EX. NO: 02

DATE :

MULTILAYER PERCEPTRON WITH HYPERPARAMETER TUNING

AIM:

To build a Multilayer Perceptron (MLP) model using the student-mat.csv dataset and improve its performance through hyperparameter tuning to classify students as pass or fail.

ALGORITHM:

STEP 1: Install required libraries (numpy, pandas, tensorflow).

STEP 2: Import packages- numpy, pandas, Sequential, and Dense from Keras.

STEP 3: Create the dataset using a dictionary and convert it to a Pandas DataFrame.

STEP 4: Separate features and labels X → input features (feature1, feature2), y → output labels (label)

STEP 5: Build the model using Sequential Add a hidden layer with 12 neurons and ReLU activation. Add an output layer with 1 neuron and sigmoid activation.

STEP 6: Compile the model with Loss-binary_crossentropy , Optimizer- adam , Metricaccuracy

STEP 7: Train the model using fit() with 100 epochs and batch size of 10.

STEP 8: Split the dataset into training and testing sets (80-20 split).

STEP 9: Build the MLP model using Sequential, with multiple dense layers and Dropout to avoid overfitting.

STEP 10: Compile the model with the Adam optimizer and binary_crossentropy loss.

STEP 11: Apply EarlyStopping to prevent overfitting during training.

STEP 12: Train the model using fit() with validation split and early stopping.

STEP 13: Evaluate the model on the same dataset using evaluate().

STEP 14: Print the results.

PROGRAM:

```
import pandas as pd  import
numpy as np
from sklearn.model_selection import train_test_split  from
sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.metrics import classification_report,confusion_matrix import
matplotlib.pyplot as plt import seaborn as sns
import tensorflow as tf from tensorflow.keras.models
import Sequential from tensorflow.keras.layers import
Dense, Dropout from
tensorflow.keras.callbacks import EarlyStopping
 df = pd.read_csv("/content/drive/MyDrive/student-mat.csv", sep=";")
df['pass'] = (df['G3'] >= 10).astype(int)   for col in df.columns:     if
df[col].dtype == 'object':          df[col] =
LabelEncoder().fit_transform(df[col])  X=df.drop(['G3','pass'],axis=1)
y=df['pass']  scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=42
)
model = Sequential([
    Dense(128, activation='relu', input_shape=(X.shape[1],)),
    Dropout(0.3),
    Dense(64, activation='relu'),
    Dropout(0.2),
   Dense(1, activation='sigmoid')
])
```

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])  early_stop
= EarlyStopping(monitor='val_loss', patience=5,
restore_best_weights=True)

history = model.fit(
X_train, y_train,
validation_split=0.2,
epochs=30,     batch_size=32,
callbacks=[early_stop],
verbose=1
)
loss, acc = model.evaluate(X_test, y_test, verbose=0)
print(f"\n Final Test Accuracy: {acc * 100:.2f}%")
y_pred = (model.predict(X_test) > 0.5).astype(int)
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(6, 4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")  plt.show()
```

OUTPUT:

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 2s 40ms/step - accuracy: 0.4463 - loss: 0.7446 - val_accuracy: 0.5625 - val_loss: 0.7137
Epoch 2/30
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.7027 - loss: 0.6038 - val_accuracy: 0.6406 - val_loss: 0.6741
Epoch 3/30
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.6924 - loss: 0.5932 - val_accuracy: 0.6406 - val_loss: 0.6300
Epoch 4/30
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.7718 - loss: 0.4859 - val_accuracy: 0.6875 - val_loss: 0.5907
Epoch 5/30
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.8024 - loss: 0.4184 - val_accuracy: 0.7500 - val_loss: 0.5526
Epoch 6/30
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.8148 - loss: 0.4154 - val_accuracy: 0.7812 - val_loss: 0.5161
Epoch 7/30
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.8642 - loss: 0.3599 - val_accuracy: 0.7812 - val_loss: 0.4907
Epoch 8/30
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9315 - loss: 0.2973 - val_accuracy: 0.7969 - val_loss: 0.4802
Epoch 9/30
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.8937 - loss: 0.2966 - val_accuracy: 0.7969 - val_loss: 0.4668
Epoch 10/30
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.8656 - loss: 0.3014 - val_accuracy: 0.7969 - val_loss: 0.4450
Epoch 11/30
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.8851 - loss: 0.2573 - val_accuracy: 0.8125 - val_loss: 0.4328
Epoch 12/30
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9358 - loss: 0.2031 - val_accuracy: 0.8125 - val_loss: 0.4190
Epoch 13/30
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9657 - loss: 0.1963 - val_accuracy: 0.8281 - val_loss: 0.4241
Epoch 14/30
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9729 - loss: 0.1471 - val_accuracy: 0.8125 - val_loss: 0.4167
Epoch 15/30
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9462 - loss: 0.1632 - val_accuracy: 0.8281 - val_loss: 0.4018
Epoch 16/30
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9338 - loss: 0.1729 - val_accuracy: 0.8125 - val_loss: 0.3988
Epoch 17/30
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9684 - loss: 0.1217 - val_accuracy: 0.8281 - val_loss: 0.3948
Epoch 18/30
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9254 - loss: 0.1610 - val_accuracy: 0.8438 - val_loss: 0.4030
Epoch 19/30
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9928 - loss: 0.0820 - val_accuracy: 0.8281 - val_loss: 0.4011
Epoch 20/30
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.9462 - loss: 0.1396 - val_accuracy: 0.7969 - val_loss: 0.4024
Epoch 21/30
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9827 - loss: 0.0898 - val_accuracy: 0.7969 - val_loss: 0.4101
Epoch 22/30
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.9676 - loss: 0.1071 - val_accuracy: 0.7969 - val_loss: 0.4219
```

Final Test Accuracy: 87.34%

1/3 ━━━━━━━━━━━━━━━━━━━━ **0s** 50ms/step **3/3**

━━━━━━━━━━━━━━━━━━━━ **0s** 30ms/step

Classification Report:

precision recall f1-score support

0 0.87 0.74 0.80 27

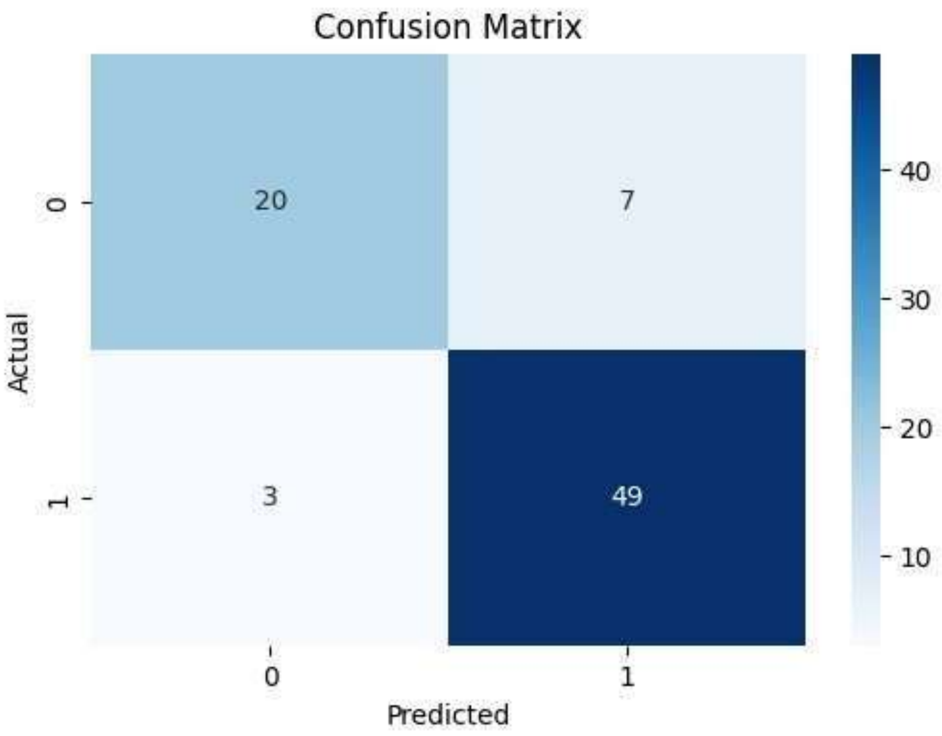
1 0.88 0.94 0.91 52 accuracy

0.87 79

macro avg 0.87 0.84 0.85 79 weighted

avg 0.87 0.87 0.87 79

RESULT:



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

RESULT:

The MLP model was successfully trained and tested. It accurately predicted student pass/fail outcomes based on academic and personal features with high classification performance.

