INDEX

Experiment No: 01

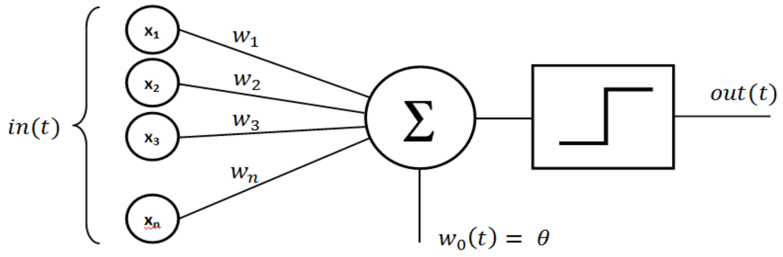
Experiment Name: Write a program to evaluate AND function with bipolar input and targets and also show the convergence curves and the decision boundary lines.

Objective:

- ❖ To implement a perceptron learning algorithm using bipolar inputs and targets for the AND logic function.
- ❖ To plot the convergence curve showing training accuracy over epochs.
- ❖ To visualize the learned decision boundary that separates the output classes.

Theory:

Perceptron Model:
A perceptron is a simple type of artificial neural network used for binary classification. It is composed of input nodes connected to an output neuron via weighted connections. It computes a weighted sum of the inputs, adds a bias, and passes the result through an activation function.



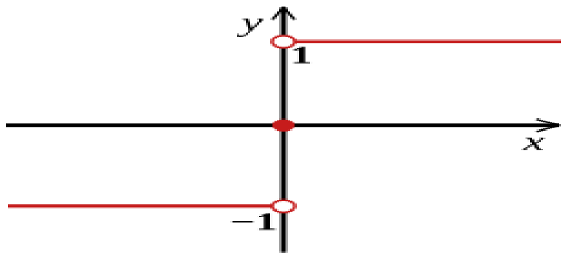
Bipolar Inputs and Targets:
In bipolar representation, the input and target values are either -1 or +1, unlike binary representation (0 and 1). For the AND function in bipolar form:

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

Activation Function – Bipolar Step Function:
The bipolar step function (also called sign function) is defined as:

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

Figure shows the sign function,



This function is used to map the net input to bipolar output.

Learning Rule:
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.

The algorithm continues updating weights until all training samples are correctly classified or a maximum number of epochs is reached.

Convergence Curve:
The convergence curve is a plot of accuracy (or error) versus the number of epochs. It helps in analyzing how quickly and effectively the perceptron is learning.

Decision Boundary:
In a 2D input space, the decision boundary is a line defined by the equation:

$$w_1x_1 + w_2x_2 + b = 0$$

It separates the input space into two regions corresponding to the two output classes.

Code:

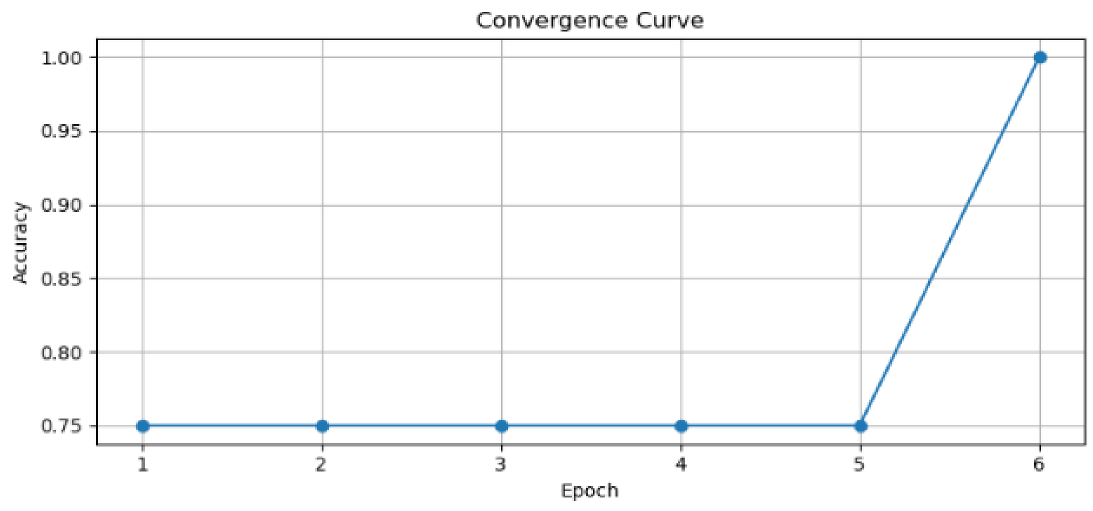
```
import numpy as np
import matplotlib.pyplot as plt

# Bipolar activation function
```

