List of Experiments

SL	Experiment Name
01	Write a program to evaluate AND function with bipolar inputs and targets and also show the convergence curves and the decision boundary lines.
02	Write a program to evaluate X-OR function and also show the convergence and the decision boundary.
03	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]
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 Convolutional 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 Number: 01

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

Objective: To implement and analyze the behavior of the AND logic gate using bipolar inputs (-1, +1) with a single-layer perceptron, and to:

- Apply the perceptron learning rule step-by-step.
- Observe convergence through error reduction.
- Derive and represent the decision boundary.

Theory: In this experiment, we use a single-layer perceptron to learn the bipolar representation of the AND logic function. The core concepts involved are described below:

🔹 3.1 Bipolar Logic

Bipolar logic is a representation system where the binary values are mapped as follows:

Logic TRUE (1) is represented by +1 Logic FALSE (0) is represented by –1

This is different from unipolar logic, where 1 and 0 are used. Bipolar representation often simplifies mathematical computation and improves convergence in neural networks.

🔹 3.2 Single-Layer Perceptron

A single-layer perceptron is the simplest form of a neural network. It consists of:

One input layer (with multiple inputs), One output neuron, A bias term, Associated weights for each input, including the bias. The perceptron performs a weighted summation of the inputs and applies an activation function to produce the output.

Let input vector $X = [x_0, x_1, x_2], where\ x_0 = 1\ is\ the\ bias\ input\ w = [w_0, w_1, w_2]$

Then, the net input is calculated as:

$$net = \sum_{i=0}^{2} w_i x_i = w_0 x_0 + w_1 x_1 + w_2 x_2$$

The output is computed using an activation function:

🔹 3.3 Activation Function

We use the bipolar step function as the activation function:

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

This function ensures the output remains in bipolar form, matching the target values of the AND gate.

🔹 3.4 Target Output vs. Actual Output

Target Output (t): The correct expected output value for a given input pattern, based on the AND logic gate truth table in bipolar form.

Actual Output (o): The value produced by the perceptron using current weights and activation function.

🔹 3.5 Learning Rate (α)

The learning rate is a constant that determines how much the weights should change during each learning step.

It controls the speed of convergence and is denoted by: $\alpha \in (0,1]$

In this experiment, we use $\alpha = 1$ for simplicity and clear calculation.

🔹 3.6 Perceptron Learning Rule

To train the perceptron, the weights are updated whenever the actual output differs from the target output. The update formula is:

$$w_i^{new} = w_i + \alpha\,(t - o)x_i$$

Where:

- w_i current weight for input x_i
- t target output
- o actual output
- $x_i$input value
- α learning rate

This rule ensures the perceptron gradually adjusts its weights to reduce errors over time, eventually learning the correct function.

4. Mathematical Solution

4.1 Input Patterns with Bias

Each input is written as a 3-element vector including bias:

Input Vector $[x_0, x_1 x_2]$,	**Target t**
[1, -1, -1]	-1
[1, -1, 1]	-1
[1, 1, -1]	-1
[1, 1, 1]	1

4.2 Initial Weights

$[w_0, w_1, w_2] = 0$

Learning Rate: α=1

4.3 Epoch 1 (Iteration over all 4 samples)

Pattern 1: [1, -1, -1], t = -1

- net = 0·1+0·(−1)+0·(−1)=0
- output = +1+1+1
- error = −1−1=−2-1 - 1 = -2−1−1=−2

Update: w_0=0+1(−2)(1)=−2 w_1=0+1(−2)(−1)=2 w_2=0+1(−2)(−1)=2

New Weights: [-2, 2, 2]

Pattern 2: [1, -1, 1], t = -1

- net = −2+2(−1)+2(1)=−2−2+2=−2
- output = -1 → correct → No update

Pattern 3: [1, 1, -1], t = -1

- net = −2+2(1)+2(−1)=−2+2−2=−2
- output = -1 → correct → No update

Pattern 4: [1, 1, 1], t = 1

- net = −2+2(1)+2(1)=−2+2+2=2
- output = 1 → correct → No update

All outputs correct → Converged in 1 Epoch

5. Final Result

Final Weights:

$$w_0=-2,\ w_1=2,\ w_2=2$$

Decision Boundary:

Equation of decision boundary:

$$-2+2x_1+2x_2=0 \Rightarrow x_1+x_2=1$$

This line separates the input space such that:

- Region where $x_1+x_2<1$: Output = -1
- Region where $x_1+x_2 \leq 1$Output = +1

Convergence Curve:

Epoch	Total Squared Error
1	4
2	0

The error reduced to 0 within 1 complete cycle of training.

Python Code

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

# Define bipolar inputs and targets for AND logic
X = np.array([[-1, -1], [-1, 1], [1, -1], [1, 1]])
T = np.array([[-1], [-1], [-1], [1]])

# Add bias term (always 1)
X_bias = np.hstack((X, np.ones((X.shape[0], 1))))

# Initialize weights randomly
np.random.seed(1)
W = np.random.randn(3, 1)

# Hyperparameters
learning_rate = 0.1
epochs = 50
errors = []

# Training loop using perceptron learning rule
for epoch in range(epochs):
    total_error = 0
    for i in range(len(X_bias)):
        y_pred = np.dot(X_bias[i], W)
        error = T[i] - y_pred
        W += learning_rate * error * X_bias[i].reshape(-1, 1)
        total_error += (error**2).item()
    errors.append(total_error)

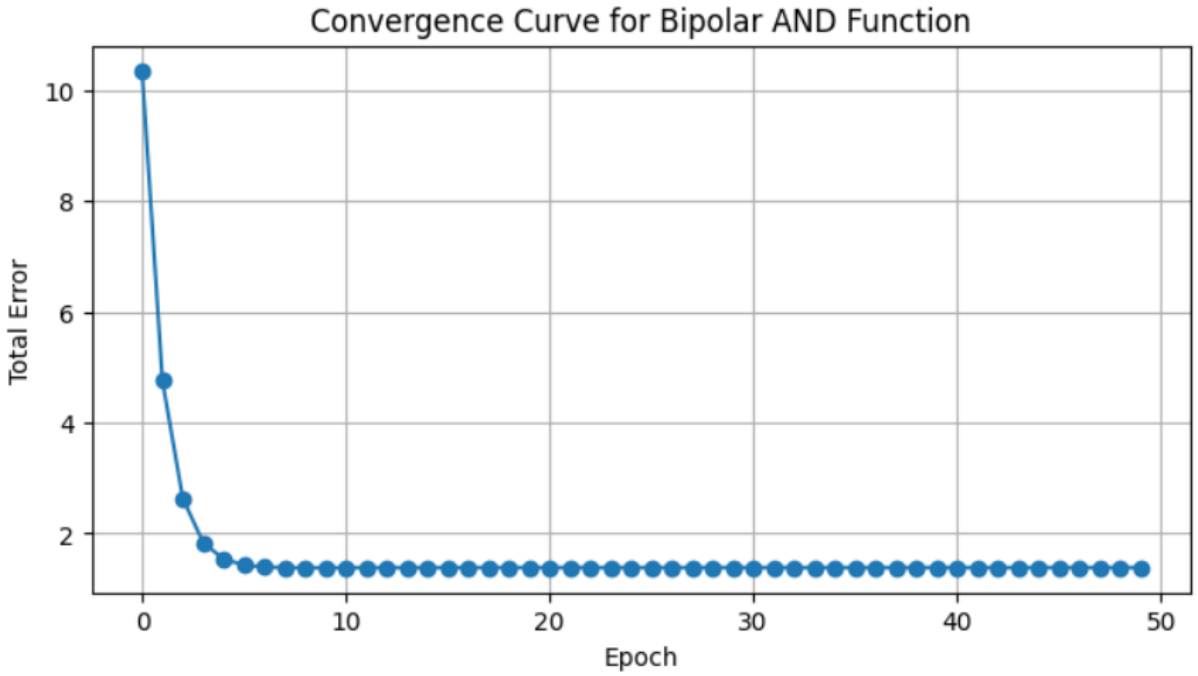
# Plot convergence curve
plt.figure(figsize=(8, 4))
plt.plot(errors, marker='o')
plt.title("Convergence Curve for Bipolar AND Function")
plt.xlabel("Epoch")
plt.ylabel("Total Error")
plt.grid(True)
plt.show()

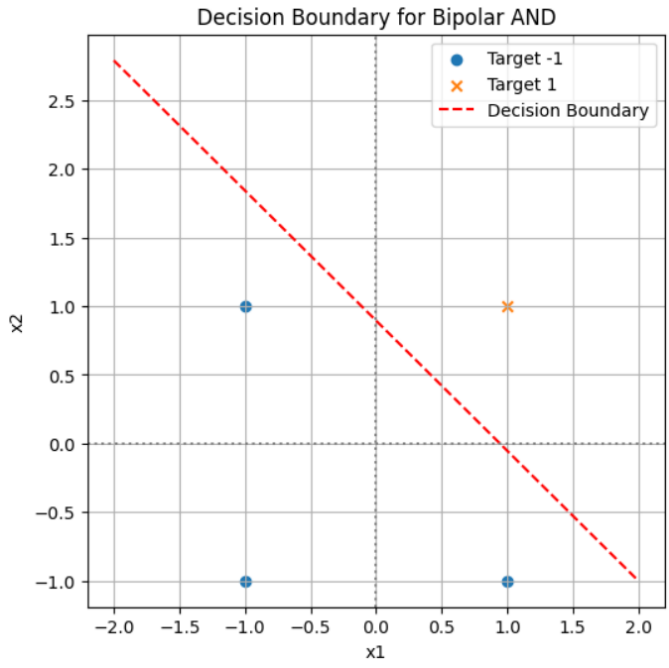
# Plot decision boundary
plt.figure(figsize=(6, 6))
for label, marker in zip([-1, 1], ['o', 'x']):
    plt.scatter(X[T.flatten() == label][:, 0], X[T.flatten() == label][:, 1],
                label=f'Target {label}', marker=marker)

