INDEX

Problem No.	Problem Name
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.

Experiment No: 01

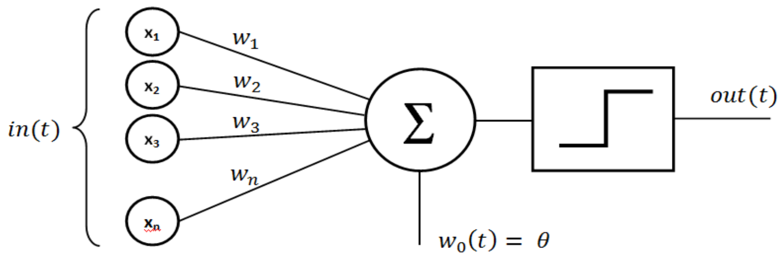
Experiment Name: Write a program to evaluate AND function with bipolar input and targets and also show the convergence curves and the decision boundary lines.

Objective:

- ✓ Implement the perceptron algorithm using bipolar inputs for the AND function.
- ✓ Plot training accuracy across epochs to show convergence.
- ✓ Visualize the final decision boundary between classes.

Theory:

A perceptron is a basic neural network model used for binary classification tasks. It connects inputs to a single output through adjustable weights. The model calculates a weighted sum of inputs, adds a bias, and applies an activation function to determine the output.



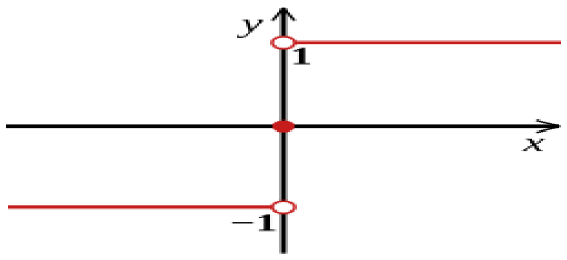
In bipolar representation, the input and target values are either -1 or +1, unlike binary representation (0 and 1). For the AND function in bipolar form:

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

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.

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:

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

where:

ω(n) is the weight vector

time t η is the learning rate.

n 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.

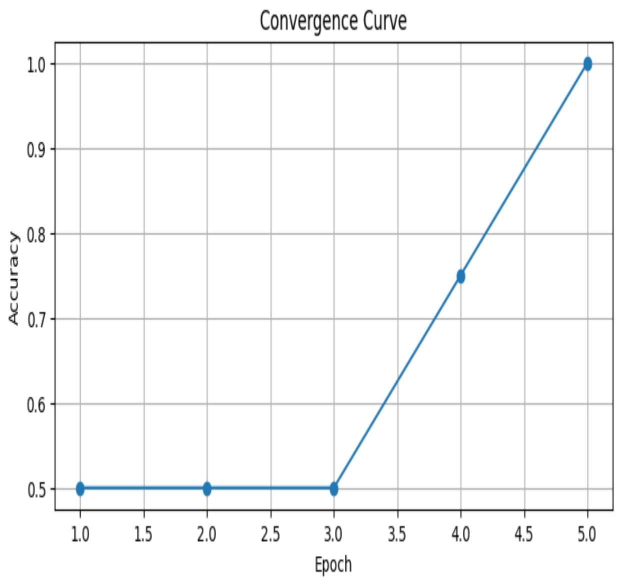
Code:

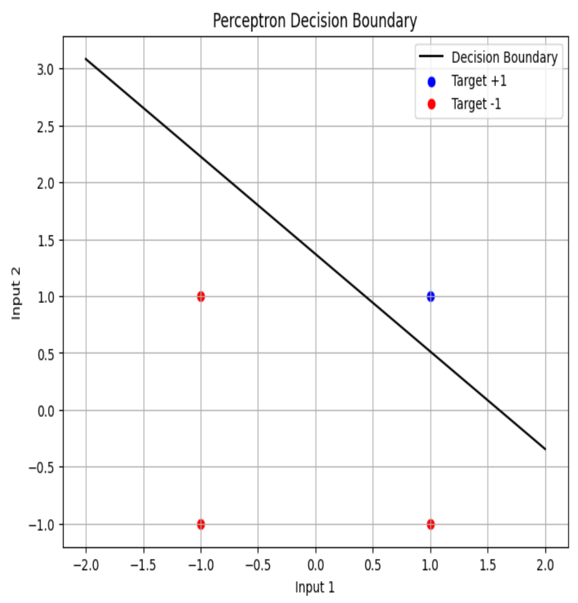
```
import numpy as np
import matplotlib.pyplot as plt

# 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 5 epochs.
```





Experiment No: 02

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

Objective:

- ✓ Develop a multi-layer neural network to learn the XOR function, which is not linearly separable.
- ✓ Track learning progress by plotting error over epochs.
- ✓ Visualize the decision boundaries and regions formed by the trained model.

Theory:

The XOR (Exclusive OR) function is a classic example used to demonstrate the limitations of linear models in neural networks. It outputs 1 when the two binary inputs differ and 0 when they are the same. The truth table reveals that the XOR function is not linearly separable—there is no straight line that can separate its output classes in the input space. Because of this, a single-layer perceptron cannot solve the XOR problem, highlighting the need for more complex network structures.