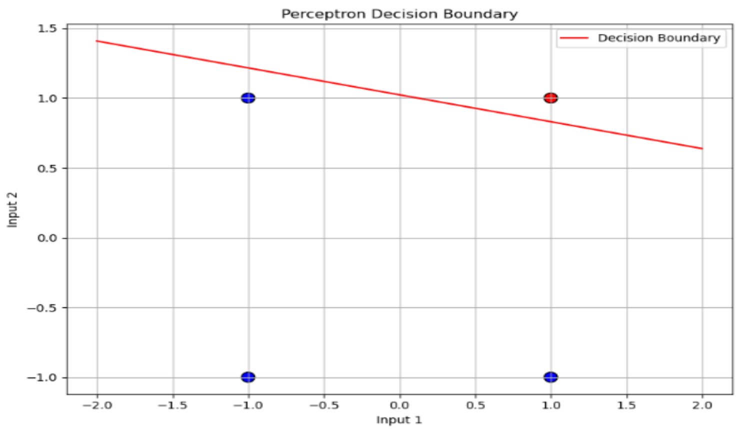
```
def bipolar_activation(x):
    return 1 if x >= 0 else -1

# 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 logic)
    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()
plt.tight_layout()
plt.show()

# Plot the decision boundary and data points
plt.figure(figsize=(8, 6))
plt.plot(x, y, label='Decision Boundary', color='red')
plt.scatter(inputs[:, 0], inputs[:, 1], c=targets, cmap='bwr', s=100, edgecolors='k')
plt.xlabel('Input 1')
plt.ylabel('Input 2')
plt.title('Perceptron Decision Boundary')
plt.legend()
plt.grid()
plt.tight_layout()
plt.show()
```

Output:

```
Converged in 6 epochs.
```




Perceptron Decision Boundary
Decision Boundary
Input 1
Input 2
1.5
1.0
0.5
0.0
−0.5
−1.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

Experiment No: 02

Experiment Name: Write a program to evaluate X-OR function and also show the convergence and the decision boundary.

Objective:

- ❖ To implement a multi-layer neural network that can learn the non-linearly separable XOR function.
- ❖ To analyze convergence by plotting the error reduction over training epochs.
- ❖ To visualize the decision regions and boundaries formed by the neural network.

Theory:

The XOR (Exclusive OR) function is a fundamental concept in digital logic and neural network studies. It serves as a benchmark problem for understanding the limitations of linear models and the power of multi-layer neural networks in solving non-linearly separable problems.

Understanding the XOR Function

The XOR function takes two binary inputs and outputs 1 if the inputs are different, and 0 if they are the same. Its truth table is as follows:

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

This simple logic gate highlights an important issue: the XOR function is not linearly separable**.** This means it is impossible to draw a single straight line that separates the two output classes (0 and 1) in the input space. As a result, a simple perceptron or single-layer neural network cannot model the XOR function.

Solving XOR with a Multi-Layer Neural Network

To solve the XOR problem, we use a multi-layer perceptron (MLP) consisting of:

- ➢ An input layer with two neurons (representing the inputs A and B),
- ➢ A hidden layer with at least two neurons and non-linear activation functions (such as sigmoid),
- ➢ An output layer with one neuron (producing the final output).

The non-linearity introduced by the hidden layer allows the model to learn a more complex decision boundary that can correctly separate the XOR outputs.

Training Using Backpropagation

The network is trained using the backpropagation algorithm with gradient descent optimization. During training:

1. A forward pass computes the outputs based on current weights and biases.
2. The error is calculated between the predicted and actual outputs.
3. A backward pass adjusts the weights using the error gradients.

This process is repeated for several iterations (epochs), and the convergence of the training process is monitored using a loss function, typically the sum of squared errors. As the training progresses, the total error should decrease, indicating that the network is learning.

Visualizing the Convergence and Decision Boundary

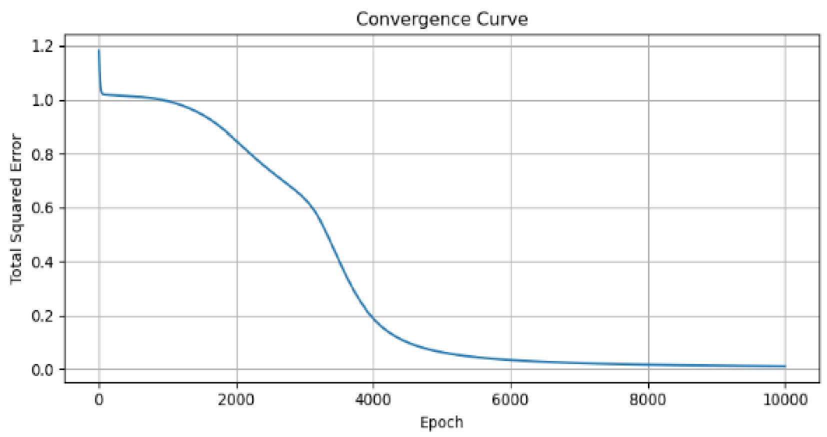
- The convergence curve shows how the training error reduces over epochs. It helps in analyzing how quickly and effectively the model learns.
- The decision boundary visualizes how the trained model separates the input space into different output classes. For XOR, the boundary is non-linear and typically consists of two regions dividing the input quadrants in a way that a linear model cannot achieve.

