EX.NO:02

DATE:

MULTILAYER PERCEPTRON WITH HYPERPARAMETER TUNING

AIM:

To construct a multilayer perceptron with a hyper parameter tuning using sales Dataset.

ALGORITHM:

STEP 1: Start the process.

STEP 2: Import required libraries – pandas, numpy, scikit-learn, matplotlib, seaborn, and tensorflow.

STEP 3: Load the dataset (sales_data_sample.csv) and display its first few rows.

STEP 4: Drop irrelevant columns and extract Year, Month, Day from the ORDERDATE column.

STEP 5: Handle missing values by removing rows with null entries. Encode categorical features using Label Encoding.

STEP 6: Separate features (X) and target (y), then split data into training and testing sets (80-20).

STEP 7: Scale the features using StandardScaler to normalize the data.

STEP 8: Build a Neural Network model with two hidden layers (128 & 64 neurons, ReLU activation), Dropout layers, and an output layer with sigmoid activation.

STEP 9: Create a Perceptron object by calling Perceptron(input_size=2) to initialize it with two inputs.

STEP 10: Compile the model with Adam optimizer, binary crossentropy loss, and accuracy metric, and apply EarlyStopping to prevent overfitting.

STEP 11: Train the model, evaluate accuracy on test data, generate predictions, print classification report, and visualize the confusion matrix.

STEP 12: Stop the process.

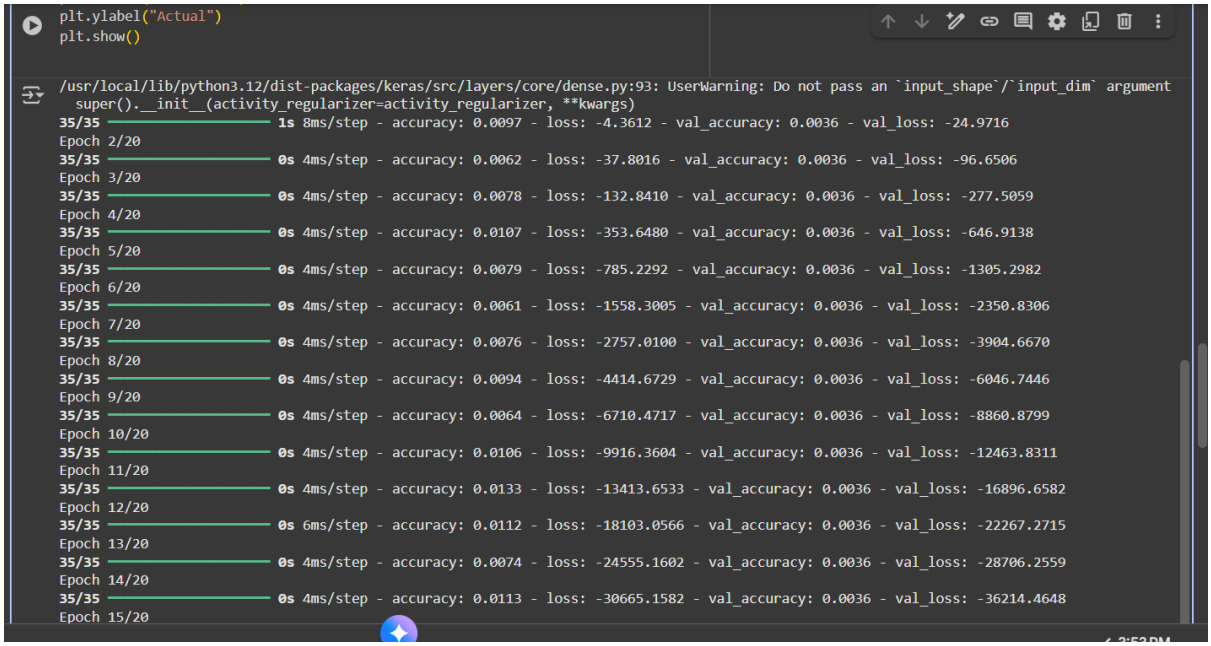
CODING:

```
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
# Step 2: Load dataset
df = pd.read_csv("sales_data_sample.csv", encoding="latin1")
print(" Dataset loaded successfully!\n")
print(df.head())
# Step 3: Drop columns not useful
df = df.drop(columns=[
    "ORDERNUMBER","ADDRESSLINE1","ADDRESSLINE2","CITY",
    "STATE","POSTALCODE","CONTACTLASTNAME","CONTACTFIRSTNAME"
], errors="ignore")
# Step 4: Convert ORDERDATE to datetime features
df["ORDERDATE"] = pd.to_datetime(df["ORDERDATE"], errors="coerce")
df["YEAR"] = df["ORDERDATE"].dt.year
df["MONTH"] = df["ORDERDATE"].dt.month
df["DAY"] = df["ORDERDATE"].dt.day
df = df.drop(columns=["ORDERDATE"])
# Step 5: Handle missing values
```

```
df = df.dropna()
# Step 6: Encode categorical columns
label_encoders = {}
for col in df.select_dtypes(include="object").columns:
    le = LabelEncoder()
    df[col] = le.fit_transform(df[col])
    label_encoders[col] = le
# Step 7: Define features and target
X = df.drop(columns=["STATUS"])   # Features
y = df["STATUS"]                  # Target
# Step 8: Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
# Step 9: Scale features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
# Step 10: Build Neural Network model
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"])
# Step 11: Early stopping
early_stop = EarlyStopping(monitor="val_loss", patience=5, restore_best_weights=True)
```

```
# Step 12: Train model
history = model.fit(
    X_train, y_train,
    validation_split=0.2,
    epochs=20,
    batch_size=32,
    callbacks=[early_stop],
    verbose=1
)
# Step 13: Evaluate model
loss, acc = model.evaluate(X_test, y_test, verbose=0)
print(f"\n Final Test Accuracy: {acc * 100:.2f}%")
# Step 14: Predictions
y_pred = (model.predict(X_test) > 0.5).astype(int)
print("\n Classification Report:")
print(classification_report(y_test, y_pred))
# Step 15: Confusion Matrix
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()
```

