KONGU ENGINEERING COLLEGE

(Autonomous)

Perundurai, Erode – 638060

DEPARTMENT OF COMPUTER APPLICATIONS

24MCF07 – DEEP LEARNING LABORATORY

Name	:	**KALISWARY K**
Register Number	:	**24MCR048**
Branch	:	**Computer Applications**
Semester	:	**III**

Certified that this is a bonafide record of work done by the above student for **24MCF07 – DEEP LEARNING LABORATORY** during the academic year **2025- 2026.**

Course-in-Charge **Head of the Department**

Submitted for the End Semester Examination held on ___________

Examiner -1 **Examiner -2**

KONGU ENGINEERING COLLEGE

(Autonomous)

Perundurai, Erode – 638 060

DEPARTMENT OF COMPUTER APPLICATIONS

24MCF07 – DEEP LEARNING LABORATORY

LIST OF EXPERIMENTS

S.No	Date	Exercise Name	Page No.	Marks	Signature
1		IMPLEMENT SIMPLE PERCEPTRON LEARNING			
2		MULTILAYER PERCEPTRON WITH HYPERPARAMETER TUNING			
3		GENERATE SYNTHETIC IMAGES USING DATA AUGMENTATION			
4		ROLE OF IMAGE DATA GENERATOR CLASS IN DATA AUGMENTATION			
5		CNN PROCESS FOR IMAGE CLASSIFICATION.			
6		RNN ARCHITECTURE FOR TIME SERIES DATA.			
7		NLP TEXT ANALYSIS STEPS			
8		DEEPDREAM & NEURAL STYLE TRANSFER			
9		VARIATIONAL AUTOENCODER (VAE) FOR SYNTHETIC IMAGES			
10		GAN (GENERATIVE ADVERSARIAL NETWORK)			

EX NO : 01

DATE:

IMPLEMENT SIMPLE PERCEPTRON LEARNING

AIM:

To design and implement a single-layer perceptron in Python and train it using the perceptron learning rule to realize the OR logic gate.

ALGORITHM:

Step 1: Start the program.

Step 2: Set all weights to 0, including one extra weight for the bias.

Step 3: Choose a small learning rate (like 0.1) and set the number of training times (epochs).

Step 4: For each input, add a bias value 1 at the beginning.

Step 5: Multiply each input by its weight and add them to get the total (this is the weighted sum)

Step 6: If the total is greater than or equal to 0, the output is 1. Otherwise, the output is 0 (this is the step function).

Step 7: Subtract the predicted output from the actual output to get the error.

Step 8: Update each weight using this formula:

new weight = old weight + (learning rate × error × input)

Step 9: Repeat Steps 4 to 8 for all inputs and for all epochs.

Step 10: After training is complete, test the model with the inputs.

Step 11: Show the final predicted outputs.

Step 12: End the program.

PROGRAM:

```python
import numpy as np


def step_function(value):
    return 1 if value >= 0 else 0
```

```python
class Perceptron:
    def __init__(self, input_size, learning_rate=0.1):
        self.weights = np.zeros(input_size + 1)  # Including bias
        self.learning_rate = learning_rate
```

```python
    def predict(self, inputs):
        inputs_with_bias = np.insert(inputs, 0, 1)
        total = np.dot(self.weights, inputs_with_bias)
        return step_function(total)
```

```python
    def train(self, X, y, epochs=10):
        for epoch in range(epochs):
            print(f"Epoch {epoch+1}")
            for i in range(len(X)):
                prediction = self.predict(X[i])
                error = y[i] - prediction
                x_with_bias = np.insert(X[i], 0, 1)
                self.weights += self.learning_rate * error * x_with_bias
                    print(f" Input: {X[i]}, Predicted: {prediction}, Actual: {y[i]}, Updated
Weights: {self.weights}")
```

```python
if __name__ == "__main__":
    # Step 1: Training data for OR logic gate
    X = np.array([
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1]
    ])
    y = np.array([0, 1, 1, 1])
```

```python
    # Step 2: Create perceptron and train it
    perceptron = Perceptron(input_size=2)
```

```python
perceptron.train(X, y, epochs=10)
```

```python
# Step 3: Test the trained perceptron
print("\nFinal Predictions:")
for x in X:
    output = perceptron.predict(x)
    print(f"Input: {x}, Predicted Output: {output}")
```

OUTPUT:

```
Epoch 1
 Input: [0 0], Predicted: 1, Actual: 0, Updated Weights: [-0.1  0.   0. ]
 Input: [0 1], Predicted: 0, Actual: 1, Updated Weights: [0.  0.  0.1]
 Input: [1 0], Predicted: 1, Actual: 1, Updated Weights: [0.  0.  0.1]
 Input: [1 1], Predicted: 1, Actual: 1, Updated Weights: [0.  0.  0.1]
Epoch 2
 Input: [0 0], Predicted: 1, Actual: 0, Updated Weights: [-0.1  0.   0.1]
 Input: [0 1], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.   0.1]
 Input: [1 0], Predicted: 0, Actual: 1, Updated Weights: [0.  0.1 0.1]
 Input: [1 1], Predicted: 1, Actual: 1, Updated Weights: [0.  0.1 0.1]
Epoch 3
 Input: [0 0], Predicted: 1, Actual: 0, Updated Weights: [-0.1  0.1  0.1]
 Input: [0 1], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
 Input: [1 0], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
 Input: [1 1], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
Epoch 4
 Input: [0 0], Predicted: 0, Actual: 0, Updated Weights: [-0.1  0.1  0.1]
 Input: [0 1], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
 Input: [1 0], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
 Input: [1 1], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
Epoch 5
 Input: [0 0], Predicted: 0, Actual: 0, Updated Weights: [-0.1  0.1  0.1]
 Input: [0 1], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
 Input: [1 0], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
 Input: [1 1], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
```

```
Epoch 6
 Input: [0 0], Predicted: 0, Actual: 0, Updated Weights: [-0.1  0.1  0.1]
 Input: [0 1], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
 Input: [1 0], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
 Input: [1 1], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
Epoch 7
 Input: [0 0], Predicted: 0, Actual: 0, Updated Weights: [-0.1  0.1  0.1]
 Input: [0 1], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
 Input: [1 0], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
 Input: [1 1], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
Epoch 8
 Input: [0 0], Predicted: 0, Actual: 0, Updated Weights: [-0.1  0.1  0.1]
 Input: [0 1], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
 Input: [1 0], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
 Input: [1 1], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
Epoch 9
 Input: [0 0], Predicted: 0, Actual: 0, Updated Weights: [-0.1  0.1  0.1]
 Input: [0 1], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
 Input: [1 0], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
 Input: [1 1], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
Epoch 10
 Input: [0 0], Predicted: 0, Actual: 0, Updated Weights: [-0.1  0.1  0.1]
 Input: [0 1], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
 Input: [1 0], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]
 Input: [1 1], Predicted: 1, Actual: 1, Updated Weights: [-0.1  0.1  0.1]

Final Predictions:
Input: [0 0], Predicted Output: 0
Input: [0 1], Predicted Output: 1
Input: [1 0], Predicted Output: 1
Input: [1 1], Predicted Output: 1
```

COE (20)	
RECORD (20)	
VIVA (10)	
TOTAL (50)	

RESULT:

The perceptron successfully learned the OR gate and correctly predicted the output for all input combinations after training.

EX NO : 02

DATE :

A MULTILAYER PERCEPTRON WITH A HYPERPARAMETER TUNING

AIM:

To develop a Multilayer Perceptron (MLP) model with hyperparameter tuning using Keras Tuner for predicting high spending behavior based on demographic and lifestyle features from the given dataset.

ALGORITHM:

Step 1: Import necessary libraries.

Step 2: Load the dataset into Python.

Step 3: Create a binary target column based on Spending_Score.

Step 4: Remove the ID and Spending_Score columns.

Step 5: Handle missing values in the dataset.

Step 6: Encode all categorical columns.

Step 7: Normalize the input features.

Step 8: Split the dataset into training and testing sets.

Step 9: Write a function to build the MLP model.

Step 10: Use Keras Tuner to tune the model's hyperparameters.

Step 11: Get the best model from the tuner.

Step 12: Train and evaluate the best model on the test data.

CODING:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

```python
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import classification_report

# Step 1: Load and preprocess your dataset
df = pd.read_csv("/content/drive/MyDrive/DL/Test.csv")

df['target'] = (df['Spending_Score'] == 'High').astype(int)
df.drop(columns=['ID', 'Spending_Score'], inplace=True)

# Fill missing values
for col in df.columns:
    if df[col].dtype == 'object':
        df[col].fillna(df[col].mode()[0], inplace=True)
    else:
        df[col].fillna(df[col].mean(), inplace=True)

# Encode categorical variables
cat_cols = df.select_dtypes(include='object').columns
for col in cat_cols:
    le = LabelEncoder()
    df[col] = le.fit_transform(df[col])

# Split features and target
X = df.drop(columns='target')
y = df['target']

# Normalize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split data
```

```
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2,
random_state=42)
```

```
# Step 2: Define different learning rates and epochs
learning_rates = [0.001, 0.0005, 0.0001]
epoch_list = [20, 30, 50]
```

```
results = []
```

```
# Step 3: Loop through different settings
for lr in learning_rates:
    for epochs in epoch_list:
        print(f"\n    Training with LR={lr}, Epochs={epochs}")

        # Build model
        model = Sequential([
            Dense(32, activation='relu', input_shape=(X_train.shape[1],)),
            Dense(16, activation='relu'),
            Dense(1, activation='sigmoid')
        ])

        optimizer = Adam(learning_rate=lr)
        model.compile(optimizer=optimizer, loss='binary_crossentropy',
metrics=['accuracy'])

        # Train
        history = model.fit(
            X_train, y_train,
            validation_split=0.1,
            epochs=epochs,
            batch_size=8,
            verbose=0
        )
```

```
        # Evaluate on test data
        test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)

        # Save metrics
        results.append({
            'Learning Rate': lr,
            'Epochs': epochs,
            'Train Accuracy': history.history['accuracy'][-1] * 100,
            'Validation Accuracy': history.history['val_accuracy'][-1] * 100,
            'Test Accuracy': test_acc * 100,
            'Train Loss': history.history['loss'][-1],
            'Validation Loss': history.history['val_loss'][-1]
        })
```

```
# Step 4: Display table
df_results = pd.DataFrame(results)
print("\n    Comparison Table:")
display(df_results)
```

```
# Step 5: Plot Accuracy and Loss
plt.figure(figsize=(14, 6))
sns.set_style("whitegrid")
```

```
# Accuracy plot
plt.subplot(1, 2, 1)
for lr in learning_rates:
    subset = df_results[df_results['Learning Rate'] == lr]
    plt.plot(subset['Epochs'], subset['Train Accuracy'], marker='o', label=f'Train Acc (LR={lr})')
    plt.plot(subset['Epochs'], subset['Validation Accuracy'], marker='o', linestyle='--', label=f'Val Acc (LR={lr})')
    plt.plot(subset['Epochs'], subset['Test Accuracy'], marker='x', linestyle=':', label=f'Test Acc (LR={lr})')
```