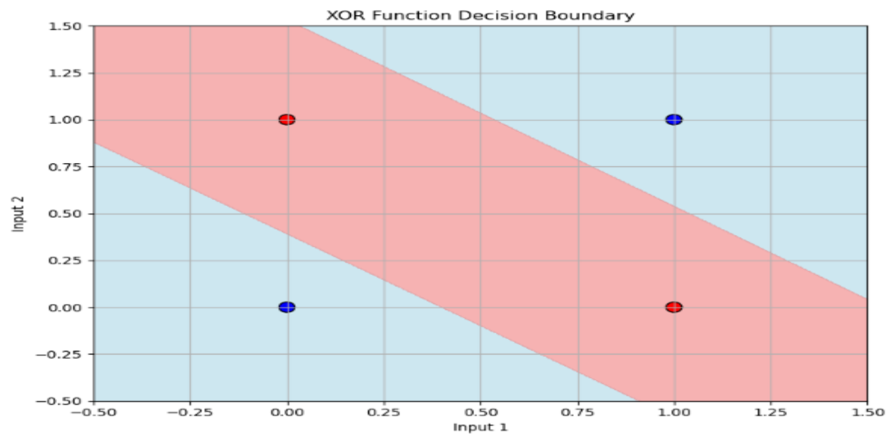
Code:

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

# Activation functions
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

# Input and target data for XOR
inputs = np.array([[0, 0],
                   [0, 1],
                   [1, 0],
                   [1, 1]])
targets = np.array([[0], [1], [1], [0]])  # reshaped to column vector

# 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_sample = inputs[i]
        target_sample = targets[i]

        hidden_input = np.dot(input_sample, weights_input_hidden) + bias_hidden
        hidden_layer_output = sigmoid(hidden_input)

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

        # Compute error
        output_error = target_sample - final_output
        total_error += np.sum(output_error**2)

        # Backpropagation
        output_delta = output_error * sigmoid_derivative(final_output)
        hidden_error = output_delta.dot(weights_hidden_output.T)
        hidden_delta = hidden_error * sigmoid_derivative(hidden_layer_output)

        # Update weights and biases
        weights_hidden_output += np.outer(hidden_layer_output, output_delta) *
learning_rate
        bias_output += output_delta * learning_rate

        weights_input_hidden += np.outer(input_sample, hidden_delta) * learning_rate
        bias_hidden += hidden_delta * learning_rate
    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)
plt.xlabel('Epoch')
plt.ylabel('Total Squared Error')
plt.title('Convergence Curve')
plt.grid()
plt.tight_layout()
plt.show()

