Ex.no: 1
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

```
import numpy as np

def step_function(value):
    """
    Activation function for the perceptron.
    Returns 1 if the value is non-negative, otherwise returns 0.
```

```
    """
    return 1 if value >= 0 else 0

class Perceptron:
    """
    A simple single-layer perceptron implementation.
    """
    def __init__(self, input_size, learning_rate=0.1):
        """
        Initializes the perceptron with weights and learning rate.
        """
        self.weights = np.zeros(input_size + 1)  # Including bias
        self.learning_rate = learning_rate

    def predict(self, inputs):
        """
        Makes a prediction based on the current weights and inputs.
        """
        inputs_with_bias = np.insert(inputs, 0, 1)  # Add bias input
        total = np.dot(self.weights, inputs_with_bias)
        return step_function(total)

    def train(self, X, y, epochs=10):
        """
        Trains the perceptron on a given dataset.
        """
        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]}, UpdatedWeights: {self.weights}")

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])
```

```
# Step 2: Create perceptron and train it
perceptron = Perceptron(input_size=2)
perceptron.train(X, y, epochs=10)

# 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, UpdatedWeights: [-0.1  0.   0. ]
 Input: [0 1], Predicted: 0, Actual: 1, UpdatedWeights: [0.  0.  0.1]
 Input: [1 0], Predicted: 1, Actual: 1, UpdatedWeights: [0.  0.  0.1]
 Input: [1 1], Predicted: 1, Actual: 1, UpdatedWeights: [0.  0.  0.1]
Epoch 2
 Input: [0 0], Predicted: 1, Actual: 0, UpdatedWeights: [-0.1  0.   0.1]
 Input: [0 1], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.   0.1]
 Input: [1 0], Predicted: 0, Actual: 1, UpdatedWeights: [0.  0.1 0.1]
 Input: [1 1], Predicted: 1, Actual: 1, UpdatedWeights: [0.  0.1 0.1]
Epoch 3
 Input: [0 0], Predicted: 1, Actual: 0, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [0 1], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [1 0], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [1 1], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
Epoch 4
 Input: [0 0], Predicted: 0, Actual: 0, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [0 1], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [1 0], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [1 1], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
Epoch 5
 Input: [0 0], Predicted: 0, Actual: 0, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [0 1], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [1 0], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [1 1], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
Epoch 6
 Input: [0 0], Predicted: 0, Actual: 0, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [0 1], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [1 0], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [1 1], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
Epoch 7
 Input: [0 0], Predicted: 0, Actual: 0, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [0 1], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [1 0], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [1 1], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
Epoch 8
 Input: [0 0], Predicted: 0, Actual: 0, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [0 1], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [1 0], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [1 1], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
Epoch 9
 Input: [0 0], Predicted: 0, Actual: 0, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [0 1], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [1 0], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [1 1], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
Epoch 10
 Input: [0 0], Predicted: 0, Actual: 0, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [0 1], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [1 0], Predicted: 1, Actual: 1, UpdatedWeights: [-0.1  0.1  0.1]
 Input: [1 1], Predicted: 1, Actual: 1, UpdatedWeights: [-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: 2
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.

Program

```
# Step 0: Install Keras Tuner if you haven't already
!pip install keras-tuner -q

import os
import kagglehub
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
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
import keras_tuner as kt

# Step 1: Load and preprocess your dataset (This part remains the same)
path =
kagglehub.dataset_download("vjchoudhary7/customer-segmentation-tutorial-
in-python")
full_path = path + '/' + os.listdir(path)[0]
df = pd.read_csv(full_path)

df['target'] = (df['Spending Score (1-100)'] >= 70).astype(int)
df.drop(columns=['CustomerID', 'Spending Score (1-100)'], inplace=True)

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)

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

X = df.drop(columns='target')
y = df['target']
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)

# Step 2: Create a model-building function (Remains the same)
def build_model(hp):
    model = Sequential()
    hp_units = hp.Int('units', min_value=8, max_value=64, step=8)
    model.add(Dense(units=hp_units, activation='relu',
input_shape=(X_train.shape[1],)))
    model.add(Dense(16, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    hp_learning_rate = hp.Choice('learning_rate', values=[0.001, 0.0005,
0.0001])
```

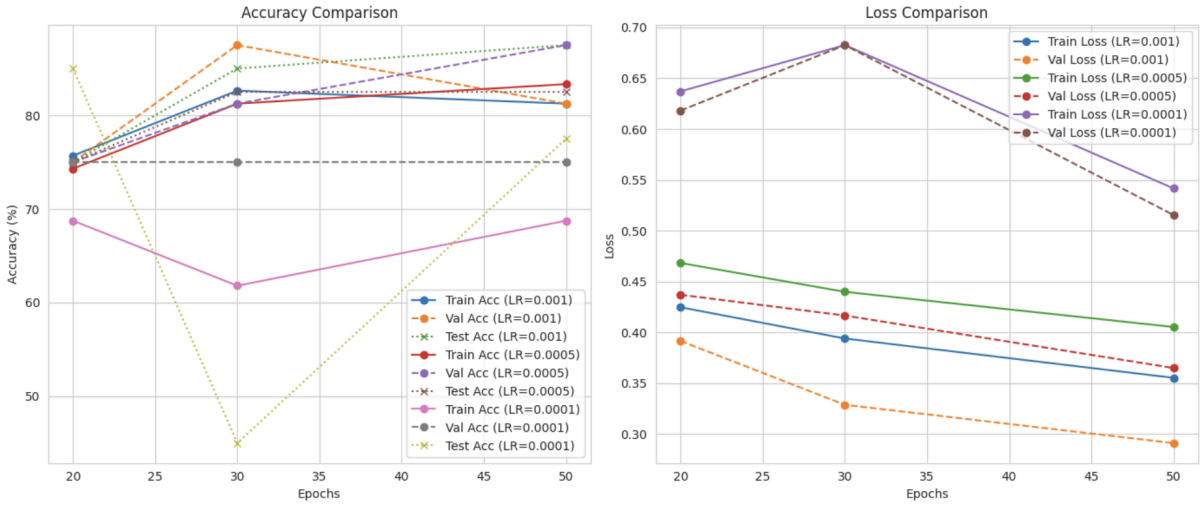
```
    model.compile(optimizer=Adam(learning_rate=hp_learning_rate),
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
    return model

# Step 3: Instantiate the tuner (Remains the same)
tuner = kt.Hyperband(
    build_model,
    objective='val_accuracy',
    max_epochs=50,
    factor=3,
    directory='my_dir',
    project_name='customer_segmentation'
)

# Step 4: Run the hyperparameter search (Remains the same)
print("Starting hyperparameter search...")
tuner.search(X_train, y_train, epochs=50, validation_split=0.2,
batch_size=8, verbose=0)
print("\nSearch complete!")