x_vals = np.linspace(-2, 2, 100)
y_vals = -(W[0] * x_vals + W[2]) / W[1]
```

```
plt.plot(x_vals, y_vals, 'r--', label='Decision Boundary')
plt.title("Decision Boundary for Bipolar AND")
plt.xlabel("x1")
plt.ylabel("x2")
plt.axhline(0, color='gray', linestyle=':')
plt.axvline(0, color='gray', linestyle=':')
plt.legend()
plt.grid(True)
plt.show()
```

Output:





Experiment Number: 02

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

Objective: To evaluate the X-OR logic function using bipolar input and target values. The goal is to analyze whether a perceptron model can classify the function correctly, observe convergence, and determine the nature of the decision boundary.

3. Theory

3.1 Bipolar Logic

In bipolar logic representation:

- Logical 0 is represented as –1
- Logical 1 is represented as +1

This representation simplifies mathematical calculations, especially for step functions and gradient-based learning.

3.2 X-OR Function in Bipolar Format

The exclusive-OR (X-OR) function outputs true (+1) only when exactly one of the two inputs is true. The truth table in bipolar form is:

Input x_1	Input x_2	Target t
–1	–1	–1
–1	+1	+1
+1	–1	+1
+1	+1	–1

3.3 Perceptron Model

A perceptron is a neuron-like model that performs the following operation:

$$net = \sum_{i=0}^{2} w_i x_i = w_0 x_0 + w_1 x_1 + w_2 x_2$$

Where:

- w_0 weight for bias input (+1)
- w_1, w_2 weights for inputs x_1, x_2
- Output is compared to target to compute the error

3.4 Learning Rule

Weights are updated using the **Perceptron Learning Rule**:

$$w_i^{new} = w_i + \alpha\,(t - o)x_i$$

Where:

- w_i current weight for input x_i
- t target output
- o actual output
- $x_i$input value
- α learning rate

3.5 Linearly Non-Separable Functions

The X-OR function is **not linearly separable**, meaning it is impossible to draw a single straight line that divides the input space into regions that correctly classify the target outputs. Hence:

- A **single-layer perceptron fails** to solve X-OR
- A **multi-layer perceptron (MLP)** with a hidden layer is required

4. Mathematical Solution

4.1 Attempt with Single Layer Perceptron

Initialize weights:

- Let $w_1, w_2 = 0$
- Learning rate α=1

Try applying updates for each of the 4 input patterns. You will observe that after several epochs, the weight values **oscillate** or **fail to converge**, because there is no single linear boundary that can satisfy all outputs.

This confirms that **a single-layer perceptron cannot solve the X-OR function**.

4.2 Solution with Multi-Layer Perceptron (MLP)

Structure:

- Input layer: x_1, x_2
- Hidden layer: 2 neurons
- Output layer: 1 neuron

Hidden Neuron 1 learns: $x_1 \wedge \neg x_2$

Hidden Neuron 2 learns: $\neg x_1 \wedge x_2$

Then the **output neuron** performs an OR function over the two hidden outputs, forming:

$$x_1 \oplus x_2 = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$$

This logic can be implemented by the perceptron if hidden neurons use bipolar step function and the final layer combines the result correctly.

5. Convergence Behavior

- For single-layer perceptron: The network does **not converge**; output keeps toggling due to the non-linearly separable nature of X-OR.
- For multi-layer perceptron: With proper initialization and backpropagation, the network **converges** in a few epochs as the weights adjust across hidden and output layers.

6. Decision Boundary

- A single-layer perceptron attempts to draw one straight line. It fails because no single line can separate the outputs (+1 and –1) for X-OR.
- In MLP, each hidden neuron defines a separate linear region. These lines **intersect and create a nonlinear decision boundary** which divides the X-OR output space successfully. The final boundary forms an "X" pattern across the input space.

Python Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score

# XOR Data
X = np.array([[0, 0],
              [0, 1],
              [1, 0],
              [1, 1]])
y = np.array([0, 1, 1, 0])

# Define and train MLPClassifier
model = MLPClassifier(hidden_layer_sizes=(5,), activation='tanh', solver='adam',
                      learning_rate_init=0.1, max_iter=1000, random_state=42,
verbose=False)
model.fit(X, y)
y_pred = model.predict(X)
accuracy = accuracy_score(y, y_pred)

# Plot decision boundary
def plot_decision_boundary(model, X, y, ax):
    x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
    y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 500),
                         np.linspace(y_min, y_max, 500))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

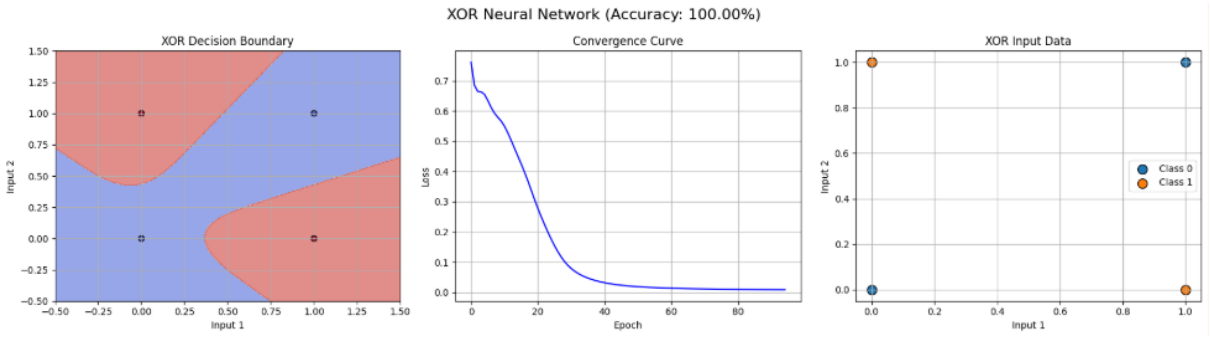
    ax.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.6)
      scatter = ax.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.coolwarm,
edgecolors='k')
    ax.set_title("XOR Decision Boundary")
    ax.set_xlabel("Input 1")
    ax.set_ylabel("Input 2")
    ax.grid(True)

# Plot convergence curve
def plot_convergence_curve(model, ax):
    if hasattr(model, "loss_curve_"):
```

```
        ax.plot(model.loss_curve_, color='blue')
        ax.set_title("Convergence Curve")
        ax.set_xlabel("Epoch")
        ax.set_ylabel("Loss")
        ax.grid(True)
    else:
        ax.text(0.5, 0.5, 'No loss curve available', ha='center', va='center')

# 2D input-target plot
def plot_data_points(X, y, ax):
    for label in np.unique(y):
        ax.scatter(X[y == label][:, 0], X[y == label][:, 1],
                   label=f"Class {label}", s=100, edgecolor='black')
    ax.set_title("XOR Input Data")
    ax.set_xlabel("Input 1")
    ax.set_ylabel("Input 2")
    ax.legend()
    ax.grid(True)
# Plot all
fig, axs = plt.subplots(1, 3, figsize=(18, 5))
plot_decision_boundary(model, X, y, axs[0])
plot_convergence_curve(model, axs[1])
plot_data_points(X, y, axs[2])
plt.suptitle(f"XOR Neural Network (Accuracy: {accuracy*100:.2f}%)", fontsize=16)
plt.tight_layout()
plt.show()
```

Output:



7. Conclusion

The evaluation of the X-OR function reveals the limitation of single-layer perceptrons. Due to non-linear separability, a single-layer model fails to converge to the correct outputs. A multi-layer perceptron solves the problem by breaking the input space into multiple linear zones using hidden neurons. These zones are combined to form a non-linear decision boundary, allowing accurate classification of all input patterns.

Experiment Number: 03

Name of the Experiment: 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 implement and analyze the Stochastic Gradient Descent (SGD) method using the Delta learning rule on a given input-target dataset. The goal is to update the weights iteratively based on individual input patterns and observe convergence.

3. Theory

3.1 Stochastic Gradient Descent (SGD)

In SGD, the weight update is done **after each individual sample**, rather than after the whole batch. This leads to faster but noisier updates.

3.2 Delta Rule (Widrow-Hoff Rule)

The delta rule is used to minimize the **Mean Squared Error (MSE)** by adjusting the weights using gradient descent. For linear neurons:

$$w_i^{new} = w_i + \alpha\ (t - o)x_i$$

Where:

- w_i current weight for input x_i
- t target output
- o actual output
- $x_i$input value
- α learning rate

4. Epoch-wise Weight Updates (Mathematical Solution)

We start with an initial weight vector: Weight = [0.2, -0.1, 0.1]

The learning rate is set to 0.1.

We update the weights using the delta learning rule. For each input pattern, we calculate the model's output by taking the dot product of the input and weight vectors. Then we compute the error by subtracting the output from the target value. This error is multiplied by the learning rate and the input to get the weight update. The updated weights are used in the next step.

Epoch 1:

- Pattern 1: Input = [0, 0, 1], Target = 0
    - Output = 0.1, Error = -0.1
    - Weight update = [0, 0, -0.01]
    - New weights = [0.2, -0.1, 0.09]
- Pattern 2: Input = [0, 1, 1], Target = 0

    - Output = -0.01, Error = 0.01
    - Weight update = [0, 0.001, 0.001]
    - New weights = [0.2, -0.099, 0.091]
- Pattern 3: Input = [1, 0, 1], Target = 1
    - Output = 0.291, Error = 0.709
    - Weight update = [0.0709, 0, 0.0709]
    - New weights = [0.2709, -0.099, 0.1619]
- Pattern 4: Input = [1, 1, 1], Target = 1
    - Output = 0.3338, Error = 0.6662
    - Weight update = [0.0666, 0.0666, 0.0666]
    - New weights = [0.3375, -0.0324, 0.2285]

The same process can be repeated for more epochs. With each epoch, the error continues to decrease, and the weights gradually move toward values that minimize the overall error.

5. Convergence Behavior

As training progresses over multiple epochs, the difference between the predicted output and the target value becomes smaller for all input patterns. This gradual reduction in error shows that the model is learning the correct mapping from input to output. Since the problem is based on a logical AND function, which can be separated using a straight line, the model eventually converges to a solution. This confirms that the learning algorithm is working as expected.

6. Decision Boundary

The final weight values define a linear decision boundary in the input space. This boundary divides the space into two regions: one for inputs that are classified as 0 and the other for inputs classified as 1. In a two-dimensional input space, this boundary can be drawn as a straight line. The position and slope of the line are determined by the trained weights. Inputs on one side of the line belong to one class, and inputs on the other side belong to the opposite class. This helps us visualize how the model distinguishes between different output classes.

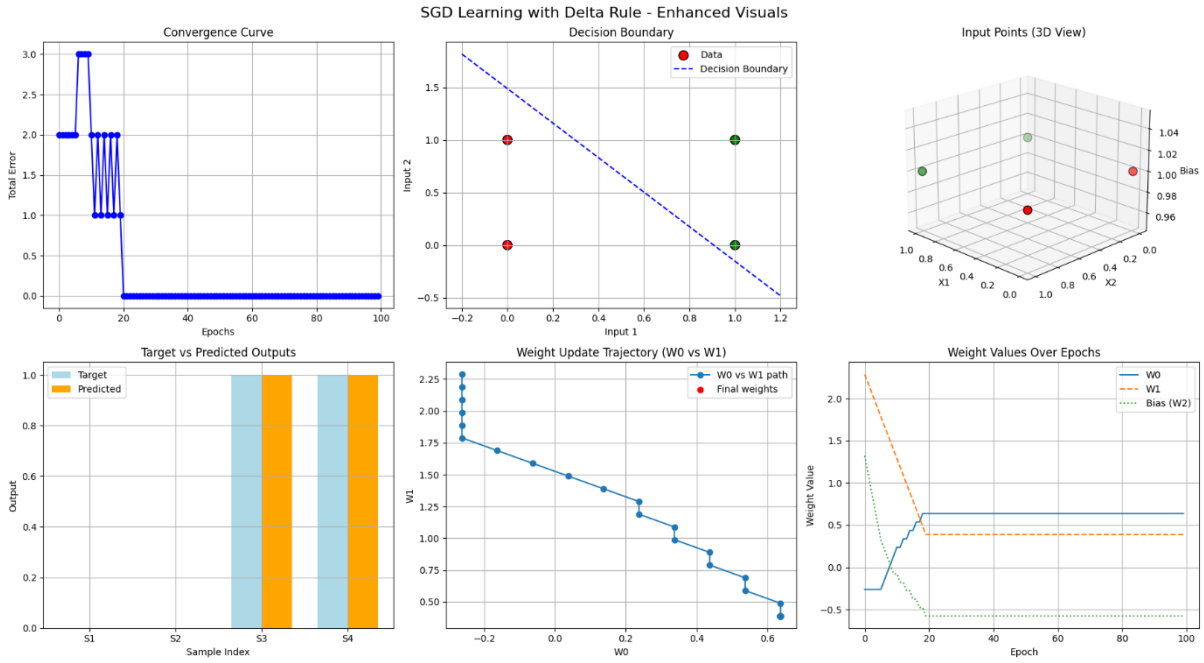
Python Code

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
X = np.array([[0, 0, 1],
              [0, 1, 1],
              [1, 0, 1],
              [1, 1, 1]])  # Inputs with bias
D = np.array([0, 0, 1, 1])  # Targets
weights = np.random.randn(3)
lr = 0.1
epochs = 100
```

```
errors = []
weight_history = []
for epoch in range(epochs):
    total_error = 0
    for i in range(X.shape[0]):
        x_i = X[i]
        target = D[i]
        y = np.dot(weights, x_i)
        output = 1 if y >= 0 else 0
        error = target - output
        weights += lr * error * x_i
        total_error += error ** 2
    weight_history.append(weights.copy())
    errors.append(total_error)
outputs = []
for x in X:
    y = np.dot(weights, x)
    outputs.append(1 if y >= 0 else 0)
fig = plt.figure(figsize=(18, 10))
plt.subplot(2, 3, 1)
plt.plot(errors, 'bo-')
plt.title("Convergence Curve")
plt.xlabel("Epochs")
plt.ylabel("Total Error")
plt.grid(True)
plt.subplot(2, 3, 2)
colors = ['red' if label == 0 else 'green' for label in D]
plt.scatter(X[:, 0], X[:, 1], c=colors, s=100, edgecolors='k', label='Data')
if weights[1] != 0:
    x_vals = np.linspace(-0.2, 1.2, 100)
    y_vals = -(weights[0]*x_vals + weights[2]) / weights[1]
    plt.plot(x_vals, y_vals, 'b--', label='Decision Boundary')
else:
         plt.axvline(-weights[2]/weights[0],    color='blue',    linestyle='--',
label='Boundary')
plt.title("Decision Boundary")
plt.xlabel("Input 1")
plt.ylabel("Input 2")
plt.legend()
plt.grid(True)
# Plot 3: 3D Input visualization
ax = fig.add_subplot(2, 3, 3, projection='3d')
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=colors, s=80, edgecolor='k')
ax.set_title("Input Points (3D View)")
ax.set_xlabel("X1")
ax.set_ylabel("X2")
ax.set_zlabel("Bias")
ax.view_init(elev=20, azim=135)
plt.subplot(2, 3, 4)
bar_width = 0.35
index = np.arange(len(D))
plt.bar(index, D, bar_width, label='Target', color='lightblue')
plt.bar(index + bar_width, outputs, bar_width, label='Predicted', color='orange')
plt.title("Target vs Predicted Outputs")
plt.xlabel("Sample Index")
plt.ylabel("Output")
plt.xticks(index + bar_width / 2, ['S1', 'S2', 'S3', 'S4'])
plt.legend()
plt.grid(True)
```