EX. NO: 03

DATE :

DATA AUGMENTATION FOR IMAGE

AIM:

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

ALGORITHM:

STEP 1: Install gdown, import cv2, numpy, and matplotlib.

STEP 2: Download image from Google Drive using gdown.download().

STEP 3: Load image with cv2.imread() and convert BGR to RGB.

STEP 4: Resize image to 224×224 and display with matplotlib.

STEP 5: Define augmentations: horizontal flip, 30° rotation, brightness increase, and zoom crop.

STEP 6: Apply augmentations to the image.

STEP 7: Display augmented images side by side using subplots.

PROGRAM:

```
!pip install -q gdown import gdown import cv2
import numpy as np import matplotlib.pyplot as plt
file_id =
"1q7zwngJePSn43Gpx9VC8A5cNrTNaaPx"  #
New file ID url =
f"https://drive.google.com/uc?id={file_id}"
output_path = "sample_image.jpg"  # You can name
the file as you like
```

```
gdown.download(url, output_path, quiet=False)
original = cv2.imread(output_path) if original
is None:
    print(f"❌ Error: Image not found at
{output_path}") else:
    original = cv2.cvtColor(original, cv2.COLOR_BGR2RGB)
    image = cv2.resize(original, (224,
224))     plt.figure(figsize=(5, 5))
plt.imshow(image)     plt.title("Original
Image")     plt.axis("off")     plt.show()
def traditional_augmentations(image):
        flipped = cv2.flip(image, 1)         M =
cv2.getRotationMatrix2D((112, 112),
angle=30, scale=1.0)
        rotated = cv2.warpAffine(image, M, (224,
224))
        bright = cv2.convertScaleAbs(image,
alpha=1.2, beta=30)
        cropped = image[30:194, 30:194]  # (164,
164)         zoomed = cv2.resize(cropped, (224,
224))         return [flipped, rotated, bright, zoomed]
augmented_images =
traditional_augmentations(image)     titles =
["Flipped", "Rotated", "Brighter",
"Zoomed"]
    plt.figure(figsize=(15, 4))     for i, aug_img in
enumerate(augmented_images):         plt.subplot(1,
4, i + 1)         plt.imshow(aug_img)
```

```
    plt.title(titles[i])
plt.axis("off")
plt.tight_layout()     plt.show()
```

OUTPUT:

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

RESULT:

The image was successfully downloaded, preprocessed, and augmented. Traditional augmentations including horizontal flip, rotation, brightness adjustment, and zoom were effectively applied, and the transformed images were visualized for comparison with the original.

EX. NO: 04

DATE :

IMAGE DATA GENERATOR FOR IMAGE DATA AUGMENTATION

AIM:

To demonstrate the use of the Keras ImageDataGenerator class for performing real-time image data augmentation in order to increase dataset diversity and improve model generalization.

ALGORITHM:

STEP 1: Install required libraries (numpy, matplotlib, tensorflow).

STEP 2: Import Packages From tensorflow.keras.preprocessing.image, import ImageDataGenerator, load_img, and img_to_array. Import matplotlib.pyplot for displaying images.

STEP 3: Create an ImageDataGenerator object with augmentation parameters such as rotation_range, width_shift_range, height_shift_range, shear_range, zoom_range, horizontal_flip, and fill_mode.

STEP 4: Load a sample image using load_img(). Convert it to a NumPy array using img_to_array() and reshape it to (1, height, width, channels) for batch processing.

STEP 5: Use datagen.flow() to create an iterator that generates augmented image batches in real time.

STEP 6: Visualize Augmentations Iterate through the batches, display several augmented images using matplotlib.pyplot.imshow().

STEP 7: Use flow_from_directory() or flow() with a training directory or dataset to supply augmented images directly to a deep learning model during the model.fit() training process

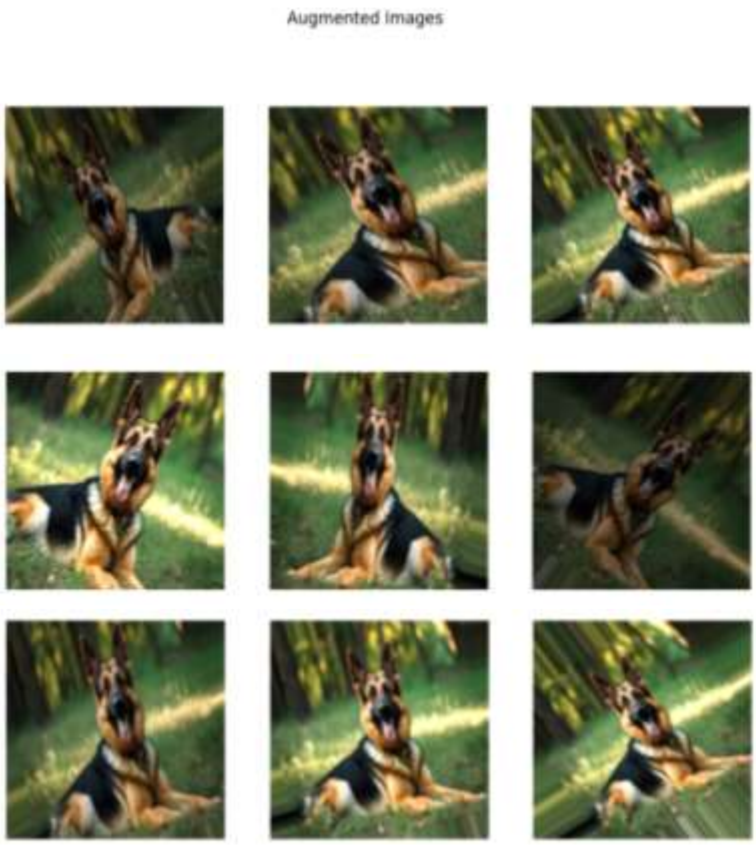
STEP 8: Train the model using the augmented data and evaluate its performance on a validation or test set to observe improved generalization and reduced overfitting.

PROGRAM:

```
import gdown

import numpy as np

import matplotlib.pyplot as plt

from tensorflow.keras.preprocessing.image import ImageDataGenerator, img_to_array,
load_img

file_id = "1q3tzA8L0rpZvwQG_5MxsjvhZi5q6DCJH"

gdown.download(f"https://drive.google.com/uc?id={file_id}", "simba.jpg", quiet=False)

img_path = "simba.jpg"

image = load_img(img_path, target_size=(224, 224))

image_array = img_to_array(image)

image_array = np.expand_dims(image_array, axis=0)

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

plt.imshow(image_array[0].astype("uint8"))

plt.title("Original Image")

plt.axis("off")

plt.show()

datagen = ImageDataGenerator(

    rotation_range=40,

    zoom_range=0.2,

    brightness_range=[0.5, 1.5],

    shear_range=0.2,

    horizontal_flip=True,

    fill_mode='nearest'

)

plt.figure(figsize=(10, 10))

i = 0

for batch in datagen.flow(image_array, 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.suptitle("Augmented Images")
plt.show()
```

OUTPUT:



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

RESULT:

The image was successfully downloaded, processed, and augmented. Multiple transformed versions including rotations, zooms, brightness changes, shears, and horizontal flips were generated and displayed, demonstrating effective real-time image data augmentation using Keras ImageDataGenerator.

EX. NO: 05

DATE :

CNN MODEL FOR IMAGE CLASSIFICATION

AIM:

To build, train, and evaluate a Convolutional Neural Network (CNN) to classify images into ten categories (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck) using the CIFAR-10 dataset.

ALGORITHM:

STEP 1: Import required libraries: tensorflow, matplotlib, and required Keras modules (datasets, layers, models).

STEP 2: Load the CIFAR-10 dataset and split into training and testing sets.

STEP 3: Normalize pixel values of images to the range `[0,1]` for faster convergence.

STEP 4: Build the CNN by stacking Conv2D and MaxPooling2D layers for feature extraction, then flatten the output, add a Dense layer with ReLU for non-linear learning, and finish with a 10-unit Dense layer for class prediction.

STEP 5: Compile the model using the Adam optimizer, SparseCategoricalCrossentropy loss, and accuracy as the metric.

STEP 6: Train the model for 10 epochs with the training set and validate on the test set.

STEP 7: Evaluate the trained model on the test data and print test accuracy.

STEP 8: Plot the training and validation accuracy curves to visualize performance.

