        output_delta = error * sigmoid_derivative(final_output)
        hidden_delta = sigmoid_derivative(hidden_output) * np.dot(weights_hidden_output,
output_delta)

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

        weights_input_hidden += learning_rate * input_layer[:, np.newaxis] * hidden_delta
```

```
        bias_hidden += learning_rate * hidden_delta

    convergence_curve.append(total_error)

    if total_error < 0.01:
        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, label='Error')
plt.xlabel('Epoch')
plt.ylabel('Total Error')
plt.title('Convergence Curve')
plt.grid(True)
plt.legend()
plt.show()

# Function to predict the output (for decision boundary)
def predict(x1, x2):
    input_grid = np.array([x1, x2]).T
    hidden_input = np.dot(input_grid, weights_input_hidden) + bias_hidden
    hidden_output = sigmoid(hidden_input)
    final_input = np.dot(hidden_output, weights_hidden_output) + bias_output
    final_output = sigmoid(final_input)
    return final_output.reshape(x1.shape)

# 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.ravel(), X2.ravel())
Z = Z.reshape(X1.shape)

# Plot decision boundaries
plt.figure(figsize=(8, 6))
plt.contourf(X1, X2, Z, levels=[0, 0.5, 1], colors=['red', 'blue'], alpha=0.3)
plt.scatter(inputs[:, 0], inputs[:, 1], c=targets.flatten(), cmap='bwr', edgecolors='k')
plt.xlabel('Input 1')
plt.ylabel('Input 2')
plt.title('XOR Function Decision Boundary')
plt.grid(True)
plt.show()
```

Output:


Convergence Curve
Error
Total Error
Epoch
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
2000
4000
6000
8000
10000


XOR Function Decision Boundary
Input 2
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
1.25
1.50

Problem No: 03

Problem Name: Implement the SGD method using delta learning rules 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:

In this experiment, we implement the Stochastic Gradient Descent (SGD) method using the Delta Learning Rule to train a simple neural network. The network is trained on a given dataset of input-target pairs to learn a linear mapping between the input and the desired output.

Input and Target Set:

We are given the following input matrix Xinput and target output Dtarget:

Input matrix (X_{input}):

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Here, each row is an input vector of 3 values. The third value 1 is added as a bias input.

Target Output (D_{target}):

$$\begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

This dataset represents the AND logic function, where the output is 1 only when both first two inputs are 1.

The objective of this experiment is to train the neural network using the Delta Rule with Stochastic Gradient Descent (SGD) and learn the correct mapping from inputs to target outputs.

Delta Learning Rule:

The Delta Rule (also known as the Widrow-Hoff rule) is a supervised learning rule used for single-layer networks. It updates the weights in the direction that reduces the error between actual output and desired output.

Weight update formula:

$$w_{new} = w_{old} + \eta \cdot (d - y) \cdot x$$

Where:

- η = learning rate (a small positive constant)
- d = desired output (target)
- y = actual output (prediction)
- x = input vector

This formula is applied for each training sample, which is why it is called Stochastic Gradient Descent (SGD).

Training Procedure:

1. Initialize weights randomly.
2. For each input vector:
    - Compute the output: $y=w^T \cdot x$
    - Calculate the error: $e=d-y$
    - Update the weights using the delta rule.
3. Repeat for multiple epochs (passes through the whole dataset) until the total error is minimized.

Convergence Curve:

We keep track of the mean squared error (MSE**)** in each epoch. The convergence curve plots the MSE over epochs, which shows how quickly and effectively the model is learning. A decreasing curve indicates successful training.

Decision Boundary:

After training, we can plot the decision boundary using the learned weights. This boundary separates the input space into two regions: one for output `0` and one for output `1`. Since this is an AND function (which is linearly separable), the decision boundary will be a straight line.

Code:

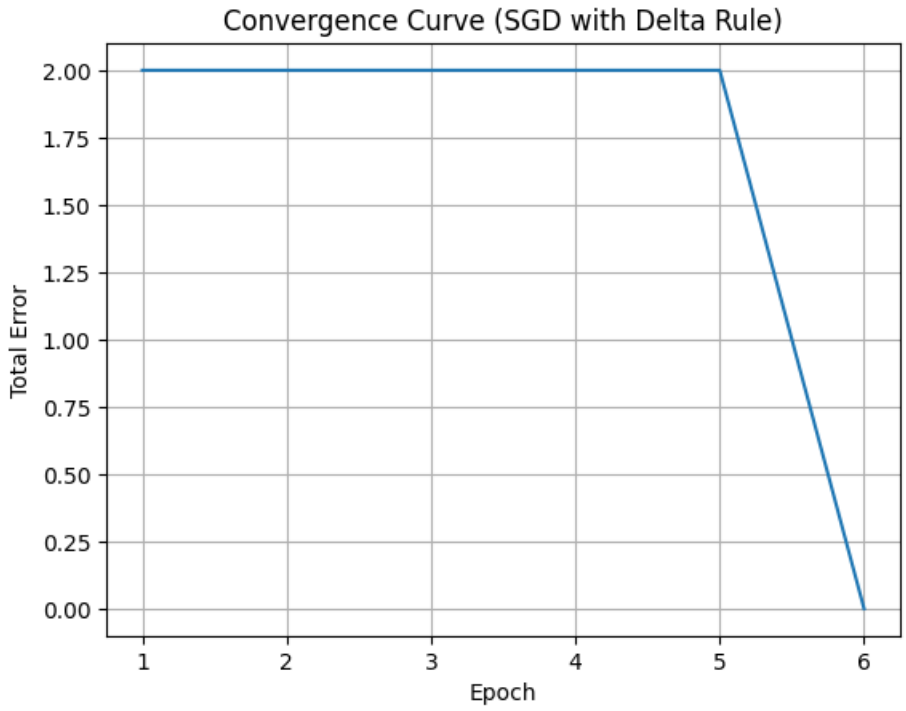
```
import numpy as np
import matplotlib.pyplot as plt