```
plt.subplot(2, 3, 5)
weight_history = np.array(weight_history)
plt.plot(weight_history[:, 0], weight_history[:, 1], 'o-', label='W0 vs W1 path')
plt.scatter(weights[0], weights[1], color='red', label='Final weights')
plt.title("Weight Update Trajectory (W0 vs W1)")
plt.xlabel("W0")
plt.ylabel("W1")
plt.legend()
plt.grid(True)
plt.subplot(2, 3, 6)
plt.plot(weight_history[:, 0], label='W0', linestyle='-')
plt.plot(weight_history[:, 1], label='W1', linestyle='--')
plt.plot(weight_history[:, 2], label='Bias (W2)', linestyle=':')
plt.title("Weight Values Over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Weight Value")
plt.legend()
plt.grid(True)
plt.suptitle("SGD Learning with Delta Rule - Enhanced Visuals", fontsize=16)
plt.tight_layout()
plt.show()
```

Output:



7. Conclusion

The single-layer model trained using the delta learning rule and stochastic gradient descent successfully learned to classify the input patterns. The model converged after several updates, and the final weights formed a clear decision boundary between the two classes. This experiment demonstrates that the delta rule is effective for problems that are linearly separable, such as this logical AND operation.

Experiment Number: 04

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

Objective: The purpose of this experiment is to train a simple neural network model that can recognize and classify handwritten digits (0 to 9) using the MNIST dataset. This is a basic task in image classification and helps us understand how machines can learn to recognize patterns like humans do.

Required Libraries

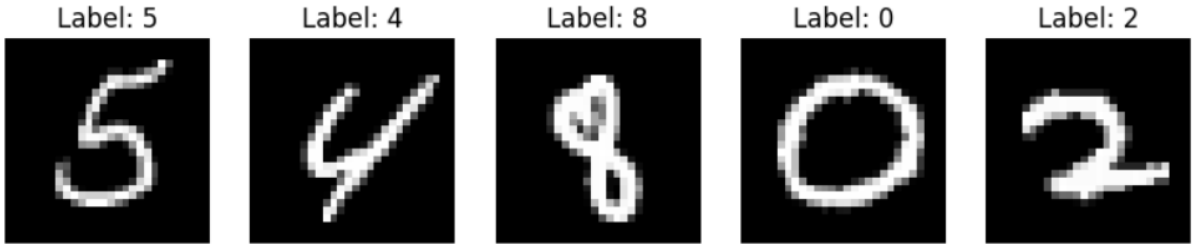
- scikit-learn – for loading data, model creation, training, and evaluation
- matplotlib, seaborn – for visualization
- numpy – for numerical operations
- warnings – to manage convergence warnings

Dataset Description

The experiment uses the **MNIST dataset**, which is a standard and widely-used dataset in machine learning. It contains:

- **70,000 grayscale images** of handwritten digits.
- Each image is **28×28 pixels**, which means it has **784 total values** (since 28 × 28 = 784).
- The images are labeled with numbers from 0 to 9, which represent the digit shown in each image.

MNIST is great for learning and testing basic machine learning and deep learning models. It's simple but meaningful.



3. Methodology

3.1 Data Preprocessing

Before training the model, the data needs to be prepared:

- The images are **flattened**: Instead of keeping them in 2D (28×28), they are converted into a 1D list of 784 numbers.
- The values are **normalized**: All pixel values, originally from 0 to 255, are scaled between 0 and 1. This helps the model learn faster and more accurately.

- The dataset is split into:
    - **Training set (80%)** – Used to teach the model.
    - **Testing set (20%)** – Used to check how well the model learned.

3.2 Model Selection

We used a **Feedforward Neural Network**, also known as a **Multilayer Perceptron (MLP)**.

What is a Feedforward Neural Network?

- It is a type of artificial neural network where information moves in one direction — from input to output.
- It has:
    - An **input layer** that takes the pixel values.
    - One or more **hidden layers** that process the data.
    - An **output layer** that gives the final prediction (0–9 digit).

Model Details:

- One hidden layer with 128 units (neurons).
- Uses the **ReLU** (Rectified Linear Unit) activation function to introduce non-linearity.
- Uses the **Adam optimizer**, which is an algorithm that helps the model adjust itself to minimize mistakes.
- The model is trained for multiple rounds (called **epochs**). In each epoch, it sees the training data once.

3.3 Training the Model

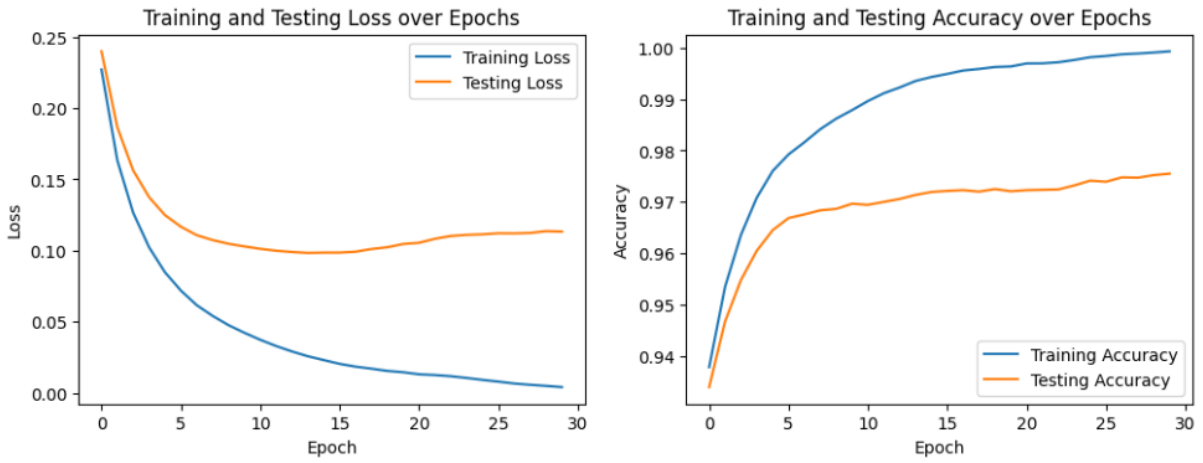
- The model was trained **for 30 epochs**, meaning it went through the entire training dataset 30 times.
- During each epoch, the model tried to improve its predictions.
- We recorded:
    - **Loss**: Measures how wrong the model's predictions are.
    - **Accuracy**: Measures how many predictions were correct.

Both **training** and **testing** loss and accuracy were tracked.

4. Results and Evaluation

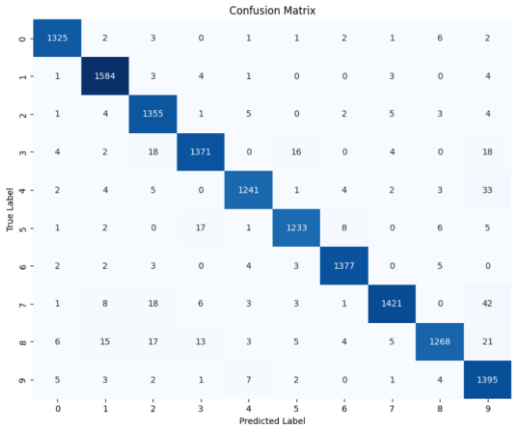
4.1 Final Accuracy

- The model achieved a **high accuracy on the test set**, meaning it learned to correctly classify most of the handwritten digits.



4.2 Confusion Matrix

- A **confusion matrix** was used to see where the model made mistakes.
- It shows how many times the model correctly or incorrectly predicted each digit.



For example:

- If the model predicted a "3" when it was actually a "5", this mistake is shown in the matrix.
- The diagonal of the matrix shows correct predictions; off-diagonal values are mistakes.

4.3 Classification Report

This report gives more detailed performance:

- **Precision**: Out of all predicted digit ‘3’s, how many were correct?
- **Recall**: Out of all actual digit ‘3’s, how many were found?
- **F1-Score**: A combined measure of precision and recall.

4.4 ROC Curve and AUC

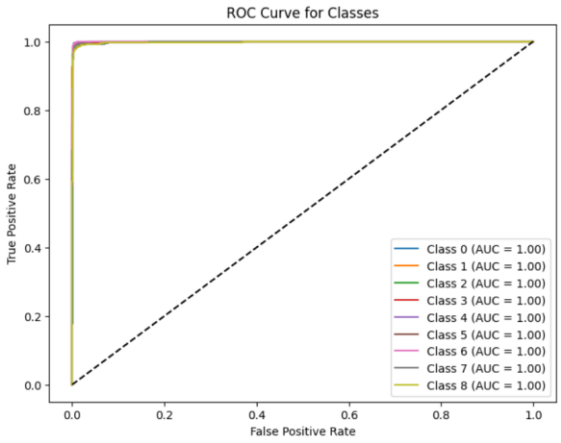
- A **ROC curve** was plotted for each digit. It helps visualize how well the model can separate one digit from the others.

- The **AUC (Area Under the Curve)** tells how good the model is: the higher the better (close to 1 means excellent).

```
Classification Report:
              precision    recall  f1-score   support

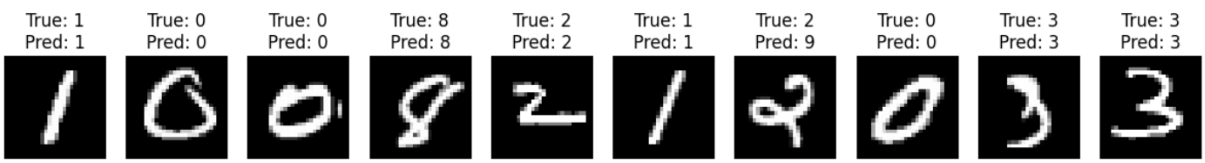
           0       0.98      0.99      0.98      1343
           1       0.97      0.99      0.98      1600
           2       0.95      0.98      0.97      1380
           3       0.97      0.96      0.96      1433
           4       0.98      0.96      0.97      1295
           5       0.98      0.97      0.97      1273
           6       0.98      0.99      0.99      1396
           7       0.99      0.95      0.97      1503
           8       0.98      0.93      0.96      1357
           9       0.92      0.98      0.95      1420

    accuracy                           0.97     14000
   macro avg       0.97      0.97      0.97     14000
weighted avg       0.97      0.97      0.97     14000
```



4.5 Visual Results

- A few random test images were shown with their **true digit** and the **model's predicted digit**.
- Most predictions were correct, showing the model's ability to recognize digits well.



5. Conclusion

This experiment successfully showed how a simple neural network can learn to recognize handwritten digits from images. Using the MNIST dataset:

- We trained a feedforward neural network using normalized pixel data.
- The model performed well, achieving high accuracy and good generalization.
- We visualized how learning improved over time (using loss and accuracy plots).
- We evaluated performance using classification metrics and visualization tools.

This lab introduced important concepts like **data preprocessing**, **model training**, **accuracy evaluation**, and **visual result interpretation**, all of which are crucial in machine learning and AI.

Python Code

```
import warnings
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
from sklearn.exceptions import ConvergenceWarning
import matplotlib.pyplot as plt

def load_data():
    mnist = fetch_openml('mnist_784', version=1)
    X = mnist.data.to_numpy() / 255.0  # Normalize pixel values to [0,1]
    y = mnist.target.astype(int).to_numpy()
    return train_test_split(X, y, test_size=0.2, random_state=42)

X_train, X_test, y_train, y_test = load_data()

import matplotlib.pyplot as plt

def show_samples(X, y, samples=5):
    plt.figure(figsize=(10, 2))
    for i in range(samples):
        plt.subplot(1, samples, i+1)
        plt.imshow(X[i].reshape(28, 28), cmap='gray')
        plt.title(f"Label: {y[i]}")
        plt.axis('off')
    plt.show()

show_samples(X_train, y_train)

from sklearn.neural_network import MLPClassifier
from sklearn.metrics import log_loss, accuracy_score
import numpy as np
import matplotlib.pyplot as plt

def train_model_epoch_by_epoch(X_train, y_train, X_test, y_test, epochs=100):
    model = MLPClassifier(
        hidden_layer_sizes=(128,),
        activation='relu',
        solver='adam',
        max_iter=1,
```

```
        warm_start=True,
        random_state=42,
        verbose=False
    )

    train_losses = []
    test_losses = []
    train_accs = []
    test_accs = []

    classes = np.unique(y_train)

    for epoch in range(epochs):
        # Train one epoch
        model.partial_fit(X_train, y_train, classes=classes)

        # Predict probabilities for loss calculation
        train_probs = model.predict_proba(X_train)
        test_probs = model.predict_proba(X_test)

        # Calculate log loss
        train_loss = log_loss(y_train, train_probs)
        test_loss = log_loss(y_test, test_probs)

        train_losses.append(train_loss)
        test_losses.append(test_loss)