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

To overcome this limitation, a multi-layer perceptron (MLP) is employed. This network includes an input layer with two neurons, a hidden layer with non-linear activation functions (like sigmoid), and an output layer with a single neuron. The hidden layer introduces the necessary non-linearity, allowing the network to learn and represent the complex decision boundary required to solve the XOR function. During training, the backpropagation algorithm is used in combination with gradient descent to iteratively adjust weights and minimize the output error.

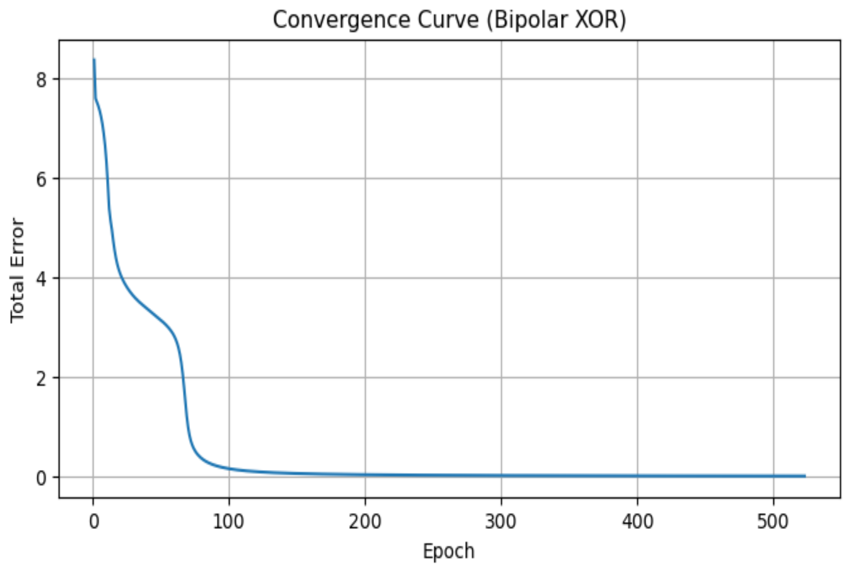
As the network trains over several epochs, the learning progress is typically observed through a convergence curve, which plots error reduction over time. A properly trained model will show a decreasing error trend, indicating successful learning. Additionally, the decision boundary can be visualized to see how the network partitions the input space into distinct output regions. In the case of XOR, this boundary is curved or segmented, clearly demonstrating the ability of the multi-layer network to handle non-linear classification tasks that a single-layer network cannot achieve.

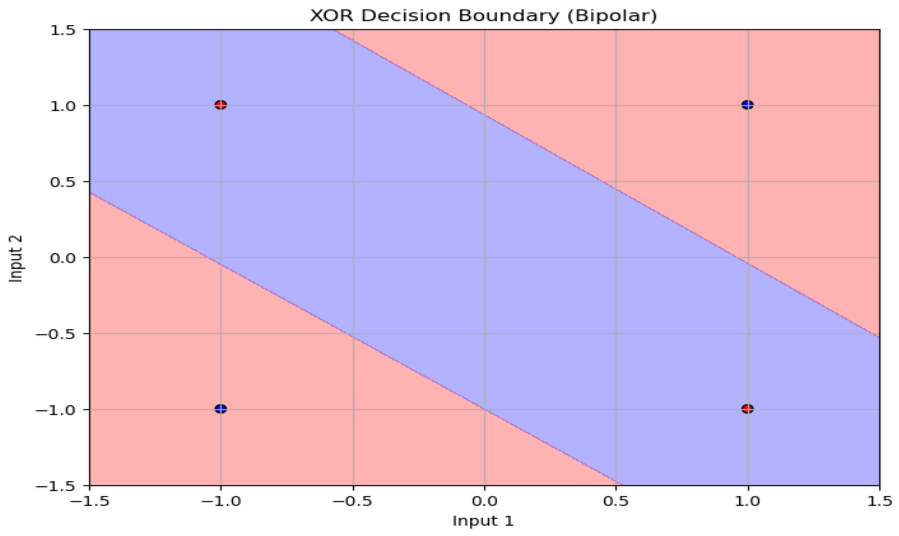
Code:

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

# Activation function: tanh and its derivative
def tanh(x):
    return np.tanh(x)

def tanh_derivative(x):
    return 1 - np.tanh(x) ** 2

# XOR dataset with bipolar inputs and targets
inputs = np.array([[-1, -1],
                   [-1,  1],
                   [ 1, -1],
                   [ 1,  1]])
targets = np.array([[-1],
                    [ 1],
                    [ 1],
                    [-1]])

# Network parameters
input_layer_size = 2
hidden_layer_size = 2
output_layer_size = 1
learning_rate = 0.1
max_epochs = 10000

# Random weight initialization
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 = tanh(hidden_input)
        final_input = np.dot(hidden_output, weights_hidden_output) + bias_output
        final_output = tanh(final_input)

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

        # Backpropagation
        output_delta = error * tanh_derivative(final_input)
        hidden_delta = tanh_derivative(hidden_input) * np.dot(weights_hidden_output,
output_delta).flatten()