# Step 5: Display results and get the best model (Remains the same)
print("\n--- Hyperparameter Search Results ---")
tuner.results_summary()
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
best_model = tuner.get_best_models(num_models=1)[0]
print("\n--- Evaluating the Best Model on Test Data ---")
test_loss, test_acc = best_model.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {test_acc*100:.2f}%")
print(f"Test Loss: {test_loss:.4f}")


# Step 6: NEW - Extract and process trial data for plotting
print("\n--- Preparing data for visualization ---")
# Get all the trials that were run
trials =
tuner.oracle.get_best_trials(num_trials=tuner.oracle.max_trials)

results_list = []
for trial in trials:
    hps = trial.hyperparameters.values
    # The 'score' is the final validation accuracy for that trial
    results_list.append({
        'learning_rate': hps['learning_rate'],
        'units': hps['units'],
        'score': trial.score
```

```
    })

# Create a DataFrame from the results
df_tuner_results = pd.DataFrame(results_list)
print("Tuner Results DataFrame:")
print(df_tuner_results.head())


# Step 7: NEW - Visualize the tuner results
print("\n--- Generating plots ---")
plt.figure(figsize=(14, 6))
sns.set_style("whitegrid")

# Plot 1: Score vs. Number of Units, colored by Learning Rate
plt.subplot(1, 2, 1)
sns.scatterplot(
    data=df_tuner_results,
    x='units',
    y='score',
    hue='learning_rate',
    palette='viridis', # Use a nice color map
    s=150, # Make points bigger
    alpha=0.8
)
plt.xlabel("Number of Units (1st Layer)")
plt.ylabel("Validation Accuracy")
plt.title("Tuner Performance: Units vs. Accuracy")
plt.legend(title='Learning Rate')

# Plot 2: Score distribution for each Learning Rate
plt.subplot(1, 2, 2)
sns.boxplot(
    data=df_tuner_results,
    x='learning_rate',
    y='score'
)
plt.xlabel("Learning Rate")
plt.ylabel("Validation Accuracy")
plt.title("Tuner Performance: Score by Learning Rate")

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

Output

```
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	75.694442	75.00	75.000000	0.424661	0.391552
1	0.0010	30	82.638890	87.50	85.000002	0.393910	0.328659
2	0.0010	50	81.250000	81.25	87.500000	0.355217	0.290972
3	0.0005	20	74.305558	75.00	75.000000	0.468240	0.436946
4	0.0005	30	81.250000	81.25	82.499999	0.439860	0.416503
5	0.0005	50	83.333331	87.50	82.499999	0.405251	0.364878
6	0.0001	20	68.750000	75.00	85.000002	0.637119	0.618194
7	0.0001	30	61.805558	75.00	44.999999	0.682632	0.682594
8	0.0001	50	68.750000	75.00	77.499998	0.541597	0.515432



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: 3
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 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.

Program

```
# Step 1: Mount Google Drive
from google.colab import drive
import os

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/Dataset/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)
x = x.reshape((1,) + x.shape)

# 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

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

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: 4
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 usingimage.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.

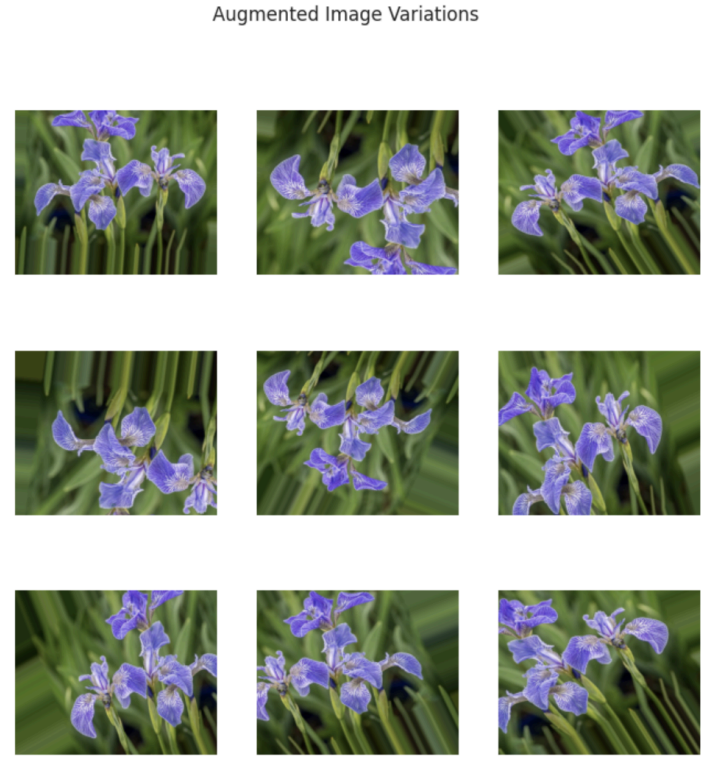
Program

```
# Step 1: Mount Google Drive
from google.colab import drive

drive.mount("/content/drive")

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

# Step 3: Set dataset path
dataset_path = "/content/drive/MyDrive/Dataset/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
sample_img_name = os.listdir(dataset_path)[2]  # First image in folder
img_path = os.path.join(dataset_path, sample_img_name)

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

# 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.suptitle("Augmented Image Variations")
plt.show()
```

Output

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```



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 usingtheImageDataGenerator class.

Ex.no: 5
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: Import the Cats and Dogs Classification Dataset from kaggle

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.

Program

```
import kagglehub
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import os

# --- 1. DATA PREPARATION ---

# Download the dataset from Kaggle Hub and get the local path
path =
kagglehub.dataset_download("bhavikjikadara/dog-and-cat-classification-da
taset")

data_path = os.path.join(path, os.listdir(path)[0])

# --- 2. DATA AUGMENTATION AND GENERATORS ---

train_datagen = ImageDataGenerator(
    rescale=1./255,            # Rescale pixel values from [0, 255] to
[0, 1] for better model performance.
    rotation_range=40,         # Randomly rotate images by up to 40
degrees.
    width_shift_range=0.2,     # Randomly shift images horizontally.
    height_shift_range=0.2,    # Randomly shift images vertically.
    shear_range=0.2,           # Apply shearing transformations
(tilting).
    zoom_range=0.3,            # Randomly zoom in on images.
    horizontal_flip=True,      # Randomly flip images horizontally.
    vertical_flip=True,        # Randomly flip images vertically.
_mode='nearest',    # Strategy to fill in new pixels that may appear
after a transformation.
    validation_split=0.2       # Reserve 20% of the images for
validation.
)

# Create a generator for the training data.
# It reads images from the specified directory, applies augmentations,
```

```
and provides them in batches.
train_data = train_datagen.flow_from_directory(
    data_path,                 # Path to the dataset directory.
    target_size=(64, 64),      # Resize all images to 64x64 pixels.
    batch_size=32,             # Load images in batches of 32.
    class_mode='categorical',  # Use categorical labels (one-hot
encoded) since we have multiple classes.
    subset='training'          # Specify that this is the training set
(uses 80% of the data).
)

# Create a generator for the validation data.
# It uses the same augmentation object but pulls from the validation
split.
# Note: Augmentations like flips and rotations are also applied to
validation data here,
# which is generally fine, but for pure evaluation, you might create a
separate generator with only rescaling.
val_data = train_datagen.flow_from_directory(
    data_path,                 # Path to the dataset directory.
    target_size=(64, 64),      # Resize all images to 64x64 pixels.
    batch_size=32,             # Load images in batches of 32.
    class_mode='categorical',  # Use categorical labels.
    subset='validation'        # Specify that this is the validation set
(uses 20% of the data).
)

# --- 3. CNN MODEL ARCHITECTURE ---

# Define the Convolutional Neural Network (CNN) model using Keras's
Sequential API.
model = tf.keras.models.Sequential([
    # First convolutional block: Extracts basic features like edges and
textures.
    # Conv2D: 32 filters, 3x3 kernel size, ReLU activation function.
    # input_shape must be specified for the first layer.
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu',
input_shape=(64, 64, 3)),
    tf.keras.layers.MaxPooling2D(2, 2), # Downsamples the feature map to
reduce dimensions.
    tf.keras.layers.Dropout(0.25),      # Regularization technique to
prevent overfitting by randomly dropping 25% of neurons.

    # Second convolutional block: Extracts more complex features from
the previous layer's output.
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
```