# Input and target values (bias included as the third column)
Xinput = np.array([
    [0, 0, 1],
    [0, 1, 1],
    [1, 0, 1],
    [1, 1, 1]
])
```

```
Dtarget = np.array([0, 0, 1, 1])  # Target output (AND gate)

# Initialize weights randomly
np.random.seed(0)  # For reproducibility
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 not converged:
        print(f"Converged in {epoch + 1} epochs.")
        converged = True
        break

# 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 6 epochs.
Final weights: [ 1.76405235 -0.09984279 -0.02126202]
```



Problem No:04

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

Theory

Handwritten digit classification is a classic problem in the field of machine learning and computer vision. It involves building a model that can recognize digits (0–9) from images. One of the most commonly used datasets for this task is the MNIST dataset, which contains 60,000 training images and 10,000 test images of 28x28 grayscale handwritten digits.

In this experiment, we use a **Simple Feedforward Neural Network (FFNN)**, also known as a **Multi-Layer Perceptron (MLP)**, to classify these digits. Unlike Convolutional Neural Networks (CNNs), FFNNs do not use spatial hierarchy or convolution layers but still provide valuable insights into the workings of basic neural architectures.

Working Principle

A feedforward neural network consists of layers of artificial neurons:

- **Input Layer:** Receives the raw pixel values of the image.
- **Hidden Layers:** Applies weights, bias, and an activation function (like ReLU) to learn patterns from the data.
- **Output Layer:** Contains 10 neurons (for digits 0–9) with a softmax activation to classify input into one of 10 classes.

The learning process involves:

1. **Forward Pass:** Data flows from input to output to compute predictions.
2. **Loss Calculation:** The difference between predicted and true label is measured using categorical crossentropy.
3. **Backpropagation:** Gradients of loss w.r.t. weights are calculated and used to update the weights using **Stochastic Gradient Descent (SGD)** or **Adam** optimizer.
4. **Epochs:** The model learns over multiple passes (epochs) through the dataset.

Why Use FFNN for MNIST?

- FFNN is simple and provides a baseline for performance.
- It helps students understand fundamental training mechanisms.
- MNIST's small image size (28x28 = 784 features) is well-suited for FFNNs.

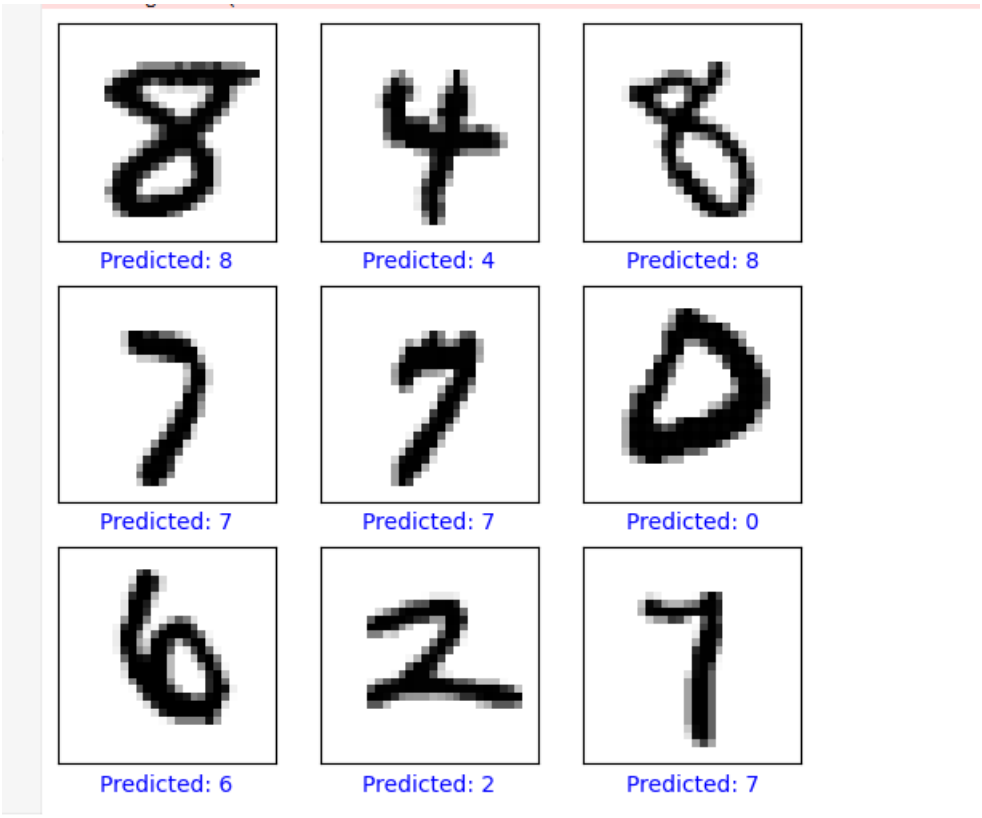
- It performs reasonably well (~97% accuracy) even without using convolution layers.

Code:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score

# Load the MNIST dataset
mnist = fetch_openml('mnist_784', version=1, as_frame=False)
X, y = mnist.data, mnist.target.astype(int)

# Normalize and split the dataset
X = X / 255.0
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train the neural network model
model = MLPClassifier(hidden_layer_sizes=(128,), activation='relu', solver='adam',
                      max_iter=10, random_state=42, verbose=True)
model.fit(X_train, y_train)

# Evaluate the model
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f'\nTest accuracy: {accuracy}')

# Function to plot image and prediction
def plot_image(i, true_label, img):
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(img.reshape(28, 28), cmap=plt.cm.binary)
    plt.xlabel(f"Predicted: {y_pred[i]}", color='blue')

# Display sample predictions
num_rows, num_cols = 3, 3
num_images = num_rows * num_cols
plt.figure(figsize=(2*num_cols, 2*num_rows))
for i in range(num_images):
    plt.subplot(num_rows, num_cols, i+1)
    plot_image(i, y_test[i], X_test[i])
plt.show()
```