PROGRAM:

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
 (train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
```

```
train_images, test_images = train_images / 255.0, test_images / 255.0
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10)  # 10 classes for CIFAR-10
])
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print(f"\nTest Accuracy: {test_acc * 100:.2f}%")
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training and Validation Accuracy')
plt.grid(True)
plt.show()
```

OUTPUT:

Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz

170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 4s 0us/step

/usr/local/lib/python3.12/dist-packages/keras/src/layers/convolutional/base_conv.py:113: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

super().__init__(activity_regularizer=activity_regularizer, **kwargs)

Epoch 1/10

1563/1563 ━━━━━━━━━━━━━━━━━━━━ 67s 42ms/step - accuracy: 0.3552 - loss: 1.7377 - val_accuracy: 0.5393 - val_loss: 1.2665

Epoch 2/10

1563/1563 ━━━━━━━━━━━━━━━━━━━━ 76s 38ms/step - accuracy: 0.5684 - loss: 1.2048 - val_accuracy: 0.6026 - val_loss: 1.1164

Epoch 3/10

1563/1563 ━━━━━━━━━━━━━━━━━━━━ 84s 40ms/step - accuracy: 0.6377 - loss: 1.0284 - val_accuracy: 0.6516 - val_loss: 0.9907

Epoch 4/10

1563/1563 ━━━━━━━━━━━━━━━━━━━━ 61s 39ms/step - accuracy: 0.6727 - loss: 0.9251 - val_accuracy: 0.6660 - val_loss: 0.9627

Epoch 5/10

1563/1563 ━━━━━━━━━━━━━━━━━━━━ 83s 40ms/step - accuracy: 0.7040 - loss: 0.8493 - val_accuracy: 0.6810 - val_loss: 0.9222

Epoch 6/10

1563/1563 ━━━━━━━━━━━━━━━━━━━━ 83s 40ms/step - accuracy: 0.7249 - loss: 0.7879 - val_accuracy: 0.6711 - val_loss: 0.9427

Epoch 7/10

1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 40ms/step - accuracy: 0.7422 - loss: 0.7412 - val_accuracy: 0.6942 - val_loss: 0.9027

Epoch 8/10

1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 40ms/step - accuracy: 0.7600 - loss: 0.6888 - val_accuracy: 0.7103 - val_loss: 0.8535

Epoch 9/10

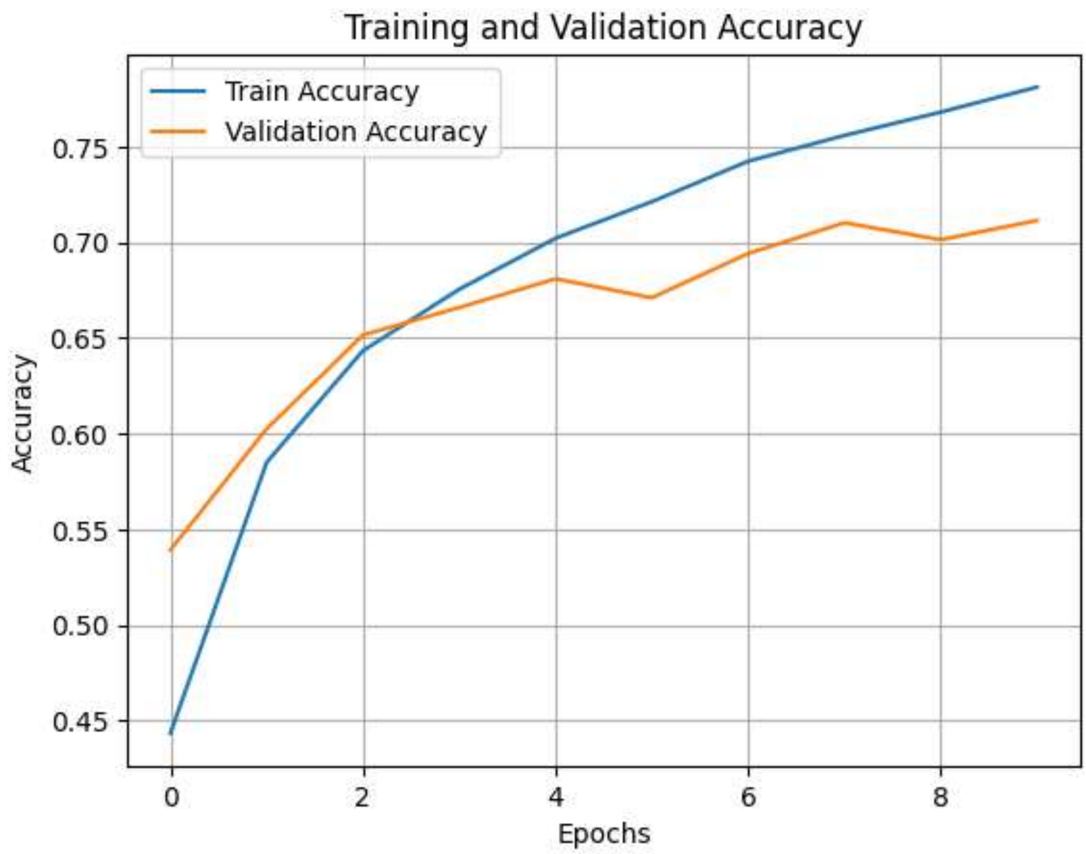
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 62s 39ms/step - accuracy: 0.7774 - loss: 0.6355 - val_accuracy: 0.7014 - val_loss: 0.8914

Epoch 10/10

1563/1563 ━━━━━━━━━━━━━━━━━━━━ 81s 39ms/step - accuracy: 0.7893 - loss: 0.6075 - val_accuracy: 0.7115 - val_loss: 0.8591

313/313 - 4s - 14ms/step - accuracy: 0.7115 - loss: 0.8591

Test Accuracy: 71.15%



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

RESULT:

The CNN was successfully trained and tested on the CIFAR-10 dataset. It accurately classified images into their respective categories, achieving high test accuracy and demonstrating effective feature extraction for image classification.

EX. NO: 06

DATE :

RNN ARCHITECTURE FOR TIME SERIES PREDICTION

AIM:

To design and train a Recurrent Neural Network (RNN) using a Simple RNN layer to predict the next value in a time-series sequence generated from a sine wave.

ALGORITHM:

STEP 1: Import required libraries: tensorflow, matplotlib, Simple RNN and Dense.

STEP 2: Generate synthetic time-series data (sine wave) and create input-output pairs where each input sequence predicts its next value.

STEP 3: Reshape the data into 3D format [samples, timesteps, features] required by RNN layers.

STEP 4: Build the RNN model using a SimpleRNN layer with 50 units and tanh activation, followed by a Dense layer with 1 neuron for output.

STEP 5: Compile the model with the Adam optimizer and mean squared error (MSE) loss function.

STEP 6: Train the model for 20 epochs with a batch size of 32 and a validation split of 20%.

STEP 7: Evaluate the model by making predictions on sample inputs and comparing them with true values.

STEP 8: Plot the training and validation loss to visualize learning performance.

PROGRAM:

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