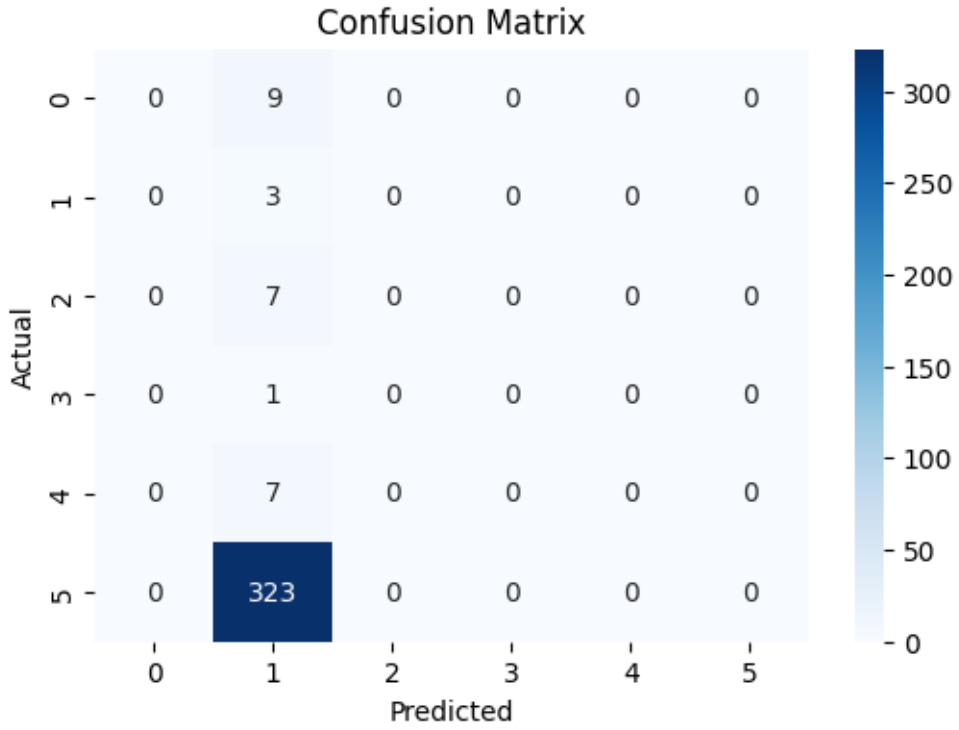
OUTPUT:



```
plt.ylabel("Actual")
plt.show()

/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93: UserWarning: Do not pass an `input_shape`/`input_dim` argument
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
35/35 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - accuracy: 0.0097 - loss: -4.3612 - val_accuracy: 0.0036 - val_loss: -24.9716
Epoch 2/20
35/35 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0062 - loss: -37.8016 - val_accuracy: 0.0036 - val_loss: -96.6506
Epoch 3/20
35/35 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0078 - loss: -132.8410 - val_accuracy: 0.0036 - val_loss: -277.5059
Epoch 4/20
35/35 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0107 - loss: -353.6480 - val_accuracy: 0.0036 - val_loss: -646.9138
Epoch 5/20
35/35 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0079 - loss: -785.2292 - val_accuracy: 0.0036 - val_loss: -1305.2982
Epoch 6/20
35/35 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0061 - loss: -1558.3005 - val_accuracy: 0.0036 - val_loss: -2350.8306
Epoch 7/20
35/35 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0076 - loss: -2757.0100 - val_accuracy: 0.0036 - val_loss: -3904.6670
Epoch 8/20
35/35 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0094 - loss: -4414.6729 - val_accuracy: 0.0036 - val_loss: -6046.7446
Epoch 9/20
35/35 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0064 - loss: -6710.4717 - val_accuracy: 0.0036 - val_loss: -8860.8799
Epoch 10/20
35/35 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0106 - loss: -9916.3604 - val_accuracy: 0.0036 - val_loss: -12463.8311
Epoch 11/20
35/35 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0133 - loss: -13413.6533 - val_accuracy: 0.0036 - val_loss: -16896.6582
Epoch 12/20
35/35 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.0112 - loss: -18103.0566 - val_accuracy: 0.0036 - val_loss: -22267.2715
Epoch 13/20
35/35 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0074 - loss: -24555.1602 - val_accuracy: 0.0036 - val_loss: -28706.2559
Epoch 14/20
35/35 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.0113 - loss: -30665.1582 - val_accuracy: 0.0036 - val_loss: -36214.4648
Epoch 15/20
```