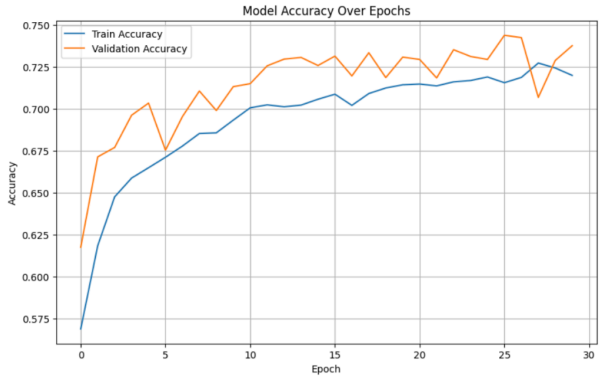
```
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Dropout(0.25),

    # Flattening and Dense layers for classification.
    tf.keras.layers.Flatten(), # Flattens the 2D feature maps into a 1D
vector to feed into the dense layers.
    tf.keras.layers.Dense(128, activation='relu'), # A fully connected
layer with 128 neurons.
    tf.keras.layers.Dropout(0.5), # A higher dropout rate for the dense
layer, which is prone to overfitting.

    # Output layer: Produces the final classification probability.
    # The number of neurons equals the number of classes.
    # 'softmax' activation ensures the output is a probability
distribution (all outputs sum to 1).
    tf.keras.layers.Dense(train_data.num_classes, activation='softmax')
])

# --- 4. MODEL COMPILATION ---

# Configure the model for training.
model.compile(
    optimizer='adam',                   # Adam is an efficient and
popular optimization algorithm.
    loss='categorical_crossentropy',    # Loss function for multi-class
classification with one-hot encoded labels.
    metrics=['accuracy']                # Metric to monitor during
training.
)

# --- 5. CALLBACKS ---

# Define an EarlyStopping callback to prevent overfitting and save time.
# It monitors the validation loss and stops training if it doesn't
improve for a set number of epochs.
early_stop = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss',           # Metric to monitor.
    patience=5,                   # Number of epochs with no improvement
after which training will be stopped.
    restore_best_weights=True     # Restores model weights from the
epoch with the best value of the monitored metric.
)

# --- 6. MODEL TRAINING ---

# Train the model using the .fit() method.
```

```
history = model.fit(
    train_data,          # The generator for training data.
    validation_data=val_data, # The generator for validation data.
    epochs=30,           # The maximum number of times to iterate over
the entire dataset.
    callbacks=[early_stop] # List of callbacks to apply during training.
)

# --- 7. VISUALIZATION AND EVALUATION ---

# Plot the training and validation accuracy over epochs to visualize
performance.
plt.figure(figsize=(10, 6))
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy Over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.show()

# Evaluate the model's final performance on the validation dataset.
loss, acc = model.evaluate(val_data)
print(f"Final Validation Accuracy: {acc*100:.2f}%")
```

Output

```
Using Colab cache for faster access to the 'dog-and-cat-classification-dataset' dataset.
Found 20000 images belonging to 2 classes.
t actions 4998 images belonging to 2 classes.
/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 usin
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
/usr/local/lib/python3.12/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in its constructor. `**kwarg
  self._warn_if_super_not_called()
Epoch 1/30
537/625 ━━━━━━━━━━━━━━━━━━━━ 40s 464ms/step - accuracy: 0.5410 - loss: 0.7104/usr/local/lib/python3.12/dist-packages/PIL/TiffImagePlugin.py:950: UserWarning: Truncated File Read
  warnings.warn(str(msg))
625/625 ━━━━━━━━━━━━━━━━━━━━ 351s 557ms/step - accuracy: 0.5446 - loss: 0.7064 - val_accuracy: 0.6176 - val_loss: 0.6565
Epoch 2/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 205s 328ms/step - accuracy: 0.6147 - loss: 0.6546 - val_accuracy: 0.6715 - val_loss: 0.6090
Epoch 3/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 203s 324ms/step - accuracy: 0.6423 - loss: 0.6292 - val_accuracy: 0.6771 - val_loss: 0.5974
Epoch 4/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 215s 344ms/step - accuracy: 0.6628 - loss: 0.6154 - val_accuracy: 0.6963 - val_loss: 0.5828
Epoch 5/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 215s 344ms/step - accuracy: 0.6630 - loss: 0.6139 - val_accuracy: 0.7035 - val_loss: 0.5750
Epoch 6/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 245s 318ms/step - accuracy: 0.6706 - loss: 0.6023 - val_accuracy: 0.6755 - val_loss: 0.5880
Epoch 7/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 222s 355ms/step - accuracy: 0.6756 - loss: 0.5987 - val_accuracy: 0.6955 - val_loss: 0.5730
Epoch 8/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 217s 347ms/step - accuracy: 0.6873 - loss: 0.5849 - val_accuracy: 0.7107 - val_loss: 0.5660
Epoch 9/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 223s 358ms/step - accuracy: 0.6846 - loss: 0.5897 - val_accuracy: 0.6991 - val_loss: 0.5667
Epoch 10/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 203s 325ms/step - accuracy: 0.6976 - loss: 0.5730 - val_accuracy: 0.7133 - val_loss: 0.5492
Epoch 11/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 210s 336ms/step - accuracy: 0.7022 - loss: 0.5751 - val_accuracy: 0.7151 - val_loss: 0.5516
Epoch 12/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 201s 322ms/step - accuracy: 0.7017 - loss: 0.5725 - val_accuracy: 0.7257 - val_loss: 0.5495
Epoch 13/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 199s 318ms/step - accuracy: 0.6984 - loss: 0.5766 - val_accuracy: 0.7297 - val_loss: 0.5492
Epoch 14/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 210s 336ms/step - accuracy: 0.7035 - loss: 0.5654 - val_accuracy: 0.7307 - val_loss: 0.5477
Epoch 15/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 214s 342ms/step - accuracy: 0.7015 - loss: 0.5763 - val_accuracy: 0.7259 - val_loss: 0.5440
Epoch 16/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 213s 341ms/step - accuracy: 0.7109 - loss: 0.5663 - val_accuracy: 0.7315 - val_loss: 0.5415
Epoch 17/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 208s 333ms/step - accuracy: 0.7065 - loss: 0.5641 - val_accuracy: 0.7197 - val_loss: 0.5477
Epoch 18/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 205s 328ms/step - accuracy: 0.7063 - loss: 0.5697 - val_accuracy: 0.7335 - val_loss: 0.5461
Epoch 19/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 209s 335ms/step - accuracy: 0.7089 - loss: 0.5650 - val_accuracy: 0.7187 - val_loss: 0.5465
Epoch 20/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 209s 334ms/step - accuracy: 0.7134 - loss: 0.5594 - val_accuracy: 0.7309 - val_loss: 0.5372
Epoch 21/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 200s 321ms/step - accuracy: 0.7161 - loss: 0.5546 - val_accuracy: 0.7295 - val_loss: 0.5321
Epoch 22/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 200s 320ms/step - accuracy: 0.7134 - loss: 0.5637 - val_accuracy: 0.7185 - val_loss: 0.5351
Epoch 23/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 208s 333ms/step - accuracy: 0.7121 - loss: 0.5519 - val_accuracy: 0.7353 - val_loss: 0.5430
Epoch 24/30
```

```
Epoch 25/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 197s 315ms/step - accuracy: 0.7207 - loss: 0.5507 - val_accuracy: 0.7295 - val_loss: 0.5333
Epoch 26/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 201s 322ms/step - accuracy: 0.7220 - loss: 0.5485 - val_accuracy: 0.7439 - val_loss: 0.5233
Epoch 27/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 202s 322ms/step - accuracy: 0.7220 - loss: 0.5412 - val_accuracy: 0.7425 - val_loss: 0.5226
Epoch 28/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 195s 313ms/step - accuracy: 0.7217 - loss: 0.5471 - val_accuracy: 0.7069 - val_loss: 0.5604
Epoch 29/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 200s 319ms/step - accuracy: 0.7204 - loss: 0.5462 - val_accuracy: 0.7289 - val_loss: 0.5308
Epoch 30/30
625/625 ━━━━━━━━━━━━━━━━━━━━ 201s 322ms/step - accuracy: 0.7173 - loss: 0.5484 - val_accuracy: 0.7377 - val_loss: 0.5241
```




```
157/157 ━━━━━━━━━━━━━━━━━━━━ 27s 169ms/step - accuracy: 0.7216 - loss: 0.5279
Final Validation Accuracy: 73.33%
```

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: 6
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