```
from tensorflow.keras.layers import SimpleRNN, Dense
def generate_sine_wave(seq_length, num_samples):
    X, y = [], []
    for _ in range(num_samples):
        start = np.random.rand() * 2 * np.pi
        xs = np.linspace(start, start + 3 * np.pi, seq_length + 1)
        data = np.sin(xs)
        X.append(data[:-1])
        y.append(data[-1])
    return np.array(X), np.array(y)
seq_length = 50
num_samples = 1000
X, y = generate_sine_wave(seq_length, num_samples)
X = X.reshape((num_samples, seq_length, 1))
model = Sequential([
    SimpleRNN(50, activation='tanh', input_shape=(seq_length, 1)),
    Dense(1)   # Predict next value in sequence])
model.compile(optimizer='adam', loss='mse')
history = model.fit(X, y, epochs=20, batch_size=32, validation_split=0.2)
pred = model.predict(X[:10])
print("\nSample Predictions:")
for i in range(5):
    print(f"True: {y[i]:.3f}, Predicted: {pred[i][0]:.3f}")
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel("Epochs")
plt.ylabel("Loss (MSE)")
plt.legend()
plt.title("RNN Training Performance")
plt.show()
```

OUTPUT:

Epoch 1/20

/usr/local/lib/python3.12/dist-packages/keras/src/layers/rnn/rnn.py:199: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

super().__init__(**kwargs)

25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 17ms/step - loss: 0.3049 - val_loss: 0.0153

Epoch 2/20

25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 0.0076 - val_loss: 6.1550e-04

Epoch 3/20

25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 6.1812e-04 - val_loss: 4.2186e-05

Epoch 4/20

25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 7.8323e-05 - val_loss: 2.5385e-05

Epoch 5/20

25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 2.4766e-05 - val_loss: 1.4559e-05

Epoch 6/20

25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 1.3221e-05 - val_loss: 1.2748e-05

Epoch 7/20

25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 1.2374e-05 - val_loss: 1.1299e-05

Epoch 8/20

25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 9.4861e-06 - val_loss: 9.9077e-06

Epoch 9/20

25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 8.6906e-06 - val_loss: 7.1489e-06

Epoch 10/20

25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 6.9545e-06 - val_loss: 5.6677e-06

Epoch 11/20

25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 5.2897e-06 - val_loss: 5.1961e-06

Epoch 12/20

25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 4.9279e-06 - val_loss: 4.0503e-06

Epoch 13/20

25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 4.0215e-06 - val_loss: 3.7501e-06

Epoch 14/20

25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 3.2531e-06 - val_loss: 3.1022e-06

Epoch 15/20

25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 2.8326e-06 - val_loss: 2.0488e-06

Epoch 16/20

25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 1.8347e-06 - val_loss: 1.5976e-06

Epoch 17/20

25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 1.6108e-06 - val_loss: 1.5111e-06

Epoch 18/20

25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 1.1025e-06 - val_loss: 1.0988e-06

Epoch 19/20

25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 9.9462e-07 - val_loss: 8.3697e-07

Epoch 20/20

25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 7.3977e-07 - val_loss: 7.0010e-07

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 131ms/step

Sample Predictions:

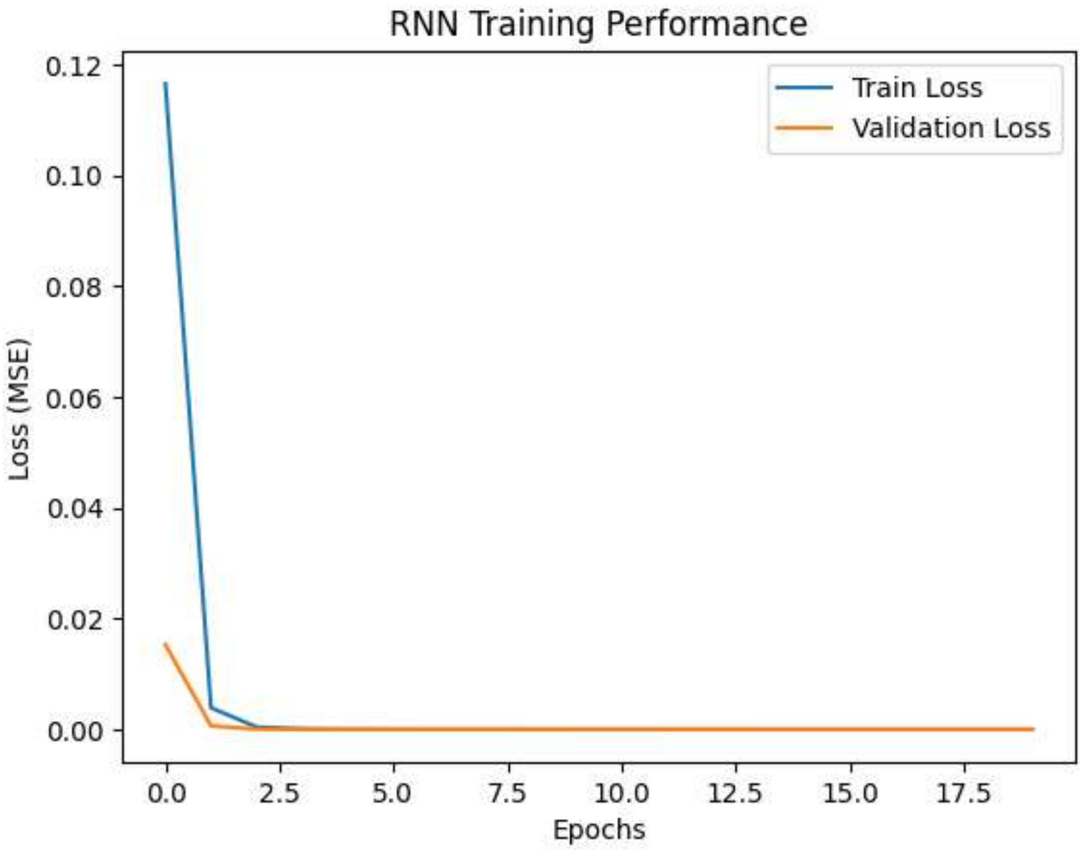
True: 0.391, Predicted: 0.391

True: 0.528, Predicted: 0.529

True: 0.432, Predicted: 0.433

True: -0.365, Predicted: -0.365

True: 0.599, Predicted: 0.600



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

RESULT:

The RNN was successfully implemented and trained on synthetic sine-wave data. It accurately predicted the next time-step values, and the training plot showed decreasing loss, demonstrating the effectiveness of the RNN architecture for time-series forecasting tasks.

EX. NO: 07

DATE :

TEXT ANALYSIS USING NATURAL LANGUAGE PROCESSING

AIM:

To perform end-to-end text analysis cleaning, feature extraction, and visualization on a sample text dataset using Natural Language Processing techniques.

ALGORITHM:

STEP 1: Install required libraries (nltk, scikit-learn, pandas, matplotlib, and wordcloud). Import pandas for data handling, re for regex, nltk for NLP utilities, TfidfVectorizer from sklearn for feature extraction, matplotlib.pyplot for plotting, and WordCloud for visualization.

STEP 2: Create or load a text dataset into a Pandas DataFrame for analysis.

STEP 3: Text preprocessing converts text to lowercase, removes punctuation and special characters, tokenizes into words, filters out stopwords, and rejoins the cleaned tokens into a processed string for analysis.

STEP 4: Combine all cleaned text and compute word frequencies using pandas.Series.value_counts() to identify the most frequent words.

STEP 4: Use TfidfVectorizer to transform the cleaned text into numerical features, capturing the importance of words in each document relative to the entire dataset. Extract and display top TF-IDF words per document.

STEP 5: Create a WordCloud from the cleaned text to visually highlight frequently occurring words in the dataset.

PROGRAM:

```
!pip install nltk scikit-learn pandas matplotlib wordcloud --quiet
import pandas as pd
import re
import nltk
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
import matplotlib.pyplot as plt
from wordcloud import WordCloud
nltk.download('stopwords')
data = {
    'text': [
        "The movie had stunning visuals but the plot was weak.",
    "Customer service was prompt and very helpful.",
    "I found the book to be quite boring and slow-paced.",
    "Excellent craftsmanship and attention to detail in this product.",
    "The app crashes every time I try to open it.",
    "Had an amazing dinner at the new Italian restaurant downtown.",
    "Terrible experience. I will not return.",
    "The software update improved performance significantly.",
    "Delivery was late and the package was damaged.",
    "Great user interface and very intuitive controls.",
    "Music quality is outstanding, especially the bass.",
    "Not impressed. Expected more for the price.",
    "Enjoyed the hiking trail — beautiful scenery and fresh air.",
    "Keyboard keys are too stiff and unresponsive.",
    "Friendly staff and a clean environment at the clinic.",
    "The laptop heats up quickly when gaming.",
    "Loved the plot twists in the final episodes!",
    "Battery life is shorter than advertised."
```

```
    ]}
df = pd.DataFrame(data)
print("Original Data:")
print(df)
stop_words = set(stopwords.words('english'))
def preprocess(text):
    text = text.lower()                    # lowercase
    text = re.sub(r'[^a-z\s]', '', text)   # remove punctuation
    tokens = text.split()                  # simple tokenization
    tokens = [word for word in tokens if word not in stop_words]  # remove stopwords
    return " ".join(tokens)
df['clean_text'] = df['text'].apply(preprocess)
print("\nCleaned Text:")
print(df[['text', 'clean_text']])
all_words = ' '.join(df['clean_text']).split()
word_freq = pd.Series(all_words).value_counts()
print("\nTop 10 Most Frequent Words:")
print(word_freq.head(10))
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(df['clean_text'])
feature_names = vectorizer.get_feature_names_out()
print("\nTop 5 TF-IDF Words per Document:")
for i, doc in enumerate(df['clean_text']):
    tfidf_scores = X[i].toarray()[0]
    top_indices = tfidf_scores.argsort()[-5:][::-1]
    top_words = [(feature_names[idx], tfidf_scores[idx]) for idx in top_indices if
tfidf_scores[idx] > 0]
    print(f"Document {i+1}: {top_words}")
text_combined = ' '.join(df['clean_text'])
wordcloud = WordCloud(width=800, height=400,
background_color='white').generate(text_combined)
```