📊 **Classification Report:**

	precision	recall	f1-score	support
0	0.00	0.00	0.00	9
1	0.01	1.00	0.02	3
2	0.00	0.00	0.00	7
3	0.00	0.00	0.00	1
4	0.00	0.00	0.00	7
5	0.00	0.00	0.00	323
accuracy			0.01	350
macro avg	0.00	0.17	0.00	350
weighted avg	0.00	0.01	0.00	350



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

RESULT:

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

EX.NO:03

DATE:

DATA AUGMENTATION FOR IMAGE

AIM:

To Generate synthetic images using traditional data augmentation function.

ALGORITHM:

STEP 1: Start the process.

STEP 2: Import necessary libraries: matplotlib.pyplot for plotting, and tensorflow.keras .preprocessing.image for image loading and augmentation.

STEP 3: Load the original image using load_img() and resize it to a fixed size (150×150 pixels).

STEP 4: Convert the loaded image to a NumPy array using img_to_array(). Expand the dimensions of the array to create a batch of size 1 (required by ImageDataGenerator).

STEP 5: Define an ImageDataGenerator with augmentation parameters such as:

- Rotation (rotation_range)
- Width & height shifts (width_shift_range, height_shift_range)
- Shear (shear_range)
- Zoom (zoom_range)
- Horizontal flip (horizontal_flip)
- Fill mode (fill_mode)

STEP 6: Create an iterator using datagen.flow() to generate augmented images from the input image batch.

STEP 7: Initialize a plot using matplotlib to display multiple images.

STEP 8 Loop through the iterator to generate a specified number of augmented images (6 images).

STEP 9: Convert each augmented image to uint8 format and plot it in a subplot, turning off axes for clarity.

STEP 10: Add titles and a main figure title, then display the figure using plt.show().