# Function to generate sine wave sequences
def generate_sine_wave(seq_length, num_samples):
    X, y = [], []
    for _ in range(num_samples):
```

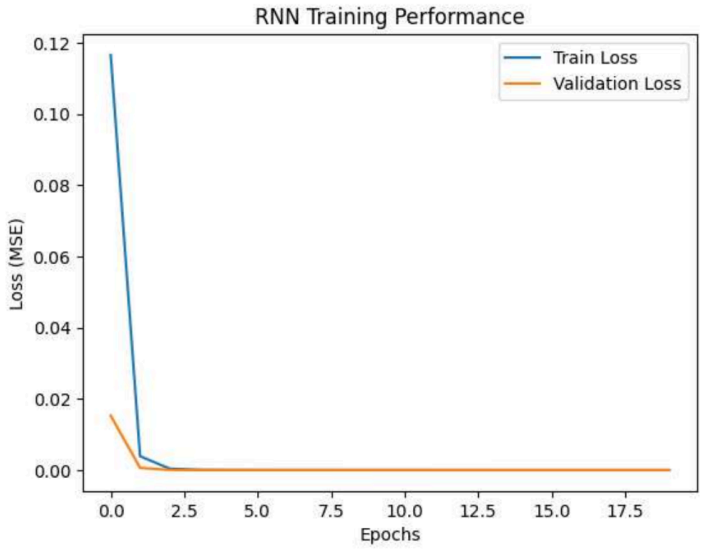
```
        start = np.random.rand() * 2 * np.pi
        xs = np.linspace(start, start + 3 * np.pi, seq_length + 1)
        data = np.sin(xs)
        X.append(data[:-1])
        y.append(data[-1])
    return np.array(X), np.array(y)


# Parameters
seq_length = 50
num_samples = 1000

# Generate dataset
X, y = generate_sine_wave(seq_length, num_samples)
X = X.reshape((num_samples, seq_length, 1))

# Build RNN model
model = Sequential([
    SimpleRNN(50, activation='tanh', input_shape=(seq_length, 1)),
    Dense(1)  # Predict next value in sequence
])

# Compile model
model.compile(optimizer='adam', loss='mse')

# Train model
history = model.fit(X, y, epochs=20, batch_size=32,
validation_split=0.2)

# Predict on first few samples
pred = model.predict(X[:10])

# Display sample predictions
print("\nSample Predictions:")
for i in range(5):
    print(f"True: {y[i]:.3f}, Predicted: {pred[i][0]:.3f}")

# Plot training performance
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: 7
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 5: 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 6: Create a WordCloud from the cleaned text to visually highlight frequently occurring words in the dataset.

Program

```
# Install required packages
!pip install nltk scikit-learn pandas matplotlib wordcloud --quiet

# Import necessary libraries
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
```

```
# Download stopwords
nltk.download('stopwords')

# Sample dataset
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."
    ]
}

# Create a DataFrame
df = pd.DataFrame(data)
print("Original Data:")
print(df)

# Load stopwords
stop_words = set(stopwords.words('english'))

# Preprocessing function
def preprocess(text):
    text = text.lower()  # convert to lowercase
    text = re.sub(r'[^a-z\s]', '', text)  # remove punctuation and
special characters
    tokens = text.split()  # tokenize text
    tokens = [word for word in tokens if word not in stop_words]  #
remove stopwords
    return " ".join(tokens)
```

```
# Apply preprocessing
df['clean_text'] = df['text'].apply(preprocess)

print("\nCleaned Text:")
print(df[['text', 'clean_text']])

# Get word frequency
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))

# TF-IDF vectorization
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(df['clean_text'])
feature_names = vectorizer.get_feature_names_out()

# Display top 5 TF-IDF words per document
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}")

# Generate WordCloud
text_combined = ' '.join(df['clean_text'])
wordcloud = WordCloud(width=800, height=400,
background_color='white').generate(text_combined)

# Plot WordCloud
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(0.400930738863647))]
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: 8(a)
Date:

DEEPDREAM IMAGE GENERATION

Aim

To implement the deepdream image generation.

Algorithm

Step 1: Install and Import Libraries

- Install required Python libraries: tensorflow, matplotlib, Pillow.
- Import libraries for:
    - TensorFlow (for deep learning models),
    - NumPy (for numerical operations),
    - PIL (for image handling),
    - Matplotlib (for displaying images),
    - Google Colab files (for uploading images).

Step 2: Load Pre-trained CNN

- Load a pre-trained InceptionV3 model without the top classification layers (include_top=False).
- Select the layers to enhance for DeepDream. Early layers capture textures; deeper layers capture objects/faces.
- Create a feature extraction model using the selected layers.

Step 3: Define Utility Functions

- load_img(path, max_dim): Loads an image, resizes it to a maximum dimension while maintaining aspect ratio, and adds a batch dimension [1, H, W, 3].
- deprocess(img): Converts processed tensor values ([-1,1]) back to standard image pixel values [0,255].
- show_img(img): Displays an image using Matplotlib.

Step 4: Define DeepDream Functions

- calc_loss(img, model): Passes the image through the selected CNN layers and calculates the loss as the sum of mean activations from all layers.

- deepdream_step(img, model, step_size):
    - Computes gradients of the loss with respect to the image.
    - Normalizes gradients and updates the image using gradient ascent:

        img = img + grads * step_size

    - Clips image values to [-1,1] to maintain valid pixel range.
- run_deep_dream(img, steps, step_size):
    - Iteratively applies deepdream_step for a fixed number of steps.
    - Prints intermediate loss every 20 steps.

Step 5: Upload and Preprocess Image

- Upload an image via Colab.
- Load and preprocess the image using load_img().
- Display the original image.

Step 6: Run DeepDream

- Convert the image to the expected range [-1,1].
- Pass the image through run_deep_dream to iteratively enhance patterns.

Step 7: Postprocess and Save

- Convert the output tensor back to pixel values using deprocess().
- Remove the extra batch dimension using np.squeeze().
- Display the final DeepDream image.
- Save the image as "deepdream_result.jpg".

Step 8: Experimentation Options

- Adjust steps (more → stronger effect).
- Adjust step size (larger → more psychedelic patterns).
- Change layers to enhance different features (textures vs objects).

Program

```
# Import libraries
import tensorflow as tf
import numpy as np
import PIL.Image
import matplotlib.pyplot as plt
from google.colab import files

# Load a pre-trained InceptionV3 model
base_model = tf.keras.applications.InceptionV3(include_top=False,
weights='imagenet')

# Choose layers to enhance (you can experiment with these)
names = ['mixed3', 'mixed5']
layers = [base_model.get_layer(name).output for name in names]

# Build the feature extraction model
dream_model = tf.keras.Model(inputs=base_model.input, outputs=layers)


# ---------------- Utility Functions ---------------- #

def deprocess(img):
    """Convert tensor image to uint8 format for display."""
    img = 255 * (img + 1.0) / 2.0
    return np.uint8(img)


def load_img(path, max_dim=512):
    """Load and preprocess image for DeepDream."""
    img = tf.io.read_file(path)
    img = tf.image.decode_image(img, channels=3)
    img = tf.image.convert_image_dtype(img, tf.float32)

    # Resize while keeping aspect ratio
    shape = tf.cast(tf.shape(img)[:2], tf.float32)
    scale = max_dim / max(shape)
    new_shape = tf.cast(shape * scale, tf.int32)
    img = tf.image.resize(img, new_shape)