        # Predict labels for accuracy
        train_preds = model.predict(X_train)
        test_preds = model.predict(X_test)

        train_acc = accuracy_score(y_train, train_preds)
        test_acc = accuracy_score(y_test, test_preds)

        train_accs.append(train_acc)
        test_accs.append(test_acc)

        print(f"Epoch {epoch+1}/{epochs} - "
              f"Train Loss: {train_loss:.4f}, Test Loss: {test_loss:.4f} - "
              f"Train Acc: {train_acc:.4f}, Test Acc: {test_acc:.4f}")

    return model, train_losses, test_losses, train_accs, test_accs



# Example call
model,     train_losses,     test_losses,     train_accs,     test_accs     =
train_model_epoch_by_epoch(X_train, y_train, X_test, y_test, epochs=30)

from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns

# Predict on test set
y_test_preds = model.predict(X_test)

# Final test accuracy
final_test_acc = accuracy_score(y_test, y_test_preds)
print(f"Final Test Accuracy: {final_test_acc:.4f}")
```

```
# Classification report
report = classification_report(y_test, y_test_preds)
print("Classification Report:\n", report)
# Confusion matrix
cm = confusion_matrix(y_test, y_test_preds)

plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()



from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
import numpy as np

# Assuming 'model' is your trained MLPClassifier, X_test, y_test are your test
data

# For multi-class ROC, binarize labels
y_test_bin = label_binarize(y_test, classes=np.arange(10))
y_score = model.predict_proba(X_test)

# ROC and AUC for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(10):
    fpr[i], tpr[i], _ = roc_curve(y_test_bin[:, i], y_score[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Plot ROC for first 3 classes as example
plt.figure(figsize=(8,6))
for i in range(9):
    plt.plot(fpr[i], tpr[i], label=f'Class {i} (AUC = {roc_auc[i]:.2f})')
plt.plot([0,1], [0,1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for Classes')
plt.legend(loc='lower right')
plt.show()

# Plotting both losses and accuracies
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training Loss')
plt.plot(test_losses, label='Testing Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Testing Loss over Epochs')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(train_accs, label='Training Accuracy')
plt.plot(test_accs, label='Testing Accuracy')
```

```
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Testing Accuracy over Epochs')
plt.legend()

plt.show()

import matplotlib.pyplot as plt
import numpy as np

# Predict on the test set
y_pred = model.predict(X_test)

def plot_random_test_images(images, true_labels, pred_labels, num_images=10):
    plt.figure(figsize=(15, 4))
    indices = np.random.choice(len(images), num_images, replace=False)
    for i, idx in enumerate(indices):
        plt.subplot(1, num_images, i + 1)
        plt.imshow(images[idx].reshape(28, 28), cmap='gray')
        plt.title(f"True: {true_labels[idx]}\nPred: {pred_labels[idx]}")
        plt.axis('off')
    plt.show()

# Show 10 random test images with true and predicted labels
plot_random_test_images(X_test, y_test, y_pred, num_images=10)
```

Experiment Number: 05

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

Objective: To build and evaluate a Convolutional Neural Network (CNN) that can automatically recognize and classify images into different categories using the CIFAR-10 dataset, which contains small color images of common objects like cats, dogs, cars, airplanes, etc.

Dataset Overview

- The **CIFAR-10** dataset contains **60,000 color images**, each of size **32x32 pixels**, divided into 10 different classes.
- Examples of classes: *airplane, car, bird, cat, deer, dog, frog, horse, ship, truck*.
- The dataset is split into:
    - **50,000 images** for training the model
    - **10,000 images** for testing how well the model works

Each image also comes with a label (a number) that indicates its class.

Data Preprocessing

Before we train the model, we do some cleaning and preparation:

- **Normalization**: We scale the pixel values of the images (originally between 0 and 255) to be between **0 and 1**. This helps the model learn more effectively.
- **Label Encoding**: The class labels (numbers) are converted into a special format called **one-hot encoding**. This is needed because our model gives probabilities for each class.

Data Augmentation

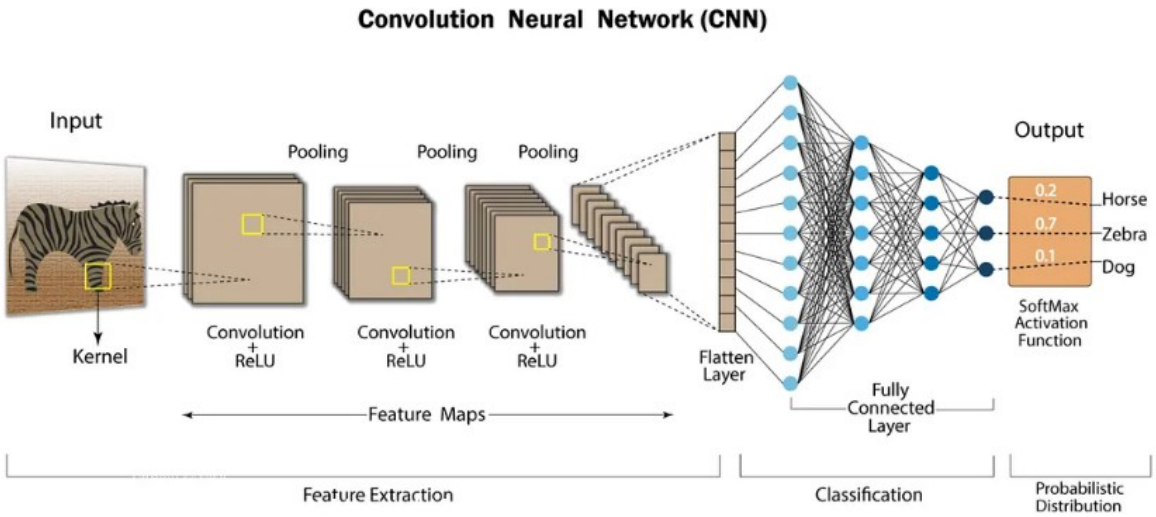
To help the model become more flexible and accurate, we apply **data augmentation**, which means we make slight changes to the training images, such as:

- Rotating them

- Shifting them left or right, up or down
- Zooming in
- Flipping them horizontally

This technique creates different versions of the same image, which helps the model handle real-world variations.

Convolutional Neural Network (CNN) Model



Convolutional Neural Network — CNN architecture

A CNN is a type of deep learning model that is especially good at recognizing patterns in images. Here's a breakdown of the layers used in the model and their roles:

Layer 1 & 2: Convolutional + Batch Normalization

- These layers help the model **detect basic features**, like edges and textures.
- We use **multiple filters** to scan across the image and find these features.
- **Activation function**: ReLU (Rectified Linear Unit) is applied to add non-linearity. This means the model can learn complex things, not just simple straight-line patterns.
- **Batch Normalization** stabilizes and speeds up training by keeping the values in the network well-balanced.

Layer 3: Max Pooling

- This layer **reduces the size** of the image but keeps the important parts.
- It helps reduce memory usage and makes the model faster.
- It also helps the model focus on the most important features.

Layer 4: Dropout

- Dropout is a technique to **prevent overfitting**. During training, it randomly removes some neurons temporarily so the model doesn't become too dependent on specific parts of the data.
- This helps the model perform better on new, unseen images.

Layer 5–8: Repeat of Convolution, Batch Normalization, Pooling, and Dropout

- These layers go deeper into the image features and learn **more abstract and complex patterns**, such as combinations of shapes or textures.

Layer 9: Flatten

- After all the image features are extracted, we **flatten** the image into a 1D array so it can be used by the final layers.

Layer 10: Fully Connected (Dense Layer)

- This layer takes the features and tries to **combine them to understand the object** in the image.
- It uses 512 neurons and ReLU activation for powerful learning.

Layer 11: Dropout Again

- Dropout is again applied here to prevent overfitting at this fully connected stage.

Final Layer: Output (Softmax)

- This layer has **10 outputs**, one for each class.
- It uses **softmax activation**, which converts numbers into probabilities, and the class with the highest probability is selected as the final prediction.

Optimizer and Loss Function

Optimizer: Adam

- **Adam** is an advanced optimization algorithm.
- It automatically adjusts how much the model should update itself at every step.
- It combines the best ideas from two other methods (momentum and RMSProp).
- It helps the model **train faster and more reliably**.

Loss Function: Categorical Crossentropy

- This function tells the model **how wrong its predictions are** during training.
- It compares the predicted class probabilities with the true class and helps the model learn to improve.

Training Process

- The model is trained for **multiple rounds (called epochs)**.

- It uses the **augmented images** and updates itself step-by-step.
- Two techniques are used to improve training:
    - **Early Stopping**: Stops training if the model isn't improving anymore, which saves time and avoids overfitting.
    - **Reduce Learning Rate on Plateau**: If progress slows down, the model will take smaller steps, allowing more precise improvements.

Evaluation and Results

After training, we test the model on **10,000 new images** that it has never seen before.

Accuracy

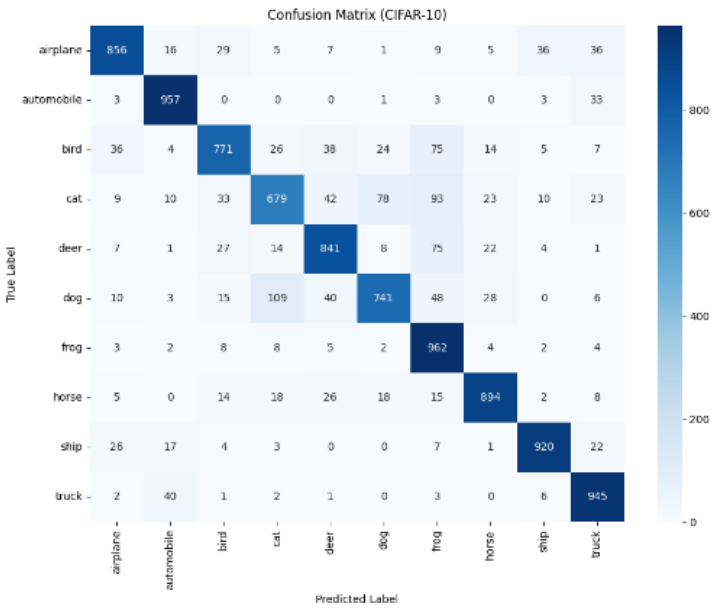
- The model correctly predicts the class of most test images.
- We calculate the **overall accuracy** and **per-class accuracy** to see which classes perform better or worse.

```
Classification Report:

              precision    recall  f1-score   support

    airplane       0.89      0.86      0.87      1000
  automobile       0.91      0.96      0.93      1000
        bird       0.85      0.77      0.81      1000
         cat       0.79      0.68      0.73      1000
        deer       0.84      0.84      0.84      1000
         dog       0.85      0.74      0.79      1000
        frog       0.75      0.96      0.84      1000
       horse       0.90      0.89      0.90      1000
        ship       0.93      0.92      0.93      1000
       truck       0.87      0.94      0.91      1000