```
plt.figure(figsize=(12,6))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title("WordCloud of All Documents", fontsize=16)
plt.show()
```

OUTPUT:

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
Original Data:
                                                text
0   The movie had stunning visuals but the plot wa...
1       Customer service was prompt and very helpful.
2   I found the book to be quite boring and slow-p...
3   Excellent craftsmanship and attention to detai...
4        The app crashes every time I try to open it.
5   Had an amazing dinner at the new Italian resta...
6            Terrible experience. I will not return.
7   The software update improved performance signi...
8      Delivery was late and the package was damaged.
9   Great user interface and very intuitive controls.
10  Music quality is outstanding, especially the b...
11        Not impressed. Expected more for the price.
12  Enjoyed the hiking trail — beautiful scenery a...
13      Keyboard keys are too stiff and unresponsive.
14  Friendly staff and a clean environment at the ...
15           The laptop heats up quickly when gaming.
16        Loved the plot twists in the final episodes!
17           Battery life is shorter than advertised.
```

Cleaned Text:

```
                                                text  \
0   The movie had stunning visuals but the plot wa...
1        Customer service was prompt and very helpful.
2   I found the book to be quite boring and slow-p...
3   Excellent craftsmanship and attention to detai...
4         The app crashes every time I try to open it.
5   Had an amazing dinner at the new Italian resta...
6              Terrible experience. I will not return.
7   The software update improved performance signi...
8     Delivery was late and the package was damaged.
9   Great user interface and very intuitive controls.
10  Music quality is outstanding, especially the b...
11        Not impressed. Expected more for the price.
12  Enjoyed the hiking trail — beautiful scenery a...
13        Keyboard keys are too stiff and unresponsive.
14  Friendly staff and a clean environment at the ...
15             The laptop heats up quickly when gaming.
16         Loved the plot twists in the final episodes!
17             Battery life is shorter than advertised.

                                         clean_text
0                    movie stunning visuals plot weak
1                     customer service prompt helpful
2                   found book quite boring slowpaced
3    excellent craftsmanship attention detail product
4                      app crashes every time try open
5     amazing dinner new italian restaurant downtown
6                           terrible experience return
```

```
7   software update improved performance significa...
8                      delivery late package damaged
9              great user interface intuitive controls
10          music quality outstanding especially bass
11                             impressed expected price
12   enjoyed hiking trail beautiful scenery fresh air
13                    keyboard keys stiff unresponsive
14             friendly staff clean environment clinic
15                          laptop heats quickly gaming
16                    loved plot twists final episodes
17                       battery life shorter advertised

Top 10 Most Frequent Words:
plot        2
movie       1
stunning    1
visuals     1
weak        1
customer    1
service     1
prompt      1
helpful     1
found       1
Name: count, dtype: int64

Top 5 TF-IDF Words per Document:
Document 1: [('weak', np.float64(0.4580541841950169)), ('visuals', np.float64(0.4580541841950169)), ('stunning', np.float64(0.4580541841950169)), ('movie', np.float64(0.4580541841950169)), ('plot', np.float64(0.4009307388636471))]
Document 2: [('service', np.float64(0.5)), ('customer', np.float64(0.5)), ('helpful', np.float64(0.5)), ('prompt', np.float64(0.5))]
```

Document 3: [('slowpaced', np.float64(0.4472135954999579)), ('boring', np.float64(0.4472135954999579)), ('book', np.float64(0.4472135954999579)), ('quite', np.float64(0.4472135954999579)), ('found', np.float64(0.4472135954999579))]

Document 4: [('excellent', np.float64(0.4472135954999579)), ('detail', np.float64(0.4472135954999579)), ('attention', np.float64(0.4472135954999579)), ('product', np.float64(0.4472135954999579)), ('craftsmanship', np.float64(0.4472135954999579))]

Document 5: [('time', np.float64(0.408248290463863)), ('try', np.float64(0.408248290463863)), ('app', np.float64(0.408248290463863)), ('open', np.float64(0.408248290463863)), ('every', np.float64(0.408248290463863))]

Document 6: [('dinner', np.float64(0.408248290463863)), ('italian', np.float64(0.408248290463863)), ('new', np.float64(0.408248290463863)), ('restaurant', np.float64(0.408248290463863)), ('amazing', np.float64(0.408248290463863))]

Document 7: [('return', np.float64(0.5773502691896258)), ('terrible', np.float64(0.5773502691896258)), ('experience', np.float64(0.5773502691896258))]

Document 8: [('update', np.float64(0.4472135954999579)), ('significantly', np.float64(0.4472135954999579)), ('software', np.float64(0.4472135954999579)), ('performance', np.float64(0.4472135954999579)), ('improved', np.float64(0.4472135954999579))]

Document 9: [('damaged', np.float64(0.5)), ('delivery', np.float64(0.5)), ('late', np.float64(0.5)), ('package', np.float64(0.5))]

Document 10: [('user', np.float64(0.4472135954999579)), ('intuitive', np.float64(0.4472135954999579)), ('great', np.float64(0.4472135954999579)), ('interface', np.float64(0.4472135954999579)), ('controls', np.float64(0.4472135954999579))]

Document 11: [('especially', np.float64(0.4472135954999579)), ('music', np.float64(0.4472135954999579)), ('outstanding', np.float64(0.4472135954999579)), ('quality', np.float64(0.4472135954999579)), ('bass', np.float64(0.4472135954999579))]

Document 12: [('impressed', np.float64(0.5773502691896258)), ('expected', np.float64(0.5773502691896258)), ('price', np.float64(0.5773502691896258))]

Document 13: [('scenery', np.float64(0.3779644730092272)), ('trail', np.float64(0.3779644730092272)), ('beautiful', np.float64(0.3779644730092272)), ('hiking', np.float64(0.3779644730092272)), ('fresh', np.float64(0.3779644730092272))]

Document 14: [('stiff', np.float64(0.5)), ('unresponsive', np.float64(0.5)), ('keys', np.float64(0.5)), ('keyboard', np.float64(0.5))]

Document 15: [('staff', np.float64(0.4472135954999579)), ('clean', np.float64(0.4472135954999579)), ('clinic', np.float64(0.4472135954999579)), ('friendly', np.float64(0.4472135954999579)), ('environment', np.float64(0.4472135954999579))]

Document 16: [('laptop', np.float64(0.5)), ('gaming', np.float64(0.5)), ('heats', np.float64(0.5)), ('quickly', np.float64(0.5))]

Document 17: [('twists', np.float64(0.4580541841950169)), ('episodes', np.float64(0.4580541841950169)), ('final', np.float64(0.4580541841950169)), ('loved', np.float64(0.4580541841950169)), ('plot', np.float64(0.400930738863647))]

Document 18: [('shorter', np.float64(0.5)), ('advertised', np.float64(0.5)), ('battery', np.float64(0.5)), ('life', np.float64(0.5))]

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

RESULT:

The NLP pipeline successfully cleaned and normalized the raw text, identified the most frequent terms along with top TF-IDF features for each document, and generated a WordCloud that visually highlights the most significant words across the dataset.

EX.NO:08

DATE:

AI GENERATOR IMAGE WITH DEEP DREAM AND NEURAL STYLE TRANSFER

AIM:

To Construct Experiment with Al generator such as Deep Dream and Neural Style Transfer.

ALGORITHM:

STEP 1: Start the process.

STEP 2: Import required libraries: tensorflow for model operations, tensorflow_hub for style transfer, numpy for array handling, PIL.Image for image loading, and matplotlib.pyplot for visualization.

STEP 3: load_image(path, max_dim) → load an image, convert to RGB, resize, scale pixels to [0,1], and return a batched tensor. show_image(img, title) → display a batched tensor as an image with a title.

STEP 4: Load a pretrained convolutional model (e.g., InceptionV3 with include_top=False) for Deep Dream. Select layers (mixed3, mixed5) to maximize activations.

STEP 5: Create a Deep Dream function that:

- Computes the mean activation of selected layers as a loss.
- Uses GradientTape to compute gradients of loss w.r.t. the image.
- Normalizes the gradients and updates the image using step_size.
- Clips image values to [0,1] for valid pixels.

STEP 6: Load the Neural Style Transfer model from TF-Hub (arbitrary-image-stylization-v1-256) and define a function to stylize a content image with a style image.

STEP 7: Load content and style images using load_image() with appropriate max values.