# Function to predict output for grid points
def predict(x1, x2):
    x = np.c_[x1.ravel(), x2.ravel()]
    hidden_input = np.dot(x, 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 for decision boundary plot
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.contourf(X1, X2, Z, levels=[0, 0.5, 1], colors=["lightblue", "lightcoral"], alpha=0.6)
plt.scatter(inputs[:, 0], inputs[:, 1], c=targets.flatten(), cmap='bwr', edgecolors='k', s=100)
plt.xlabel('Input 1')
plt.ylabel('Input 2')
plt.title('XOR Function Decision Boundary')
plt.grid()
plt.tight_layout()
plt.show()
```

Output:


Convergence Curve
Total Squared 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
Input 1
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

Experiment No: 03

Experiment 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]

Objective:

- ❖ To understand and apply the Stochastic Gradient Descent (SGD) algorithm using the Delta learning rule.
- ❖ To train a perceptron to solve a binary classification task with linearly separable data.
- ❖ To visualize the convergence behavior by plotting total error across epochs.

Theory:

The Perceptron Model

A perceptron is the simplest type of artificial neural network, consisting of a single neuron. It computes a weighted sum of the input features and passes the result through a step (threshold) activation function to produce a binary output.

The general formula for the perceptron output is:

$$y = \text{step}\ (w \cdot x)$$

Where:

- w is the weight vector,
- x is the input vector (including the bias as an additional input),
- The step function outputs 1 if the weighted sum is ≥ 0, otherwise 0.

Delta Learning Rule

The Delta rule, also known as the Widrow-Hoff rule, is a supervised learning rule used for adjusting the weights in a perceptron. It updates the weights based on the error between the predicted output and the target output. The update rule is:

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

Where:

- η is the learning rate (a small positive constant),
- d is the desired (target) output,

- y is the predicted output,
- x is the input vector.

This rule drives the weights in a direction that minimizes the error on each sample.

Stochastic Gradient Descent (SGD)

In SGD**,** weights are updated after each training sample, rather than after the entire dataset (as in batch gradient descent). This often leads to faster convergence and helps the model escape local minima in more complex problems.

In each epoch:

- The data samples are shuffled.
- The network processes each sample individually and updates the weights immediately using the Delta rule.

Convergence and Learning Behavior

During training, the total error is recorded for each epoch to visualize the learning process. The model is said to have converged when the total error for all samples becomes zero, meaning the perceptron has correctly classified all input samples.

The convergence curve shows the total error per epoch, helping to understand how quickly the network learns and whether it stabilizes.

Dataset and Learning Goal

The input and target pairs used in this experiment are:

$$\mathbf{X}_{\text{input}} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \quad \mathbf{D}_{\text{target}} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

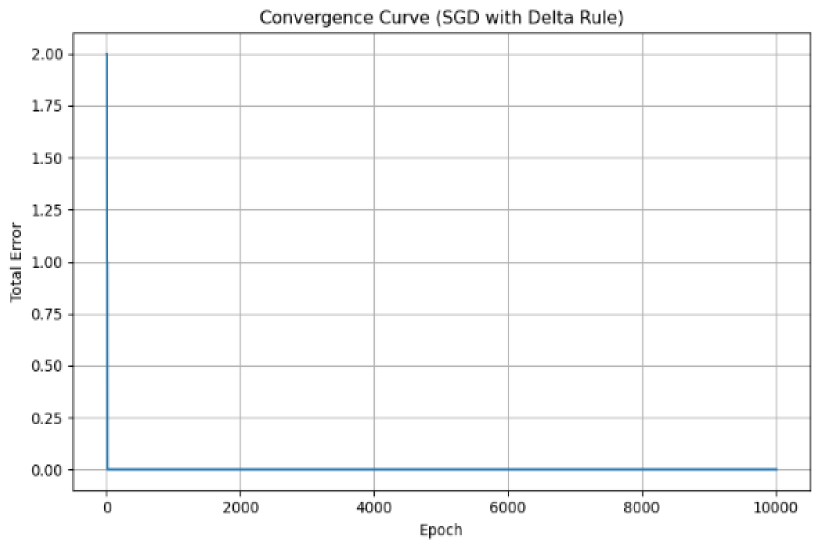
This setup corresponds to a linearly separable function, allowing a perceptron to successfully learn the correct mapping using the Delta rule and SGD.

Code:

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

# Input data (bias included as third column)
Xinput = np.array([
    [0, 0, 1],
    [0, 1, 1],
    [1, 0, 1],
    [1, 1, 1]
])

# Target outputs (OR function)
Dtarget = np.array([0, 0, 1, 1])

# Random initialization of weights (3 for 2 inputs + bias)
weights = np.random.randn(3)

# Hyperparameters
learning_rate = 0.1
epochs = 10010

# Step activation function
def step_function(x):
    return np.where(x >= 0, 1, 0)

# For tracking convergence over time
convergence_curve = []
converged = False

# Training loop
for epoch in range(epochs):
    total_error = 0

    # Shuffle the data (for stochastic training)
    indices = np.random.permutation(len(Xinput))
    Xinput_shuffled = Xinput[indices]
    Dtarget_shuffled = Dtarget[indices]

    # Loop through each training sample
    for i in range(len(Xinput)):
        x = Xinput_shuffled[i]
        d = Dtarget_shuffled[i]

        # Compute perceptron output
        y = step_function(np.dot(x, weights))

        # Calculate the error
        error = d - y
        total_error += abs(error)

        # Apply the Delta rule
        weights += learning_rate * error * x

    # Track error per epoch
    convergence_curve.append(total_error)

    # Check for convergence
    if total_error == 0 and not converged:
        print(f"Converged in {epoch + 1} epochs.")
        converged = True

# Display final weights
print("Final weights:", weights)

# Plot convergence curve
plt.figure(figsize=(8, 5))
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(True)
plt.tight_layout()
plt.show()
```

Output:

```
Converged in 13 epochs.
Final weights: [ 1.69375463  0.60729248 -0.65523283]
```



Experiment No: 04

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

Objective:

- ❖ To load and preprocess the MNIST dataset for training a neural network.
- ❖ To design and train a feedforward neural network for multi-class digit classification.
- ❖ To evaluate model performance using accuracy metrics and visualize training loss.

Theory:

A Feedforward Neural Network (FNN) is one of the most basic types of artificial neural networks where the connections between the nodes do not form a cycle. This type of network is often used for supervised learning tasks like image classification, pattern recognition, and regression.

MNIST Dataset:

The MNIST (Modified National Institute of Standards and Technology) dataset is a benchmark dataset in machine learning. It contains 70,000 grayscale images of handwritten digits (0–9), each of size 28x28 pixels, resulting in 784 features per image**.**

- ➢ Features (X): Pixel values of the image (flattened to 784-dimensional vectors).
- ➢ Labels (y): Corresponding digit values from 0 to 9.

Data Preprocessing:

Before training, the dataset is preprocessed:

- ➢ Normalization: Pixel values are scaled between 0 and 1 by dividing by 255. This helps improve the convergence speed of the model.
- ➢ Train-Test Split: The dataset is split into a training set (80%) and a test set (20%) to evaluate the model's generalization.

Neural Network Model (MLPClassifier):

The neural network used in this experiment is implemented using MLPClassifier from Scikit-learn. It is a type of Multilayer Perceptron that supports backpropagation.

- ➢ Hidden Layer Size: One hidden layer with 128 neurons.
- ➢ Activation Function: ReLU (Rectified Linear Unit), which introduces non-linearity and helps avoid the vanishing gradient problem.
- ➢ Solver: ‘adam’, an efficient stochastic gradient-based optimizer.
- ➢ Max Iterations: The model is trained for 10 epochs.