    # Add batch dimension
    img = tf.expand_dims(img, axis=0)
    return img
```

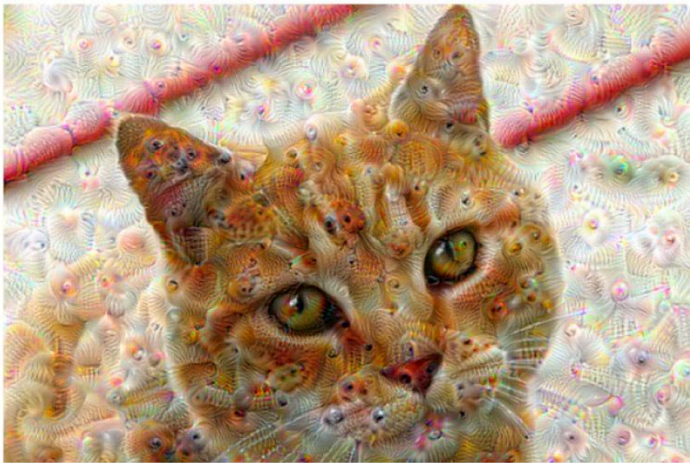
```
def show_img(img):
    """Display image using Matplotlib."""
    plt.figure(figsize=(8, 8))
    plt.imshow(np.squeeze(img))
    plt.axis('off')
    plt.show()


# ---------------- DeepDream Core Functions ---------------- #

def calc_loss(img, model):
    """Calculate DeepDream loss from selected layer activations."""
    layer_activations = model(img)
    if len(layer_activations) == 1:
        layer_activations = [layer_activations]
    losses = [tf.reduce_mean(act) for act in layer_activations]
    return tf.reduce_sum(losses)


@tf.function
def deepdream_step(img, model, step_size):
    """Perform one DeepDream gradient ascent step."""
    with tf.GradientTape() as tape:
        tape.watch(img)
        loss = calc_loss(img, model)
    grads = tape.gradient(loss, img)
    grads /= tf.math.reduce_std(grads) + 1e-8  # Normalize gradients
    img = img + grads * step_size
    img = tf.clip_by_value(img, -1, 1)
    return img, loss


def run_deep_dream(img, steps=100, step_size=0.01):
    """Run DeepDream process for a number of steps."""
    for step in range(steps):
        img, loss = deepdream_step(img, dream_model, step_size)
        if step % 20 == 0:
            print(f"Step {step}, Loss {loss}")
    return img


# ---------------- Run DeepDream in Google Colab ---------------- #

# Upload an image
uploaded = files.upload()
image_path = list(uploaded.keys())[0]
```

```
# Load and display original image
original_img = load_img(image_path)
show_img(original_img)

# Run DeepDream
dream_img = run_deep_dream(original_img * 2 - 1, steps=100,
step_size=0.01)

# Convert and display final image
dream_img = deprocess(dream_img.numpy())
dream_img = np.squeeze(dream_img, axis=0)  # Remove batch dimension
show_img(dream_img)

# Save the result
result = PIL.Image.fromarray(dream_img)
result.save("deepdream_result.jpg")

print("✅ DeepDream image saved as 'deepdream_result.jpg'")
```

Output

Choose files output_step_0.png
output_step_0.png(image/png) - 348356 bytes, last modified: 29/09/2025 - 100% done
Saving output_step_0.png to output_step_0 (2).png

```
Step 0, Loss 0.5668067336082458
Step 20, Loss 1.4887861013412476
Step 40, Loss 1.8054102659225464
Step 60, Loss 1.9969892501831055
Step 80, Loss 2.1354920864105225
```



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

Result

Thus, the implementation of deepdream image generation is successfully executed.

Ex. no: 8(b)
Date:

NEURAL STYLE TRANSFER

Aim

To implement the neural style transfer

Algorithm

Step 1: Inputs

- Content Image (C) → carries the structure, objects, and layout you want to preserve.
- Style Image (S) → carries textures, colors, and artistic style to apply.

Step 2: Image Transformation Network

- This is a trainable feed-forward CNN (not VGG).
- It takes the Content Image C as input and produces an Output Image Y (stylized).
- Instead of optimizing pixels directly (like Gatys' original NST), here we train this network to generate stylized images quickly.

Step 3: VGG Network (Feature Extractor)

- A fixed, pre-trained VGG network (e.g., VGG19 trained on ImageNet).
- It is not trained here — only used to extract features.
- It processes:
  - Content Image (C) → produces content representation.
  - Style Image (S) → produces style representation (via Gram matrices).
  - Output Image (Y) → produces both content & style representations.

Step 4: Loss Calculation

- Compare the Output Image's features with both Content and Style features:
- Content Loss = difference between content features of C and Y.
- Style Loss = difference between style features (Gram matrices) of S and Y.
- Both losses are combined:

$$L_{\text{total}} = \alpha \cdot L_{\text{content}} + \beta \cdot L_{\text{style}}$$

Step 5: Optimization

- The Image Transformation Network is trained to minimize this loss.
- Over time, the network learns to generate stylized output images in one forward pass (fast style transfer).

Program

```
# ----------------------------------
# Import libraries
# ----------------------------------
```

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
import torchvision.models as models
from PIL import Image
import matplotlib.pyplot as plt
import copy
import os

# ---------- User settings ----------
content_img_path = "/content/drive/MyDrive/Colab Notebooks/Cat03.jpg"
style_img_path = "/content/drive/MyDrive/Colab Notebooks/wave.jpg"
output_dir = "nst_outputs"

imsize = 512            # Set 512 for GPU, 256 for CPU if memory limited
num_steps = 300         # Total optimization steps
style_weight = 1e6      # Style loss weight (β)
content_weight = 1e0    # Content loss weight (α)

use_gpu = torch.cuda.is_available()
device = torch.device("cuda" if use_gpu else "cpu")
# -----------------------------------

# Create output directory if not exists
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# -----------------------------------
# Image loader / preprocessor
# -----------------------------------
loader = transforms.Compose([
    transforms.Resize(imsize),
    transforms.CenterCrop(imsize),
    transforms.ToTensor()
])
unloader = transforms.ToPILImage()


def load_image(path):
    """Load and preprocess an image."""
    image = Image.open(path).convert('RGB')
    image = loader(image).unsqueeze(0)  # Add batch dimension
    return image.to(device, torch.float)
```

```
def imsave(tensor, filename):
    """Save a tensor as an image."""
    image = tensor.cpu().clone().detach()
    image = image.squeeze(0)
    image = unloader(image)
    image.save(filename)


# ----------------------------------
# Gram matrix for style representation
# ----------------------------------
def gram_matrix(input_tensor):
    """Compute the Gram matrix of a feature map."""
    b, ch, h, w = input_tensor.size()
    features = input_tensor.view(b * ch, h * w)
    G = torch.mm(features, features.t())
    return G.div(b * ch * h * w)


# ----------------------------------
# Load VGG19 model for feature extraction
# ----------------------------------
cnn = models.vgg19(pretrained=True).features.to(device).eval()

# Freeze parameters (we won't train the CNN)
for param in cnn.parameters():
    param.requires_grad = False

# Define which layers to use for content and style
content_layers = ['21']  # conv4_2
style_layers = ['0', '5', '10', '19', '28']  # conv1_1, conv2_1,
conv3_1, conv4_1, conv5_1


# ----------------------------------
# Feature extraction helper
# ----------------------------------
def get_features(x, model, layers):
    """Extract intermediate features from selected layers."""
    features = {}
    i = 0
    for layer in model.children():
        x = layer(x)
        if str(i) in layers:
            features[str(i)] = x
        i += 1
```

```
    return features


# ------------------------------------
# Load content and style images
# ------------------------------------
content_img = load_image(content_img_path)
style_img = load_image(style_img_path)

assert content_img.size() == style_img.size(), "Style and content images
must be the same size"

# Initialize target image as a copy of content image (or random noise)
target = content_img.clone().requires_grad_(True).to(device)

# Compute features for content and style
content_feats = get_features(content_img, cnn, content_layers)
style_feats = get_features(style_img, cnn, style_layers)

# Compute Gram matrices for style features
style_grams = {layer: gram_matrix(style_feats[layer]) for layer in
style_feats}

# Define MSE loss
mse_loss = nn.MSELoss()

# Use L-BFGS optimizer (works well for NST)
optimizer = optim.LBFGS([target])
# Alternative: optimizer = optim.Adam([target], lr=0.02)

print("Starting optimization on", device)

# ------------------------------------
# Optimization loop
# ------------------------------------
run = [0]
while run[0] <= num_steps:

    def closure():
        with torch.no_grad():
            target.clamp_(0, 1)

        optimizer.zero_grad()

        # Extract target features
        target_feats = get_features(target, cnn, content_layers +
```

```
style_layers)

        # Compute content loss
        content_loss = 0.0
        for layer in content_layers:
            target_f = target_feats[layer]
            content_f = content_feats[layer]
            content_loss += mse_loss(target_f, content_f)

        # Compute style loss
        style_loss = 0.0
        for layer in style_layers:
            target_f = target_feats[layer]
            target_gram = gram_matrix(target_f)
            style_gram = style_grams[layer]
            style_loss += mse_loss(target_gram, style_gram)

        # Total loss
        total_loss = content_weight * content_loss + style_weight *
style_loss
        total_loss.backward()

        # Logging and saving intermediate images
        if run[0] % 50 == 0:
            print("Step {}: Total Loss: {:.4e}, Content: {:.4e}, Style:
{:.4e}".format(
                run[0], total_loss.item(), content_loss.item(),
style_loss.item()))

            out_path = os.path.join(output_dir,
f"output_step_{run[0]}.png")
            with torch.no_grad():
                tmp = target.clone().detach()
                tmp.clamp_(0, 1)
                imsave(tmp, out_path)

        run[0] += 1
        return total_loss

    optimizer.step(closure)

# ------------------------------------
# Save final output
# ------------------------------------
with torch.no_grad():
    target.clamp_(0, 1)
```

```
    imsave(target, os.path.join(output_dir, "final_output.png"))

print("✅ Finished. Final image saved to", os.path.join(output_dir,
"final_output.png"))
```