Output:

```
Iteration 1, loss = 0.42473059
Iteration 2, loss = 0.19824993
Iteration 3, loss = 0.14820670
Iteration 4, loss = 0.11724433
Iteration 5, loss = 0.09662182
Iteration 6, loss = 0.08035723
Iteration 7, loss = 0.06790252
Iteration 8, loss = 0.05895092
Iteration 9, loss = 0.05065377
Iteration 10, loss = 0.04440961

Test accuracy: 0.9717142857142858
```



Problem No:05

Problem Name: Write a program to evaluate a convolution neural network (CNN) for image classification.

Theory

Image classification has become a critical task in computer vision, and Convolutional Neural Networks (CNNs**)** have proven to be one of the most effective models for this purpose. CNNs are a specialized type of neural network model that are particularly well-suited for processing data with a grid-like topology, such as images. They have significantly outperformed traditional machine learning techniques in image recognition, object detection, and classification tasks.

Introduction to CNN

A CNN is a deep learning model that automatically learns to extract meaningful features from images through a hierarchical structure of layers. These features can include simple patterns such as edges in the first layers, and complex patterns such as objects or shapes in the deeper layers. Unlike traditional neural networks that use fully connected layers for all inputs, CNNs use convolutional layers that reduce the number of parameters and computations while maintaining the ability to learn powerful features.

Key Components of CNN

1. **Convolutional Layer**:
    - This is the core building block of a CNN.
    - It applies a number of filters (also called kernels) to the input image to produce feature maps.
    - Each filter detects specific features like horizontal edges, vertical edges, or patterns.
    - The result of convolution is passed through an activation function (usually ReLU) to introduce non-linearity.
2. **Activation Function (ReLU)**:
    - ReLU (Rectified Linear Unit) replaces all negative pixel values with zero.
    - It helps the network learn non-linear relationships and speeds up convergence.
3. **Pooling Layer**:
    - Pooling (usually Max Pooling) is used to reduce the spatial dimensions (width and height) of the feature maps.

- It helps in reducing the computation, preventing overfitting, and improving model generalization.

4. **Flattening**:
    - After several convolution and pooling layers, the multidimensional output is flattened into a 1D vector.
    - This vector is then passed to the fully connected layers.
5. **Fully Connected Layer (Dense Layer)**:
    - These are traditional neural network layers.
    - Each neuron is connected to all the neurons in the previous layer.
    - It combines the features learned by convolutional layers and makes predictions.
6. **Output Layer**:
    - This layer uses the Softmax activation function for multi-class classification.
    - It provides the final probability distribution across all possible output classes.

Advantages of CNN in Image Classification

- **Automatic Feature Extraction**: CNNs learn useful image features automatically without manual feature engineering.
- **Parameter Efficiency**: By using local connections and shared weights, CNNs drastically reduce the number of parameters compared to fully connected networks.
- **Robustness to Translation**: CNNs can recognize patterns in different parts of the image, making them invariant to image translation and local distortions.
- **Improved Accuracy**: CNNs consistently achieve state-of-the-art results in image classification tasks.

Code:

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

# Load the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
```

```
# Normalize pixel values to [0, 1]
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

# Class names for CIFAR-10
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

# Define the CNN model
model = keras.Sequential([
    keras.layers.Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(32, 32,
3)),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
    keras.layers.Flatten(),
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dense(10, activation='softmax')  # 10 classes for CIFAR-10
])

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

# Print model summary
model.summary()

# Train the model
history = model.fit(x_train, y_train, epochs=10,
                    batch_size=64,
                    validation_data=(x_test, y_test),
                    verbose=2)

# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f'\nTest accuracy: {test_acc:.4f}')