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

        weights_input_hidden += learning_rate * np.outer(input_layer, hidden_delta)
        bias_hidden += learning_rate * hidden_delta

    convergence_curve.append(total_error)

    if total_error < 0.01:
        print(f"Converged in {epoch+1} epochs.")
        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 Error')
plt.title('Convergence Curve (Bipolar XOR)')
plt.grid(True)
plt.show()

# Predict function for decision boundary
def predict(x1, x2):
    input_data = np.array([x1, x2]).T
    hidden_input = np.dot(input_data, weights_input_hidden) + bias_hidden
    hidden_output = tanh(hidden_input)
    final_input = np.dot(hidden_output, weights_hidden_output) + bias_output
    final_output = tanh(final_input)
    return final_output.reshape(x1.shape)

# Plot decision boundary
x1 = np.linspace(-1.5, 1.5, 200)
```

```
x2 = np.linspace(-1.5, 1.5, 200)
X1, X2 = np.meshgrid(x1, x2)
Z = predict(X1.ravel(), X2.ravel())
Z = Z.reshape(X1.shape)

plt.figure(figsize=(8, 6))
plt.contourf(X1, X2, Z, levels=[-1, 0, 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 Decision Boundary (Bipolar)')
plt.grid(True)
plt.show()
```

Output:





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:

- ✓ Learn and implement the Stochastic Gradient Descent (SGD) algorithm guided by the Delta rule.
- ✓ Train a perceptron model for binary classification using linearly separable data.
- ✓ Plot total error over epochs to observe the model's convergence behavior.

Theory:

A perceptron is the most basic form of a neural network, composed of a single neuron that performs binary classification. It calculates the weighted sum of input features and uses a threshold-based activation function (step function) to determine the output as either 0 or 1. This simple model is effective for solving linearly separable problems.

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.

To train the perceptron, the Delta learning rule (also known as the Widrow-Hoff rule) is applied. This rule adjusts the weights based on the error between the model's prediction and the actual target output. Stochastic Gradient Descent (SGD) is used in this process, where weights are updated after each individual data point rather than after processing the full dataset. This approach can speed up convergence and reduce the chance of getting stuck in suboptimal solutions, especially in more complex tasks. 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.

During training, each epoch involves shuffling the data and updating weights for each sample one at a time using the Delta rule. To monitor learning progress, the total classification error for all samples is tracked at each epoch. A convergence curve is then plotted, showing how the total error changes over time. When the total error drops to zero, it indicates that the perceptron has successfully learned to classify all inputs correctly.

The input and target pairs used in this experiment are:

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

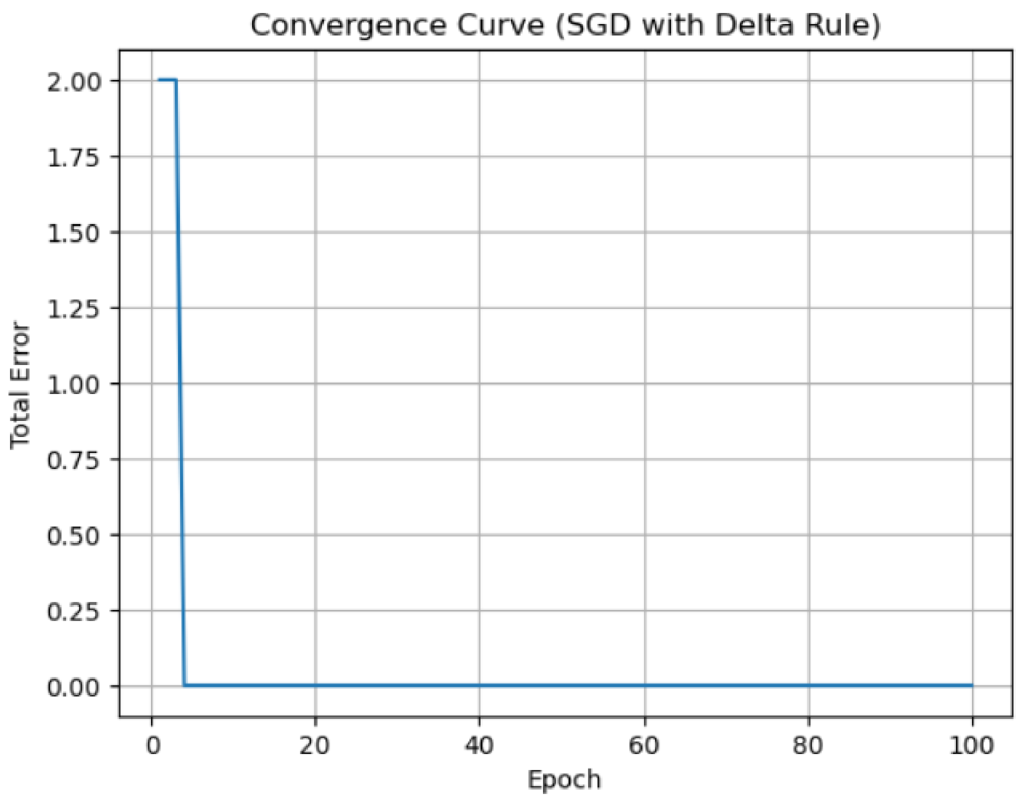
This setup corresponds to a linearly separable function, allowing a perceptron to successfully learn the correct mapping using the Delta rule and SGD.

Code:

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

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

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

# Initialize weights randomly
weights = np.random.randn(3)
learning_rate = 0.1
epochs = 100

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

# Training using SGD and Delta learning rule
convergence_curve = []
converged=False

for epoch in range(epochs):
    total_error = 0
    # Shuffle the data for each epoch (stochastic gradient descent)
    indices = np.random.permutation(len(Xinput))
    Xinput_shuffled = Xinput[indices]
    Dtarget_shuffled = Dtarget[indices]

    for i in range(len(Xinput)):
        x = Xinput_shuffled[i]
        d = Dtarget_shuffled[i]

        # Forward pass (calculate output)
        y = step_function(np.dot(x, weights))

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

        # Delta rule: weight update
        weights += learning_rate * error * x

    convergence_curve.append(total_error)

    # Stop early if there is no error
    if total_error == 0 and converged==False:
        print(f"Converged in {epoch + 1} epochs.")
        converged=True

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

# Plot convergence curve
plt.plot(range(1, len(convergence_curve) + 1), convergence_curve)
plt.xlabel('Epoch')
plt.ylabel('Total Error')
plt.title('Convergence Curve (SGD with Delta Rule)')
plt.grid()
plt.show()
```