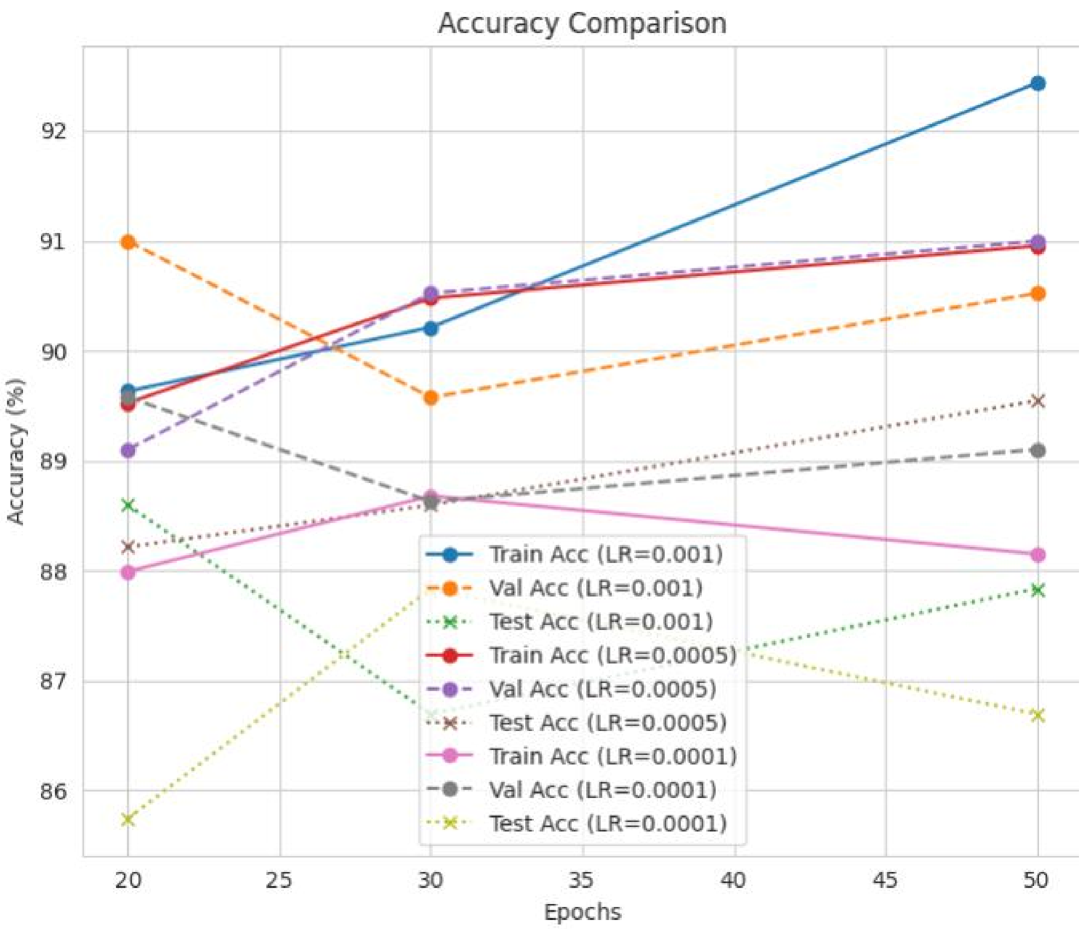
```python
plt.xlabel("Epochs")
plt.ylabel("Accuracy (%)")
plt.title("Accuracy Comparison")
plt.legend()

# Loss plot
plt.subplot(1, 2, 2)
for lr in learning_rates:
    subset = df_results[df_results['Learning Rate'] == lr]
    plt.plot(subset['Epochs'], subset['Train Loss'], marker='o', label=f'Train Loss (LR={lr})')
    plt.plot(subset['Epochs'], subset['Validation Loss'], marker='o', linestyle='--', label=f'Val Loss (LR={lr})')

plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Loss Comparison")
plt.legend()

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

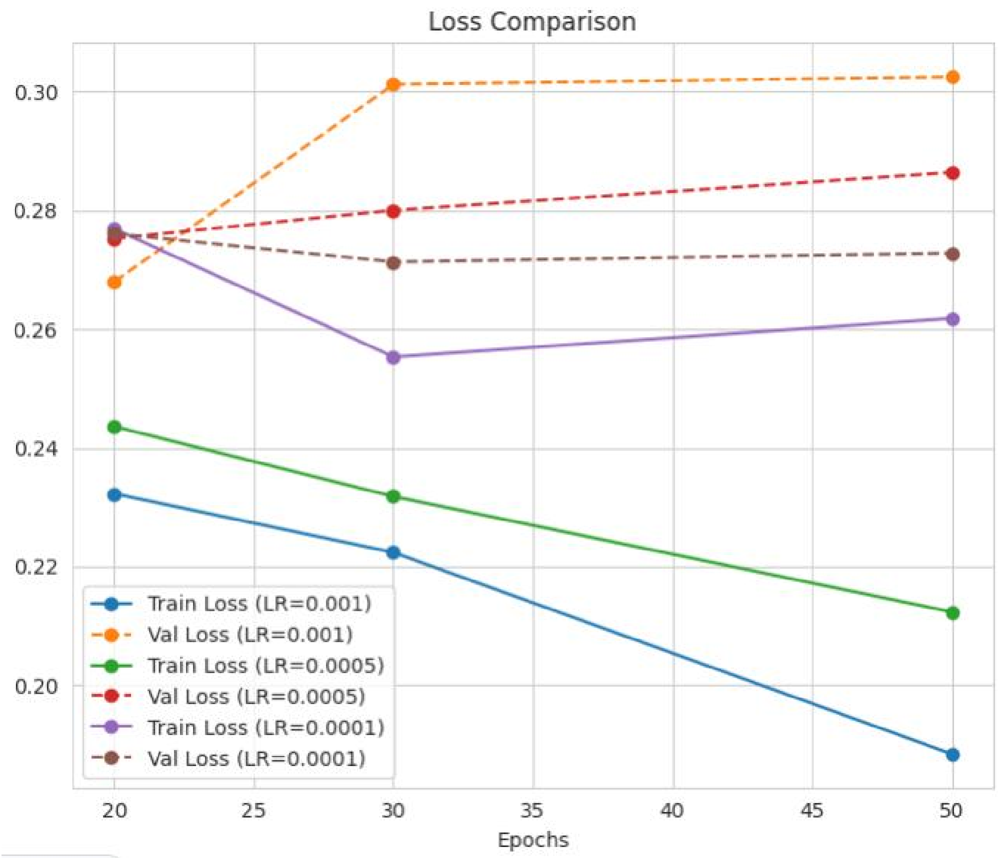
OUTPUT:

```
🔧 Training with LR=0.001, Epochs=20
🔧 Training with LR=0.001, Epochs=30
🔧 Training with LR=0.001, Epochs=50
🔧 Training with LR=0.0005, Epochs=20
🔧 Training with LR=0.0005, Epochs=30
🔧 Training with LR=0.0005, Epochs=50
🔧 Training with LR=0.0001, Epochs=20
🔧 Training with LR=0.0001, Epochs=30
🔧 Training with LR=0.0001, Epochs=50
```

📊 Comparison Table:

	Learning Rate	Epochs	Train Accuracy	Validation Accuracy	Test Accuracy	Train Loss	Validation Loss
0	0.0010	20	89.629632	90.995258	88.593155	0.232290	0.267986
1	0.0010	30	90.211642	89.573461	86.692017	0.222391	0.301207
2	0.0010	50	92.433864	90.521330	87.832701	0.188469	0.302496
3	0.0005	20	89.523810	89.099526	88.212925	0.243595	0.275269
4	0.0005	30	90.476191	90.521330	88.593155	0.231816	0.280010
5	0.0005	50	90.952379	90.995258	89.543724	0.212384	0.286377
6	0.0001	20	87.989420	89.573461	85.741442	0.276916	0.276082
7	0.0001	30	88.677251	88.625592	87.832701	0.255324	0.271320
8	0.0001	50	88.148147	89.099526	86.692017	0.261793	0.272812





COE (20)	
RECORD (20)	
VIVA (10)	
TOTAL (50)	

RESULT:

The Multilayer Perceptron (MLP) model was successfully implemented with hyperparameter tuning using Keras Tuner.

EX NO : 03

DATE :

DATA AUGMENTATION FOR IMAGE

AIM:

To generate augmented ECG image data using traditional augmentation techniques in order to improve model generalization and reduce overfitting in the image classification process.

ALGORITHM:

Step 1: Mount Google Drive to access the dataset stored in /content/drive/MyDrive/DL/iris-setosa.

Step 2: Import required libraries such as TensorFlow, Keras's ImageDataGenerator, Matplotlib, and OS for file handling.

Step 3: Set the dataset path to the folder containing the sample images for augmentation.

Step 4: Initialize the ImageDataGenerator object with augmentation parameters:

rotation_range=40 for random rotations

width_shift_range=0.2 and height_shift_range=0.2 for shifting

shear_range=0.2 for shearing transformation

zoom_range=0.2 for zoom in/out

horizontal_flip=True and vertical_flip=True for flipping

fill_mode='nearest' for filling empty pixels

Step 5: Load a sample image from the dataset folder using Keras's image.load_img() function.

Step 6: Convert the loaded image into a NumPy array and reshape it to match the input format expected by ImageDataGenerator.

Step 7: Generate augmented images by iterating over the batches produced by the .flow() method of the data generator.

Step 8: Display the augmented results using Matplotlib to visualize multiple transformations applied to the same input image.

CODING:

```
# Step 1: Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')

# Step 2: Import libraries
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
import os
```

```
# Step 3: Set dataset path
dataset_path = "/content/drive/MyDrive/DL/iris-setosa"
```