    accuracy                           0.86     10000
   macro avg       0.86      0.86      0.86     10000
weighted avg       0.86      0.86      0.86     10000
```

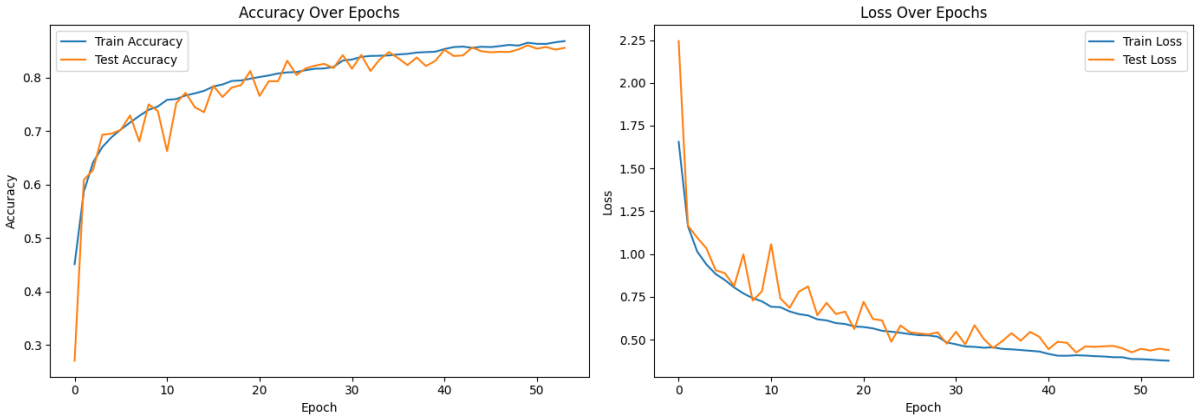


Confusion Matrix

- A confusion matrix is used to show how often the model correctly or incorrectly predicts each class.
- Each row shows actual classes, and each column shows predicted classes.

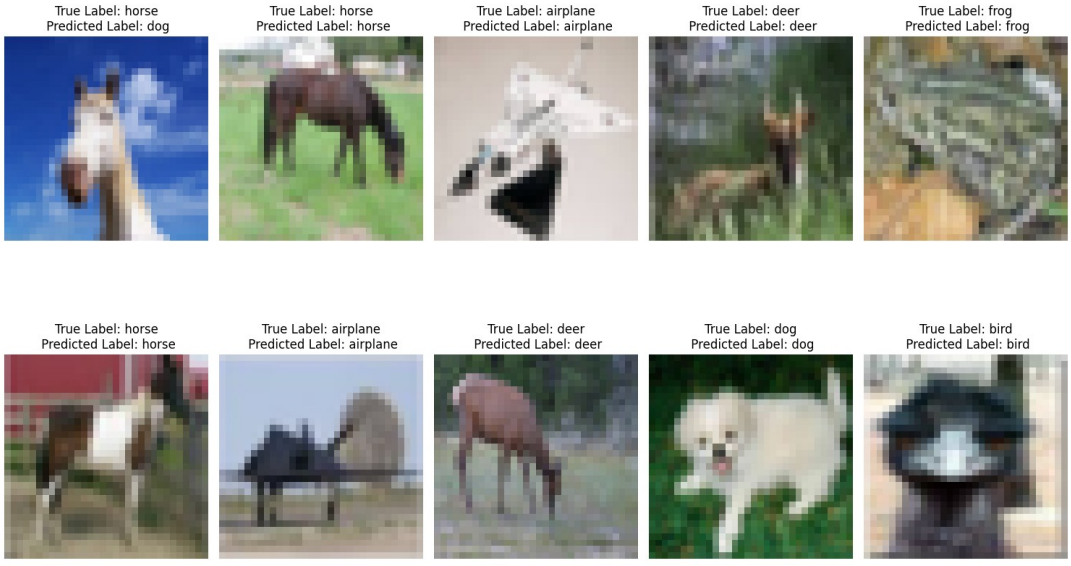
Graphs

- We plot how the **accuracy and loss changed over time** during training.
- This helps visualize whether the model improved steadily or faced problems like overfitting.



Sample Predictions

- A few test images are displayed along with their **true labels and the model's predicted labels**.
- This helps visually check how good the model is doing.
-



Conclusion

This experiment demonstrates how a Convolutional Neural Network (CNN) can effectively classify images into categories. Using techniques like data augmentation, dropout, and optimization, the model learns to recognize visual patterns with high accuracy. The use of various layers and functions in CNN helps the system understand everything from simple edges to complex shapes in an image.

Python Code:

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.metrics import classification_report, confusion_matrix
# Load CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

# Normalize pixel values
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
import matplotlib.pyplot as plt

plt.figure(figsize=(15, 6), dpi=100)
for i in range(10):
    plt.subplot(2, 5, i + 1)
    plt.imshow(x_train[i], interpolation='nearest')
    plt.title(class_names[y_train[i][0]])  # or .item()
    plt.axis('off')
plt.tight_layout()
plt.show()

num_classes = 10
y_train_cat = to_categorical(y_train, num_classes)
y_test_cat = to_categorical(y_test, num_classes)

# Data augmentation
datagen = ImageDataGenerator(
    rotation_range=10,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
    zoom_range=0.1
)
datagen.fit(x_train)
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.utils import to_categorical

# Define the CNN + Transformer model
# Define a simple CNN model
def simple_cnn(input_shape=(32, 32, 3), num_classes=10):
    model = models.Sequential([
        layers.Conv2D(32,        (3,3),        activation='relu',        padding='same',
input_shape=input_shape),
```

```
        layers.BatchNormalization(),
        layers.Conv2D(32, (3,3), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2,2)),
        layers.Dropout(0.25),

        layers.Conv2D(64, (3,3), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.Conv2D(64, (3,3), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2,2)),
        layers.Dropout(0.25),

        layers.Flatten(),
        layers.Dense(512, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation='softmax')
    ])
    return model



# Compile model
model = simple_cnn(input_shape=x_train.shape[1:], num_classes=num_classes)
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Callbacks
early_stopping          =          EarlyStopping(monitor='val_loss',          patience=10,
restore_best_weights=True)
lr_scheduler = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5, min_lr=1e-6)


model.summary()
# Train model with data augmentation
history = model.fit(
    datagen.flow(x_train, y_train_cat, batch_size=128),
    epochs=100,
    validation_data=(x_test, y_test_cat),
    callbacks=[early_stopping, lr_scheduler],
    verbose=2
)
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
import numpy as np

# Get predicted class labels from model
y_pred_probs = model.predict(x_test)
y_pred = np.argmax(y_pred_probs, axis=1)
y_true = y_test.flatten()

# Classification Report
report = classification_report(y_true, y_pred, target_names=class_names)
print("Classification Report:\n")
print(report)

# Calculate overall test accuracy
accuracy = np.mean(y_pred == y_true)
print(f"Overall Test Accuracy: {accuracy:.4f}")

# Per-class accuracy
correct_per_class = np.zeros(10)
total_per_class = np.zeros(10)
```

```
for true, pred in zip(y_true, y_pred):
    total_per_class[true] += 1
    if true == pred:
        correct_per_class[true] += 1

print("\nPer-Class Accuracy:")
for i in range(10):
    acc = correct_per_class[i] / total_per_class[i]
    print(f"{class_names[i]}: {acc:.4f}")

# Generate confusion matrix
cm = confusion_matrix(y_true, y_pred)

# Display raw confusion matrix
print("Confusion Matrix:\n")
print(cm)

# Plot confusion matrix heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
            xticklabels=class_names,
            yticklabels=class_names)
plt.title("Confusion Matrix (CIFAR-10)")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.tight_layout()
plt.show()

plt.figure(figsize=(14, 5), dpi=100)

# Loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Test Loss')
plt.title('Loss Over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

# Accuracy
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Test Accuracy')
plt.title('Accuracy Over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()

import random

plt.figure(figsize=(15, 10))
indices = random.sample(range(len(x_test)), 10)

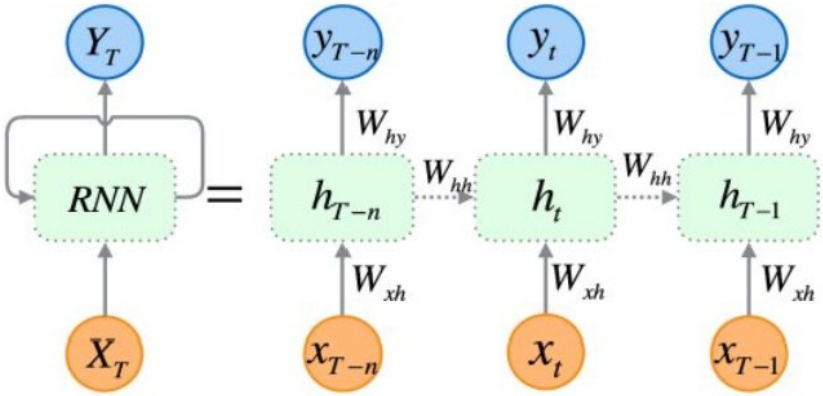
for i, idx in enumerate(indices):
    plt.subplot(2, 5, i+1)
    plt.imshow(x_test[idx])
    true_label = class_names[y_test[idx][0]]
    pred_label = class_names[y_pred[idx]]
    plt.title(f"True Label: {true_label}\n Predicted Label: {pred_label}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Experiment Number: 06

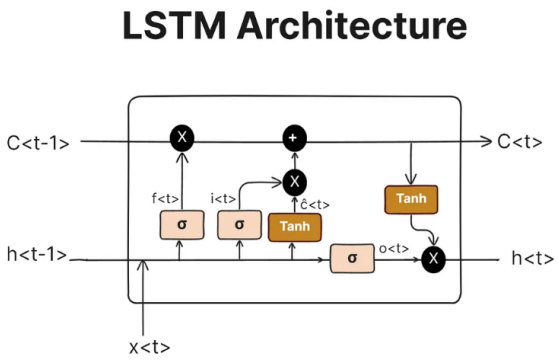
Name of the Experiment: Write a program to evaluate a Recurrent Neural Network (RNN) for text classification.

Objective: The goal is to build a model that can read movie reviews and decide whether each review expresses a positive or negative opinion. This task is known as **binary text classification** because there are two categories: positive and negative.

Recurrent Neural Network (RNN): A **Recurrent Neural Network (RNN)** is a type of neural network specially designed to process **sequence data**, such as sentences, speech, or time series data. Unlike traditional neural networks that treat inputs independently, RNNs have a form of **memory** that allows them to use information from previous steps in the sequence to influence the current step's output. This is crucial for understanding context, since the meaning of words often depends on the words before them.



A Special Kind of RNN, LSTM (Long Short-Term Memory)



An **LSTM (Long Short-Term Memory)** network is a type of RNN that overcomes a key problem faced by basic RNNs: the difficulty in remembering information over long sequences. Simple RNNs tend to forget earlier information as they process longer texts, which limits their ability to capture important context far back in the sequence.

LSTMs include special components called **gates** that control the flow of information. These gates decide which parts of the previous information to keep, which to discard, and what new information to add. This structure allows LSTMs to maintain important information over longer sequences, making them much better at tasks like understanding the sentiment of entire reviews.

Because LSTM is designed with this memory control mechanism, it is considered a **special kind of RNN** that is more effective for long sequences.

Dataset and Data Preparation

The model uses the **IMDb movie reviews dataset**, which contains thousands of reviews labeled as positive or negative. To make the data manageable and uniform:

- Only the **10,000 most frequent words** are considered. Less frequent words are ignored to reduce complexity.
- Each review is processed so that its length is exactly **200 words** by cutting longer reviews and padding shorter ones with a special symbol. This ensures all input sequences have the same length, which is necessary for training the model efficiently.

```
Review #1
[START] this has to be one of the worst films of the 1990s when my friends i were watching this film being the target audience it was aimed at we j
Label: Positive
--------------------------------------------------------------------------------------------------------

Review #2
[START] the [UNK] [UNK] at storytelling the traditional sort many years after the event i can still see in my [UNK] eye an elderly lady my friend's
Label: Negative
--------------------------------------------------------------------------------------------------------

Review #3
[START] worst mistake of my life br br i picked this movie up at target for 5 because i figured hey it's sandler i can get some cheap laughs i was
Label: Negative
```

Model Architecture Explained

- **Word Embedding Layer:** Each word is converted into a vector (a list of numbers) that captures its meaning and relationship with other words. This allows the model to understand semantic similarities between words.
- **Bidirectional LSTM Layers:** The model has two layers of LSTM that read the sequence both **forward and backward**. This helps the model capture context from both past and future words in the sentence, improving understanding.
    - The first LSTM layer has 64 units (neurons).
    - The second LSTM layer has 32 units.
    - Both layers use **regularization** (L2 penalty) to prevent overfitting, which means the model is discouraged from simply memorizing training data.
- **Batch Normalization and Dropout:** Batch normalization stabilizes and speeds up training by normalizing the output of each layer. Dropout randomly deactivates some neurons during training, which helps the model generalize better to new data by preventing over-reliance on any single neuron.
- **Self-Attention Layer:** This layer allows the model to assign different importance to different words in the review. Instead of treating all words equally, the attention mechanism helps the model focus more on the words or phrases that are most relevant for determining the sentiment.
- **Dense (Fully Connected) Layers:** After extracting features from the sequence, the model uses dense layers to combine this information and produce the final prediction.

The last layer uses a sigmoid activation function, which outputs a value between 0 and 1, interpreted as the probability of the review being positive.

Training Procedure

The model learns by comparing its predicted outputs to the actual labels (positive or negative) and minimizing the difference using a **loss function** called binary crossentropy, appropriate for two-class problems.

An **Adam optimizer** is used to adjust the model parameters efficiently.

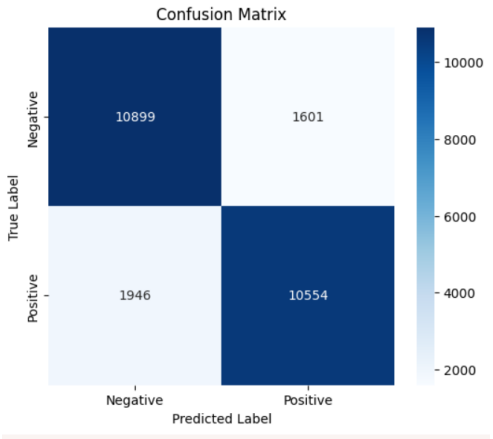
Two techniques improve training:

- **Early Stopping:** Training stops if the validation accuracy does not improve for several epochs, which prevents overfitting.
- **Learning Rate Reduction:** If the model stops improving, the learning rate is reduced to allow finer adjustments.

Model Evaluation

After training, the model is tested on new reviews to check its performance.

- **Accuracy** measures the percentage of reviews correctly classified.
- A **confusion matrix** shows how many reviews were correctly or incorrectly predicted for each class.



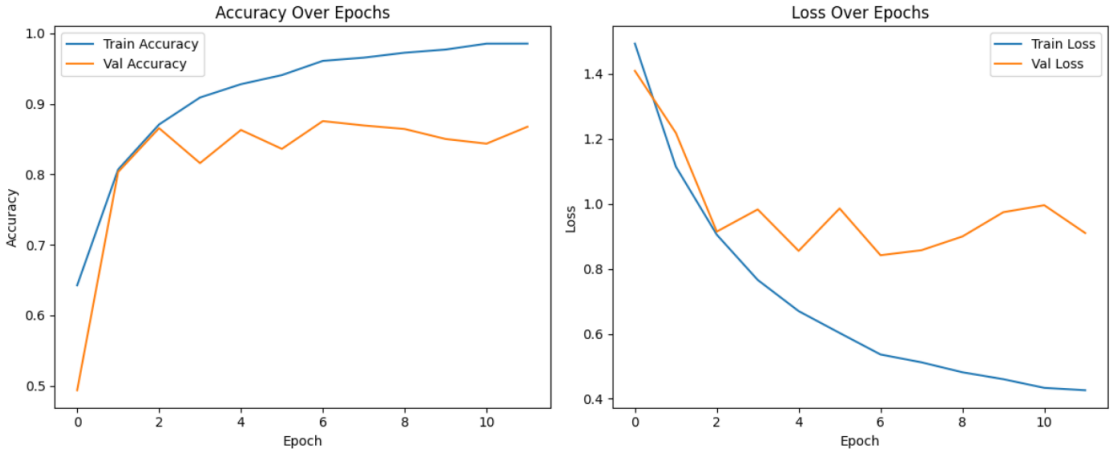
- A **classification report** provides detailed metrics like precision, recall, and F1-score, which give insight into how well the model distinguishes positive and negative reviews.

	precision	recall	f1-score	support
Negative	0.85	0.87	0.86	12500
Positive	0.87	0.84	0.86	12500
accuracy			0.86	25000
macro avg	0.86	0.86	0.86	25000
weighted avg	0.86	0.86	0.86	25000

Visualization of Training

Graphs display the model's accuracy and loss during training and validation phases. This helps identify if the model is learning well or overfitting.

The confusion matrix is shown as a heatmap, making it easier to interpret prediction errors visually.



Making Predictions

For each review, the model outputs a score between 0 and 1 representing the confidence that the review is positive. Scores greater than 0.5 are classified as positive, while scores below 0.5 are classified as negative.

The words of the review can be decoded back from their numeric representation into readable text for inspection.