Output:

```
Converged in 4 epochs.
Final weights: [ 1.13863788 -0.21294792 -0.80435195]
```



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 a fundamental type of artificial neural network where information moves in a single direction—from input to output—without any feedback loops. It is commonly applied in supervised learning tasks such as classification, regression, and pattern recognition. In this experiment, the MNIST dataset is used to evaluate the performance of an FNN. The MNIST dataset consists of 70,000 grayscale images of handwritten digits ranging from 0 to 9. Each image is 28x28 pixels, resulting in 784 input features when flattened.

Before training the model, data preprocessing is essential to enhance performance. First, pixel values are normalized by dividing them by 255 to scale them between 0 and 1, which helps accelerate model convergence. The dataset is then divided into training and testing sets, with 80% used for training and 20% for evaluation. The neural network is implemented using Scikit-learn's MLPClassifier, a type of multilayer perceptron. It includes a hidden layer with 128 neurons, uses the ReLU activation function to handle non-linearity, and applies the 'adam' optimizer for efficient weight updates. The model is trained over 10 iterations (epochs).

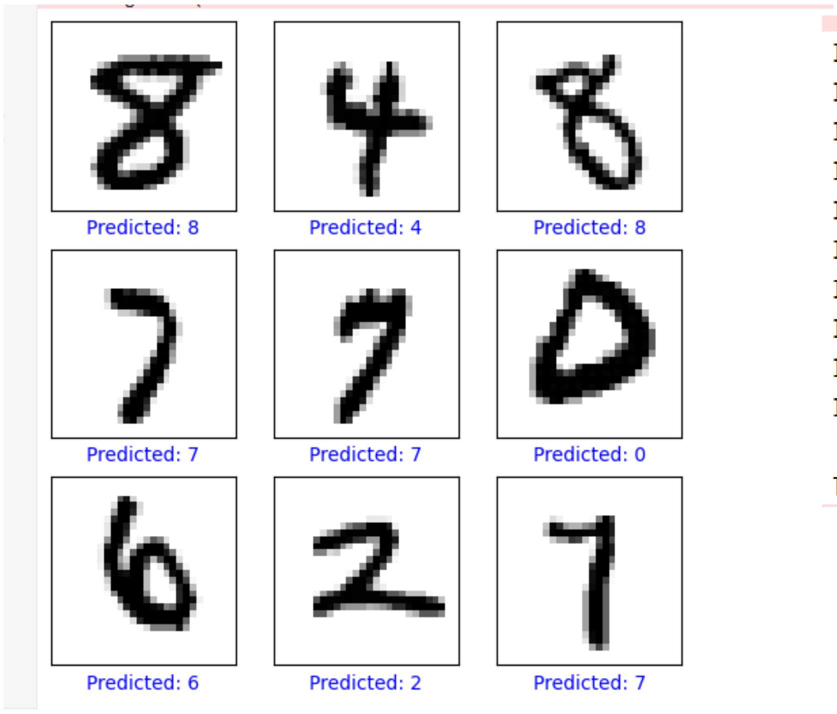
After training, the model is evaluated by predicting digit labels for the test set. Its accuracy, defined as the percentage of correct predictions, is calculated to assess performance. Additionally, a few test samples are visualized to compare the model's predictions against the actual digits. For display, the flattened 784-element input vectors are reshaped back into 28x28 images using matplotlib, providing an intuitive way to interpret model outputs and visually inspect prediction quality.