```
# Step 4: Create ImageDataGenerator with augmentations
datagen = ImageDataGenerator(
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    vertical_flip=True,
    fill_mode='nearest'
)
```

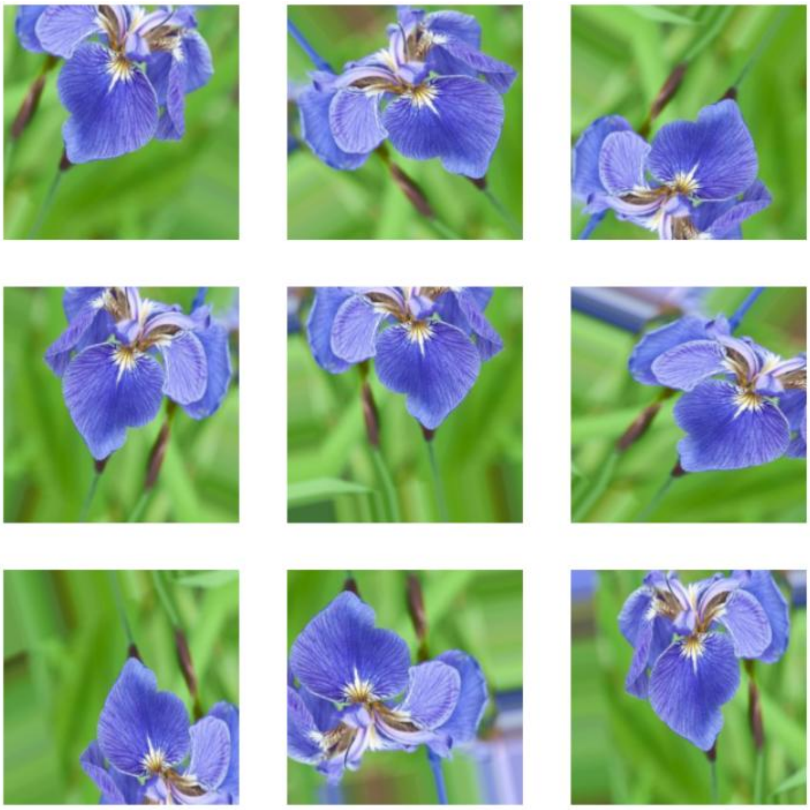
```
# Step 5: Load one sample image
from tensorflow.keras.preprocessing import image
sample_img = os.listdir(dataset_path)[0]
img_path = os.path.join(dataset_path, sample_img)
```

```
img = image.load_img(img_path)
x = image.img_to_array(img)
```

```python
x = x.reshape((1,) + x.shape)
```

```python
# Step 6: Generate and plot augmented images
plt.figure(figsize=(8,8))
i = 0
for batch in datagen.flow(x, batch_size=1):
    plt.subplot(3, 3, i + 1)
    plt.imshow(batch[0].astype('uint8'))
    plt.axis('off')
    i += 1
    if i == 9:
        break
plt.show()
```

OUTPUT:



COE(20)	
RECORD(20)	
VIVA(10)	
TOTAL(50)	

RESULT:

Traditional data augmentation techniques such as rotation, shift, shear, zoom, and flip were applied to the iris-setosa images. The augmentation process was implemented successfully.

EX NO : 04

DATE :

IMAGE DATA GENERATOR IN DATA AUGMENTATION FOR IMAGE

AIM:

To apply traditional image augmentation techniques such as rotation, shifting, shearing, zooming, and flipping to iris-setosa images using the ImageDataGenerator class, in order to generate multiple variations of an existing image and enhance dataset diversity for better model generalization.

ALGORITHM:

Step 1: Mount Google Drive to access the dataset stored in /content/drive/MyDrive/DL/iris-setosa.

Step 2: Import the required Python libraries including TensorFlow, Keras's ImageDataGenerator, Matplotlib, and OS for file handling.

Step 3: Set the dataset path to the folder containing the sample images for augmentation.

Step 4: Create an ImageDataGenerator object with the following augmentation parameters:

rotation_range=40 for random rotations

width_shift_range=0.2 and height_shift_range=0.2 for horizontal and vertical shifting

shear_range=0.2 for shearing transformation

zoom_range=0.2 for zoom in/out

horizontal_flip=True and vertical_flip=True for flipping

fill_mode='nearest' to fill empty pixels after transformation

Step 5: Select a sample image from the dataset folder and load it using image.load_img().

Step 6: Convert the loaded image into a NumPy array and reshape it to match the input format expected by the ImageDataGenerator.

Step 7: Use the .flow() method to generate augmented images in batches from the sample image.

Step 8: Display the augmented images using Matplotlib to visualize the transformations applied to the original image.

CODING:

```
# Step 1: Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')

# Step 2: Import libraries
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
import os
```

```
# Step 3: Set dataset path
dataset_path = "/content/drive/MyDrive/DL/iris-setosa"
```

```
# Step 4: Create ImageDataGenerator object with various augmentations
datagen = ImageDataGenerator(
    rotation_range=40,       # Rotate up to 40 degrees
    width_shift_range=0.2,    # Horizontal shift
    height_shift_range=0.2,   # Vertical shift
    shear_range=0.2,          # Shear transformation
    zoom_range=0.2,           # Zoom in/out
    horizontal_flip=True,     # Flip horizontally
    vertical_flip=True,       # Flip vertically
    fill_mode='nearest'       # Fill empty pixels after transformation
)
```

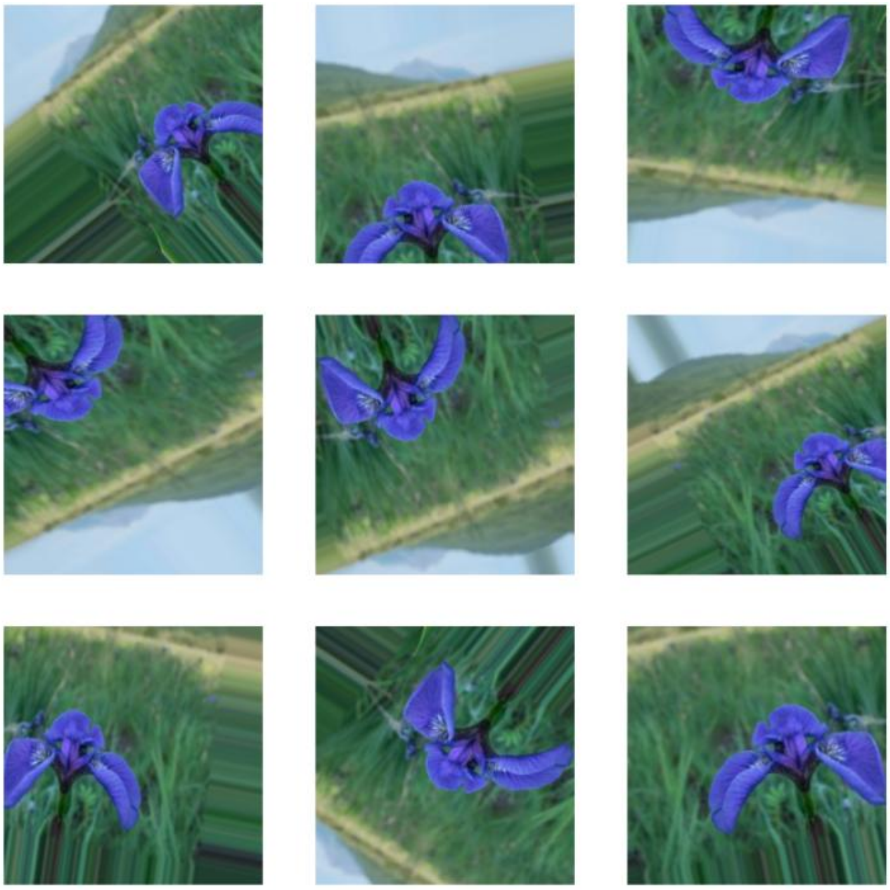
```
# Step 5: Pick one sample image from the folder
from tensorflow.keras.preprocessing import image
```

```python
sample_img = os.listdir(dataset_path)[0]  # First image in folder
img_path = os.path.join(dataset_path, sample_img)
```

```python
img = image.load_img(img_path)
x = image.img_to_array(img)     # Convert to NumPy array
x = x.reshape((1,) + x.shape)   # Reshape for the generator
```

```python
# Step 6: Display augmented images
plt.figure(figsize=(8,8))
i = 0
for batch in datagen.flow(x, batch_size=1):
    plt.subplot(3, 3, i + 1)
    plt.imshow(batch[0].astype('uint8'))
    plt.axis('off')
    i += 1
    if i == 9:   # Display 9 augmented versions
        break
plt.show()
```

OUTPUT:



COE(20)	
RECORD(20)	
VIVA(10)	
TOTAL(50)	

RESULT:

Traditional augmentation methods including rotation, shifting, shearing, zooming, and flipping were successfully applied to the iris-setosa images using the ImageDataGenerator class.

EX NO : 05

DATE :

CNN FOR IMAGE CLASSIFICATION

AIM:

To build and train a Convolutional Neural Network (CNN) for image classification using augmented images, applying strong data augmentation techniques to reduce overfitting and improve generalization performance on unseen data.

ALGORITHM:

Step 1: Mount Google Drive in Google Colab to access the dataset stored in /content/drive/MyDrive/DL/.

Step 2: Import required libraries including TensorFlow, Keras's ImageDataGenerator, and Matplotlib for model building, augmentation, and visualization.

Step 3: Specify the dataset path containing the images to be classified.

Step 4: Create an ImageDataGenerator object with the following augmentation parameters for the training and validation split (80%-20%):

Rescaling pixel values to the range [0,1]

Rotation range: 40 degrees

Width shift and height shift range: 0.2

Shear range: 0.2

Zoom range: 0.3

Horizontal and vertical flips enabled

Fill mode: nearest

Step 5: Generate training and validation data batches from the directory using .flow_from_directory() with target image size (64, 64) and batch size 32.

Step 6: Build a CNN model with the following layers:

Conv2D + MaxPooling2D layers for feature extraction

Dropout layers for regularization

Flatten + Dense layers for classification

Output Dense layer with softmax activation for multi-class classification

Step 7: Compile the model using the Adam optimizer, categorical crossentropy loss, and accuracy as the evaluation metric.

Step 8: Define an EarlyStopping callback to stop training if the validation loss does not improve for 5 consecutive epochs, restoring the best weights.

Step 9: Train the model for up to 30 epochs using the training and validation generators.

Step 10: Plot training and validation accuracy over epochs for performance visualization.

Step 11: Evaluate the trained model on the validation dataset and display the final validation accuracy.

CODING:

```
# Step 1: Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')

# Step 2: Import libraries
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
```

```
# Step 3: Dataset path
data_path = "/content/drive/MyDrive/DL/"
```

```
# Step 4: Data Augmentation with stronger transformations
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.3,
    horizontal_flip=True,
```

```
    vertical_flip=True,
    fill_mode='nearest',
    validation_split=0.2
)
```

```
# Train and Validation Generators
train_data = train_datagen.flow_from_directory(
    data_path,
    target_size=(64, 64),
    batch_size=32,
    class_mode='categorical',
    subset='training'
)
```

```
val_data = train_datagen.flow_from_directory(
    data_path,
    target_size=(64, 64),
    batch_size=32,
    class_mode='categorical',
    subset='validation'
)
```