STEP 8: Apply Deep Dream on the content image to generate a "dreamed" image and visualize it with show_image().

STEP 9: Apply Neural Style Transfer on:

- Original content image → stylized original image
- Dreamed content image → combined stylized image

STEP 10: Display all results using show_image() to visualize: Deep Dream result, Neural Style Transfer result, and Deep Dream + Style Transfer combined result.

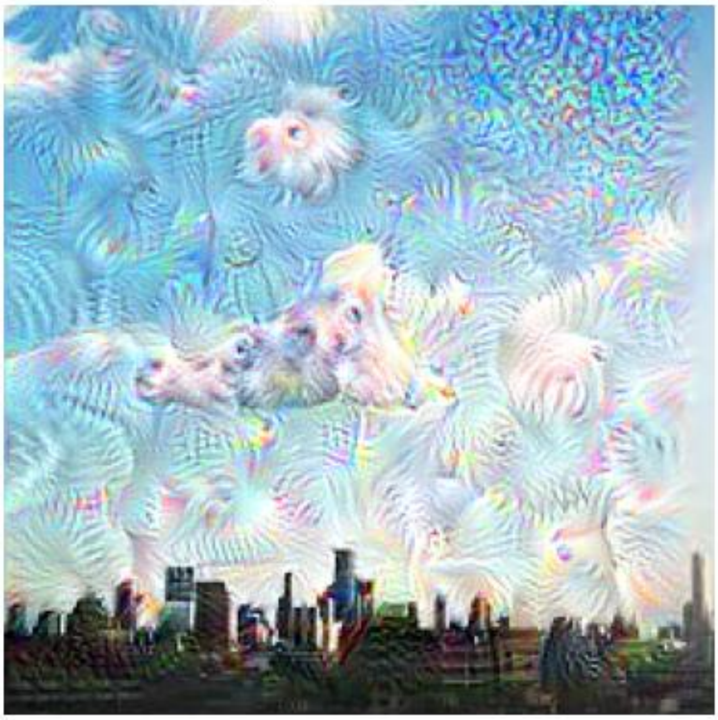
STEP 12: Stop the Process.

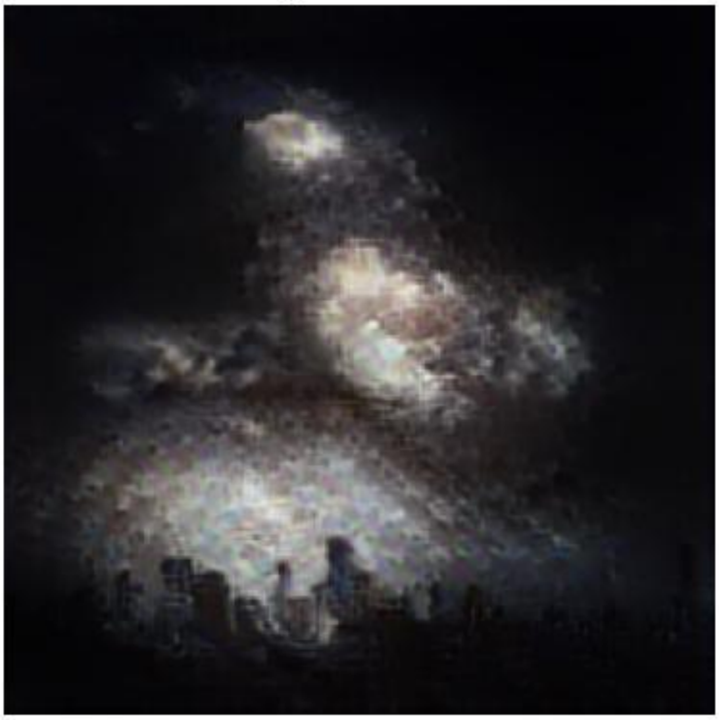
CODING:

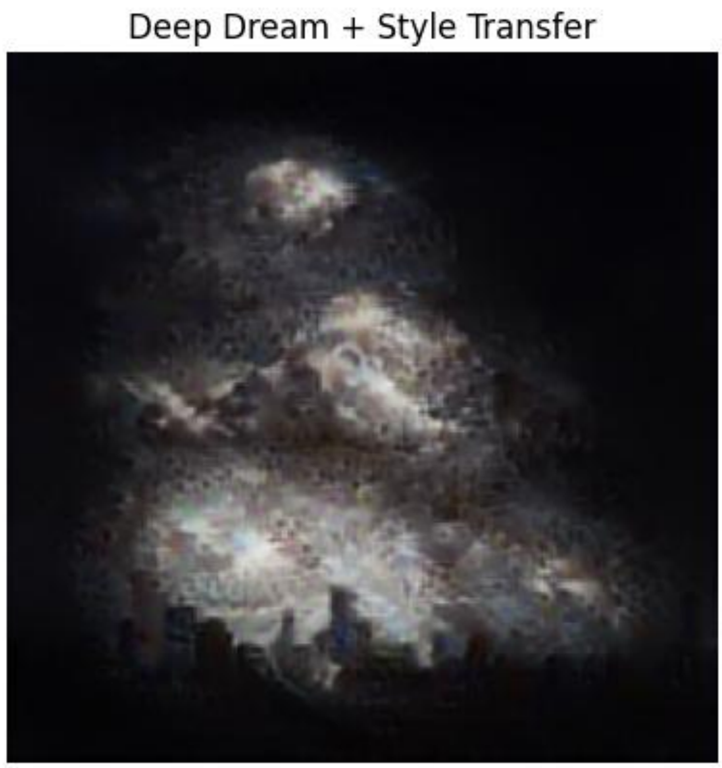
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import tensorflow_hub as hub
# Utility Functions
def load_image(path, max_dim=512):
    img = Image.open(path)
    img = img.convert("RGB")
    img.thumbnail((max_dim, max_dim))
    img = np.array(img) / 255.0
    img = tf.convert_to_tensor(img, dtype=tf.float32)
    return img[tf.newaxis, :]
def show_image(img, title=""):
    if len(img.shape) == 4:
        img = img[0]
    plt.imshow(np.clip(img, 0, 1))
    plt.axis("off")
    plt.title(title)
    plt.show()
# Deep Dream Implementation
def deepdream(image, model, steps=100, step_size=0.01):
    image = tf.convert_to_tensor(image)
    for step in range(steps):
        with tf.GradientTape() as tape:
            tape.watch(image)
            loss = tf.reduce_mean(model(image))
        grads = tape.gradient(loss, image)
```