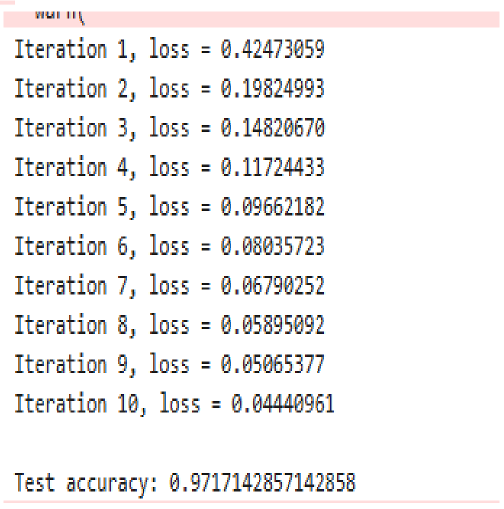
Code:

```
%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
```

Experiment No: 05

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

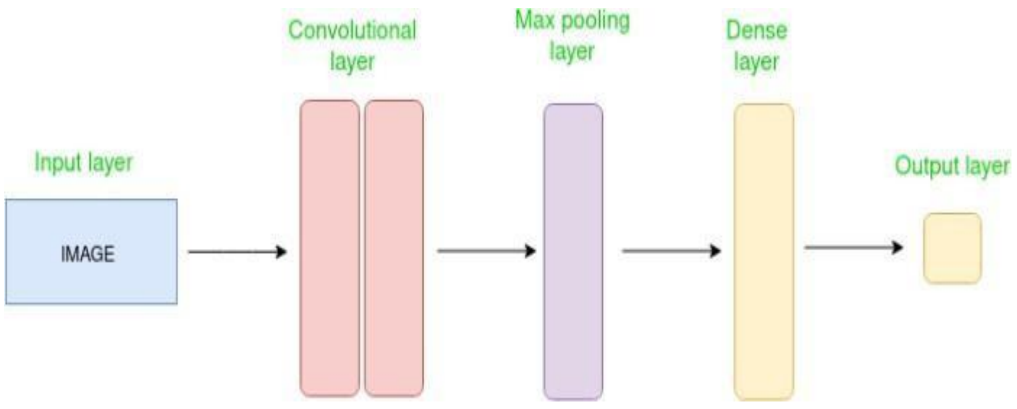
Objective:

- ✓ 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 and CNNs: Image classification is a core task in computer vision, involving the identification of objects within images by assigning them to predefined categories such as animals, vehicles, or other entities. Traditional machine learning methods often fall short when dealing with the high dimensionality and spatial complexity of image data. To overcome this, deep learning approaches—particularly Convolutional Neural Networks (CNNs)—are widely used. CNNs are designed to efficiently extract spatial features from images, making them ideal for classification tasks.

CNN Architecture and Layers: A CNN processes images through a series of layers that each serve a specific purpose. The **convolutional layers** apply filters to detect local patterns like edges and textures. These patterns are then passed through **ReLU activation functions** to introduce non-linearity. **Pooling layers** follow to reduce spatial dimensions and help control overfitting. The resulting feature maps are then **flattened** into a one-dimensional vector before being passed to **fully connected (dense) layers**, which interpret the extracted features to make final predictions. The **output layer**, typically using the Softmax function, produces probability scores for each class in multi-class classification problems.



CIFAR-10 Dataset and Model Design For this experiment, the CIFAR-10 dataset is used, containing 60,000 color images sized 32×32 pixels, spread across 10 categories including animals and vehicles (e.g., airplane, cat, truck). The dataset is divided into 50,000 training and 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)

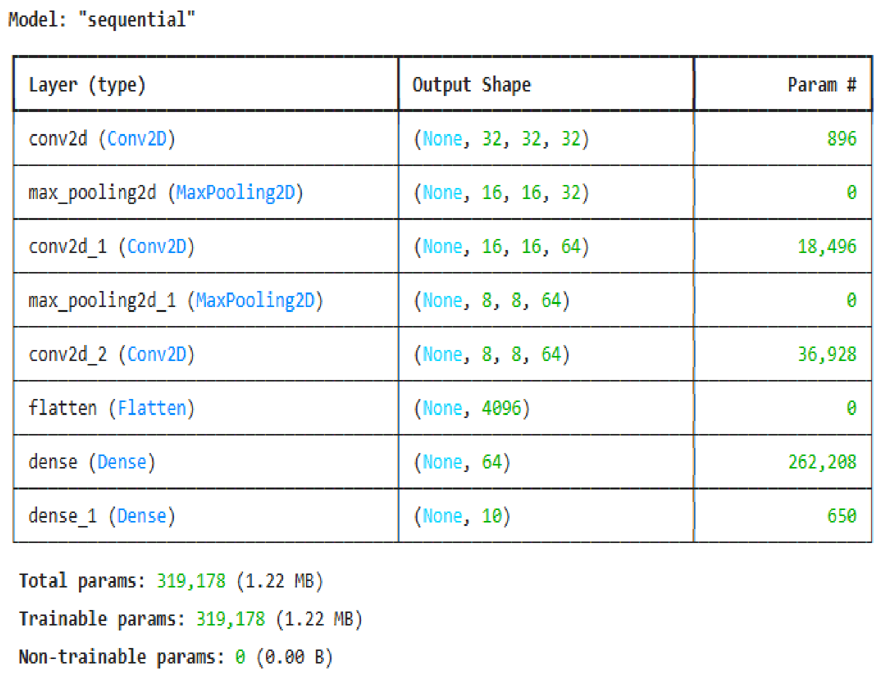
Model Training and Evaluation The model is compiled using the Adam optimizer, known for its adaptive learning capabilities and efficiency in training deep networks. The loss function used is sparse categorical crossentropy, which is well-suited for multi-class problems with integer-based labels. Training is carried out over 10 epochs with performance evaluated on the test set. Accuracy on the test data is used to assess how well the model generalizes to unseen images, offering insight into the effectiveness of the learned representations.