```
# Step 5: CNN Model with Dropout
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(64,64,3)),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Dropout(0.25),
```

```
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Dropout(0.25),
```

```
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
```

```python
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(train_data.num_classes, activation='softmax')
])
```

```python
# Step 6: Compile Model
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
```

```python
# Step 7: Early Stopping Callback
early_stop = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=5,
    restore_best_weights=True
)
```

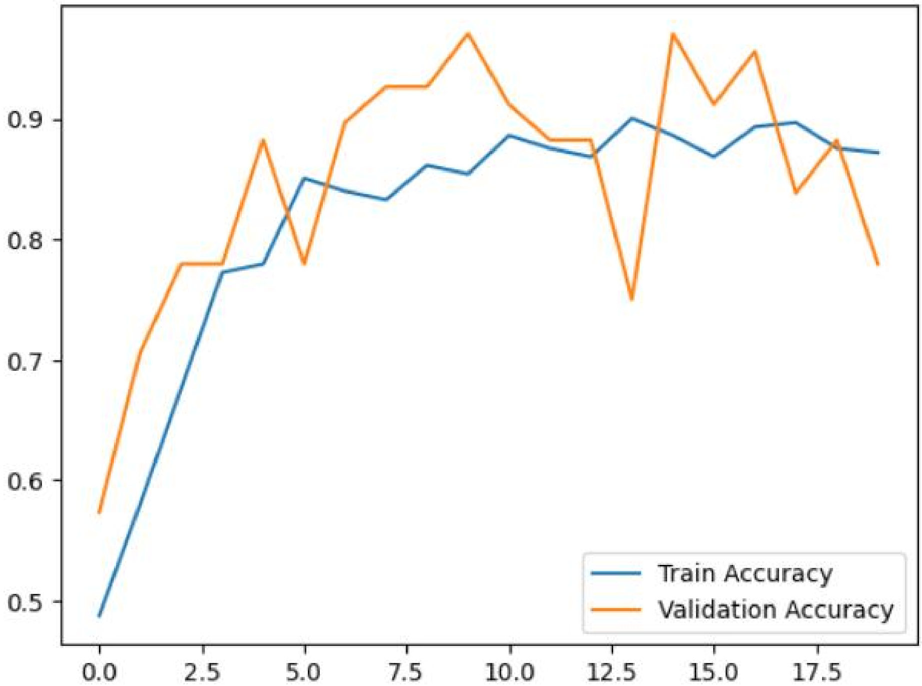
```python
# Step 8: Train Model
history = model.fit(
    train_data,
    validation_data=val_data,
    epochs=30,
    callbacks=[early_stop]
)
```

```python
# Step 9: Plot Accuracy
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.legend()
plt.show()
```

```python
# Step 10: Evaluate Model
loss, acc = model.evaluate(val_data)
print(f"Validation Accuracy: {acc*100:.2f}%")
```

OUTPUT:

```
Found 281 images belonging to 3 classes.
Found 68 images belonging to 3 classes.
Epoch 1/30
9/9 ━━━━━━━━━━━━━━━━━━━━ 10s 987ms/step - accuracy: 0.4308 - loss: 1.2474 - val_accuracy: 0.5735 - val_loss: 0.9750
Epoch 2/30
9/9 ━━━━━━━━━━━━━━━━━━━━ 5s 542ms/step - accuracy: 0.6129 - loss: 0.8886 - val_accuracy: 0.7059 - val_loss: 0.9785
Epoch 3/30
9/9 ━━━━━━━━━━━━━━━━━━━━ 5s 578ms/step - accuracy: 0.6422 - loss: 0.8640 - val_accuracy: 0.7794 - val_loss: 0.6931
Epoch 4/30
9/9 ━━━━━━━━━━━━━━━━━━━━ 6s 628ms/step - accuracy: 0.7308 - loss: 0.6319 - val_accuracy: 0.7794 - val_loss: 0.5291
Epoch 5/30
9/9 ━━━━━━━━━━━━━━━━━━━━ 5s 566ms/step - accuracy: 0.7841 - loss: 0.5482 - val_accuracy: 0.8824 - val_loss: 0.3912
Epoch 6/30
9/9 ━━━━━━━━━━━━━━━━━━━━ 6s 658ms/step - accuracy: 0.8822 - loss: 0.4313 - val_accuracy: 0.7794 - val_loss: 0.4400
Epoch 7/30
9/9 ━━━━━━━━━━━━━━━━━━━━ 9s 529ms/step - accuracy: 0.8443 - loss: 0.4308 - val_accuracy: 0.8971 - val_loss: 0.3026
Epoch 8/30
9/9 ━━━━━━━━━━━━━━━━━━━━ 6s 723ms/step - accuracy: 0.8238 - loss: 0.4190 - val_accuracy: 0.9265 - val_loss: 0.3000
Epoch 9/30
9/9 ━━━━━━━━━━━━━━━━━━━━ 5s 507ms/step - accuracy: 0.8358 - loss: 0.4497 - val_accuracy: 0.9265 - val_loss: 0.2688
Epoch 10/30
9/9 ━━━━━━━━━━━━━━━━━━━━ 5s 571ms/step - accuracy: 0.8534 - loss: 0.3749 - val_accuracy: 0.9706 - val_loss: 0.2076
Epoch 11/30
9/9 ━━━━━━━━━━━━━━━━━━━━ 6s 688ms/step - accuracy: 0.8864 - loss: 0.2894 - val_accuracy: 0.9118 - val_loss: 0.2157
Epoch 12/30
9/9 ━━━━━━━━━━━━━━━━━━━━ 5s 541ms/step - accuracy: 0.8755 - loss: 0.3303 - val_accuracy: 0.8824 - val_loss: 0.2979
Epoch 13/30
9/9 ━━━━━━━━━━━━━━━━━━━━ 6s 731ms/step - accuracy: 0.8770 - loss: 0.3579 - val_accuracy: 0.8824 - val_loss: 0.2676
Epoch 14/30
9/9 ━━━━━━━━━━━━━━━━━━━━ 5s 560ms/step - accuracy: 0.9038 - loss: 0.2658 - val_accuracy: 0.7500 - val_loss: 0.4123
Epoch 15/30
9/9 ━━━━━━━━━━━━━━━━━━━━ 5s 554ms/step - accuracy: 0.8874 - loss: 0.3489 - val_accuracy: 0.9706 - val_loss: 0.1682
Epoch 16/30
9/9 ━━━━━━━━━━━━━━━━━━━━ 7s 790ms/step - accuracy: 0.8608 - loss: 0.3096 - val_accuracy: 0.9118 - val_loss: 0.2059
Epoch 17/30
9/9 ━━━━━━━━━━━━━━━━━━━━ 9s 684ms/step - accuracy: 0.9008 - loss: 0.3335 - val_accuracy: 0.9559 - val_loss: 0.2484
Epoch 18/30
9/9 ━━━━━━━━━━━━━━━━━━━━ 7s 808ms/step - accuracy: 0.8934 - loss: 0.2954 - val_accuracy: 0.8382 - val_loss: 0.3635
Epoch 19/30
9/9 ━━━━━━━━━━━━━━━━━━━━ 5s 524ms/step - accuracy: 0.8767 - loss: 0.3694 - val_accuracy: 0.8824 - val_loss: 0.3620
Epoch 20/30
9/9 ━━━━━━━━━━━━━━━━━━━━ 6s 660ms/step - accuracy: 0.8842 - loss: 0.3307 - val_accuracy: 0.7794 - val_loss: 0.4377
```




```
3/3 ━━━━━━━━━━━━━━━━━━━━ 1s 298ms/step - accuracy: 0.9775 - loss: 0.1941
Validation Accuracy: 97.06%
```

COE(20)	
RECORD(20)	
VIVA(10)	
TOTAL(50)	

RESULT:

A CNN model with strong data augmentation and dropout regularization was successfully trained and tested for image classification

EX NO : 06

DATE :

RNN ARCHITECTURE FOR TIME SERIES DATA

AIM:

To implement a Recurrent Neural Network (RNN) for stock market prediction using S&P 500 company data, where historical stock prices are used to forecast future values.

ALGORITHM:

Step 1: Install and import required libraries (yfinance, pandas, numpy, matplotlib, sklearn, keras).

Step 2: Mount Google Drive in Google Colab and set the CSV file path for S&P 500 company list.

Step 3: Load the company data from sp500_companies.csv and check availability.

Step 4: Select a stock ticker from the S&P 500 list (e.g., AAPL, MSFT).

Step 5: Download historical stock prices using yfinance (with date range and interval).

Step 6: Preprocess the data: keep Close price, handle missing values, and normalize with MinMaxScaler.

Step 7: Create input-output sequences (e.g., past 60 days → next day).

Step 8: Split the data into training (80%) and testing (20%) sets.

Step 9: Build an RNN model with SimpleRNN and a Dense output layer.

Step 10: Compile the model with Adam optimizer and MSE loss function.

Step 11: Train the model with early stopping and checkpoint callbacks.

Step 12: Predict stock prices on test data and inverse transform results.

Step 13: Evaluate performance using RMSE and MAE.

Step 14: Plot actual vs. predicted stock prices for comparison.

CODING:

```
!pip install yfinance --quiet
```

```
# Imports
import os
import pandas as pd
import numpy as np
```

```
import yfinance as yf
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error
from math import sqrt
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
# Mount Google Drive (Colab)
from google.colab import drive
drive.mount('/content/drive')
# Provided CSV path (as given by user)
CSV_PATH = "/content/drive/MyDrive/DL/sp500_companies.csv"
print("CSV exists?", os.path.exists(CSV_PATH))



df_csv = pd.read_csv(CSV_PATH)
print("CSV columns:", df_csv.columns.tolist())
print("CSV preview:")
display(df_csv.head())
has_date_close = any(col.lower() == 'date' for col in df_csv.columns) and any(col.lower() in
('close', 'adj close', 'adj_close') for col in df_csv.columns)
if has_date_close:
    print("Detected historical prices in CSV. Will use this file as the dataset.")
else:
    print("CSV looks like a tickers list. Will pick a ticker and download history via yfinance.")



def load_prices_from_csv(df):
    # Attempt to normalize column names
    cols = {c: c.strip() for c in df.columns}
    df = df.rename(columns=cols)
```