Output

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

Result

Thus, the implementation of neural style transfer is successfully executed.

Ex. no: 9
Date:

GENERATING SYNTHETIC IMAGES USING VAE

Aim

To implement the VAE by generating the synthetic images

Algorithm

Step 1: Initialize Parameters
- Set latent dimension size, batch size, learning rate, and number of epochs.
- Choose the device (GPU if available, else CPU).
- Create an output directory to save generated images.

Step 2: Prepare Dataset
- Load MNIST dataset (28×28 grayscale images).
- Apply transformation (ToTensor) to convert images into tensors scaled between [0,1].
- Use DataLoader to batch and shuffle the dataset.

Step 3: Define VAE Model
- Encoder:
  - Flatten input image.
  - Pass through fully connected layers with activation (ReLU).
  - Output two vectors: mean (μ) and log-variance ($\log\sigma^2$).
- Reparameterization Trick:
  - Sample $\varepsilon \sim N(0, I)$.
  - Compute latent vector $z = \mu + \sigma \cdot \varepsilon$.
- Decoder:
  - Pass latent vector z through fully connected layers with activation (ReLU).
  - Output a reconstructed image using Sigmoid activation (ensures values between 0–1).

Step 4: Define Loss Function
- Reconstruction Loss (BCE): Measures pixel-wise difference between input and reconstructed image.
- KL Divergence: Forces latent distribution $q(z|x)$ to be close to the standard normal distribution $p(z)$.
- Total Loss = BCE + KL Divergence.

Step 5: Training Loop
- For each epoch:
  - For each batch:
    - Forward pass: input image → encoder → latent z → decoder → reconstruction.
    - Compute loss (BCE + KL).
    - Backpropagate and update model parameters using Adam

optimizer.
    - Print average loss after each epoch.

Step 6: Image Generation

- After training:
    - Sample random latent vectors z ~ N(0, I).
    - Pass z through decoder to generate synthetic images.
    - Save images to the output folder.

Step 7: Reconstruction

- Pass real images through encoder → reparameterization → decoder.
- Compare original images with reconstructions and save output.

Step 8: Interpolation in Latent Space

- Choose two real images.
- Encode them into latent vectors z_1 and z_2.
- Linearly interpolate between z_1 and z_2.
- Decode intermediate z values to generate a smooth transition of images.

Program

```
# ============================================
# Variational Autoencoder (VAE) on MNIST
# ============================================

# Install dependencies (uncomment if needed)
# !pip install torch torchvision tqdm matplotlib

import os
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms, utils
from torch.utils.data import DataLoader
from tqdm import tqdm

# ----------------------
# Config / Hyperparameters
# ----------------------
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
batch_size = 128
epochs = 20
latent_dim = 20
lr = 1e-3
log_interval = 200
output_dir = "vae_outputs"
os.makedirs(output_dir, exist_ok=True)
```

```
# ----------------------
# Dataset (MNIST)
# ----------------------
transform = transforms.Compose([
    transforms.ToTensor()
])

train_dataset = datasets.MNIST(
    root="data",
    train=True,
    download=True,
    transform=transform
)

train_loader = DataLoader(
    train_dataset,
    batch_size=batch_size,
    shuffle=True,
    num_workers=2,
    pin_memory=True
)

# ----------------------
# Variational Autoencoder (VAE)
# ----------------------
class VAE(nn.Module):
    def __init__(self, latent_dim=20):
        super().__init__()

        # Encoder: simple MLP
        self.enc = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28 * 28, 400),
            nn.ReLU(),
        )

        self.fc_mu = nn.Linear(400, latent_dim)
        self.fc_logvar = nn.Linear(400, latent_dim)

        # Decoder
        self.dec_fc = nn.Sequential(
            nn.Linear(latent_dim, 400),
            nn.ReLU(),
            nn.Linear(400, 28 * 28),
            nn.Sigmoid()  # outputs in [0, 1]
        )
```

```
    def encode(self, x):
        h = self.enc(x)
        mu = self.fc_mu(h)
        logvar = self.fc_logvar(h)
        return mu, logvar

    def reparameterize(self, mu, logvar):
        # Reparameterization trick
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def decode(self, z):
        x_recon = self.dec_fc(z)
        x_recon = x_recon.view(-1, 1, 28, 28)
        return x_recon

    def forward(self, x):
        mu, logvar = self.encode(x)
        z = self.reparameterize(mu, logvar)
        x_recon = self.decode(z)
        return x_recon, mu, logvar


# -----------------------
# Loss Function: BCE + KL Divergence
# -----------------------
def vae_loss(recon_x, x, mu, logvar):
    # Reconstruction term (binary cross-entropy per pixel)
    BCE = nn.functional.binary_cross_entropy(recon_x, x,
reduction='sum')

    # KL divergence between posterior N(mu, sigma^2) and prior N(0,1)
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())

    return BCE + KLD, BCE, KLD


# -----------------------
# Training
# -----------------------
model = VAE(latent_dim=latent_dim).to(device)
optimizer = optim.Adam(model.parameters(), lr=lr)
```

```
def train(epoch):
    model.train()
    train_loss = 0.0
    pbar = tqdm(enumerate(train_loader), total=len(train_loader))

    for batch_idx, (data, _) in pbar:
        data = data.to(device)
        optimizer.zero_grad()

        recon_batch, mu, logvar = model(data)
        loss, bce, kld = vae_loss(recon_batch, data, mu, logvar)

        loss.backward()
        train_loss += loss.item()
        optimizer.step()

        if batch_idx % log_interval == 0:
            pbar.set_description(
                f"Epoch {epoch} Batch {batch_idx} "
                f"Loss {loss.item() / len(data):.4f} "
                f"BCE {bce / len(data):.4f} "
                f"KLD {kld / len(data):.4f}"
            )

    avg_loss = train_loss / len(train_loader.dataset)
    print(f"====> Epoch {epoch} Average loss: {avg_loss:.4f}")


# -----------------------
# Utilities: Sampling & Visualization
# -----------------------
@torch.no_grad()
def sample_and_save(epoch, n=64):
    model.eval()
    z = torch.randn(n, latent_dim).to(device)
    samples = model.decode(z).cpu()
    utils.save_image(samples, f"{output_dir}/samples_epoch_{epoch}.png",
nrow=8, padding=2)


@torch.no_grad()
def reconstruct_and_save(x, filename="recon.png", n=8):
    model.eval()
    x = x.to(device)[:n]
    recon, _, _ = model(x)
    comparison = torch.cat([x.cpu(), recon.cpu()])
```

```
    utils.save_image(comparison, filename, nrow=n, padding=2)


@torch.no_grad()
def interpolate_and_save(index_a, index_b, dataset, steps=8,
filename="interpolate.png"):
    # Pick two images from dataset and encode them, then interpolate in
latent space
    model.eval()
    xa, _ = dataset[index_a]
    xb, _ = dataset[index_b]
    xa = xa.unsqueeze(0).to(device)
    xb = xb.unsqueeze(0).to(device)

    mu_a, logvar_a = model.encode(xa)
    mu_b, logvar_b = model.encode(xb)

    za = mu_a  # deterministic interpolation
    zb = mu_b

    all_imgs = []
    for alpha in torch.linspace(0, 1, steps):
        z = (1 - alpha) * za + alpha * zb
        img = model.decode(z).cpu()
        all_imgs.append(img)

    grid = torch.cat(all_imgs, dim=0)
    utils.save_image(grid, filename, nrow=steps, padding=2)


# ----------------------
# Main Loop
# ----------------------
if __name__ == "__main__":
    for epoch in range(1, epochs + 1):
        train(epoch)
        sample_and_save(epoch, n=64)

    # Save trained model
    torch.save(model.state_dict(), os.path.join(output_dir,
"vae_mnist.pth"))
    print("Training complete. Samples saved in", output_dir)

    # Example: reconstruction
    test_loader = DataLoader(train_dataset, batch_size=8, shuffle=True)
    data_iter = iter(test_loader)
```

```
    x, _ = next(data_iter)
    reconstruct_and_save(x, filename=os.path.join(output_dir,
"reconstruction.png"), n=8)

    # Example: interpolation
    interpolate_and_save(
        1, 10, train_dataset, steps=10,
        filename=os.path.join(output_dir, "interp.png")
    )

    print("Saved reconstruction and interpolation images.")
```