```
Review:
lucio don't torture a paints an exceptionally portrait of small town plagued by series of brutal murders of young boys this surprisingly well directed

True Label: Positive
Predicted Label: Positive
```

Python Code:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout, BatchNormalization,
Bidirectional
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.datasets import imdb
import matplotlib.pyplot as plt
import numpy as np
def preprocess_data(vocab_size=10000, maxlen=200):
    """
    Loads and preprocesses the IMDb dataset for binary text classification.
    Returns padded sequences and labels for training and testing.
    """
    (x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=vocab_size)

    # Pad sequences to fixed length for uniform input size
    x_train = pad_sequences(x_train, maxlen=maxlen, padding='post')
    x_test = pad_sequences(x_test, maxlen=maxlen, padding='post')

    return x_train, y_train, x_test, y_test, vocab_size, maxlen
x_train, y_train, x_test, y_test, vocab_size, maxlen = preprocess_data()
```

```
# Load IMDB dataset
(x_train_raw, y_train), (x_test_raw, y_test) = imdb.load_data(num_words=vocab_size)

# Word index for decoding
word_index = imdb.get_word_index()
reverse_word_index = {value + 3: key for key, value in word_index.items()}
reverse_word_index[0] = "[PAD]"
reverse_word_index[1] = "[START]"
reverse_word_index[2] = "[UNK]"
reverse_word_index[3] = "[UNUSED]"

def decode_review(text):
    return ' '.join([reverse_word_index.get(i, '?') for i in text])

for i in range(3):
    print(f"\nReview #{i+1}")
    print(decode_review(x_train_raw[i+2]))
    print("Label:", "Positive" if y_train[i] == 1 else "Negative")
    print("-" * 100)

x_train = pad_sequences(x_train_raw, maxlen=maxlen)
x_test = pad_sequences(x_test_raw, maxlen=maxlen)

import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Embedding, Bidirectional, LSTM, Dense, Dropout,
BatchNormalization
from tensorflow.keras.layers import Layer
from tensorflow.keras.regularizers import l2
from tensorflow.keras.optimizers import Adam

# Custom self-attention layer
class SelfAttention(Layer):
    def __init__(self, **kwargs):
        super(SelfAttention, self).__init__(**kwargs)

    def build(self, input_shape):
        self.W = self.add_weight(shape=(input_shape[-1], input_shape[-1]),
                                 initializer='glorot_uniform',
                                 trainable=True)
        self.b = self.add_weight(shape=(input_shape[-1],),
                                 initializer='zeros',
                                 trainable=True)
        self.u = self.add_weight(shape=(input_shape[-1], 1),
                                 initializer='glorot_uniform',
                                 trainable=True)
        super(SelfAttention, self).build(input_shape)

    def call(self, inputs):
        u_it = tf.nn.tanh(tf.tensordot(inputs, self.W, axes=1) + self.b)
        a_it = tf.nn.softmax(tf.tensordot(u_it, self.u, axes=1), axis=1)
        output = tf.reduce_sum(inputs * a_it, axis=1)
        return output

# Model definition
input_layer = Input(shape=(maxlen,))
x = Embedding(input_dim=vocab_size, output_dim=256, input_length=maxlen)(input_layer)

x = Bidirectional(LSTM(64, return_sequences=True, kernel_regularizer=l2(0.001)))(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)

x = Bidirectional(LSTM(32, return_sequences=True, kernel_regularizer=l2(0.001)))(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
```

```
x = SelfAttention()(x)

x = Dense(64, activation='relu', kernel_regularizer=l2(0.001))(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)

output_layer = Dense(1, activation='sigmoid')(x)

model = Model(inputs=input_layer, outputs=output_layer)

# Compile model
model.compile(
    loss='binary_crossentropy',
    optimizer=Adam(learning_rate=1e-4),
    metrics=['accuracy']
)

model.summary()



# 4. Define callbacks for better training
early_stopping = EarlyStopping(
    monitor='val_accuracy',
    patience=5,
    restore_best_weights=True
)

lr_scheduler = ReduceLROnPlateau(
    monitor='val_accuracy',
    factor=0.5,
    patience=3,
    min_lr=1e-6
)
# 5. Train the model
history = model.fit(
    x_train, y_train,
    epochs=20,
    batch_size=128,
    validation_split=0.2,
    callbacks=[early_stopping, lr_scheduler],
    verbose=2
)
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Predict
y_pred_probs = model.predict(x_test)
y_pred = (y_pred_probs > 0.5).astype("int32")

# Classification Report
print(classification_report(y_test, y_pred, target_names=['Negative', 'Positive']))
# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)

plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Negative', 'Positive'],
yticklabels=['Negative', 'Positive'])
plt.title("Confusion Matrix")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()

plt.figure(figsize=(12, 5))
```

```
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label="Train Accuracy")
plt.plot(history.history['val_accuracy'], label="Val Accuracy")
plt.title("Accuracy Over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label="Train Loss")
plt.plot(history.history['val_loss'], label="Val Loss")
plt.title("Loss Over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()

plt.tight_layout()
plt.show()

# Load the word index from IMDb dataset
word_index = imdb.get_word_index()
reverse_word_index = {value + 3: key for key, value in word_index.items()}
reverse_word_index[0] = "[PAD]"
reverse_word_index[1] = "[START]"
reverse_word_index[2] = "[UNK]"
reverse_word_index[3] = "[UNUSED]"

# Function to decode review properly
def decode_review(sequence):
    words = [reverse_word_index.get(i, '?') for i in sequence if i > 3]
    return ' '.join(words)

import random

# Pick a random test index
idx = random.randint(0, len(x_test) - 1)

# Predict the label for this review
pred = model.predict(np.expand_dims(x_test[idx], axis=0), verbose=0)[0][0]
pred_label = "Positive" if pred > 0.5 else "Negative"

# Print decoded review and results
print("Review:")
print(decode_review(x_test[idx]))
print("\nTrue Label:", "Positive" if y_test[idx] else "Negative")
print("Predicted Label:", pred_label)
```

Experiment Number: 07

Name of the Experiment: Write a program to evaluate a Transformer model for text classification.

Objective: Sentiment analysis is a natural language processing (NLP) task that involves classifying a piece of text as expressing a positive, negative, or neutral opinion. This lab focuses on binary sentiment classification, where the goal is to classify movie reviews as either positive or negative.

In recent years, **BERT (Bidirectional Encoder Representations from Transformers)** has become one of the most powerful tools in NLP due to its deep understanding of context and relationships in language. In this experiment, BERT is used to classify sentiment in the IMDB movie review dataset.

Dataset Description: The IMDB dataset is a commonly used benchmark for sentiment classification. It contains 50,000 movie reviews in total, labeled as either positive or negative. However, to reduce computational complexity and focus on a balanced training process, we use:

- 2,500 positive and 2,500 negative reviews for training.
- 400 positive and 400 negative reviews for testing.

This balancing helps prevent the model from being biased toward one sentiment class.

Preprocessing Steps The dataset is processed in several stages before it can be fed into the BERT model:

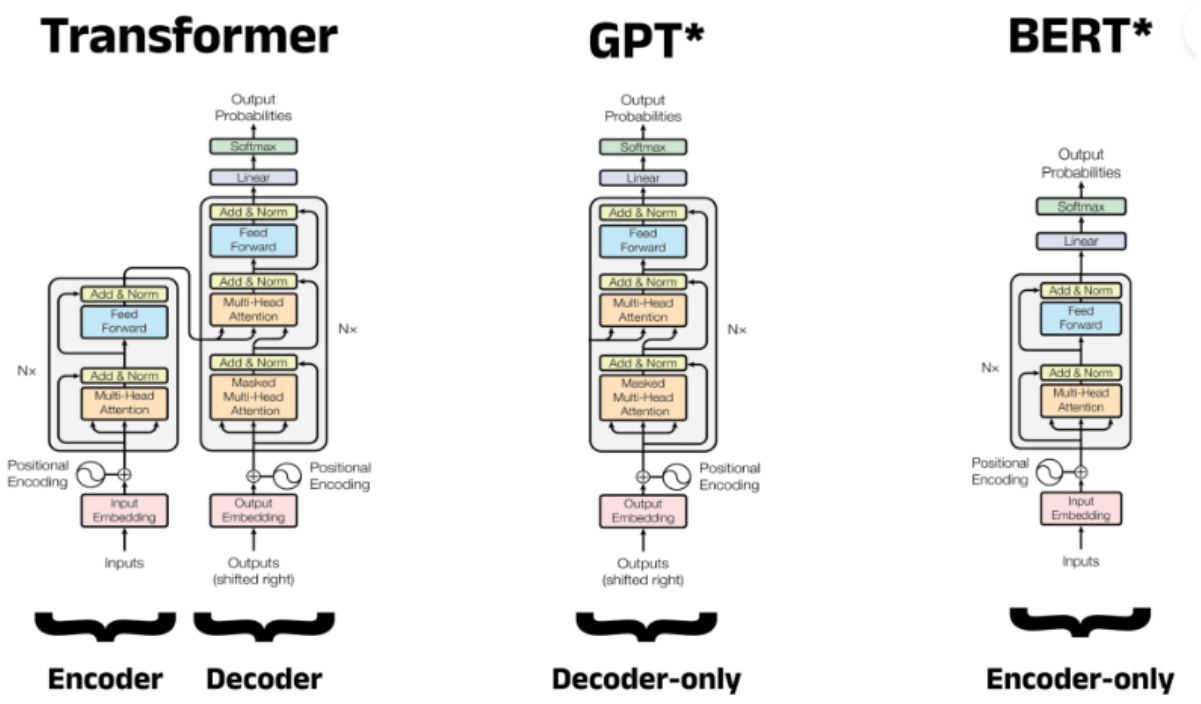
1. **Loading the dataset** The IMDB dataset is loaded using the datasets library.
2. **Filtering samples** Reviews are filtered to include only the ones labeled as positive (label = 1) or negative (label = 0).
3. **Balancing the dataset** Equal numbers of positive and negative reviews are selected for both the training and test sets.
4. **Tokenization** Each review is converted into a sequence of tokens using the BertTokenizer. This includes:
    - Breaking the review into tokens (words or subwords).
    - Padding or truncating each review to a fixed length (256 tokens in this case).
    - Creating attention masks that tell the model which tokens are actual input and which are padding.
5. **Formatting for training** The processed data is converted into PyTorch tensors and loaded into data loaders for batch processing during training.