Model Training and Evaluation:

- The model is trained using the training dataset (X_train, y_train).
- After training, predictions are made on the test dataset.
- **Accuracy Score:** The performance is measured using accuracy — the ratio of correct predictions to total predictions.

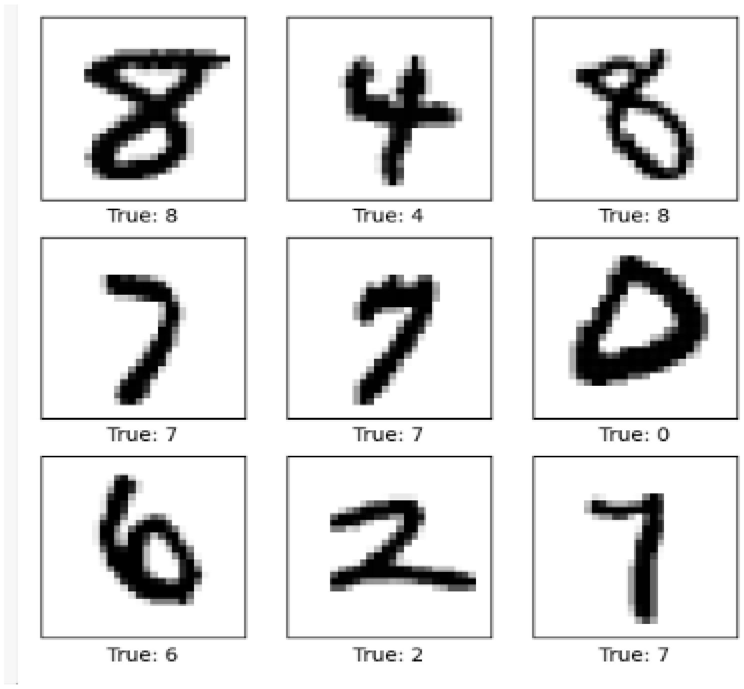
Visualization:

- A sample of the test images is displayed alongside the model's predictions.
- Each digit is reshaped from a 784-dimensional vector to a 28x28 pixel image for visualization using matplotlib.

Code:

```
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.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score

# Load the MNIST dataset
mnist = fetch_openml('mnist_784', version=1)
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)

# Initialize and train the neural network model
model = MLPClassifier(hidden_layer_sizes=(128,), activation='relu', solver='adam',
max_iter=10, random_state=42)
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.iloc[i], X_test.iloc[i])
plt.show()
```

Output:



Experiment No: 05

Experiment Name: Write a program to evaluate a Convolutional Neural Network (CNN) for image classification.

Objective:

- ❖ To construct a CNN architecture suitable for image classification tasks.
- ❖ To train the CNN on a standard image dataset (e.g., CIFAR-10 or MNIST).
- ❖ To evaluate the model performance using metrics like accuracy and confusion matrix.

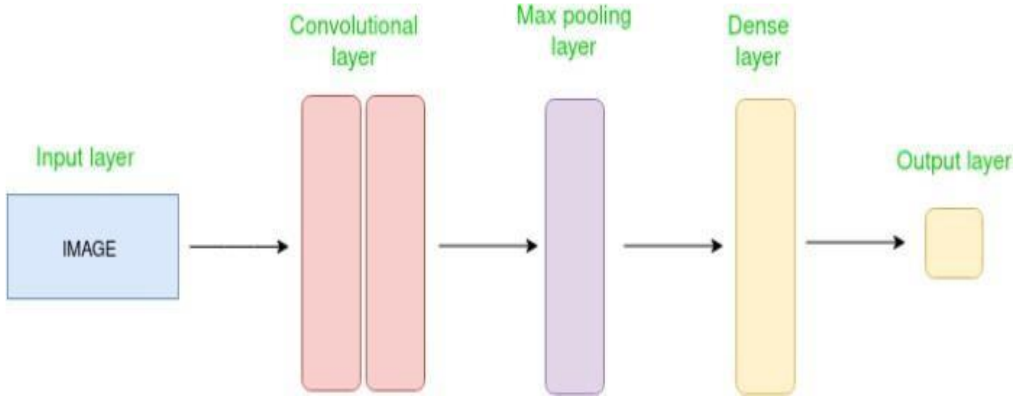
Theory:

Introduction to Image Classification

Image classification is a fundamental task in computer vision where the goal is to categorize images into predefined classes. For example, identifying whether an image contains a cat, dog, car, or airplane. Traditional machine learning algorithms struggle with high-dimensional image data due to the spatial structure and large number of features (pixels). To address this, deep learning techniques such as Convolutional Neural Networks (CNNs) are used for efficient and accurate image classification.

Convolutional Neural Network (CNN)

A CNN is a specialized type of neural network designed for processing grid-like data such as images. CNNs leverage the spatial structure of image data by applying convolutional layers that use filters (kernels) to detect local patterns (e.g., edges, textures, shapes) and preserve spatial relationships.