Output

```
Epoch 7 Batch 400 Loss 107.5740 BCE 82.8399 KLD 24.7341: 100%|██████████| 469/469 [00:07<00:00, 62.22it/s]====> Epoch 7 Average loss: 108.0125
Epoch 8 Batch 400 Loss 107.7619 BCE 82.1332 KLD 25.6287: 100%|██████████| 469/469 [00:07<00:00, 59.92it/s]====> Epoch 8 Average loss: 107.3202
Epoch 9 Batch 400 Loss 108.5338 BCE 83.0104 KLD 25.5233: 100%|██████████| 469/469 [00:06<00:00, 68.04it/s]====> Epoch 9 Average loss: 106.7956
Epoch 10 Batch 400 Loss 104.7559 BCE 79.8464 KLD 24.9095: 100%|██████████| 469/469 [00:07<00:00, 60.23it/s]====> Epoch 10 Average loss: 106.3813
Epoch 11 Batch 400 Loss 106.6543 BCE 81.5160 KLD 25.1383: 100%|██████████| 469/469 [00:07<00:00, 64.21it/s]====> Epoch 11 Average loss: 106.0447
Epoch 12 Batch 400 Loss 104.2354 BCE 79.0394 KLD 25.1961: 100%|██████████| 469/469 [00:07<00:00, 64.10it/s]====> Epoch 12 Average loss: 105.6968
Epoch 13 Batch 400 Loss 101.7215 BCE 77.1955 KLD 24.5260: 100%|██████████| 469/469 [00:07<00:00, 61.10it/s]====> Epoch 13 Average loss: 105.4054
Epoch 14 Batch 400 Loss 106.5979 BCE 80.6492 KLD 25.9487: 100%|██████████| 469/469 [00:06<00:00, 68.58it/s]====> Epoch 14 Average loss: 105.1540
Epoch 15 Batch 400 Loss 103.5979 BCE 78.6691 KLD 24.9289: 100%|██████████| 469/469 [00:07<00:00, 60.62it/s]====> Epoch 15 Average loss: 104.9338
Epoch 16 Batch 400 Loss 105.4352 BCE 80.3764 KLD 25.0587: 100%|██████████| 469/469 [00:07<00:00, 66.71it/s]====> Epoch 16 Average loss: 104.7524
Epoch 17 Batch 400 Loss 102.2124 BCE 77.9314 KLD 24.2810: 100%|██████████| 469/469 [00:07<00:00, 62.50it/s]====> Epoch 17 Average loss: 104.5408
Epoch 18 Batch 400 Loss 103.1426 BCE 77.3923 KLD 25.7503: 100%|██████████| 469/469 [00:07<00:00, 60.84it/s]====> Epoch 18 Average loss: 104.3621
Epoch 19 Batch 400 Loss 102.5561 BCE 77.5128 KLD 25.0433: 100%|██████████| 469/469 [00:06<00:00, 67.60it/s]====> Epoch 19 Average loss: 104.1620
Epoch 20 Batch 400 Loss 103.9708 BCE 78.0564 KLD 25.9145: 100%|██████████| 469/469 [00:07<00:00, 61.34it/s]
====> Epoch 20 Average loss: 104.0347
Training complete. Samples saved in vae_outputs
Saved reconstruction and interpolation images.
```

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

Result

Thus, the implementation of generating the synthetic image using VAE is executed successfully.

Ex. no: 10
Date:

SYNTHETIC IMAGE GENERATION USING GAN

Aim

To implement the synthetic image generation using GAN

Algorithm

Step 1: Initialize Parameters

- Set batch size, image size (28×28 for MNIST), latent vector size (nz), learning rate (lr), and number of epochs.
- Choose the device (GPU if available, else CPU).

Step 2: Prepare Dataset

- Use MNIST dataset (grayscale handwritten digits).
- Apply transforms:
    - Convert images to tensors.
    - Normalize pixel values to range [-1, 1].
- Load dataset with a DataLoader.

Step 3: Define Generator (G)

- Input: Random noise vector (latent space, size 100).
- Layers:
    - ConvTranspose2D → Expand noise to feature maps.
    - BatchNorm + ReLU → Normalize and activate.
    - More ConvTranspose2D layers → Upsample to 28×28.
    - Final layer: ConvTranspose2D → Tanh() → Output fake image in [-1,1].
- Output: Synthetic image (shape = 1×28×28).

Step 4: Define Discriminator (D)

- Input: Real or fake image (1×28×28).
- Layers:
    - Conv2D → Extract features from image.

    - LeakyReLU → Non-linear activation.
    - Conv2D + BatchNorm + LeakyReLU → Deep features.
    - Flatten + Linear + Sigmoid → Output probability (real=1, fake=0).
- Output: Scalar (real/fake prediction).

Step 5: Initialize Training Components

- Loss function: Binary Cross-Entropy Loss (BCELoss).
- Optimizers: Adam for both G and D.
- Create a fixed noise vector for monitoring generator progress.

Step 6: Training Loop (for each epoch)

- Train Discriminator (D):
    - Feed real images → predict real probability.
        - Compute loss_real against target label = 1.
        - Backpropagate.
    - Generate fake images using Generator.
        - Feed fake images into D.
        - Compute loss_fake against target label = 0.
        - Backpropagate.
    - Update D weights (gradient descent).
- Train Generator (G):
    - Generate fake images from random noise.
    - Pass them into D, but this time the target label = 1 (trick D).
    - Compute loss_G and backpropagate.
    - Update G weights.

Step 7: Monitor Progress

- Print losses for D and G each epoch.
- Generate sample fake images using fixed noise.
- Display them to visually check improvements.

Step 8: Output

- After training, Generator can take random noise and produce synthetic handwritten digit images that look realistic.