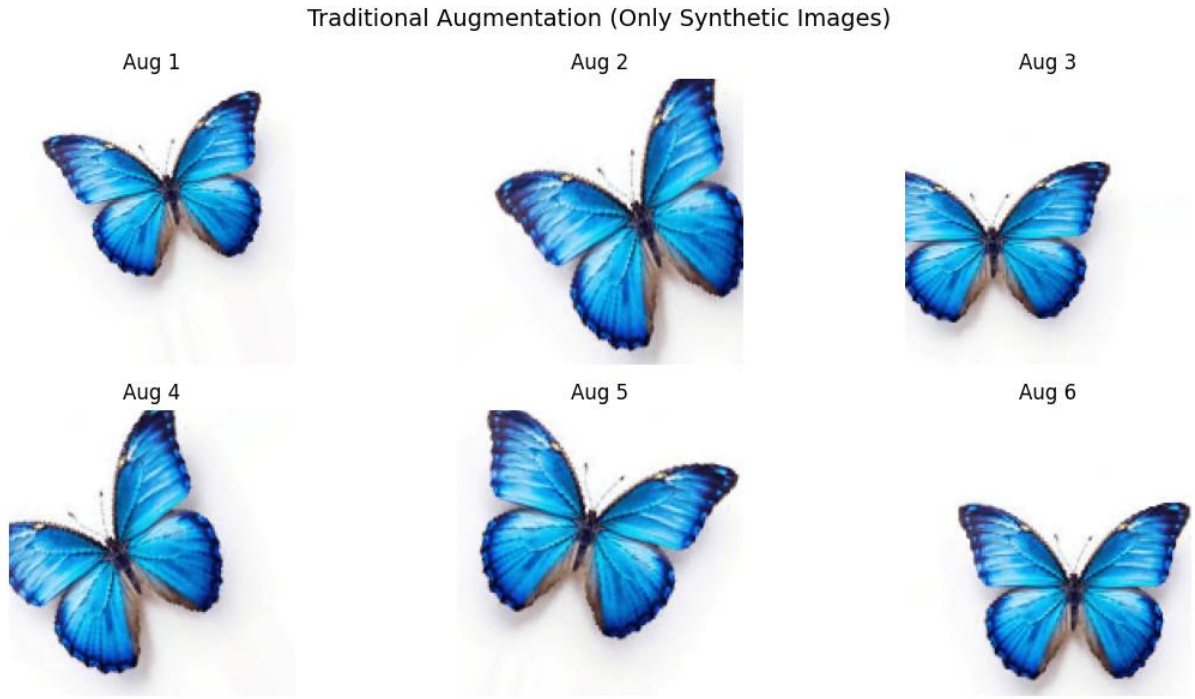
STEP 11: Stop the process.

CODING:

```
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.image import ImageDataGenerator, img_to_array, load_img
import numpy as np
# Load image
img = load_img("/content/download (1).jpeg", target_size=(150, 150))
x = img_to_array(img)
x = np.expand_dims(x, axis=0)
# Define augmentation
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,
    fill_mode='nearest'
)
# Iterator
aug_iter = datagen.flow(x, batch_size=1)
# Display 6 augmented samples (NO original shown here)
plt.figure(figsize=(12, 6))
for i in range(6):
    batch = next(aug_iter)
    aug_img = batch[0].astype('uint8')
    plt.subplot(2, 3, i+1)
    plt.imshow(aug_img)
    plt.axis("off")
    plt.title(f"Aug {i+1}")
```

```
plt.suptitle("Traditional Augmentation (Only Synthetic Images)", fontsize=14)
plt.tight_layout()
plt.show()
```

OUTPUT:



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

RESULT:

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

EX.NO:04

DATE:

IMAGE DATA GENERATOR FOR IMAGE DATA AUGMENTATION

AIM:

To Demonstrate the role of Image Data Generator class in data augmentation.

ALGORITHM:

STEP 1: Start the process.

STEP 2 Import the required libraries:

- matplotlib.pyplot for visualization
- tensorflow.keras.preprocessing.image for image loading and augmentation
- numpy for array manipulation.

STEP 3: Load the original image using load_img() and resize it to (150, 150) pixels.

STEP 4: Convert the loaded image to a NumPy array using img_to_array().

STEP 5: Expand the dimensions of the array to create a batch of size 1 using np.expand_dims().

STEP 6: Define an ImageDataGenerator with augmentation parameters such as:

- rotation_range for random rotations
- zoom_range for random zoom
- brightness_range for adjusting image brightness
- horizontal_flip to flip images randomly
- fill_mode='nearest' to fill in new pixels

STEP 7: Create an iterator from the batch using datagen.flow() to generate augmented images.

STEP 8: Initialize a matplotlib figure for displaying images.

STEP 9: Plot the original image in the first subplot.

STEP 10: Loop through the iterator to generate 5 augmented images, convert each to uint8, and plot them in subsequent subplots.

STEP 11: Add titles for each subplot and a main figure title, then display the figure using plt.show().