# Plot training and validation accuracy
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.show()
```

Output:

Model: "sequential"

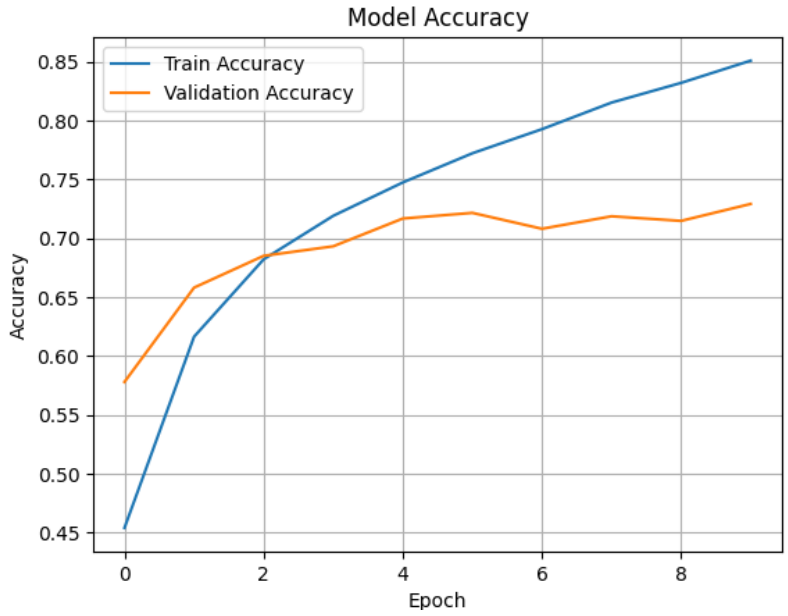
Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 32)	896
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0
conv2d_1 (Conv2D)	(None, 16, 16, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 64)	0
conv2d_2 (Conv2D)	(None, 8, 8, 64)	36,928
flatten (Flatten)	(None, 4096)	0
dense (Dense)	(None, 64)	262,208
dense_1 (Dense)	(None, 10)	650

Total params: 319,178 (1.22 MB)

Trainable params: 319,178 (1.22 MB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/10
782/782 - 34s - 43ms/step - accuracy: 0.4540 - loss: 1.5076 - val_accuracy: 0.5780 - val_loss: 1.1870
Epoch 2/10
782/782 - 27s - 34ms/step - accuracy: 0.6164 - loss: 1.0839 - val_accuracy: 0.6581 - val_loss: 0.9679
Epoch 3/10
782/782 - 26s - 34ms/step - accuracy: 0.6821 - loss: 0.9066 - val_accuracy: 0.6851 - val_loss: 0.8993
Epoch 4/10
782/782 - 27s - 34ms/step - accuracy: 0.7191 - loss: 0.8041 - val_accuracy: 0.6932 - val_loss: 0.8756
Epoch 5/10
782/782 - 27s - 34ms/step - accuracy: 0.7474 - loss: 0.7162 - val_accuracy: 0.7168 - val_loss: 0.8193
Epoch 6/10
782/782 - 28s - 36ms/step - accuracy: 0.7722 - loss: 0.6520 - val_accuracy: 0.7216 - val_loss: 0.8254
Epoch 7/10
782/782 - 28s - 36ms/step - accuracy: 0.7927 - loss: 0.5924 - val_accuracy: 0.7081 - val_loss: 0.8668
Epoch 8/10
782/782 - 30s - 38ms/step - accuracy: 0.8153 - loss: 0.5295 - val_accuracy: 0.7187 - val_loss: 0.8402
Epoch 9/10
782/782 - 32s - 40ms/step - accuracy: 0.8320 - loss: 0.4755 - val_accuracy: 0.7148 - val_loss: 0.8675
Epoch 10/10
782/782 - 32s - 41ms/step - accuracy: 0.8509 - loss: 0.4246 - val_accuracy: 0.7292 - val_loss: 0.8724
313/313 - 3s - 9ms/step - accuracy: 0.7292 - loss: 0.8724

Test accuracy: 0.7292
```



Problem No:06

Problem Name: write a program to evaluate a recurrent neural network (RNN) for text classification.

Theory

Text classification is a fundamental task in Natural Language Processing (NLP), where the goal is to assign predefined categories or labels to text documents. Common applications include spam detection, sentiment analysis, language detection, and topic classification. Traditional machine learning models often rely on manually extracted features, but these models struggle to capture the sequential nature and contextual meaning of language.

To address this, Recurrent Neural Networks (RNNs) are used. RNNs are a type of neural network architecture that are specifically designed to handle sequential data, such as text, speech, and time series. Unlike feedforward networks, RNNs have loops that allow information to persist, making them capable of remembering previous inputs in the sequence and modeling context over time.

What is an RNN?

A Recurrent Neural Network processes input sequences one element at a time, maintaining a hidden state that contains information about previous elements. This hidden state is updated with each new input. Mathematically, the RNN computes the current hidden state h_t using the current input x_t and the previous hidden state h_{t-1} :

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

Here, W_{xh} and W_{hh} are weight matrices and b is the bias.

This recurrent structure makes RNNs ideal for tasks where the sequence or order of the data is important, such as language modeling or sentiment analysis.

Challenges with Basic RNNs

While RNNs are conceptually simple and powerful, they suffer from several limitations:

Vanishing and Exploding Gradients: During training, the gradients can become too small or too large, making learning difficult.

Limited Long-Term Memory: Basic RNNs struggle to capture dependencies that are far apart in the input sequence.

To overcome these problems, more advanced RNN architectures have been developed, such as:

Long Short-Term Memory (LSTM) networks

Gated Recurrent Units (GRU)

These architectures include gates that control the flow of information and can capture long-range dependencies better than basic RNNs.

Text Classification Pipeline Using RNN

1. **Text Preprocessing**:

   Convert all text to lowercase.

   Remove punctuation and special characters.

   Tokenize text into words or subwords.

   Convert words to integers using tokenization.

2. **Embedding Layer**:

   Transforms words into dense vectors of fixed size (word embeddings).

   Captures semantic relationships between words.

3. **Recurrent Layer (RNN / LSTM / GRU)**:

   Processes the sequence of embeddings.

   Learns contextual dependencies between words.

4. **Dense Output Layer**:

   Outputs probabilities for each class using a softmax (for multi-class) or sigmoid (for binary classification) function.

Advantages of RNNs for Text Classification

1. **Sequential Understanding**: Maintains context by remembering previous inputs.
2. **Flexible Input Length**: Can handle sequences of varying lengths.
3. **Context-Aware Predictions**: Considers the meaning of words based on their order and neighboring words.