Program

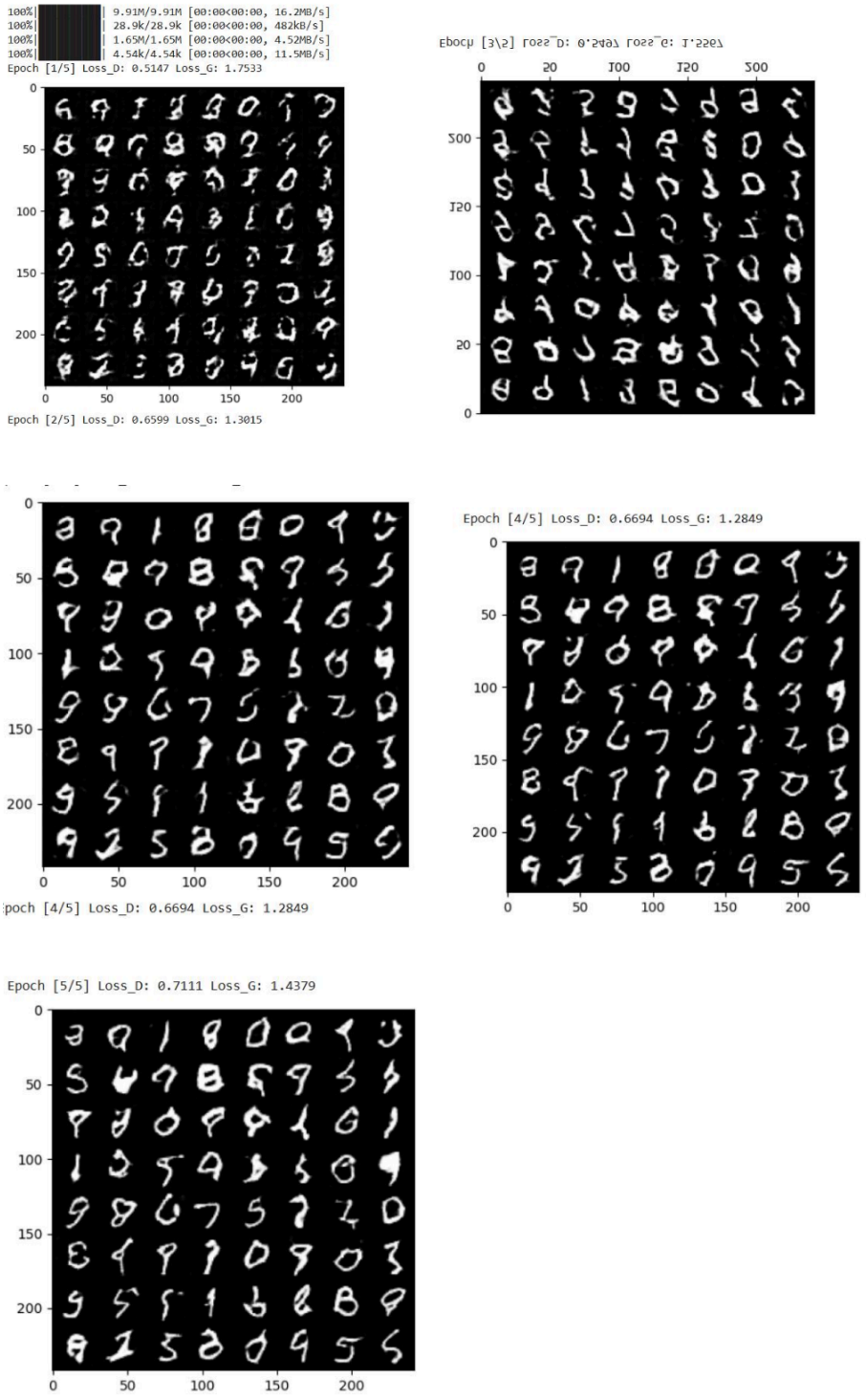
```
# ======================================================
# DCGAN on MNIST
# ======================================================

# Install dependencies (uncomment if needed)
# !pip install torch torchvision tqdm matplotlib

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

# ================== Configuration ==================
batch_size = 128
image_size = 28
nz = 100              # Size of latent vector (input noise to generator)
num_epochs = 5
lr = 0.0002
beta1 = 0.5
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# ================== Dataset & Dataloader ==================
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))  # Normalize to [-1, 1]
])

dataset = torchvision.datasets.MNIST(
    root='./data',
    train=True,
    transform=transform,
    download=True
)

dataloader = torch.utils.data.DataLoader(
    dataset, batch_size=batch_size, shuffle=True
)
```

```
# ================== Generator ==================
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            nn.ConvTranspose2d(nz, 128, 7, 1, 0, bias=False),
            nn.BatchNorm2d(128),
            nn.ReLU(True),

            nn.ConvTranspose2d(128, 64, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(True),

            nn.ConvTranspose2d(64, 1, 4, 2, 1, bias=False),
            nn.Tanh()  # Output in range [-1, 1]
        )

    def forward(self, input):
        return self.main(input)


# ================== Discriminator ==================
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            nn.Conv2d(1, 64, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(64, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Flatten(),
            nn.Linear(128 * 7 * 7, 1),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)


# ================== Initialize Models, Loss & Optimizers
==================
netG = Generator().to(device)
netD = Discriminator().to(device)
```

```
criterion = nn.BCELoss()
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))

fixed_noise = torch.randn(64, nz, 1, 1, device=device)

# ================== Training Loop ==================
for epoch in range(num_epochs):
    for i, (data, _) in enumerate(dataloader):
        # --------------------
        # Train Discriminator
        # --------------------
        netD.zero_grad()
        real = data.to(device)
        b_size = real.size(0)

        # Train on real images
        label = torch.full((b_size,), 1.0, dtype=torch.float,
device=device)
        output = netD(real).view(-1)
        errD_real = criterion(output, label)
        errD_real.backward()

        # Train on fake images
        noise = torch.randn(b_size, nz, 1, 1, device=device)
        fake = netG(noise)
        label.fill_(0.0)
        output = netD(fake.detach()).view(-1)
        errD_fake = criterion(output, label)
        errD_fake.backward()

        optimizerD.step()

        # --------------------
        # Train Generator
        # --------------------
        netG.zero_grad()
        label.fill_(1.0)  # Generator tries to fool Discriminator
        output = netD(fake).view(-1)
        errG = criterion(output, label)
        errG.backward()
        optimizerG.step()

        # Print progress
        print(
```

```
            f"Epoch [{epoch+1}/{num_epochs}] "
            f"Loss_D: {(errD_real + errD_fake):.4f}  "
            f"Loss_G: {errG:.4f}"
        )

    # --------------------
    # Generate Samples for Visualization
    # --------------------
    with torch.no_grad():
        fake = netG(fixed_noise).detach().cpu()
        grid = torchvision.utils.make_grid(fake, padding=2,
normalize=True)
        plt.figure(figsize=(8, 8))
        plt.axis("off")
        plt.title(f"Generated Images - Epoch {epoch+1}")
        plt.imshow(grid.permute(1, 2, 0))
        plt.show()
```

Output


100%| | 9.91M/9.91M [00:00<00:00, 16.2MB/s]
100%| | 28.9k/28.9k [00:00<00:00, 482kB/s]
100%| | 1.65M/1.65M [00:00<00:00, 4.52MB/s]
100%| | 4.54k/4.54k [00:00<00:00, 11.5MB/s]
Epoch [1/5] Loss_D: 0.5147 Loss_G: 1.7533
Epoch [2/5] Loss_D: 0.6599 Loss_G: 1.3015
:poch [4/5] Loss_D: 0.6694 Loss_G: 1.2849
Epoch [4/5] Loss_D: 0.6694 Loss_G: 1.2849
Epoch [5/5] Loss_D: 0.7111 Loss_G: 1.4379

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

Result

Thus, the implementation of synthetic image generation using gan is successfully executed.