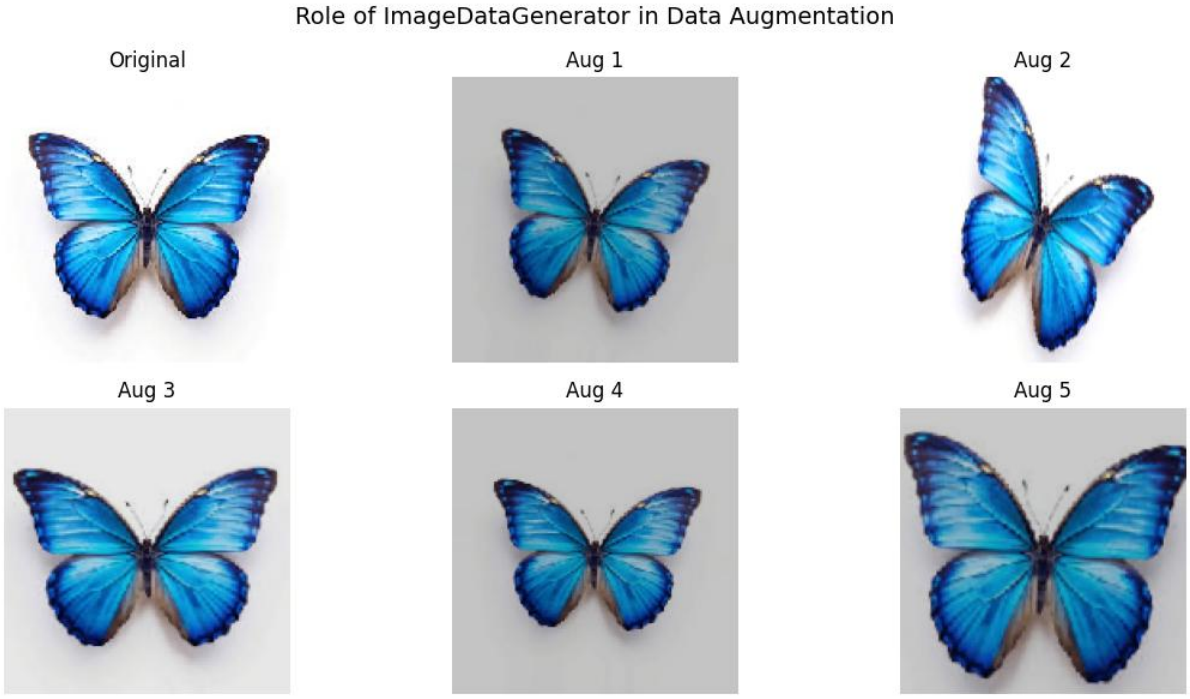
STEP 12: Stop the Process.

CODING:

```
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.image import ImageDataGenerator, img_to_array, load_img
import numpy as np
# Load image
img = load_img("/download (1).jpeg", target_size=(150, 150))
x = img_to_array(img)
x = np.expand_dims(x, axis=0)
# Define augmentation with more variety
datagen = ImageDataGenerator(
    rotation_range=30,
    zoom_range=0.3,
    brightness_range=[0.7, 1.3],
    horizontal_flip=True,
    fill_mode='nearest'
)
aug_iter = datagen.flow(x, batch_size=1)
# Display Original + 5 Augmented
plt.figure(figsize=(12, 6))
plt.subplot(2, 3, 1)
plt.imshow(img)
plt.axis("off")
plt.title("Original")
for i in range(5):
    batch = next(aug_iter)
    aug_img = batch[0].astype('uint8')
    plt.subplot(2, 3, i+2)
    plt.imshow(aug_img)
    plt.axis("off")
    plt.title(f"Aug {i+1}")
```

```
plt.suptitle("Role of ImageDataGenerator in Data Augmentation", fontsize=14)
plt.tight_layout()
plt.show()
```

OUTPUT:



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

RESULT:

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

EX.NO:05

DATE:

CNN MODEL FOR IMAGE CLASSIFICATION

AIM:

To Implement a CNN process for image classification.

ALGORITHM:

STEP 1: Start the process.

STEP 2: Import required libraries: tensorflow (for CNN), matplotlib (for plotting), and Keras modules (datasets, layers, models).

STEP 3: Load the CIFAR-10 dataset using datasets.cifar10.load_data(), which returns training and testing images and labels.