The key layers in a CNN include:

- ➢ Convolutional Layer: Applies filters over the input image to produce feature maps that highlight specific patterns.
- ➢ Activation Function (ReLU): Introduces non-linearity to allow learning of complex patterns.

- Pooling Layer: Reduces the spatial dimensions of the feature maps (downsampling) to decrease computation and control overfitting.
- Flatten Layer: Converts the 2D feature maps into a 1D feature vector to feed into fully connected layers.
- Fully Connected (Dense) Layer: Acts like a traditional neural network to perform classification based on the extracted features.
- Output Layer (Softmax): Produces a probability distribution over the target classes for multi-class classification.

CIFAR-10 Dataset

In this experiment, the CIFAR-10 dataset is used. It is a benchmark dataset for image classification and consists of 60,000 color images (32×32 pixels) categorized into 10 different classes such as: Airplane, Car, Bird, Cat, Deer, Dog, Frog, Horse, Ship, Truck.

The dataset is divided into:

- 50,000 training images
- 10,000 test images

Model Architecture Used

The CNN architecture implemented in this experiment includes the following layers:

- Conv2D (32 filters, 3×3 kernel) + ReLU + MaxPooling2D
- Conv2D (64 filters, 3×3) + ReLU + MaxPooling2D
- Conv2D (64 filters, 3×3) + ReLU
- Flatten layer to convert to 1D
- Dense (64 neurons) + ReLU
- Dense (10 neurons) + Softmax (for 10-class output)

This structure progressively extracts features and learns representations that help in classifying the input images accurately.

Training and Evaluation

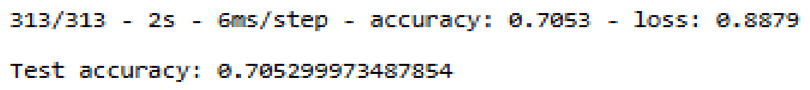
- The model is compiled using the Adam optimizer**,** known for its adaptive learning rate and efficient gradient-based optimization.
- The loss function used is sparse categorical crossentropy**,** suitable for multi-class classification problems with integer labels.
- The model is trained for 10 epochs, and validation is performed using the test set.

After training, the model’s performance is evaluated using accuracy on the test set, giving an indication of its generalization capability.

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 the dataset
x_train, x_test = x_train / 255.0, x_test / 255.0

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

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

# Train the model
model.fit(x_train, y_train, epochs=10, 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:



```
313/313 - 2s - 6ms/step - accuracy: 0.7053 - loss: 0.8879

Test accuracy: 0.705299973487854
```

Experiment No: 06

Experiment Name: Write a program to evaluate a Recurent Neural Network (RNN) for text classification.

Objective:

- ❖ To preprocess text data and convert it into sequences suitable for RNN input.
- ❖ To build and train an RNN model (e.g., using LSTM or GRU) for classifying text.
- ❖ To analyze the model's performance using accuracy, precision, and recall.

Theory:

Introduction to Text Classification:

Text classification is a common Natural Language Processing (NLP) task where a piece of text is assigned a category. One popular application is sentiment analysis, where the goal is to classify a text (like a movie review) as expressing positive or negative sentiment.

IMDB Dataset:

The IMDB (Internet Movie Database) dataset is a widely-used benchmark for sentiment classification. It consists of 50,000 movie reviews split evenly into training and testing datasets:

- ❖ Each review is preprocessed and represented as a sequence of integers (each integer is a word index).
- ❖ Labels: 0 = negative review, 1 = positive review.

Recurrent Neural Networks (RNNs):

Unlike feedforward neural networks, RNNs are specially designed to handle sequential data**.** They maintain a memory of previous inputs, making them ideal for text and time series analysis. However, basic RNNs suffer from vanishing gradient problems.

To overcome this, we use a more advanced variant:

Long Short-Term Memory (LSTM):

LSTM is a type of RNN architecture that is capable of learning long-term dependencies and is widely used for NLP tasks.

- ❖ LSTM units have memory cells**,** input**,** output, and forget gates that regulate the flow of information.
- ❖ LSTM can learn which data in a sequence is important to keep or discard.

Model Architecture:

The RNN model in this experiment is built using the Keras Sequential API:

- Embedding Layer**:** Converts word indices into dense vectors of fixed size (32). Helps the model learn word relationships.
- First LSTM Layer: Processes sequences and returns the full sequence (return_sequences=True), allowing the next LSTM layer to process it.
- Second LSTM Layer: Reduces the sequence into a fixed-size vector, summarizing the information.
- Dense Output Layer: A single neuron with a sigmoid activation function for binary classification (positive/negative).

Data Preprocessing:

- Vocabulary Limiting: Only the top 10,000 most frequent words are used to reduce dimensionality.
- Padding: Sequences are padded to a fixed length of 200 tokens to ensure uniform input size.

Training and Evaluation:

- The model is compiled with:

    - Loss function: Binary Crossentropy (used for binary classification tasks).
    - Optimizer: Adam (efficient variant of SGD).
    - Metric: Accuracy.