Code:

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

# Load the IMDB dataset
vocab_size = 10000       # Use top 10,000 words
max_length = 200         # Max review length (truncate/pad to this size)
```

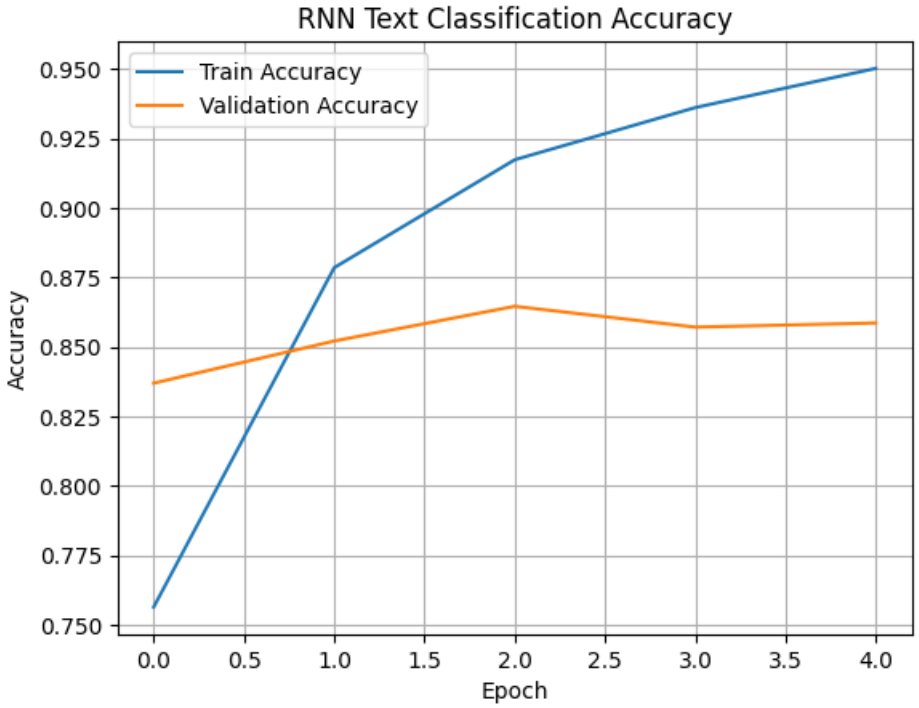
```
(x_train, y_train), (x_test, y_test) = keras.datasets.imdb.load_data(num_words=vocab_size)

# Pad sequences to ensure equal input length
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(input_dim=vocab_size,                    output_dim=32,
input_length=max_length),
   keras.layers.LSTM(64, return_sequences=True),  # First LSTM layer
   keras.layers.LSTM(32),                  # Second LSTM layer
   keras.layers.Dense(1, activation='sigmoid')    # Output layer for binary classification
])

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

# Print the model architecture
model.summary()

# Train the model
history = model.fit(x_train, y_train,
                epochs=5,
                batch_size=64,
                validation_data=(x_test, y_test),
                verbose=2)

# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f'\nTest Accuracy: {test_acc:.4f}')


#optional:

import matplotlib.pyplot as plt

plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('RNN Text Classification Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.show()
```

Output:

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	?	0 (unbuilt)
lstm_2 (LSTM)	?	0 (unbuilt)
lstm_3 (LSTM)	?	0 (unbuilt)
dense_1 (Dense)	?	0 (unbuilt)

```
Total params: 0 (0.00 B)
Trainable params: 0 (0.00 B)
Non-trainable params: 0 (0.00 B)
Epoch 1/5
391/391 - 93s - 238ms/step - accuracy: 0.7565 - loss: 0.5000 - val_accuracy: 0.8370 - val_loss: 0.3796
Epoch 2/5
391/391 - 89s - 228ms/step - accuracy: 0.8785 - loss: 0.2946 - val_accuracy: 0.8521 - val_loss: 0.3502
Epoch 3/5
391/391 - 95s - 243ms/step - accuracy: 0.9173 - loss: 0.2211 - val_accuracy: 0.8646 - val_loss: 0.3809
Epoch 4/5
391/391 - 89s - 226ms/step - accuracy: 0.9360 - loss: 0.1746 - val_accuracy: 0.8571 - val_loss: 0.3755
Epoch 5/5
391/391 - 85s - 217ms/step - accuracy: 0.9501 - loss: 0.1416 - val_accuracy: 0.8586 - val_loss: 0.3790
782/782 - 33s - 42ms/step - accuracy: 0.8586 - loss: 0.3790