```
        grads = grads / (tf.math.reduce_std(grads) + 1e-8)
        image = image + grads * step_size
        image = tf.clip_by_value(image, 0.0, 1.0)
    return image
# Load InceptionV3 for Deep Dream
base_model = tf.keras.applications.InceptionV3(include_top=False, weights="imagenet")
dream_layers = ['mixed3', 'mixed5']
dream_model = tf.keras.Model(
    inputs=base_model.input,
    outputs=[base_model.get_layer(name).output for name in dream_layers]
)
# Neural Style Transfer
style_transfer_model = hub.load("https://tfhub.dev/google/magenta/arbitrary-image-stylization-v1-256/2")
def neural_style_transfer(content_img, style_img):
    stylized_image = style_transfer_model(tf.constant(content_img), tf.constant(style_img))[0]
    return stylized_image
content_path = "content.jpeg"  # Your content image
style_path = "stly.jpeg"       # Your style image
# Load images
content_image = load_image(content_path)
style_image = load_image(style_path, max_dim=256)
dreamed_image = deepdream(content_image, dream_model, steps=50, step_size=0.01)
show_image(dreamed_image, "Deep Dream Result")
stylized_image = neural_style_transfer(content_image, style_image)
show_image(stylized_image, "Neural Style Transfer Result")
combined = neural_style_transfer(dreamed_image, style_image)
show_image(combined, "Deep Dream + Style Transfer")
```

OUTPUT:


Deep Dream Result


Neural Style Transfer Result



COE(30)	
RECORD(20)	
VIVA(10)	
TOTAL	

RESULT:

Thus, the above program has been successfully verified and executed.

EX.NO:09

DATE:

SYNTHETIC IMAGES USING VARIATIONAL AUTO ENCODERS

AIM:

To Generate synthetic images using variational auto encoders.

ALGORITHM:

STEP 1: Start the process.

STEP 2: Import required libraries: tensorflow for deep learning, numpy for array handling, matplotlib.pyplot for visualization, and Keras layers and Model for building the encoder, decoder, and VAE model.

STEP 3: Load and preprocess the MNIST dataset:

- Combine training and test sets.
- Normalize pixel values to [0,1].
- Expand dimensions to add a channel axis (28,28,1) for grayscale images.

STEP 4: Build the Encoder network:

- Input layer with shape (28,28,1).
- Flatten the image and pass through a Dense layer with 128 neurons and ReLU activation.
- Output two layers: z_mean and z_log_var (for latent space).
- Apply the reparameterization trick via a Lambda layer to sample z from the latent space.

STEP 5: Build the Decoder network:

- Input is the latent vector z of size latent_dim.
- Dense layer with 128 neurons and ReLU activation.
- Dense layer to reconstruct the image (28*28) with sigmoid activation.
- Reshape the output back to (28,28,1).

STEP 6: Create the VAE model class:

- Use custom train_step to compute combined loss:
    - Reconstruction loss: binary cross-entropy between input and output.
    - KL Divergence: regularizes latent space to approximate a standard normal distribution.

- Apply gradients and update weights using Adam optimizer.

STEP 7: Compile and train the VAE:

- Use vae.compile(optimizer="adam").
- Train with vae.fit(x, epochs=10, batch_size=128).

STEP 8: Generate synthetic images using the trained decoder:

- Sample random vectors z from a standard normal distribution.
- Pass sampled vectors through the decoder to generate new images.

STEP 9: Visualize generated images in a grid using matplotlib.pyplot

- Arrange images in n x n subplots.
- Disable axes and use grayscale colormap for proper visualization.

STEP 10: Experiment by changing latent_dim, number of epochs, or batch size to generate different styles or higher-quality synthetic images.

STEP 12: Stop the Process.

CODING:

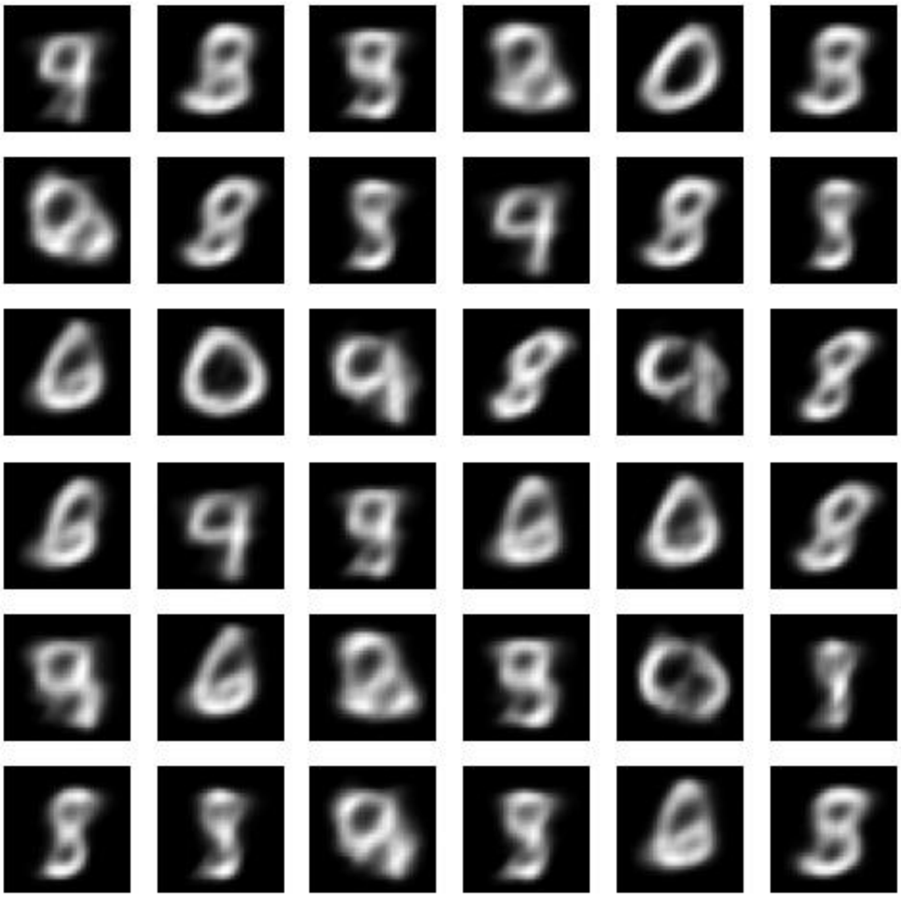
```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import layers, Model
# Load MNIST
(x_train, _), (x_test, _) = tf.keras.datasets.mnist.load_data()
x = np.concatenate([x_train, x_test], axis=0).astype("float32") / 255.0
x = np.expand_dims(x, -1)   # shape (70000, 28, 28, 1)
latent_dim = 2   # small latent space to visualize
#  Encoder
inputs = layers.Input(shape=(28,28,1))
x_enc = layers.Flatten()(inputs)
x_enc = layers.Dense(128, activation="relu")(x_enc)
z_mean = layers.Dense(latent_dim)(x_enc)
z_log_var = layers.Dense(latent_dim)(x_enc)
```

```
# Reparameterization trick
def sampling(args):
    z_mean, z_log_var = args
    eps = tf.random.normal(shape=tf.shape(z_mean))
    return z_mean + tf.exp(0.5 * z_log_var) * eps
z = layers.Lambda(sampling)([z_mean, z_log_var])
encoder = Model(inputs, [z_mean, z_log_var, z])
#  Decoder
latent_inputs = layers.Input(shape=(latent_dim,))
x_dec = layers.Dense(128, activation="relu")(latent_inputs)
x_dec = layers.Dense(28*28, activation="sigmoid")(x_dec)
outputs = layers.Reshape((28,28,1))(x_dec)
decoder = Model(latent_inputs, outputs)
# VAE Model
class VAE(Model):
    def __init__(self, encoder, decoder):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
    def train_step(self, data):
        if isinstance(data, tuple): data = data[0]
        with tf.GradientTape() as tape:
            z_mean, z_log_var, z = self.encoder(data)
            recon = self.decoder(z)
            # Reconstruction loss
            recon_loss = tf.reduce_mean(
                tf.keras.losses.binary_crossentropy(data, recon)
            ) * 28 * 28
            # KL Divergence
            kl_loss = -0.5 * tf.reduce_mean(1 + z_log_var - tf.square(z_mean) -
tf.exp(z_log_var))
```

```
            loss = recon_loss + kl_loss
        grads = tape.gradient(loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
        return {"loss": loss, "recon_loss": recon_loss, "kl_loss": kl_loss}
vae = VAE(encoder, decoder)
vae.compile(optimizer="adam")
vae.fit(x, epochs=10, batch_size=128)
# Generate Synthetic Images
def show_generated(n=10):
    z_samples = np.random.normal(size=(n*n, latent_dim))
    imgs = decoder.predict(z_samples)
    plt.figure(figsize=(n, n))
    for i in range(n*n):
        plt.subplot(n, n, i+1)
        plt.imshow(imgs[i].squeeze(), cmap="gray")
        plt.axis("off")
    plt.show()
show_generated(6)   # show 6x6 = 36 synthetic images
```

OUTPUT:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Epoch 1/10
547/547 ━━━━━━━━━━━━━━━━━━━━ 11s 12ms/step - kl_loss: 7.7264 - loss: 204.8223 - recon_loss: 197.0959
Epoch 2/10
547/547 ━━━━━━━━━━━━━━━━━━━━ 7s 12ms/step - kl_loss: 3.1713 - loss: 171.0090 - recon_loss: 167.8378
Epoch 3/10
547/547 ━━━━━━━━━━━━━━━━━━━━ 6s 11ms/step - kl_loss: 3.1720 - loss: 165.2389 - recon_loss: 162.0669
Epoch 4/10
547/547 ━━━━━━━━━━━━━━━━━━━━ 7s 13ms/step - kl_loss: 3.2109 - loss: 162.9484 - recon_loss: 159.7376
Epoch 5/10
547/547 ━━━━━━━━━━━━━━━━━━━━ 6s 11ms/step - kl_loss: 3.2174 - loss: 161.3449 - recon_loss: 158.1276
Epoch 6/10
547/547 ━━━━━━━━━━━━━━━━━━━━ 7s 13ms/step - kl_loss: 3.2323 - loss: 160.1319 - recon_loss: 156.8997
Epoch 7/10
547/547 ━━━━━━━━━━━━━━━━━━━━ 12s 16ms/step - kl_loss: 3.2380 - loss: 159.1310 - recon_loss: 155.8930
Epoch 8/10
547/547 ━━━━━━━━━━━━━━━━━━━━ 6s 11ms/step - kl_loss: 3.2543 - loss: 158.2969 - recon_loss: 155.0426
Epoch 9/10
547/547 ━━━━━━━━━━━━━━━━━━━━ 7s 14ms/step - kl_loss: 3.2705 - loss: 157.5142 - recon_loss: 154.2437
Epoch 10/10
547/547 ━━━━━━━━━━━━━━━━━━━━ 6s 11ms/step - kl_loss: 3.2962 - loss: 156.7552 - recon_loss: 153.4590
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step
```



COE(30)	
RECORD(20)	
VIVA(10)	
TOTAL	

RESULT:

Thus, the above program has been successfully verified and executed.