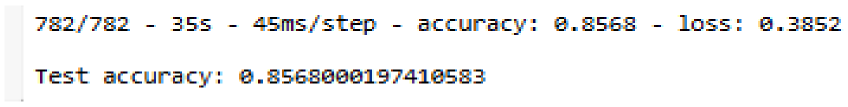
- The model is trained for 5 epochs and evaluated using the test set.
- The final accuracy is printed to assess performance.

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:



```
782/782 - 35s - 45ms/step - accuracy: 0.8568 - loss: 0.3852

Test accuracy: 0.8568000197410583
```

Experiment No: 07

Experiment Name: Write a program to evaluate a Transfer model for text classification.

Objective:

- ❖ To apply a pre-trained language model (e.g., BERT or RoBERTa) for text classification.
- ❖ To fine-tune the model on a labeled dataset for sentiment or topic classification.
- ❖ To measure the effectiveness of transfer learning using standard evaluation metrics.

Theory:

Introduction to Text Classification

Text classification is a Natural Language Processing (NLP) task that involves assigning predefined categories to textual data. Common applications include spam detection, sentiment analysis, news categorization, and topic labeling. Traditional models rely heavily on manual feature engineering and simpler models like Naive Bayes or SVM. However, with the rise of deep learning, more accurate and scalable solutions have emerged—most notably, Transformer-based models like BERT**.**

Transfer Learning in NLP

Transfer Learning refers to the process of leveraging a pre-trained model on a large corpus of data and fine-tuning it for a specific downstream task, such as text classification. This allows models to learn general language features from massive datasets (e.g., Wikipedia) and then adapt those features to solve more specific problems efficiently, even with limited labeled data.

BERT (Bidirectional Encoder Representations from Transformers)

BERT is a state-of-the-art pre-trained language model developed by Google. It is based on the Transformer architecture and is trained to understand the context of words in a sentence from both directions (left-to-right and right-to-left). This bidirectional training enables BERT to capture more nuanced relationships between words than traditional models.

Key features of BERT:

- ➢ Pre-trained on a large English corpus using masked language modeling.
- ➢ Requires only minimal architecture changes for downstream tasks like classification.
- ➢ Outputs embeddings that capture semantic meaning and context.

IMDB Dataset

In this experiment, the IMDB movie review dataset is used for binary sentiment classification (positive or negative). The dataset contains:

- ❖ 25,000 training samples

- 25,000 test samples

Each sample is a movie review labeled as either positive (1) or negative (0).

Model Architecture Used

The model in this experiment utilizes TensorFlow Hub to import a pretrained BERT model for text preprocessing and embedding generation. The key steps are:

- Preprocessing Layer**:** Uses bert_en_uncased_preprocess to tokenize, normalize, and convert text into BERT-compatible format.
- Embedding Layer**:** Extracts dense feature representations from the preprocessed text using BERT.
- Classification Layers**:** A dense layer with 128 neurons and ReLU activation. A final dense layer with sigmoid activation to output a probability score for binary classification.

Training and Evaluation

- The model is compiled with the Adam optimizer**,** binary cross-entropy loss**,** and accuracy as the metric.
- It is trained over 3 epochs using a batch size of 32.
- Accuracy on the test set is evaluated to determine the generalization performance of the model.

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
)
```

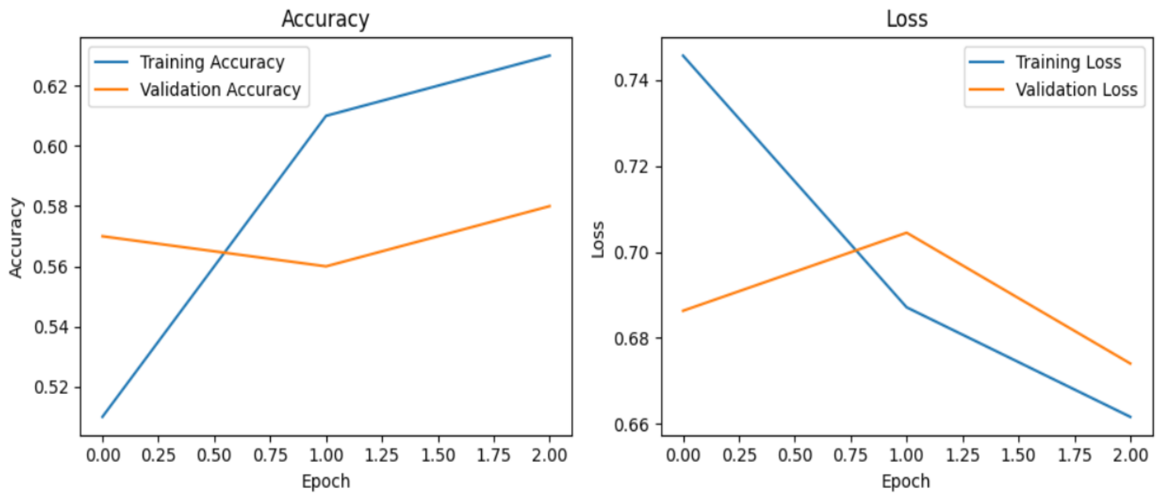
```
# Take smaller samples for faster training
train_data = train_data_full.take(200)
test_data = test_data_full.take(100)