```
    # Find date column
    date_col = None
    for c in df.columns:
        if c.lower() == 'date':
            date_col = c
            break
    # Find close-like column
    close_col = None
    for c in df.columns:
        if c.lower() in ('close', 'adj close', 'adj_close'):
            close_col = c
            break
    if date_col is None or close_col is None:
        raise ValueError("CSV has no Date and Close columns in expected form.")
    df[date_col] = pd.to_datetime(df[date_col])
    df = df[[date_col, close_col]].dropna()
    df = df.set_index(date_col).sort_index()
    df = df.rename(columns={close_col: "Close"})
    return df

if has_date_close:
    prices = load_prices_from_csv(df_csv)
else:
    # treat CSV as tickers list
    # try common ticker columns
    ticker_col = None
    for c in df_csv.columns:
        if c.lower() in ('symbol', 'ticker', 'code'):
            ticker_col = c
            break
    if ticker_col is None:
        # fallback: use a manual ticker
        TICKER = "AAPL"
        print("No ticker column found; using default ticker:", TICKER)
```

```
    else:
        TICKER = df_csv[ticker_col].dropna().astype(str).iloc[0]
        print("Using ticker from CSV:", TICKER)

    # Download historical prices (adjust start/end as needed)
    start = "2010-01-01"
    end = None  # None means up to today
    print(f"Downloading {TICKER} from yfinance (this may take a moment)...")
    prices = yf.download(TICKER, start=start, end=end, progress=False)
    if prices.empty:
        raise RuntimeError("Downloaded price DataFrame is empty. Check ticker/internet.")
    prices = prices[['Close']].dropna()

print("Prices shape:", prices.shape)
display(prices.head())



# sequence generator
def create_sequences(data, seq_len):
    X, y = [], []
    for i in range(len(data) - seq_len):
        X.append(data[i:i+seq_len])
        y.append(data[i+seq_len])
    return np.array(X), np.array(y)

X, y = create_sequences(scaled_values, SEQ_LEN)
print("Raw sequences shapes X,y:", X.shape, y.shape)
# Train/test split (time-ordered)
split_idx = int(len(X) * (1 - TEST_RATIO))
X_train, X_test = X[:split_idx], X[split_idx:]
y_train, y_test = y[:split_idx], y[split_idx:]
# Reshape for RNN: already (samples, seq_len, features). features=1 here.
print("Train shape:", X_train.shape, y_train.shape)
print("Test shape:", X_test.shape, y_test.shape)
```

```
model = Sequential([
    SimpleRNN(64, input_shape=(SEQ_LEN, 1), activation='tanh'),
    Dense(1)
])
model.compile(optimizer='adam', loss='mse', metrics=['mae'])
model.summary()
# Callbacks
checkpoint_path = "/content/drive/MyDrive/DL/rnn_best_model.h5"
es = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
mc = ModelCheckpoint(checkpoint_path, save_best_only=True, monitor='val_loss')
# Train
history = model.fit(
    X_train, y_train,
    epochs=100,
    batch_size=32,
    validation_split=0.1,
    callbacks=[es, mc],
    verbose=2
)


# Predict
y_pred_scaled = model.predict(X_test)
# Inverse transform to original price scale
y_pred = scaler.inverse_transform(y_pred_scaled.reshape(-1,1)).flatten()
y_true = scaler.inverse_transform(y_test.reshape(-1,1)).flatten()
# Metrics
mse = mean_squared_error(y_true, y_pred)
rmse = sqrt(mse)
mae = mean_absolute_error(y_true, y_pred)
print(f"Test RMSE: {rmse:.4f}, MAE: {mae:.4f}")
# Plot a slice of actual vs predicted
```

```
plt.figure(figsize=(12,6))
plt.plot(y_true, label='Actual')
plt.plot(y_pred, label='Predicted')
plt.title("Actual vs Predicted (Test set)")
plt.legend()
plt.show()
# zoom on last N points
N = 200
plt.figure(figsize=(12,6))
plt.plot(y_true[-N:], label='Actual')
plt.plot(y_pred[-N:], label='Predicted')
plt.title(f"Last {N} points: Actual vs Predicted")
plt.legend()
plt.show()
```

OUTPUT:

```
CSV columns: ['Exchange', 'Symbol', 'Shortname', 'Longname', 'Sector', 'Industry', 'Currentprice', 'Marketcap', 'Ebitda', 'Revenuegrowth', 'City', 'State', 'Country', 'Fulltimeemployees', 'Longbusinesssummar
CSV preview:
```

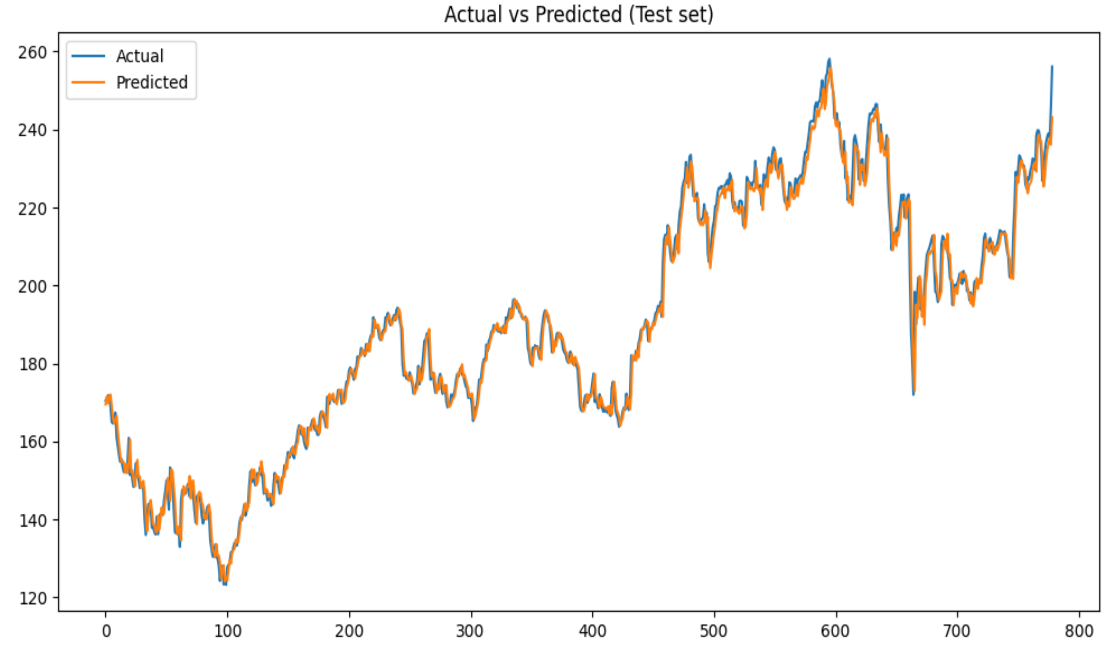
	Exchange	Symbol	Shortname	Longname	Sector	Industry	Currentprice	Marketcap	Ebitda	Revenuegrowth	City	State	Country	Fulltimeemployees	Longbusinesssummary	Weight
0	NMS	AAPL	Apple Inc.	Apple Inc.	Technology	Consumer Electronics	254.49	3846819807232	1.346610e+11	0.061	Cupertino	CA	United States	164000.0	Apple Inc. designs, manufactures, and markets ...	0.069209
1	NMS	NVDA	NVIDIA Corporation	NVIDIA Corporation	Technology	Semiconductors	134.70	3298803056640	6.118400e+10	1.224	Santa Clara	CA	United States	29600.0	NVIDIA Corporation provides graphics and compu...	0.059350
2	NMS	MSFT	Microsoft Corporation	Microsoft Corporation	Technology	Software - Infrastructure	436.60	3246068596736	1.365520e+11	0.160	Redmond	WA	United States	228000.0	Microsoft Corporation develops and supports so...	0.058401
3	NMS	AMZN	Amazon.com, Inc.	Amazon.com, Inc.	Consumer Cyclical	Internet Retail	224.92	2365033807872	1.115830e+11	0.110	Seattle	WA	United States	1551000.0	Amazon.com, Inc. engages in the retail sale of...	0.042550
4	NMS	GOOGL	Alphabet Inc.	Alphabet Inc.	Communication Services	Internet Content & Information	191.41	2351625142272	1.234700e+11	0.151	Mountain View	CA	United States	181269.0	Alphabet Inc. offers various products and plat...	0.042309

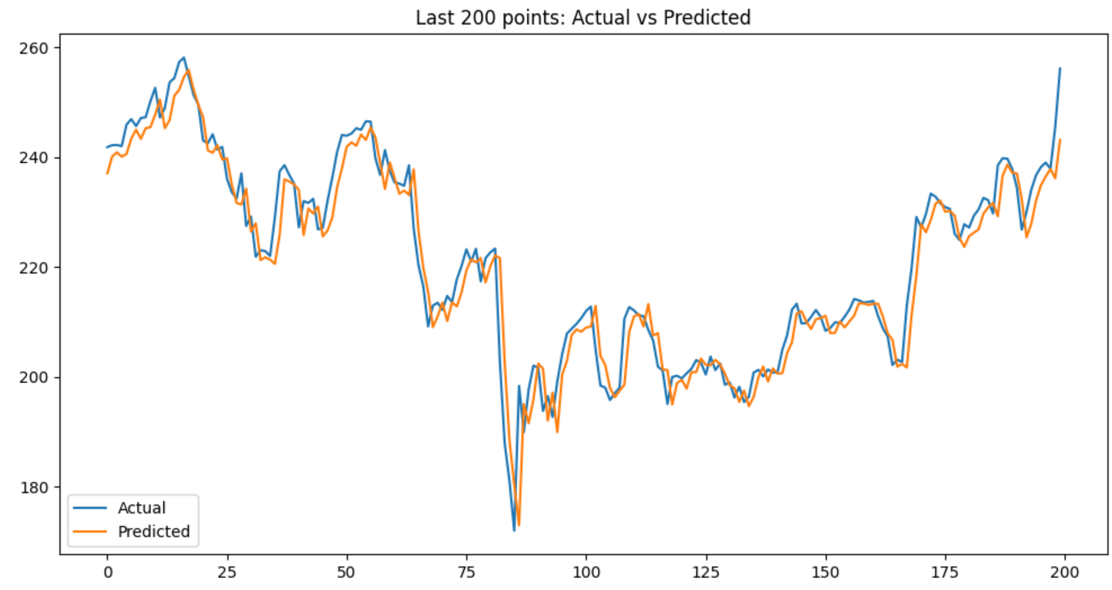
```
CSV looks like a tickers list. Will pick a ticker and download history via yfinance.
```

Price	Close
Ticker	AAPL
Date	
2010-01-04	6.424605
2010-01-05	6.435713
2010-01-06	6.333345
2010-01-07	6.321635
2010-01-08	6.363664