EX.NO:10

DATE:

SYNTHETIC IMAGES USING GENERATIVE ADVERSARIAL NETWORK

AIM:

To Generate synthetic images using Generative Adversarial Network.

ALGORITHM:

STEP 1: Start the process.

STEP 2 Import required libraries: tensorflow for building GAN, numpy for array operations, matplotlib.pyplot for visualization, and Keras layers for generator and discriminator networks.

STEP 3: Load and preprocess the MNIST dataset: Normalize pixel values to [-1,1] using (X-127.5)/127.5. Expand dimensions to (28,28,1) for grayscale images. Convert to TensorFlow Dataset, shuffle, and batch it.

STEP 4: Build the Generator network: Input is a noise vector of size LATENT_DIM. Dense layer with 128 neurons and ReLU activation. Dense layer to output 28×28 pixels with tanh activation. Reshape output to (28,28,1) to represent an image.

STEP 5: Build the Discriminator network:

- Flatten input image (28,28,1).
- Dense layer with 128 neurons and ReLU activation.
- Output layer with 1 neuron and sigmoid activation to classify real/fake images.

STEP 6: Define loss functions and optimizers:

- Use BinaryCrossentropy for both generator and discriminator.
- Use Adam optimizer with learning rate 1e-4.

STEP 7: Implement a training step function: Sample random noise for the generator. Generate fake images from noise. Compute discriminator outputs for real and fake images. Compute generator loss (how well fake images fool discriminator). Compute discriminator loss (real vs fake classification). Apply gradients to update generator and discriminator weights.

STEP 8: Implement a training loop:

- Iterate over epochs and batches.
- Call train_step() on each batch.
- Print generator and discriminator losses per epoch.

- Optionally generate sample images using a fixed seed to monitor progress.

STEP 9: Create a function to generate and plot images: Generate images from a batch of noise vectors. Scale pixel values from [-1,1] to [0,1]. Plot images in a grid with no axes for visualization.

STEP 10: Run the GAN training with a defined number of epochs (EPOCHS) and visualize generated images at each epoch to monitor learning.

STEP 12: Stop the Process.

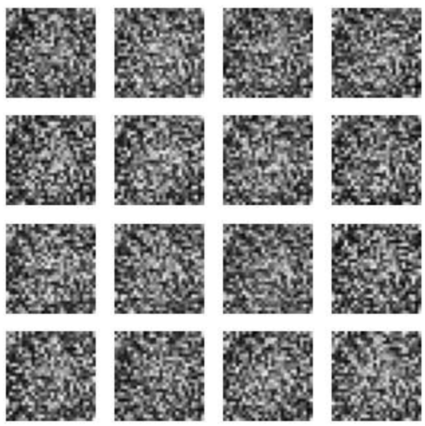
CODING:

```
import tensorflow as tf
from tensorflow.keras import layers
import numpy as np
import matplotlib.pyplot as plt
#  Load and preprocess MNIST
(X_train, _), (_, _) = tf.keras.datasets.mnist.load_data()
X_train = (X_train.astype("float32") - 127.5) / 127.5   # normalize to [-1, 1]
X_train = np.expand_dims(X_train, axis=-1)             # (60000, 28, 28, 1)
BUFFER_SIZE = 60000
BATCH_SIZE = 128
LATENT_DIM = 100   # size of noise vector
dataset =
tf.data.Dataset.from_tensor_slices(X_train).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
# Generator
def build_generator():
   model = tf.keras.Sequential([
      layers.Dense(128, activation="relu", input_shape=(LATENT_DIM,)),
      layers.Dense(28*28, activation="tanh"),
      layers.Reshape((28, 28, 1))
   ])
   return model
#  Discriminator
```

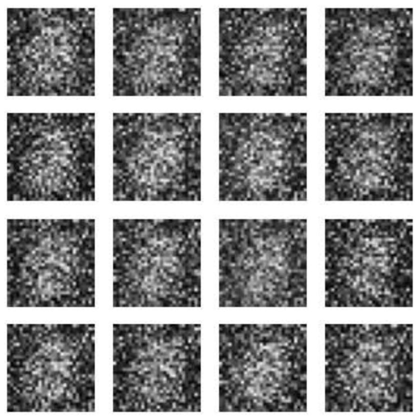
```
def build_discriminator():
    model = tf.keras.Sequential([
        layers.Flatten(input_shape=(28,28,1)),
        layers.Dense(128, activation="relu"),
        layers.Dense(1, activation="sigmoid")
    ])
    return model
generator = build_generator()
discriminator = build_discriminator()
#Loss & Optimizers
cross_entropy = tf.keras.losses.BinaryCrossentropy()
g_optimizer = tf.keras.optimizers.Adam(1e-4)
d_optimizer = tf.keras.optimizers.Adam(1e-4)
#  Training Step
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, LATENT_DIM])
    with tf.GradientTape() as g_tape, tf.GradientTape() as d_tape:
        generated = generator(noise, training=True)
        real_out = discriminator(images, training=True)
        fake_out = discriminator(generated, training=True)
        g_loss = cross_entropy(tf.ones_like(fake_out), fake_out)
        d_loss = (cross_entropy(tf.ones_like(real_out), real_out) +
                  cross_entropy(tf.zeros_like(fake_out), fake_out)) / 2
    grads_g = g_tape.gradient(g_loss, generator.trainable_variables)
    grads_d = d_tape.gradient(d_loss, discriminator.trainable_variables)
    g_optimizer.apply_gradients(zip(grads_g, generator.trainable_variables))
    d_optimizer.apply_gradients(zip(grads_d, discriminator.trainable_variables))
    return g_loss, d_loss
# Training Loop
```

```
EPOCHS = 10
seed = tf.random.normal([16, LATENT_DIM])  # for monitoring progress
def train(dataset, epochs):
    for epoch in range(epochs):
        for image_batch in dataset:
            g_loss, d_loss = train_step(image_batch)
        print(f"Epoch {epoch+1}/{epochs} | Gen Loss: {g_loss:.4f} | Disc Loss: {d_loss:.4f}")
        generate_and_plot(generator, seed)
def generate_and_plot(model, test_input):
    preds = model(test_input, training=False)
    preds = (preds + 1) / 2.0   # back to [0,1]
    plt.figure(figsize=(4,4))
    for i in range(preds.shape[0]):
        plt.subplot(4, 4, i+1)
        plt.imshow(preds[i,:,:,0], cmap="gray")
        plt.axis("off")
    plt.show()
train(dataset, EPOCHS)
```

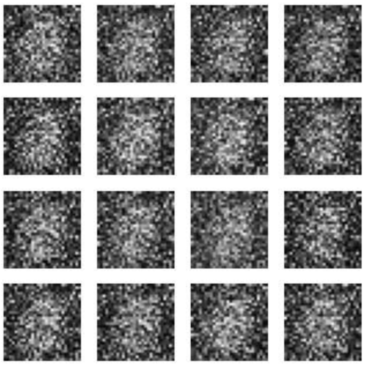
OUTPUT:



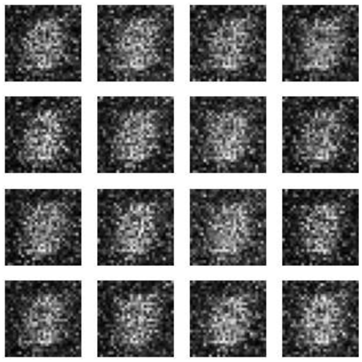
Epoch 1/10 | Gen Loss: 1.1682 | Disc Loss: 0.2248



Epoch 2/10 | Gen Loss: 1.3248 | Disc Loss: 0.2212



Epoch 3/10 | Gen Loss: 1.2188 | Disc Loss: 0.2738



Epoch 4/10 | Gen Loss: 1.0084 | Disc Loss: 0.3873

COE(30)	
RECORD(20)	
VIVA(10)	
TOTAL	

RESULT:

Thus, the above program has been successfully verified and executed.