Model Architecture

1 BERT Overview

BERT is a pre-trained language model that understands language in a deeply contextual way. It reads text **bidirectionally**, meaning it looks at both the left and right context of each word at once. This is what makes it more effective than older models like LSTMs or unidirectional Transformers.

BERT is based on the **Transformer** architecture, which is why it is often called a Transformer-based model.



2 BERT is Called a Transformer

BERT is called a Transformer because it is built entirely on the **encoder part of the Transformer architecture**. The Transformer was first introduced in the paper *"Attention is All You Need"*, which proposed a new way of processing sequences using a mechanism called **self-attention**.

The key points are:

- BERT uses **multiple layers of Transformer encoders** (12 in base, 24 in large).
- Each encoder layer includes:
    - Multi-head self-attention mechanism.
    - Feed-forward neural network.
    - Layer normalization and residual connections.

Because BERT relies entirely on this encoder stack and does not include the decoder (used in tasks like text generation), it is considered a **Transformer encoder-only model**.

3 Components of BERT Used

When we use BERT for classification, we mainly use the following:

- **Token Embeddings:** Each word is represented as a dense vector.
- **Positional Embeddings:** Since Transformers do not have any concept of word order by default, positional embeddings are added to help understand the position of each word in the sequence.
- **CLS Token:** A special [CLS] token is added at the beginning of each input. BERT uses the final hidden state of this token as the overall representation of the input sentence for classification tasks.
- **Attention Mask:** Used to tell the model which tokens are real and which are just padding.

The model outputs a vector (called pooler_output) representing the entire input review, which is then passed to a classifier.

Classifier Design

A small neural network is built on top of BERT:

- A single fully connected (linear) layer is added on top of BERT's output.
- This layer outputs a single value, which is passed through a sigmoid activation to predict the probability of the review being positive.
- The classifier is trained using Binary Cross Entropy with Logits Loss (BCEWithLogitsLoss), which is suitable for binary classification problems.

Training Process

The model is trained using the following steps:

- **Device Setup:** The model is trained using GPU if available for faster performance.
- **Optimizer:** Adam optimizer is used with a learning rate of 2e-5.
- **Gradient Scaling:** GradScaler and autocast are used to enable mixed-precision training, which speeds up training and reduces memory usage.
- **Epochs:** The model is trained for 20 epochs.
- **Metrics:** During each epoch, training loss and accuracy are calculated. After each epoch, validation loss and accuracy are also measured using the test dataset.

Evaluation

After training, the model is evaluated using several metrics:

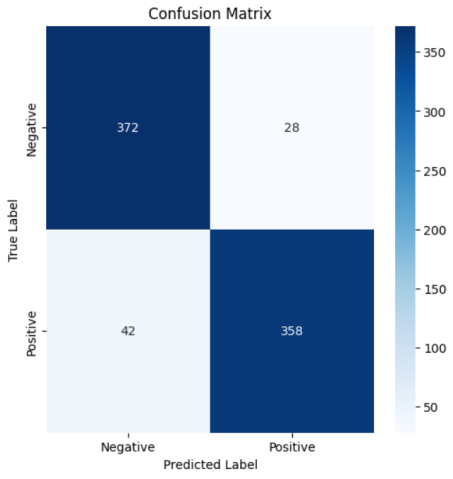
- **Test Accuracy:** The percentage of correct predictions on the test set.
- ```
  Test Accuracy: 0.9125
  ```
- **Classification Report:** This includes precision, recall, and F1-score for both positive and negative classes.

```
Classification Report (Test Data):
              precision    recall  f1-score   support

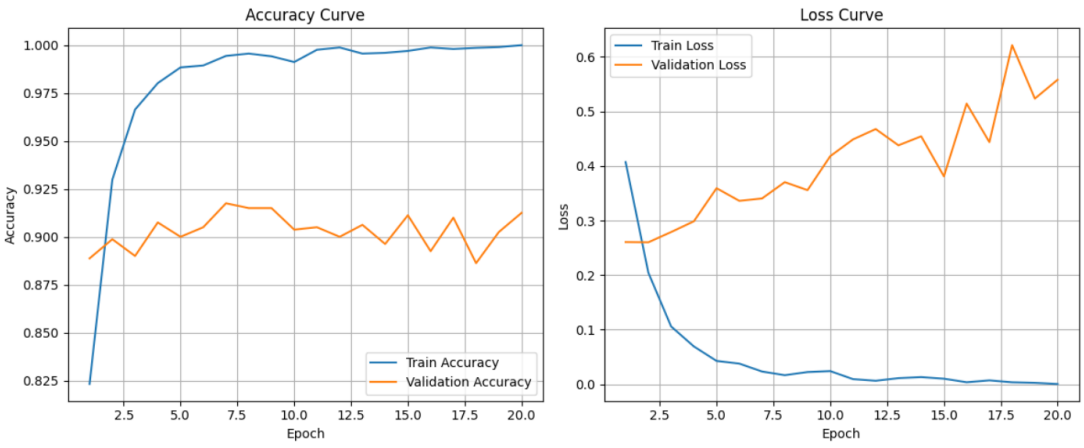
         0.0       0.90      0.93      0.91       400
         1.0       0.93      0.90      0.91       400

    accuracy                           0.91       800
   macro avg       0.91      0.91      0.91       800
weighted avg       0.91      0.91      0.91       800
```

- **Confusion Matrix:** This shows how many true positives, true negatives, false positives, and false negatives were predicted.



- **Graphs:** Loss and accuracy curves are plotted over the epochs to show the model's learning progress.


```

Prediction on New Data

To test the model's real-world usefulness, a prediction function is written to classify new user input. This function:

- Tokenizes the input review.
- Passes it through the trained BERT classifier.
- Outputs the sentiment label (positive or negative) along with a confidence score.

Two example inputs show how the model handles both positive and negative reviews.

```
Input Text: Every one should watch the movie at least once
Predicted Sentiment: Positive (Confidence: 0.67)

Input Text: This movie was boring and didn't make sense.
Predicted Sentiment: Negative (Confidence: 0.00)
```

Results Summary

- The model achieves high accuracy on the test set, showing strong performance on unseen data.
- Training and validation curves show steady learning without overfitting.
- The classification report and confusion matrix confirm balanced performance on both classes.
- Custom text predictions demonstrate the model's real-world ability to understand sentiment.

Conclusion

This experiment demonstrates how powerful pre-trained Transformer models like BERT can be for sentiment classification. By using only a small, balanced dataset and applying BERT with minimal fine-tuning, we achieve excellent classification results.

The success of BERT comes from its deep bidirectional attention, strong language understanding, and efficient encoding of input sequences through Transformer architecture. This lab confirms that Transformer-based models are well-suited for modern NLP tasks like sentiment analysis.

Python Code:

```
import os
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from transformers import BertTokenizer, BertModel
from datasets import load_dataset
from torch.amp import autocast, GradScaler
from tqdm import tqdm
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
```

```
import seaborn as sns
import numpy as np
from datasets import concatenate_datasets
# Optional: Disable symlink warning
os.environ["HF_HUB_DISABLE_SYMLINKS_WARNING"] = "1"

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

# Load the full IMDB dataset
dataset = load_dataset("imdb")

# Print label distribution for better understanding
print("Training dataset label distribution:",
dataset["train"].features["label"].names)
print("Test dataset label distribution:",
dataset["test"].features["label"].names)
# Separate the positive and negative samples from the training and test sets
positive_train_data = dataset["train"].filter(lambda example: example["label"] ==
1)
negative_train_data = dataset["train"].filter(lambda example: example["label"] ==
0)

positive_test_data = dataset["test"].filter(lambda example: example["label"] ==
1)
negative_test_data = dataset["test"].filter(lambda example: example["label"] ==
0)



# Balance the datasets by selecting an equal number of samples from both classes
balanced_train_data =
concatenate_datasets([positive_train_data.select(range(2500)),
negative_train_data.select(range(2500))])
balanced_test_data =
concatenate_datasets([positive_test_data.select(range(400)),
negative_test_data.select(range(400))])

# Load tokenizer
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")

# Tokenization function
def tokenize_function(example):
    return tokenizer(example["text"], padding="max_length", truncation=True,
max_length=256)

print("Tokenizing...")
tokenized_train = balanced_train_data.map(tokenize_function, batched=True)
tokenized_test = balanced_test_data.map(tokenize_function, batched=True)
tokenized_train.set_format(type='torch', columns=['input_ids', 'attention_mask',
'label'])
tokenized_test.set_format(type='torch', columns=['input_ids', 'attention_mask',
'label'])

# DataLoader
train_loader = DataLoader(tokenized_train, batch_size=64, shuffle=True)
test_loader = DataLoader(tokenized_test, batch_size=64, shuffle= False)
# Model definition
```

```
class BERTClassifier(nn.Module):
    def __init__(self, bert):
        super(BERTClassifier, self).__init__()
        self.bert = bert
        self.fc = nn.Linear(bert.config.hidden_size, 1)

    def forward(self, input_ids, attention_mask):
        outputs = self.bert(input_ids=input_ids, attention_mask=attention_mask)
        cls_output = outputs.pooler_output
        return self.fc(cls_output)

model           =           BERTClassifier(BertModel.from_pretrained("bert-base-
uncased")).to(device)

# Loss and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(), lr=2e-5)
scaler = GradScaler(device)
import matplotlib.pyplot as plt
from tqdm import tqdm

train_losses, val_losses = [], []
train_accuracies, val_accuracies = [], []

for epoch in range(20):  # 20 epochs
    model.train()
    total_loss = 0
    correct = 0
    total = 0
    loop = tqdm(train_loader, desc=f"Epoch {epoch+1}")

    for batch in loop:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['label'].float().unsqueeze(1).to(device)  # shape: (B, 1)

        optimizer.zero_grad()
        with   autocast(device_type='cuda'   if   torch.cuda.is_available()   else
'cpu'):
            outputs = model(input_ids, attention_mask)
            loss = criterion(outputs, labels)

        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

        total_loss += loss.item()
        preds = (torch.sigmoid(outputs) > 0.5).float()
        correct += (preds == labels).sum().item()
        total += labels.size(0)

        loop.set_postfix(loss=total_loss / (total / 8), acc=correct / total)

    # Record training metrics
    epoch_loss = total_loss / len(train_loader)
    epoch_acc = correct / total
    train_losses.append(epoch_loss)
    train_accuracies.append(epoch_acc)
```

```
    # Validation
    model.eval()
    val_loss = 0
    val_correct = 0
    val_total = 0

    with torch.no_grad():
        for batch in test_loader:
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['label'].float().unsqueeze(1).to(device)

            outputs = model(input_ids, attention_mask)
            loss = criterion(outputs, labels)

            val_loss += loss.item()
            preds = (torch.sigmoid(outputs) > 0.5).float()
            val_correct += (preds == labels).sum().item()
            val_total += labels.size(0)

    val_losses.append(val_loss / len(test_loader))
    val_accuracies.append(val_correct / val_total)

    print(f"Epoch {epoch+1} | Train Loss: {epoch_loss:.4f}, Acc: {epoch_acc:.4f}
| Val Loss: {val_losses[-1]:.4f}, Acc: {val_accuracies[-1]:.4f}")

# Evaluation of the model
model.eval()
correct = 0
total = 0
all_preds = []
all_labels = []

with torch.no_grad():
    for batch in test_loader:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['label'].float().unsqueeze(1).to(device)

        with  autocast(device_type='cuda'  if  torch.cuda.is_available()  else
'cpu'):
            outputs = model(input_ids, attention_mask)
            preds = (torch.sigmoid(outputs) > 0.5).float()

            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

            correct += (preds == labels).sum().item()
            total += labels.size(0)
accuracy = correct / total
print(f"Test Accuracy: {accuracy:.4f}")
# Classification Report
print("\nClassification Report (Test Data):")
print(classification_report(all_labels, all_preds))
# Confusion Matrix
cm = confusion_matrix(all_labels, all_preds, labels=[0.0, 1.0])

# Plot Confusion Matrix
plt.figure(figsize=(6,6))
```

```
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=["Negative",
"Positive"], yticklabels=["Negative", "Positive"])
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.title('Confusion Matrix')
plt.show()
import matplotlib.pyplot as plt

epochs = range(1, len(train_losses) + 1)  # dynamically set length

plt.figure(figsize=(12, 5))

# Plotting Loss
plt.subplot(1, 2, 2)
plt.plot(epochs, train_losses, label='Train Loss')
if len(val_losses) == len(train_losses):  # Safe check
    plt.plot(epochs, val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss Curve')
plt.legend()
plt.grid(True)
# Plotting Accuracy
plt.subplot(1, 2, 1)
plt.plot(epochs, train_accuracies, label='Train Accuracy')
if len(val_accuracies) == len(train_accuracies):
    plt.plot(epochs, val_accuracies, label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Accuracy Curve')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Define a function to predict sentiment of custom text input
def predict_sentiment(text, model, tokenizer, device):
    model.eval()
    with torch.no_grad():
        # Tokenize input
        encoding = tokenizer(
            text,
            return_tensors='pt',
            truncation=True,
            padding='max_length',
            max_length=256
        )
        input_ids = encoding['input_ids'].to(device)
        attention_mask = encoding['attention_mask'].to(device)