Test Accuracy: 0.8586
```



Problem No:07

Problem Name: Write a program to evaluate a Transformer model for text classification.

Theory

Text classification is the process of assigning predefined categories or labels to textual data. It is a core task in Natural Language Processing (NLP) with applications in spam filtering, sentiment analysis, news categorization, and customer feedback analysis. Traditional neural architectures such as Recurrent Neural Networks (RNNs) and Long Short-Term Memory networks (LSTMs) were commonly used for this task, as they can process sequential data. However, these models struggle with long-range dependencies and lack parallelization capabilities.

To overcome these limitations, the Transformer architecture was introduced by Vaswani et al. in 2017. The Transformer model has revolutionized NLP by enabling highly parallel training and achieving state-of-the-art performance across many tasks, including text classification.

What is a Transformer?

A Transformer is a deep learning architecture based solely on self-attention mechanisms and feedforward layers. Unlike RNNs, which process data sequentially, Transformers process entire sequences at once, allowing them to capture long-range dependencies more effectively and train faster on GPUs.

Key Components of Transformer Architecture:

1. **Input Embedding**:

   Each word is converted into a vector using an embedding layer.

   Positional encoding is added to retain the order of words since Transformers do not have inherent sequence-awareness like RNNs.

2. **Self-Attention Mechanism**:

   Allows the model to weigh the importance of different words in the input sequence relative to each other.

   Calculates attention scores for every pair of words to model context.

3. **Multi-Head Attention**:

   Multiple attention heads operate in parallel to capture different relationships in the input.

   Their outputs are concatenated and linearly transformed.

4. **Feedforward Layers**:

   Position-wise dense layers that process the output of the attention mechanism.

Typically includes normalization and dropout for better generalization.

5. **Output Layer**:
    - For classification tasks, the final hidden state (often from the [CLS] token) is passed through a dense layer followed by a softmax or sigmoid activation to predict class probabilities.

Advantages of Transformer Models

- **Parallelization**: Unlike RNNs, Transformers can process sequences in parallel, leading to faster training.
- **Long-Range Dependencies**: Attention mechanisms capture relationships between distant words better than sequential models.
- **State-of-the-Art Performance**: Transformers like BERT, GPT, and RoBERTa outperform traditional models in many benchmarks.
- **Pretrained Models**: Transformers can be pre-trained on massive datasets and fine-tuned for specific tasks, making them highly effective even with limited labeled data.

Transformer in Text Classification

To apply a Transformer for text classification:

1. **Preprocessing**: Tokenize input text and add special tokens
2. **Embedding**: Convert tokens into vector representations using pre-trained embeddings.
3. **Transformer Layers**: Apply self-attention and feedforward transformations to contextualize the inputs.
4. **Classification Head**: Use the representation of the [CLS] token or pooled output to classify the text into categories.

Popular transformer models used in text classification:

- **BERT (Bidirectional Encoder Representations from Transformers)**
- **DistilBERT** (lighter version of BERT)
- **RoBERTa**
- **ALBERT**

Code:

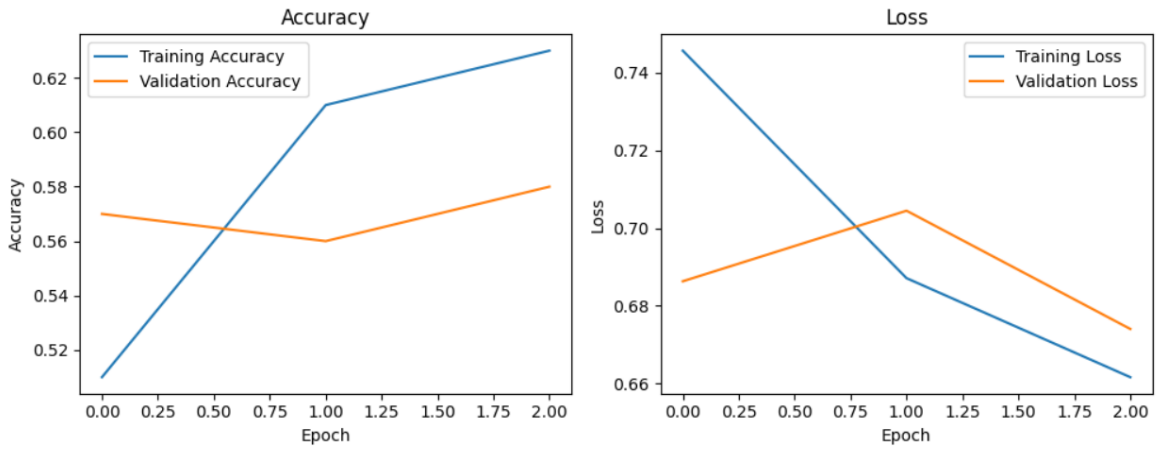
```
#  Install required libraries (uncomment if needed)
# !pip install tensorflow tensorflow-hub tensorflow-text tensorflow-datasets matplotlib scikit-learn

import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_text as text
import tensorflow_datasets as tfds
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, classification_report, roc_curve, auc

# Load a SMALL subset of the IMDB dataset
(train_data_full, test_data_full), info = tfds.load(
    'imdb_reviews',
    split=['train', 'test'],
    as_supervised=True,
    with_info=True
)

# Take smaller samples for faster training
train_data = train_data_full.take(200)
test_data = test_data_full.take(100)

# Load BERT preprocessing and encoder models
bert_preprocess_url = "https://tfhub.dev/tensorflow/bert_en_uncased_preprocess/3"
bert_encoder_url = "https://tfhub.dev/tensorflow/bert_en_uncased_L-12_H-768_A-12/3"

bert_preprocess = hub.KerasLayer(bert_preprocess_url, name="bert_preprocessing")
bert_encoder = hub.KerasLayer(bert_encoder_url, trainable=False, name="bert_encoder")

#  Build the BERT-based classification model
text_input = tf.keras.layers.Input(shape=(), dtype=tf.string, name='text')
preprocessed_text = bert_preprocess(text_input)
bert_output = bert_encoder(preprocessed_text)['pooled_output']
dense = tf.keras.layers.Dense(128, activation='relu')(bert_output)
dropout = tf.keras.layers.Dropout(0.3)(dense)
final_output = tf.keras.layers.Dense(1, activation='sigmoid')(dropout)

model = tf.keras.Model(inputs=text_input, outputs=final_output)

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

#  Prepare batched datasets
BATCH_SIZE = 8
```

```
train_ds = train_data.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
test_ds = test_data.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)

# Train for just 3 epochs
history = model.fit(
    train_ds,
    validation_data=test_ds,
    epochs=3
)

#  Evaluate the model
loss, accuracy = model.evaluate(test_ds)
print(f"\n Test Accuracy: {accuracy:.4f}")

#  Plot training history
history_dict = history.history
plt.figure(figsize=(10, 4))

# Accuracy
plt.subplot(1, 2, 1)
plt.plot(history_dict['accuracy'], label='Training Accuracy')
plt.plot(history_dict['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

# Loss
plt.subplot(1, 2, 2)
plt.plot(history_dict['loss'], label='Training Loss')
plt.plot(history_dict['val_loss'], label='Validation Loss')
plt.title('Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()

# Make predictions
y_true = []
y_pred = []
y_prob = []

for x_batch, y_batch in test_ds:
    preds = model.predict(x_batch)
    y_prob += preds.flatten().tolist()
    y_pred += (preds > 0.5).astype("int").flatten().tolist()
    y_true += y_batch.numpy().tolist()
```

```
#  Confusion Matrix
cm = confusion_matrix(y_true, y_pred)
ConfusionMatrixDisplay(cm, display_labels=["Negative", "Positive"]).plot()
plt.title("Confusion Matrix")
plt.grid(False)
plt.show()