```
Epoch 42/100
88/88 - 2s - 18ms/step - loss: 2.0863e-05 - mae: 0.0029 - val_loss: 1.5641e-04 - val_mae: 0.0096
Epoch 43/100
88/88 - 1s - 13ms/step - loss: 1.8378e-05 - mae: 0.0025 - val_loss: 1.9191e-04 - val_mae: 0.0111
Epoch 44/100
88/88 - 1s - 10ms/step - loss: 1.8592e-05 - mae: 0.0026 - val_loss: 2.2614e-04 - val_mae: 0.0122
Epoch 45/100
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g.
88/88 - 1s - 11ms/step - loss: 1.8966e-05 - mae: 0.0026 - val_loss: 1.3857e-04 - val_mae: 0.0090
Epoch 46/100
88/88 - 1s - 10ms/step - loss: 1.8496e-05 - mae: 0.0025 - val_loss: 1.8545e-04 - val_mae: 0.0106
Epoch 47/100
88/88 - 1s - 14ms/step - loss: 1.8353e-05 - mae: 0.0025 - val_loss: 3.4599e-04 - val_mae: 0.0159
Epoch 48/100
88/88 - 1s - 11ms/step - loss: 2.5472e-05 - mae: 0.0033 - val_loss: 1.6559e-04 - val_mae: 0.0102
Epoch 49/100
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g.
88/88 - 1s - 14ms/step - loss: 2.7436e-05 - mae: 0.0035 - val_loss: 1.3116e-04 - val_mae: 0.0087
Epoch 50/100
88/88 - 1s - 16ms/step - loss: 1.8050e-05 - mae: 0.0025 - val_loss: 1.4071e-04 - val_mae: 0.0092
Epoch 51/100
88/88 - 1s - 10ms/step - loss: 1.8439e-05 - mae: 0.0026 - val_loss: 1.3191e-04 - val_mae: 0.0088
Epoch 52/100
88/88 - 1s - 13ms/step - loss: 2.0538e-05 - mae: 0.0029 - val_loss: 1.3557e-04 - val_mae: 0.0090
Epoch 53/100
88/88 - 2s - 28ms/step - loss: 1.6330e-05 - mae: 0.0023 - val_loss: 1.6861e-04 - val_mae: 0.0104
Epoch 54/100
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g.
88/88 - 1s - 16ms/step - loss: 1.8558e-05 - mae: 0.0026 - val_loss: 1.2551e-04 - val_mae: 0.0085
Epoch 55/100
88/88 - 1s - 10ms/step - loss: 1.8143e-05 - mae: 0.0025 - val_loss: 1.5116e-04 - val_mae: 0.0095
Epoch 56/100
88/88 - 1s - 14ms/step - loss: 2.0496e-05 - mae: 0.0028 - val_loss: 1.2885e-04 - val_mae: 0.0087
Epoch 57/100
88/88 - 1s - 10ms/step - loss: 1.8241e-05 - mae: 0.0026 - val_loss: 1.4014e-04 - val_mae: 0.0090
Epoch 58/100
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g.
88/88 - 1s - 10ms/step - loss: 1.7099e-05 - mae: 0.0024 - val_loss: 1.2373e-04 - val_mae: 0.0084
Epoch 59/100
88/88 - 1s - 12ms/step - loss: 2.0963e-05 - mae: 0.0029 - val_loss: 1.6533e-04 - val_mae: 0.0103
Epoch 60/100
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g.
88/88 - 1s - 11ms/step - loss: 1.7079e-05 - mae: 0.0024 - val_loss: 1.2227e-04 - val_mae: 0.0084
Epoch 61/100
88/88 - 1s - 10ms/step - loss: 1.8072e-05 - mae: 0.0025 - val_loss: 1.5112e-04 - val_mae: 0.0094
Epoch 62/100
88/88 - 1s - 14ms/step - loss: 1.7854e-05 - mae: 0.0026 - val_loss: 1.2296e-04 - val_mae: 0.0084
Epoch 63/100
88/88 - 1s - 10ms/step - loss: 1.9461e-05 - mae: 0.0027 - val_loss: 1.2902e-04 - val_mae: 0.0088
Epoch 64/100
88/88 - 1s - 13ms/step - loss: 1.7690e-05 - mae: 0.0025 - val_loss: 1.2306e-04 - val_mae: 0.0084
```

```
25/25 ━━━━━━━━━━━━━━━━━━━━ 2s 56ms/step
Test RMSE: 3.4491, MAE: 2.4794
```





COE(20)	
RECORD(20)	
VIVA(10)	
TOTAL(50)	

RESULT:

The RNN model was trained on S&P 500 stock data and successfully predicted future stock prices. The results showed that predicted values closely matched actual values, proving the model's effectiveness.

EX NO : 07

DATE :

NLP TEXT ANALYSIS STEPS

AIM:

To perform text analysis on the IMDB movie review dataset using Natural Language Processing (NLP) techniques and classify the reviews into positive or negative sentiments with the help of an LSTM-based deep learning model.

ALGORITHM:

Step 1: Import the required libraries (pandas, numpy, sklearn, keras, re).

Step 2: Load the IMDB dataset from the given file path.

Step 3: Preprocess the text by converting to lowercase, removing HTML tags, non-alphabet characters, and extra spaces.

Step 4: Encode the sentiment labels → positive = 1, negative = 0.

Step 5: Tokenize the cleaned text into word sequences using Keras Tokenizer.

Step 6: Vectorize the token sequences by padding them to a fixed length.

Step 7: Split the dataset into training and testing sets.

Step 8: Build an LSTM-based deep learning model with embedding and dense layers.

Step 9: Train the model on the training set and validate using a validation split.

Step 10: Evaluate the model on the test set and predict sentiment for new input text.

CODING:

```
import pandas as pd
import numpy as np
import re
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, accuracy_score
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout
```

```
df = pd.read_csv("/content/drive/MyDrive/DL/IMDB Dataset.csv")
print(df.head())
```

```
                                              review sentiment
0  One of the other reviewers has mentioned that ...  positive
1  A wonderful little production. <br /><br />The...  positive
2  I thought this was a wonderful way to spend ti...  positive
3  Basically there's a family where a little boy ...  negative
4  Petter Mattei's "Love in the Time of Money" is...  positive
```

```
def clean_text(text):
    text = text.lower()                              # Lowercase
    text = re.sub(r"<br />", " ", text)              # Remove HTML tags
    text = re.sub(r"[^a-zA-Z]", " ", text)           # Keep only alphabets
    text = re.sub(r"\s+", " ", text)                 # Remove extra spaces
    return text.strip()
df['review'] = df['review'].apply(clean_text)
# Encode labels: positive=1, negative=0
df['sentiment'] = df['sentiment'].map({'positive':1, 'negative':0})

X = df['review'].values
y = df['sentiment'].values
tokenizer = Tokenizer(num_words=10000, oov_token="<OOV>")
tokenizer.fit_on_texts(X)
X_seq = tokenizer.texts_to_sequences(X)

maxlen = 200
X_pad = pad_sequences(X_seq, maxlen=maxlen)

X_train, X_test, y_train, y_test = train_test_split(X_pad, y, test_size=0.2, random_state=42)

model = Sequential()
model.add(Embedding(input_dim=10000, output_dim=128, input_length=maxlen))
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))

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

```
print(model.summary())
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
embedding (Embedding)	?	0 (unbuilt)
lstm (LSTM)	?	0 (unbuilt)
dense (Dense)	?	0 (unbuilt)

```
history = model.fit(X_train, y_train, batch_size=64, epochs=3, validation_split=0.2, verbose=2)

y_pred = (model.predict(X_test) > 0.5).astype("int32")
print("Accuracy:", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))

sample = ["The movie was absolutely wonderful and touching"]
sample_cleaned = [clean_text(text) for text in sample] # Apply clean_text to the sample
print("Cleaned sample:", sample_cleaned) # Print cleaned text
sample_seq = tokenizer.texts_to_sequences(sample_cleaned)
print("Sample sequence:", sample_seq) # Print generated sequence
# Filter out None values before padding
sample_seq = [seq for seq in sample_seq if seq is not None]
sample_pad = pad_sequences(sample_seq, maxlen=maxlen)
prediction = model.predict(sample_pad)
print("Sentiment:", "Positive" if prediction[0][0] > 0.5 else "Negative")
```

OUTPUT:

```
Epoch 1/3
500/500 - 297s - 594ms/step - accuracy: 0.7732 - loss: 0.4820 - val_accuracy: 0.8497 - val_loss: 0.3580
Epoch 2/3
500/500 - 318s - 635ms/step - accuracy: 0.8650 - loss: 0.3336 - val_accuracy: 0.8556 - val_loss: 0.3432
Epoch 3/3
500/500 - 287s - 573ms/step - accuracy: 0.8835 - loss: 0.2935 - val_accuracy: 0.8601 - val_loss: 0.3377
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 27s 84ms/step
Accuracy: 0.8644
              precision    recall  f1-score   support

           0       0.87      0.85      0.86      4961
           1       0.86      0.88      0.87      5039

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

```
Cleaned sample: ['the movie was absolutely wonderful and touching']
Sample sequence: [[2, 16, 14, 419, 391, 3, 1350]]
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 70ms/step
Sentiment: Positive
```

COE(20)	
RECORD(20)	
VIVA(10)	
TOTAL(50)	

RESULT:

The IMDB dataset was successfully preprocessed, tokenized, and converted into padded sequences. An LSTM model was trained on the data, and the sentiment of reviews was classified as positive.

EX NO : 08

DATE :

DEEPDREAM & NEURAL STYLE TRANSFER

AIM:

To create an artistic image by blending the style of a painting with the content of a photo. This shows how AI can generate creative visuals using deep learning.

ALGORITHMS:

STEP 1: Load the content image and style image, remove any transparency, resize them to safe dimensions, and normalize the pixel values.

STEP 2: Load the pre-trained neural style transfer model from TensorFlow Hub and access its default signature for stylization.

STEP 3: Pass both images into the model using the correct input names and extract the stylized output image from the result.

STEP 4: Display the content image, style image, and stylized output side by side using matplotlib for visual comparison.

STEP 5: Optionally save the stylized image to your Google Drive or use it in your design or presentation.