        # Predict
        output = model(input_ids, attention_mask)
        prob = torch.sigmoid(output)

        # Result
        sentiment = "Positive" if prob.item() > 0.5 else "Negative"
        print(f"\nInput Text: {text}")
        print(f"Predicted       Sentiment:       {sentiment}       (Confidence:
{prob.item():.2f})")
```

```
# Example: Try predicting sentiment for new input
user_input = "Every one should watch the movie at least once"
predict_sentiment(user_input, model, tokenizer, device)

user_input = "This movie was boring and didn't make sense."
predict_sentiment(user_input, model, tokenizer, device)
```

Experiment Number: 08

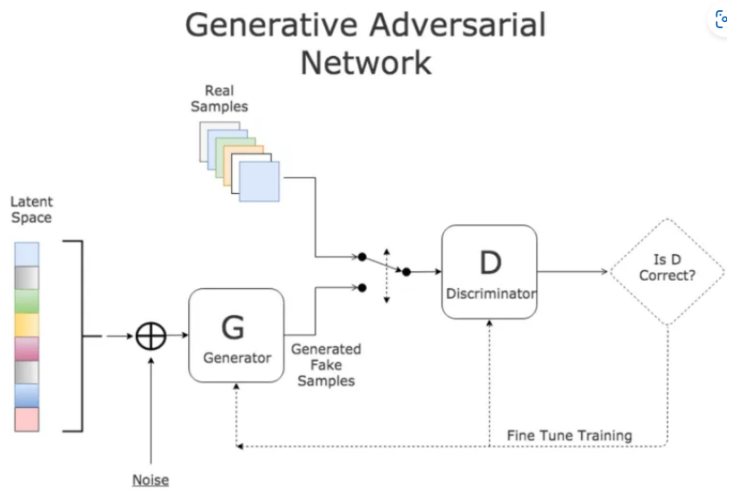
Name of the Experiment: Write a program to evaluate Generative Adversarial Network (GAN) for image generation.

Objective: Generative Adversarial Networks (GANs) are a class of deep learning models designed to generate new, synthetic data samples that closely resemble real data. They have revolutionized image synthesis by enabling the creation of highly realistic images, even from random noise inputs.

GANs consist of two competing neural networks: the **Generator** and the **Discriminator**. These two models are trained simultaneously in an adversarial manner, where the generator tries to create images that fool the discriminator, while the discriminator learns to distinguish between real and fake images.

The objective of this experiment is to evaluate the performance of a GAN in generating realistic images, both qualitatively (by visual inspection) and quantitatively (using statistical metrics). Understanding GAN evaluation helps in determining the model's ability to learn the underlying data distribution and produce diverse, high-quality images.

GAN Architecture



General GAN architecture

1 Generator

The generator is a neural network that starts from a vector of random noise (also called latent vector) and transforms it into a synthetic image through a series of upsampling and convolutional layers. It learns to generate images that resemble the training data.

2 Discriminator

The discriminator is a binary classifier that takes an image as input and outputs the probability that the image is real (from the dataset) or fake (produced by the generator). It consists of convolutional layers followed by fully connected layers.

3 Adversarial Training

Both networks are trained together using a minimax game approach:

- The generator tries to maximize the discriminator's error by generating more realistic images.
- The discriminator tries to minimize classification errors by correctly identifying real and fake images.

This adversarial process continues until the generator produces images that the discriminator cannot reliably distinguish from real images.

GANs Use Transformers

While the original GAN uses convolutional neural networks (CNNs) for image generation and discrimination, recent advances integrate **Transformers** into GANs to improve image quality. Transformers use **self-attention mechanisms**, allowing the model to understand global relationships across the entire image, which CNNs may miss due to their local receptive fields.

Because of this, Transformer-based GANs can capture more complex patterns and generate more coherent, detailed images. The **self-attention** in Transformers dynamically focuses on different parts of the image, improving realism.

Dataset Preparation

For this experiment, a dataset of real images (such as CIFAR-10 or CelebA) is prepared:

- Images are resized and normalized to match the input requirements of the GAN.
- The dataset is split into training and testing sets.
- The training images serve as real samples for the discriminator.

Training Process

The GAN training proceeds through the following steps:

1. **Generate Fake Images:** Random noise vectors are passed to the generator to create fake images.
2. **Train Discriminator:** The discriminator is trained with a batch of real images (labeled real) and generated fake images (labeled fake). It learns to classify them correctly.
3. **Train Generator:** The generator is trained to fool the discriminator by generating images that the discriminator labels as real.
4. **Iterative Updates:** The discriminator and generator are updated alternately through backpropagation, optimizing their respective objectives.
5. **Epochs:** Training continues over multiple epochs until image quality stabilizes.

Evaluation Metrics

GANs are challenging to evaluate because generated images do not have ground truth labels. Common evaluation metrics include:

- **Inception Score (IS):** Measures image quality and diversity by feeding generated images into a pretrained Inception model and analyzing the class probabilities. Higher scores indicate better performance.

- **Fréchet Inception Distance (FID):** Computes the statistical distance between feature representations of real and generated images. Lower FID scores mean the generated images are closer to real images in distribution.
- **Visual Inspection:** Human evaluation of the generated images to identify artifacts, blurriness, or unrealistic features.

Results

The generator produces synthetic images from random noise after training. The results are analyzed as follows:

- **Visual Assessment:** Generated images appear visually realistic, showing clear objects and natural textures.
- **Metric Scores:** Inception Score increases over epochs, showing improved quality and diversity. Fréchet Inception Distance decreases, confirming generated images better match the real data distribution.
- **Common Issues:** Some artifacts such as mode collapse (generator producing limited image varieties) or blurry images might appear if training is unstable.

Discussion

The GAN successfully learns to generate realistic images through adversarial training. The balance between the generator and discriminator is crucial for stable training; if one network becomes too strong, it can hinder learning.

The use of self-attention (Transformer-based components) in advanced GANs can further improve image quality by capturing long-range dependencies.

Evaluating GANs requires both quantitative metrics and visual inspection, as metrics alone may not fully capture image realism or diversity.

Conclusion

This experiment confirms that GANs can effectively generate realistic images by learning the distribution of the training data. Using adversarial training, the generator improves image quality over time while the discriminator sharpens its ability to detect fakes.

Evaluation through Inception Score, Fréchet Inception Distance, and visual inspection provides a comprehensive understanding of GAN performance.

Future work may explore Transformer-based GAN variants and improved training techniques to address common challenges like mode collapse and instability.

Python Code

```
# Cell 1: Imports and MNIST data preparation
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Dense, Reshape, Flatten, Dropout, Input
```

```
from tensorflow.keras.layers import Conv2D, Conv2DTranspose, BatchNormalization,
LeakyReLU, Activation, UpSampling2D
from tensorflow.keras.optimizers import Adam

# Load and normalize data
(x_train, _), (x_test, _) = mnist.load_data()

x_train = (x_train.astype(np.float32) - 127.5) / 127.5
x_test = (x_test.astype(np.float32) - 127.5) / 127.5

x_train = np.expand_dims(x_train, axis=-1)
x_test = np.expand_dims(x_test, axis=-1)

print(f"Train shape: {x_train.shape}, Test shape: {x_test.shape}")
# Plot some sample real images
plt.figure(figsize=(10, 2))
for i in range(10):
    plt.subplot(1, 10, i+1)
    plt.imshow(x_train[i].reshape(28, 28), cmap='gray')
    plt.axis('off')
plt.suptitle('Sample Real MNIST Images')
plt.show()

# Cell 2: Improved Generator Model (better architecture)
latent_dim = 100

def build_generator():
    model = Sequential([
        Dense(7*7*256, input_dim=latent_dim, kernel_initializer='he_normal'),
        BatchNormalization(momentum=0.8),
        Activation('relu'),
        Reshape((7, 7, 256)),

        Conv2DTranspose(128,    kernel_size=4,    strides=2,    padding='same',
kernel_initializer='he_normal'),
        BatchNormalization(momentum=0.8),
        Activation('relu'),

        Conv2DTranspose(64,    kernel_size=4,    strides=2,    padding='same',
kernel_initializer='he_normal'),
        BatchNormalization(momentum=0.8),
        Activation('relu'),

        Conv2D(1,     kernel_size=7,     activation='tanh',     padding='same',
kernel_initializer='he_normal')
    ])
    return model

# Cell 3: Improved Discriminator Model (better architecture)
def build_discriminator():
    model = Sequential([
        Conv2D(64,        kernel_size=3,        strides=2,        padding='same',
input_shape=(28,28,1), kernel_initializer='he_normal'),
        LeakyReLU(alpha=0.2),
        Dropout(0.25),

        Conv2D(128,       kernel_size=3,       strides=2,       padding='same',
kernel_initializer='he_normal'),
        BatchNormalization(momentum=0.8),
```

```
        LeakyReLU(alpha=0.2),
        Dropout(0.25),

        Conv2D(256,       kernel_size=3,       strides=2,       padding='same',
kernel_initializer='he_normal'),
        BatchNormalization(momentum=0.8),
        LeakyReLU(alpha=0.2),
        Dropout(0.25),

        Flatten(),
        Dense(1, activation='sigmoid', kernel_initializer='glorot_normal')
    ])
    return model

# Cell 4: Build and compile GAN models
optimizer = Adam(0.0002, 0.5)

discriminator = build_discriminator()
discriminator.compile(loss='binary_crossentropy',           optimizer=optimizer,
metrics=['accuracy'])

generator = build_generator()

# Freeze discriminator weights when training generator via GAN
discriminator.trainable = False

z = Input(shape=(latent_dim,))
img = generator(z)
validity = discriminator(img)

gan = Model(z, validity)
gan.compile(loss='binary_crossentropy', optimizer=optimizer)

print(generator.summary())
print(discriminator.summary())

# Cell 5: Training loop with metrics & periodic outputs
import time

batch_size = 128
epochs = 100000
# Labels for real and fake images
real_labels = np.ones((batch_size, 1))
fake_labels = np.zeros((batch_size, 1))

d_losses, d_accs, g_losses = [], [], []

start_time = time.time()

for epoch in range(1, epochs+1):
    # ---------------------
    # Train Discriminator
    # ---------------------
    idx = np.random.randint(0, x_train.shape[0], batch_size)
    real_imgs = x_train[idx]

    noise = np.random.normal(0, 1, (batch_size, latent_dim))
    fake_imgs = generator.predict(noise, verbose=0)
```

```
    d_loss_real = discriminator.train_on_batch(real_imgs, real_labels)
    d_loss_fake = discriminator.train_on_batch(fake_imgs, fake_labels)
    d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

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

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

if epoch == 1 or epoch % 10 == 0:
    elapsed = time.time() - start_time
    print(f"Epoch  {epoch}/{epochs}  -  D  loss:  {d_loss[0]:.4f},  D  acc:
{d_loss[1]*100:.2f}%, G loss: {g_loss:.4f} - Time: {elapsed:.1f}s")
    start_time = time.time()


# Cell 6: Plot training losses and accuracy
plt.figure(figsize=(14,5))

plt.subplot(1,2,1)
plt.plot(d_losses, label='Discriminator Loss', color='red')
plt.plot(g_losses, label='Generator Loss', color='blue')
plt.title('Loss over epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1,2,2)
plt.plot(d_accs, label='Discriminator Accuracy', color='green')
plt.title('Discriminator Accuracy over epochs')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim(0,1)
plt.legend()

plt.show()

# Cell 7: Show real vs generated images side-by-side for visual comparison
def plot_generated_images(generator, examples=10):
    noise = np.random.normal(0, 1, (examples, latent_dim))
    gen_imgs = generator.predict(noise)
    gen_imgs = 0.5 * gen_imgs + 0.5  # Scale back from [-1,1] to [0,1]

    plt.figure(figsize=(15, 2))
    for i in range(examples):
        plt.subplot(1, examples, i+1)
        plt.imshow(gen_imgs[i,:,:,0], cmap='gray')
        plt.axis('off')
    plt.suptitle('Generated MNIST Images')
    plt.show()

# Display 5 real and 5 generated images
plt.figure(figsize=(15,4))
```

```
for i in range(5):
    plt.subplot(2, 5, i+1)
    plt.imshow(x_test[i].reshape(28,28), cmap='gray')
    plt.title("Real")
    plt.axis('off')

noise = np.random.normal(0, 1, (5, latent_dim))
fake_imgs = generator.predict(noise)
fake_imgs = 0.5 * fake_imgs + 0.5

for i in range(5):
    plt.subplot(2, 5, 5+i+1)
    plt.imshow(fake_imgs[i,:,:,0], cmap='gray')
    plt.title("Fake")
    plt.axis('off')

plt.show()
```