# Classification Report
print("\n Classification Report:\n")
print(classification_report(y_true, y_pred, target_names=["Negative", "Positive"]))

#  ROC Curve and AUC
fpr, tpr, _ = roc_curve(y_true, y_prob)
roc_auc = auc(fpr, tpr)

plt.figure()
plt.plot(fpr, tpr, label=f'ROC Curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], linestyle='--', color='gray')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.grid(True)
plt.show()
```

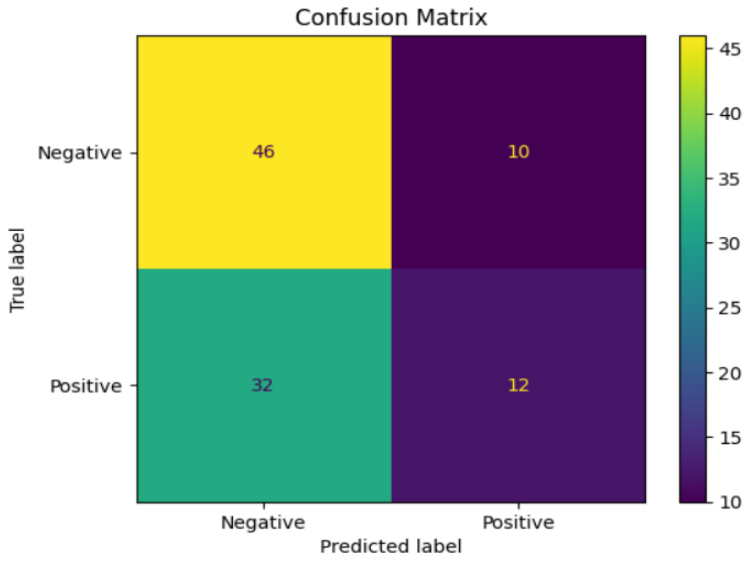
Output:

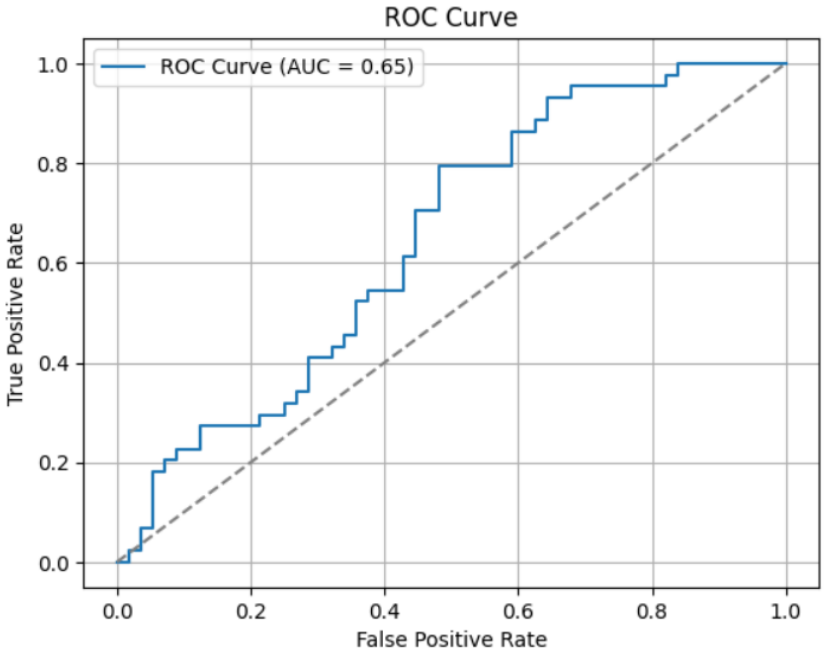
```
Epoch 1/3
25/25 [==============================] - 164s 6s/step - loss: 0.7456 - accuracy: 0.5100 - val_loss: 0.6863 - val_accuracy: 0.5700
Epoch 2/3
25/25 [==============================] - 147s 6s/step - loss: 0.6871 - accuracy: 0.6100 - val_loss: 0.7045 - val_accuracy: 0.5600
Epoch 3/3
25/25 [==============================] - 148s 6s/step - loss: 0.6617 - accuracy: 0.6300 - val_loss: 0.6741 - val_accuracy: 0.5800
13/13 [==============================] - 50s 4s/step - loss: 0.6741 - accuracy: 0.5800