CODE:

A) NEURAL STYLE TRANSFER

```
import tensorflow as tf
import tensorflow_hub as hub
import numpy as np
import PIL.Image
import matplotlib.pyplot as plt
import requests
from io import BytesIO
# Load and prepare image
def load_image_from_url(url, max_dim=512):
    img = PIL.Image.open(BytesIO(requests.get(url).content)).convert('RGB')
    img.thumbnail((max_dim, max_dim))
    img = np.array(img).astype(np.float32)[np.newaxis, ...] / 255.0
    return tf.convert_to_tensor(img, dtype=tf.float32)
# Use known working images
content_url =
"https://storage.googleapis.com/download.tensorflow.org/example_images/YellowLabradorL
ooking_new.jpg"
style_url =
"https://storage.googleapis.com/download.tensorflow.org/example_images/Vassily_Kandinsk
y%2C_1913_-_Composition_7.jpg"
content_image = load_image_from_url(content_url)
style_image = load_image_from_url(style_url)
# Load model
stylize_model = hub.load("https://tfhub.dev/google/magenta/arbitrary-image-stylization-v1-
256/2")
stylize = stylize_model.signatures['serving_default']
```

```
# Stylize
result = stylize(placeholder=content_image, placeholder_1=style_image)
stylized_image = result['output_0']
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.imshow(tf.squeeze(content_image))
plt.title("Content Image")
plt.axis('off')
plt.subplot(1, 3, 2)
plt.imshow(tf.squeeze(style_image))
plt.title("Style Image")
plt.axis('off')
plt.subplot(1, 3, 3)
plt.imshow(tf.squeeze(stylized_image))
plt.title("Stylized Output")
plt.axis('off')
plt.tight_layout()
plt.show()
```

OUTPUT:

B) DEEPDREA

```
# STEP 1: Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')
# STEP 2: Install dependencies
!pip install tensorflow matplotlib
# STEP 3: Import libraries
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.applications import inception_v3
import numpy as np
import matplotlib.pyplot as plt
# STEP 4: Load your image from Drive
base_image_path = "/content/drive/MyDrive/imagesdl/OIP.jpg"
# Display original image
plt.figure(figsize=(6, 6))
plt.axis("off")
plt.imshow(keras.utils.load_img(base_image_path))
plt.title("Original Image")
plt.show()
# STEP 5: Load pretrained InceptionV3 model
model = inception_v3.InceptionV3(weights="imagenet", include_top=False)
# STEP 6: Select layers to enhance
layer_settings = {
  "mixed3": 0.5,
  "mixed5": 1.0,
  "mixed7": 1.5,
```

```
}
outputs_dict = {
    layer.name: layer.output
    for layer in [model.get_layer(name) for name in layer_settings.keys()]
}
feature_extractor = keras.Model(inputs=model.inputs, outputs=outputs_dict)
# STEP 7: Define loss function
def compute_loss(input_image):
    features = feature_extractor(input_image)
    loss = tf.zeros(shape=())
    for name in features.keys():
        coeff = layer_settings[name]
        activation = features[name]
        loss += coeff * tf.reduce_mean(tf.square(activation[:, 2:-2, 2:-2, :]))
    return loss
# STEP 8: Gradient ascent step
@tf.function
def gradient_ascent_step(image, learning_rate):
    with tf.GradientTape() as tape:
        tape.watch(image)
        loss = compute_loss(image)
    grads = tape.gradient(loss, image)
    grads = tf.math.l2_normalize(grads)
    image += learning_rate * grads
    return loss, image
def gradient_ascent_loop(image, iterations, learning_rate, max_loss=None):
    for i in range(iterations):
```

```
 loss, image = gradient_ascent_step(image, learning_rate)
     if max_loss is not None and loss > max_loss:
        break
     if i % 10 == 0:
        print(f"... Loss at step {i}: {loss:.2f}")
  return image
# STEP 9: Preprocessing utilities
def preprocess_image(image_path):
  img = keras.utils.load_img(image_path)
  img = keras.utils.img_to_array(img)
  img = np.expand_dims(img, axis=0)
  img = inception_v3.preprocess_input(img)
  return img
def deprocess_image(img):
  img = img.reshape((img.shape[1], img.shape[2], 3))
  img /= 2.0
  img += 0.5
  img *= 255.0
  img = np.clip(img, 0, 255).astype("uint8")
  return img # STEP 10: Run DeepDream
step = 20.0
iterations = 30
max_loss = 15.0
original_img = preprocess_image(base_image_path)
dream_img = gradient_ascent_loop(tf.identity(original_img), iterations, step, max_loss)
# STEP 11: Display result
final_img = deprocess_image(dream_img.numpy())
```

```
plt.figure(figsize=(8, 8))

plt.axis("off")

plt.imshow(final_img)

plt.title("DeepDream Output")

plt.show()
```

OUTPUT:

```
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.12/dist-packages (from requests<3,>=2.21.0->tensorflow)
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.12/dist-packages (from tensorboard~=2.19.0->tensorflow) (3
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in /usr/local/lib/python3.12/dist-packages (from tensorboard~=
Requirement already satisfied: werkzeug>=1.0.1 in /usr/local/lib/python3.12/dist-packages (from tensorboard~=2.19.0->tensorflow) (3
Requirement already satisfied: MarkupSafe>=2.1.1 in /usr/local/lib/python3.12/dist-packages (from werkzeug>=1.0.1->tensorboard~=2.1
Requirement already satisfied: markdown-it-py>=2.2.0 in /usr/local/lib/python3.12/dist-packages (from rich->keras>=3.5.0->tensorflo
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /usr/local/lib/python3.12/dist-packages (from rich->keras>=3.5.0->tensorf
Requirement already satisfied: mdurl~=0.1 in /usr/local/lib/python3.12/dist-packages (from markdown-it-py>=2.2.0->rich->keras>=3.5.
```

Original Image

```
... Loss at step 0: 0.73
... Loss at step 10: 4.13
... Loss at step 20: 7.72
```

```
... Loss at step 20: 7.72
```

DeepDream Output

COE(20)	
RECORD(20)	
VIVA(10)	
TOTAL(50)	

RESULT:

The final image keeps the shape of the original photo but looks like a painting. It proves that AI can turn simple photos into creative artwork.

EX NO : 09

DATE :

VARIATIONAL AUTOENCODER (VAE) FOR SYNTHETIC IMAGES

AIM:

To train a Variational Autoencoder (VAE) on handwritten digits and generate new, realistic looking synthetic images of a specific digit. This demonstrates how deep learning can learn patterns and recreate data with variation.

ALGORITHMS:

STEP 1: Load and preprocess MNIST data, keeping only the target digit.

STEP 2: Build the VAE with encoder, sampling layer, and decoder.

STEP 3: Train using reconstruction loss and KL divergence.

STEP 4: Sample latent space to generate synthetic digit images.

STEP 5: Compare original vs reconstructed images for validation.

STEP 6: Interpolate between latent points to show smooth transitions.

STEP 7: Monitor KL and reconstruction loss separately during training.

CODE:

```
# STEP 1: Import Libraries
import tensorflow as tf
from tensorflow.keras import layers, models
import numpy as np
import matplotlib.pyplot as plt
# Load MNIST and filter only digit "3"
(x_train, y_train), _ = tf.keras.datasets.mnist.load_data()
x_train = x_train[y_train == 3]  # keep only digit 3
x_train = x_train.astype("float32") / 255.0
x_train = np.reshape(x_train, (-1, 28, 28, 1))
# STEP 3: Define Sampling Layer
class Sampling(layers.Layer):
    def call(self, inputs):
        z_mean, z_log_var = inputs
        epsilon = tf.random.normal(shape=tf.shape(z_mean))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon
# STEP 4: Build Encoder
latent_dim = 2
encoder_inputs = layers.Input(shape=(28, 28, 1))
x = layers.Flatten()(encoder_inputs)
x = layers.Dense(128, activation="relu")(x)
z_mean = layers.Dense(latent_dim)(x)
z_log_var = layers.Dense(latent_dim)(x)
z = Sampling()([z_mean, z_log_var])
encoder = models.Model(encoder_inputs, [z_mean, z_log_var, z], name="encoder")
```

```
# STEP 5: Build Decoder
decoder_inputs = layers.Input(shape=(latent_dim,))
x = layers.Dense(128, activation="relu")(decoder_inputs)
x = layers.Dense(28 * 28, activation="sigmoid")(x)
x = layers.Reshape((28, 28, 1))(x)
decoder = models.Model(decoder_inputs, x, name="decoder")
# STEP 6: Build VAE Model
class VAE(models.Model):
    def __init__(self, encoder, decoder):
        super(VAE, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
    def compile(self, optimizer):
        super(VAE, self).compile()
        self.optimizer = optimizer
        self.loss_fn = tf.keras.losses.BinaryCrossentropy()
    def train_step(self, data):
        with tf.GradientTape() as tape:
            z_mean, z_log_var, z = self.encoder(data)
            reconstruction = self.decoder(z)
            reconstruction_loss = self.loss_fn(data, reconstruction)
            kl_loss = -0.5 * tf.reduce_mean(
                z_log_var - tf.square(z_mean) - tf.exp(z_log_var) + 1
            )
            total_loss = reconstruction_loss + kl_loss
        grads = tape.gradient(total_loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
        return {"loss": total_loss}
```

```
vae = VAE(encoder, decoder)
vae.compile(optimizer=tf.keras.optimizers.Adam())
vae.fit(x_train, epochs=10, batch_size=128)
# STEP 7: Generate Synthetic Images
def plot_latent_space(decoder, n=10, figsize=10):
    digit_size = 28
    scale = 2.0
    figure = np.zeros((digit_size * n, digit_size * n))
    grid_x = np.linspace(-scale, scale, n)
    grid_y = np.linspace(-scale, scale, n)
    for i, yi in enumerate(grid_y):
        for j, xi in enumerate(grid_x):
            z_sample = np.array([[xi, yi]])
            x_decoded = decoder.predict(z_sample)
            digit = x_decoded[0].reshape(digit_size, digit_size)
            figure[i * digit_size: (i + 1) * digit_size,
                   j * digit_size: (j + 1) * digit_size] = digit
    plt.figure(figsize=(figsize, figsize))
    plt.imshow(figure, cmap="Greys_r")
    plt.axis("off")
    plt.title("Synthetic Digits from VAE")
    plt.show()
plot_latent_space(decoder)
```

OUTPUT:

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 52ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 46ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 64ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 47ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 51ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 53ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 57ms/step
```

COE(20)	
RECORD(20)	
VIVA(10)	
TOTAL(50)	

RESULT:

The model successfully generates varied versions of the digit "3", each with unique style and shape. This proves the VAE can learn and reproduce realistic patterns from limited data.

EX NO: 10

DATE:

GAN (GENERATIVE ADVERSARIAL NETWORK)

AIM:

To generate synthetic handwritten digit images using a Generative Adversarial Network (GAN) trained on the MNIST dataset.

ALGORITHMS:

STEP 1: Import TensorFlow, NumPy, and Matplotlib libraries.

STEP 2: Load the MNIST dataset and normalize the images to the range [-1, 1].

STEP 3: Build the Generator network to create fake digit images from random noise.

STEP 4: Build the Discriminator network to classify images as real or fake.

STEP 5: Define loss functions and optimizers for both networks.

STEP 6: Train the GAN by alternating between training the Discriminator on real and generated images and training the Generator to fool the Discriminator.

STEP 7: After every fixed number of epochs, generate and visualize synthetic digit images.

STEP 8: Plot generator and discriminator loss curves to analyze training stability.

CODE:

```
# STEP 1: Import Libraries
import tensorflow as tf
from tensorflow.keras import layers
import numpy as np
import matplotlib.pyplot as plt
# STEP 2: Load and Preprocess Data
(x_train, _), _ = tf.keras.datasets.mnist.load_data()
x_train = (x_train.astype("float32") - 127.5) / 127.5 # Normalize [-1,1]
x_train = np.expand_dims(x_train, axis=-1)
BUFFER_SIZE = 60000
BATCH_SIZE = 256
train_dataset =
tf.data.Dataset.from_tensor_slices(x_train).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
# STEP 3: Build Generator
def build_generator():
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
# STEP 4: Build Discriminator
def build_discriminator():
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
# STEP 5: Loss and Optimizers
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)
return real_loss + fake_loss
def generator_loss(fake_output):
return cross_entropy(tf.ones_like(fake_output), fake_output)
generator = build_generator()
discriminator = build_discriminator()
```

```
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
# STEP 6: Training Setup
EPOCHS = 300 #  Increased epochs
noise_dim = 100
num_examples_to_generate = 16
seed = tf.random.normal([num_examples_to_generate, noise_dim])
gen_losses, disc_losses = [], []
@tf.function
def train_step(images):
noise = tf.random.normal([BATCH_SIZE, noise_dim])
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
# STEP 7: Generate and Plot Images
def generate_and_plot_images(model, test_input, epoch):
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] + 1) / 2.0, cmap="gray")
plt.axis("off")
plt.suptitle(f"Epoch {epoch} – Synthetic Digits")
plt.show()
# STEP 8: Training Loop
def train(dataset, epochs):
for epoch in range(1, epochs + 1):
g_losses, d_losses = [], []
for image_batch in dataset:
g_loss, d_loss = train_step(image_batch)
g_losses.append(g_loss)
d_losses.append(d_loss)
gen_losses.append(np.mean(g_losses))
disc_losses.append(np.mean(d_losses))
print(f"Epoch {epoch}, Gen Loss: {gen_losses[-1]:.4f}, Disc Loss: {disc_losses[-1]:.4f}")
if epoch % 50 == 0 or epoch == 1:
generate_and_plot_images(generator, seed, epoch)
# Plot Loss Curves
plt.figure(figsize=(8, 4))
plt.plot(gen_losses, label="Generator Loss")
plt.plot(disc_losses, label="Discriminator Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
```

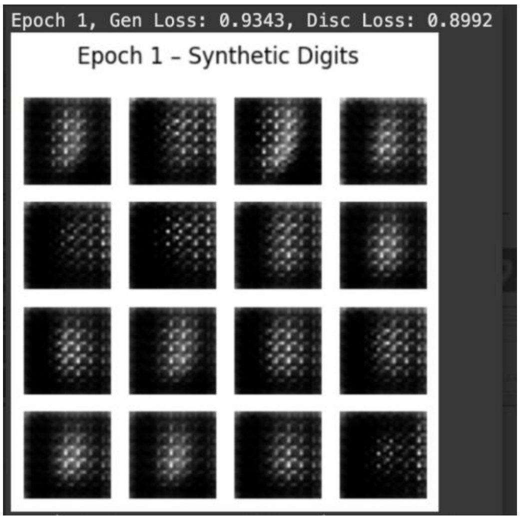
plt.title("Training Loss Curves")

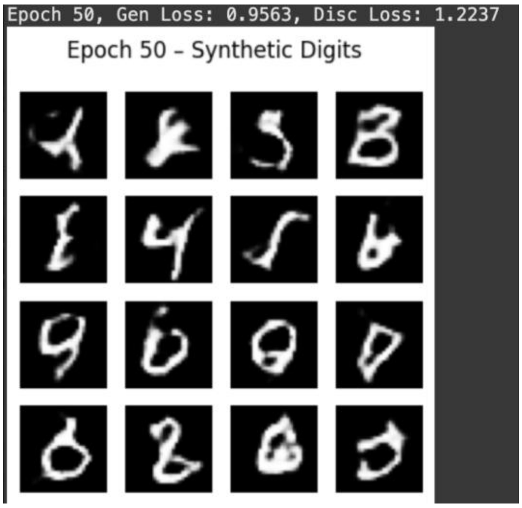
plt.show()

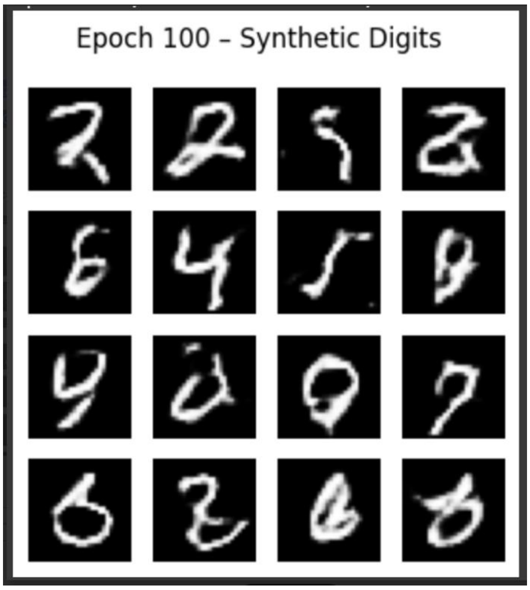
STEP 9: Run Training

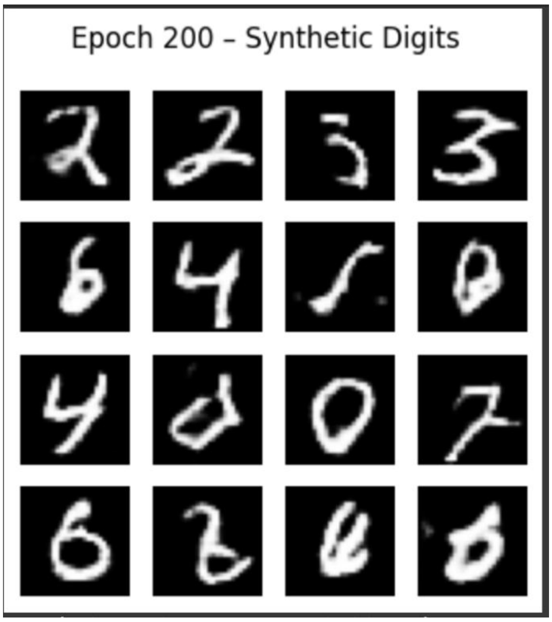
train(train_dataset, EPOCHS)

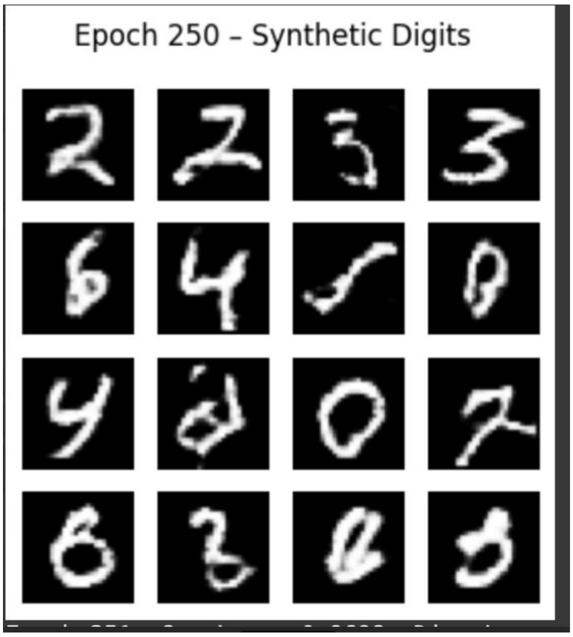
OUTPUT:

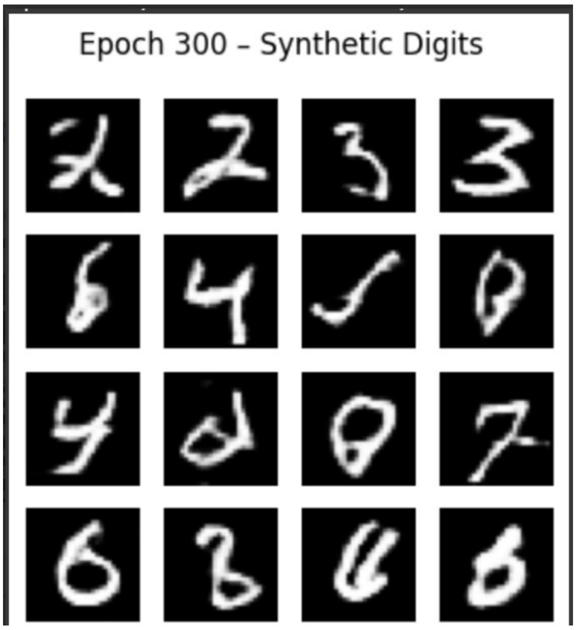


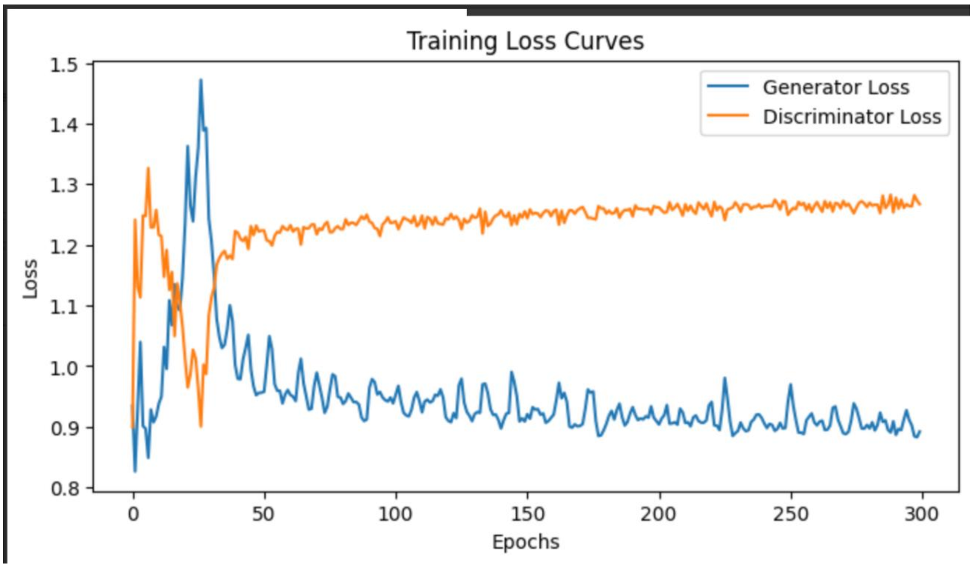












COE(20)	
RECORD(20)	
VIVA(10)	
TOTAL(50)	

RESULT:

The GAN successfully generates synthetic handwritten digits resembling MNIST images, with visual quality improving as training epochs increase.