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
])
```

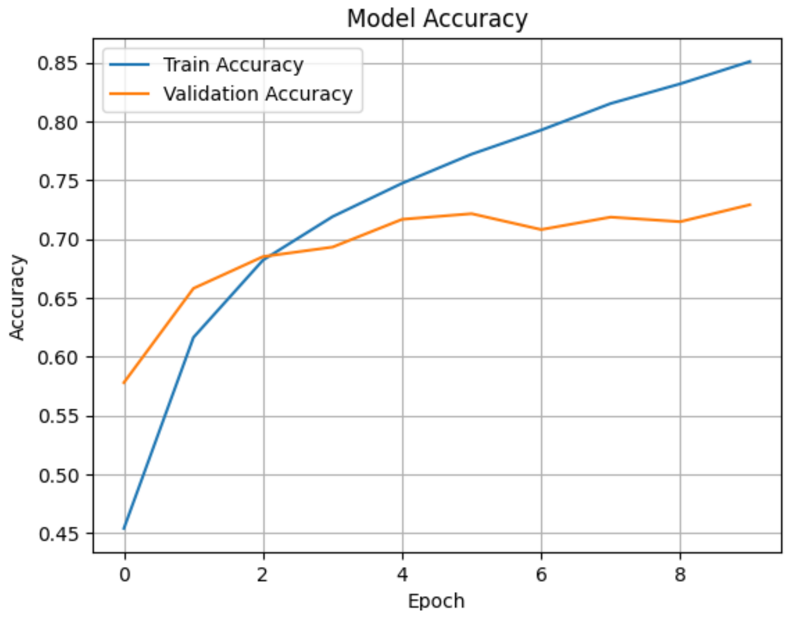
```
# Compile the model
model.compile(optimizer='adam',
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy'])

# 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
```



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 and IMDB Dataset Text classification is a vital task in Natural Language Processing (NLP) where textual data is categorized into predefined labels. A common example is sentiment analysis, which determines whether a text, such as a movie review, conveys a positive or negative opinion. The IMDB dataset is a popular benchmark for this purpose, containing 50,000 movie reviews evenly divided into training and testing sets. Each review is converted into a sequence of integers representing word indices, with labels assigned as 0 for negative and 1 for positive sentiments.

Recurrent Neural Networks and LSTM A Recurrent Neural Network processes input sequences one element at a time, maintaining a hidden state that contains information about previous elements However, standard RNNs struggle with learning long-term dependencies due to the vanishing gradient problem. To address this, Long Short-Term Memory (LSTM) networks are employed. LSTMs have specialized memory cells and gating mechanisms that control what information to remember or forget, enabling them to capture long-range context effectively.

Model Architecture The experiment’s model is built using the Keras Sequential API and includes multiple layers tailored for sequence data. It begins with an Embedding layer that transforms word indices into dense vector representations, facilitating the learning of semantic relationships between words. This is followed by two LSTM layers: the first returns the full sequence to pass rich temporal information to the next layer, while the second summarizes the sequence into a fixed-size vector. Finally, a Dense layer with a sigmoid activation produces a binary output, indicating the sentiment class.

Data Preparation, Training, and Evaluation Before training, the dataset is preprocessed by limiting the vocabulary to the 10,000 most common words to reduce complexity and padding all sequences to a uniform length of 200 tokens. The model is compiled with binary cross-entropy loss and optimized using the Adam algorithm. Training runs for 5 epochs, after which the model’s performance is evaluated on the test set by calculating accuracy. The final accuracy score reflects how well the model can classify unseen movie reviews as positive or negative

Code:

```
import tensorflow as tf
from tensorflow import keras
```

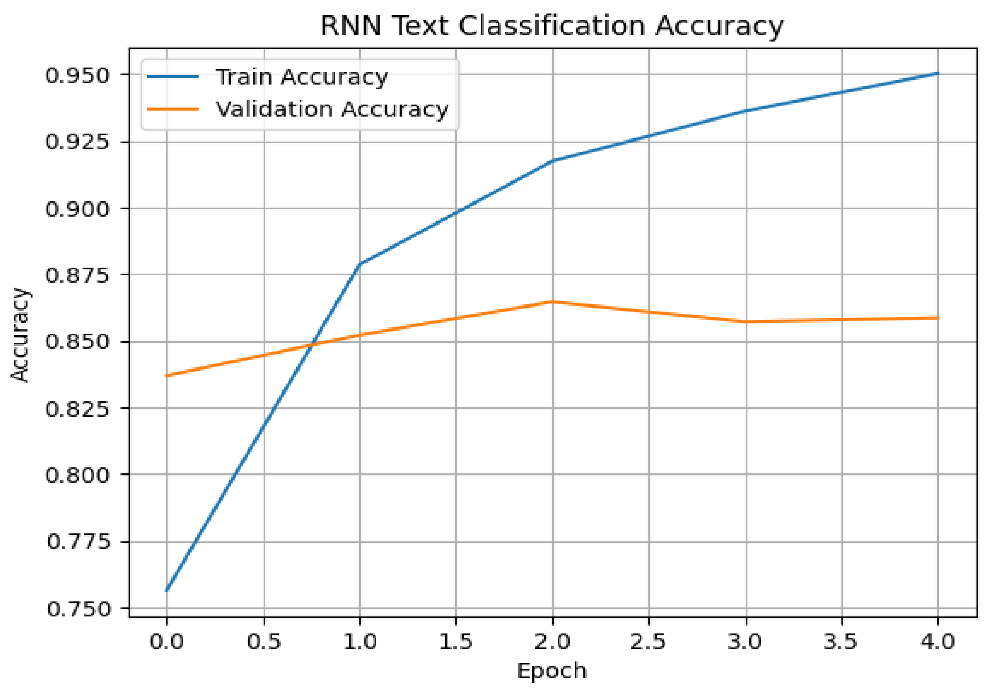
```
import numpy as np