✅ Test Accuracy: 0.5800
```



✅ Classification Report:

	precision	recall	f1-score	support
Negative	0.59	0.82	0.69	56
Positive	0.55	0.27	0.36	44
accuracy			0.58	100
macro avg	0.57	0.55	0.53	100
weighted avg	0.57	0.58	0.54	100





Problem No:08

Problem Name: Write a program to evaluate Generative adversarial Network (GAN) for image generation.

Theory:

Generative Adversarial Networks (GANs) represent one of the most innovative breakthroughs in deep learning. Proposed by Ian Goodfellow in 2014, GANs are powerful generative models that can learn to mimic any distribution of data. They are particularly known for generating realistic images, including human faces, artwork, and even synthetic data that closely resembles real-world samples.

GANs belong to the family of unsupervised learning techniques and operate on a unique idea of two neural networks — the Generator and the Discriminator — competing in a zero-sum game.

What is a GAN?

A Generative Adversarial Network (GAN) consists of two primary components:

a. Generator (G):

- Takes in random noise (usually sampled from a normal or uniform distribution).
- Learns to map this noise into realistic data (e.g., images).
- Its goal is to generate data that is indistinguishable from real data.

b. Discriminator (D):

- Takes in both real data and fake data (generated by the generator).
- Learns to correctly classify whether the input is real or generated.
- Acts like a binary classifier (real vs. fake).

How GANs Work (Game Theory Perspective)

GANs work using a **minimax game** between the Generator and the Discriminator:

- The Generator tries to maximize the Discriminator's error (fool it).
- The Discriminator tries to minimize the error (correctly identify real vs. fake).

The objective function is:

$$\min_G \max_D \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

x: real data

z: random noise vector

G(z): fake data from generator

D(x): probability that x is real

Training a GAN

GANs are trained in alternating steps:

1. **Train Discriminator**: Show real and generated samples and train it to classify correctly.
2. **Train Generator**: Update generator parameters so that it can fool the discriminator.

This adversarial process continues until the generator produces samples that the discriminator can no longer reliably distinguish from real data.

Challenges in Training GANs

- **Mode Collapse**: The generator produces limited variety of outputs.
- **Non-convergence**: Oscillatory behavior due to the adversarial dynamics.
- **Training Instability**: Sensitive to learning rate and network architecture.

To address these, variants like **DCGAN**, **WGAN**, **LSGAN**, etc., have been introduced.

Applications of GANs

- Image synthesis (faces, art, scenery)
- Super-resolution (enhancing image quality)
- Image-to-image translation (e.g., black-and-white to color)
- Data augmentation
- Deepfake generation

Code:

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
import os

# Load and preprocess the MNIST dataset
(x_train, _), (_, _) = keras.datasets.mnist.load_data()
x_train = (x_train.astype(np.float32) - 127.5) / 127.5  # Normalize to [-1, 1]
x_train = np.expand_dims(x_train, axis=-1)  # Shape: (60000, 28, 28, 1)
```

```
# Hyperparameters
latent_dim = 100
batch_size = 128
epochs = 100
sample_interval = 1000

# Create output directory for generated images
os.makedirs("generated_images", exist_ok=True)

# Build the Generator
def build_generator():
    model = keras.Sequential([
        keras.layers.Dense(256, activation="relu", input_dim=latent_dim),
        keras.layers.BatchNormalization(),
        keras.layers.Dense(512, activation="relu"),
        keras.layers.BatchNormalization(),
        keras.layers.Dense(1024, activation="relu"),
        keras.layers.BatchNormalization(),
        keras.layers.Dense(28 * 28 * 1, activation="tanh"),
        keras.layers.Reshape((28, 28, 1))
    ])
    return model

# Build the Discriminator
def build_discriminator():
    model = keras.Sequential([
        keras.layers.Flatten(input_shape=(28, 28, 1)),
        keras.layers.Dense(512, activation="relu"),
        keras.layers.Dense(256, activation="relu"),
        keras.layers.Dense(1, activation="sigmoid")
    ])
    return model

# Instantiate models
generator = build_generator()
discriminator = build_discriminator()
discriminator.compile(loss="binary_crossentropy",
optimizer=keras.optimizers.Adam(0.0002, 0.5), metrics=["accuracy"])

# Combine into GAN
discriminator.trainable = False
gan_input = keras.Input(shape=(latent_dim,))
gan_output = discriminator(generator(gan_input))
gan = keras.Model(gan_input, gan_output)
gan.compile(loss="binary_crossentropy", optimizer=keras.optimizers.Adam(0.0002, 0.5))

# Lists to store training metrics
d_losses = []
d_accuracies = []
g_losses = []
```

```
# Function to save generated images
def save_generated_images(epoch, show=True, save=True):
    noise = np.random.normal(0, 1, (16, latent_dim))
    gen_imgs = generator.predict(noise)
    gen_imgs = 0.5 * gen_imgs + 0.5  # Scale from [-1,1] to [0,1]

    fig, axs = plt.subplots(4, 4, figsize=(4, 4))
    count = 0
    for i in range(4):
        for j in range(4):
            axs[i, j].imshow(gen_imgs[count, :, :, 0], cmap="gray")
            axs[i, j].axis("off")
            count += 1
    fig.tight_layout()

    if save:
        plt.savefig(f"generated_images/mnist_{epoch}.png")

    if show:
        plt.show()
    else:
        plt.close()


# Training Loop
for epoch in range(epochs):
    # Train Discriminator
    idx = np.random.randint(0, x_train.shape[0], batch_size)
    real_imgs = x_train[idx]
    noise = np.random.normal(0, 1, (batch_size, latent_dim))
    fake_imgs = generator.predict(noise)

    d_loss_real = discriminator.train_on_batch(real_imgs, np.ones((batch_size, 1)))
    d_loss_fake = discriminator.train_on_batch(fake_imgs, np.zeros((batch_size, 1)))
    d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

    # Train Generator
    noise = np.random.normal(0, 1, (batch_size, latent_dim))
    g_loss = gan.train_on_batch(noise, np.ones((batch_size, 1)))

    # Store losses and accuracy
    d_losses.append(d_loss[0])
    d_accuracies.append(d_loss[1])
    g_losses.append(g_loss)

    # Print progress
    if epoch % sample_interval == 0:
        print(f"{epoch} [D loss: {d_loss[0]:.4f} | D acc: {100 * d_loss[1]:.2f}%] [G loss:
{g_loss:.4f}]")
```

```
        save_generated_images(epoch)

print("GAN Training Complete!")

# Plot losses and accuracy
plt.figure(figsize=(12, 4))

plt.subplot(1, 3, 1)
plt.plot(d_losses, label="Discriminator Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Discriminator Loss")
plt.legend()

plt.subplot(1, 3, 2)
plt.plot(g_losses, label="Generator Loss", color='orange')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Generator Loss")
plt.legend()

plt.subplot(1, 3, 3)
plt.plot(d_accuracies, label="Discriminator Accuracy", color='green')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("Discriminator Accuracy")
plt.legend()

plt.tight_layout()
plt.show()
```

Output:

```
1/1 [==============================] - 0s 174ms/step
```