# 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()
```

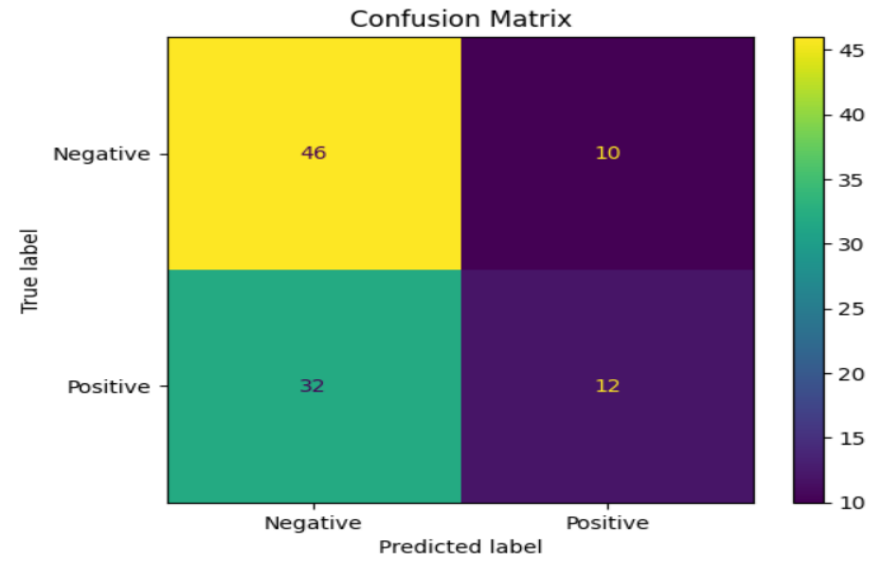
```
# Classification Report
print("\n Classification Report:\n")
print(classification_report(y_true, y_pred, target_names=["Negative", "Positive"]))

#  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
```



Experiment No: 08

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

Objective:

- ❖ To implement a basic GAN architecture consisting of generator and discriminator networks.
- ❖ To train the GAN on a dataset of images to learn the distribution of real data.
- ❖ To generate new synthetic images and evaluate their quality visually and statistically.

Theory:

Introduction to GANs

A Generative Adversarial Network (GAN) is a class of machine learning frameworks designed by Ian Goodfellow in 2014. It consists of two neural networks — the Generator and the Discriminator — that are trained simultaneously in a game-theoretic setting. The goal of a GAN is to generate new, synthetic instances of data that resemble a given training dataset.

Components of a GAN

- ➢ Generator (G)**:** Takes random noise as input and attempts to produce realistic data (e.g., images). Its goal is to "fool" the Discriminator into classifying fake data as real.
- ➢ Discriminator (D)**:** Receives both real data from the training set and fake data from the Generator, and attempts to correctly classify each as real or fake. Its goal is to distinguish genuine images from those generated.
- ➢ The two models are trained simultaneously: The Discriminator is optimized to maximize the probability of assigning correct labels. The Generator is optimized to minimize the Discriminator's ability to detect fake samples.

This adversarial process forces both models to improve over time.

Loss Functions

The training objective is a minimax loss function**:**

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

Where:

- D(x) is the probability that x is real.
- G(z) is the fake data generated from noise z.

In practice:

- Discriminator loss**:** Combines the loss of predicting real images as real and fake images as fake.
- Generator loss**:** Encourages G to generate samples that D classifies as real.

Dataset: MNIST

The MNIST dataset is a well-known dataset of handwritten digits (0–9). It contains:

- 60,000 training images
- 10,000 test images.Each image is a 28×28 grayscale image.

In this experiment, we use only the training images to train the GAN to generate new digit images.

Network Architectures

Generator Architecture:

- Starts with a 100-dimensional noise vector.
- Uses Dense, BatchNorm, and LeakyReLU layers.
- Upsamples via Conv2DTranspose layers to generate a 28×28×1 image.
- Uses a tanh activation to output values in [-1, 1].

Discriminator Architecture:

- Convolutional layers with LeakyReLU activations.
- Includes dropout to prevent overfitting.
- Ends with a Dense layer producing a single output (real or fake).

Training Process

- Random noise is sampled and fed into the Generator to produce synthetic images.
- Both real and generated images are passed to the Discriminator.
- The Discriminator and Generator are trained using gradients derived from their respective losses.
- Over time, the Generator improves in creating images that the Discriminator cannot distinguish from real ones.

The training loop includes:

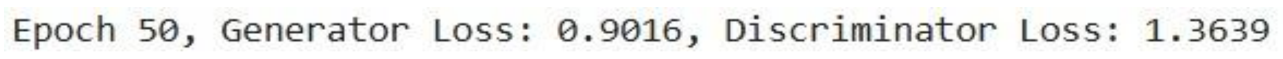
- Batch-wise updates for both networks.
- Logging of Generator and Discriminator losses per epoch.
- Visualization of generated samples every few epochs to monitor learning progress.

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: 0.9016, Discriminator Loss: 1.3639
```