# 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
```



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:

Text Classification Text classification is an important NLP task that assigns categories to text data. It's used in spam detection, sentiment analysis, and topic labeling. Traditional methods used manual feature extraction with models like Naive Bayes or SVM, but deep learning—especially Transformer models like BERT—now offers more accurate and scalable solutions.

Transfer Learning in NLP Transfer learning enables models pretrained on large datasets to be fine-tuned for specific tasks using less labeled data. This approach lets models learn general language patterns from huge corpora and apply that knowledge efficiently to downstream tasks, improving performance even with limited data.

BERT Model BERT (Bidirectional Encoder Representations from Transformers) is a powerful language model from Google. It processes text bidirectionally, capturing richer context than previous models. Pretrained on large text corpora, BERT can be easily adapted to various NLP tasks by adding simple classification layers, making it highly effective.

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.

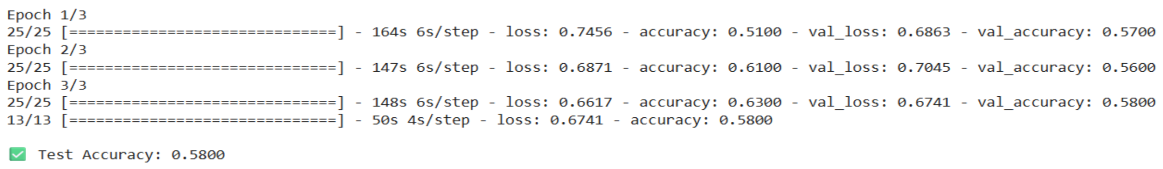
Experiment Setup and Evaluation This experiment uses the IMDB dataset for binary sentiment classification with 25,000 reviews for training and testing each. The model uses TensorFlow Hub's pretrained BERT with a preprocessing layer, followed by dense layers for classification. Trained over 3 epochs with Adam optimizer and binary cross-entropy loss, the model's accuracy on the test set measures its performance.

Code:

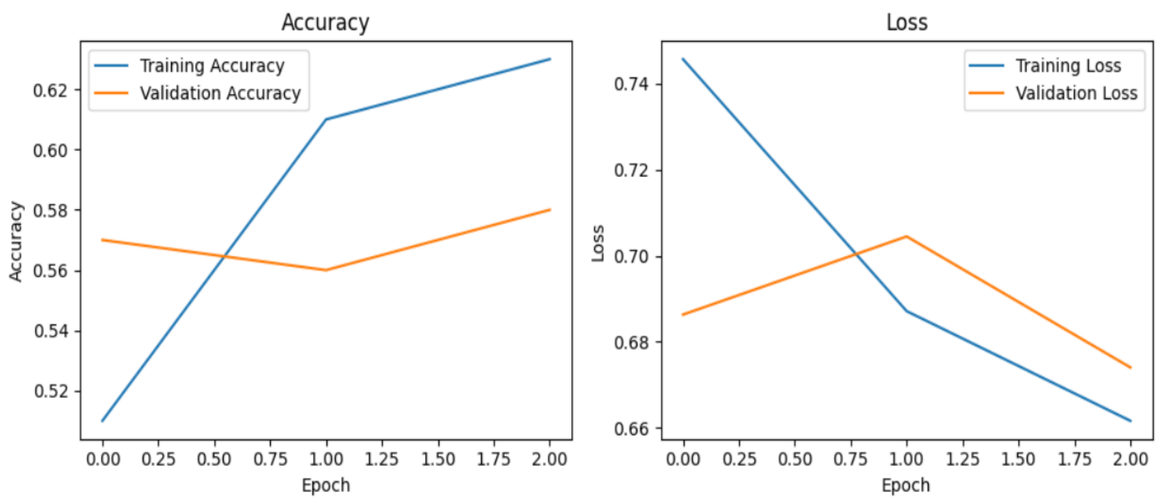
```
import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_text as text
import tensorflow_datasets as tfds
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay,
classification_report, roc_curve, auc

# 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()
```

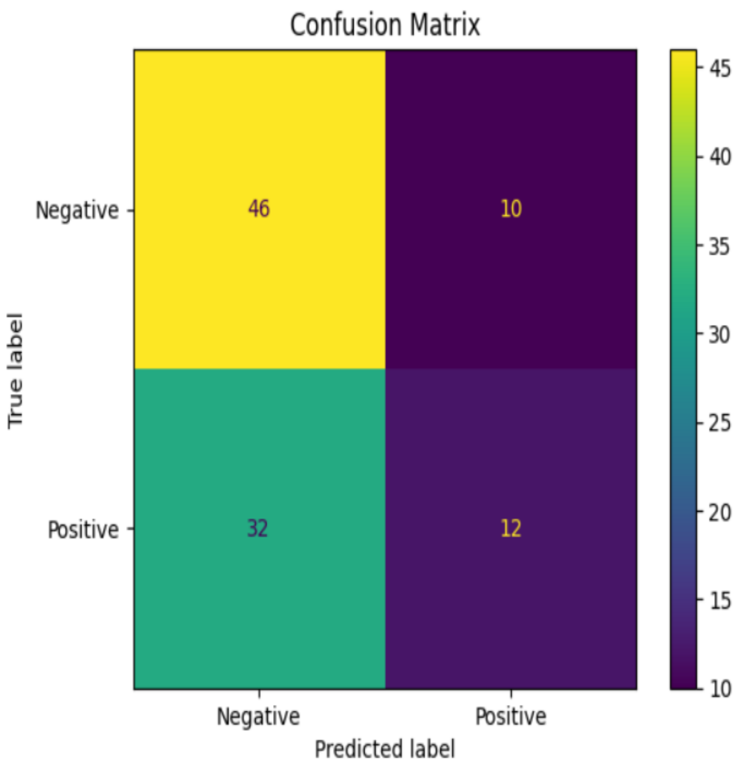
Output:

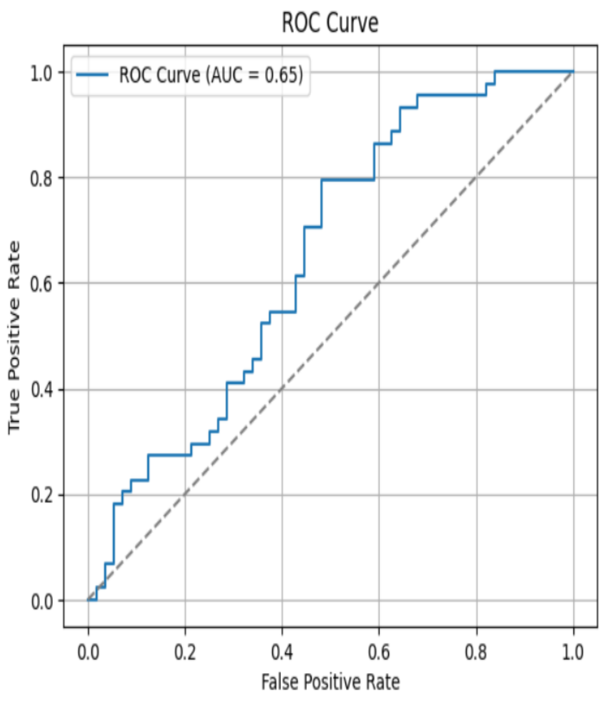


```
Epoch 1/3
25/25 [==============================] - 164s 6s/step - loss: 0.7456 - accuracy: 0.5100 - val_loss: 0.6863 - val_accuracy: 0.5700
Epoch 2/3
25/25 [==============================] - 147s 6s/step - loss: 0.6871 - accuracy: 0.6100 - val_loss: 0.7045 - val_accuracy: 0.5600
Epoch 3/3
25/25 [==============================] - 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:

Generative Adversarial Networks (GANs), introduced by Ian Goodfellow in 2014, consist of two neural networks—the Generator and the Discriminator—that compete in a game-like setting. The Generator creates synthetic data from random noise aiming to fool the Discriminator, which tries to distinguish real data from fake. Through this adversarial training, both networks improve, with the Generator producing increasingly realistic outputs over time.

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.

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: This experiment uses the MNIST dataset, which contains 60,000 training and 10,000 test grayscale images of handwritten digits sized 28×28 pixels. Only the training set is used to teach the GAN to generate new digit images.

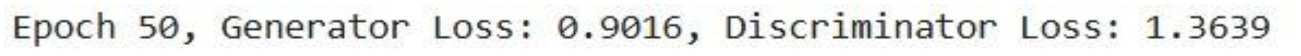
Network Architectures: The Generator transforms a 100-dimensional noise vector through layers such as Dense, BatchNorm, and Conv2DTranspose, producing images with pixel values scaled between -1 and 1. The Discriminator is built with convolutional layers and LeakyReLU activations, with dropout layers to reduce overfitting, and outputs a single value indicating whether an input image is real or fake.

Training Process: During training, random noise inputs generate fake images that, alongside real images, are fed to the Discriminator for classification. Both networks are updated batch-wise using gradients from their loss functions, with losses logged every epoch. Periodic visualization of generated images helps track progress, showing how the Generator becomes better at creating images indistinguishable from real handwritten digits.

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
```