STEP 4: Normalize image pixel values to the range **0–1** by dividing by 255.0.

STEP 5: Define class names for CIFAR-10: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck.

STEP 6: Build the CNN model using Sequential with layers:

- Conv2D + ReLU followed by MaxPooling2D (first block)
- Conv2D + ReLU followed by MaxPooling2D (second block)
- Conv2D + ReLU then Flatten (third block)
- Dense (128 neurons + ReLU) + Dropout (0.5)
- Dense output layer (10 neurons) for 10 classes

STEP 7: Compile the model using:

- Optimizer: Adam
- Loss function: Sparse Categorical Crossentropy (from logits)
- Metric**:** Accuracy

STEP 8: Train the model using fit() for 10 epochs with a batch size of 64, providing validation data as the test set.

STEP 9: Evaluate the model on the test set using evaluate() and print the **test accuracy**.

STEP 10: Plot training and validation accuracy per epoch using matplotlib to visualize learning progress.

STEP 11: Analyze the plot to check for underfitting or overfitting and performance on unseen data.

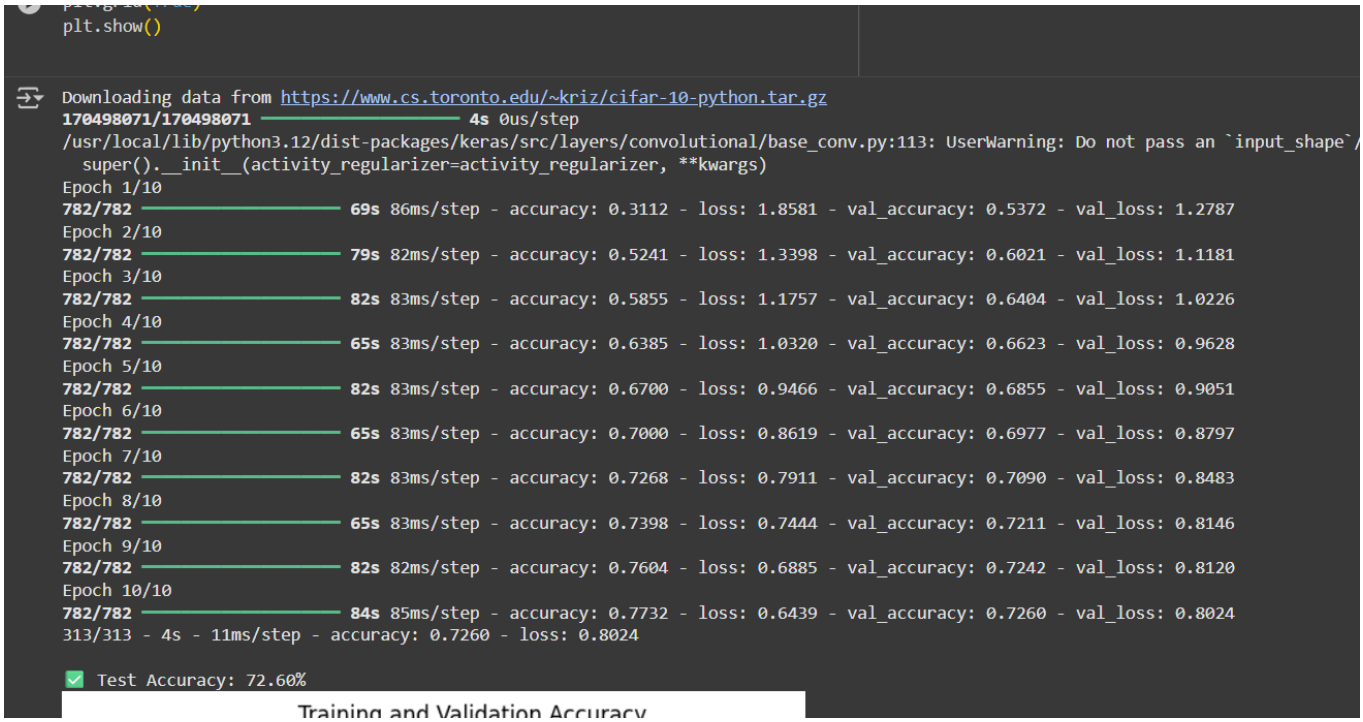
STEP 12: Stop the Process.

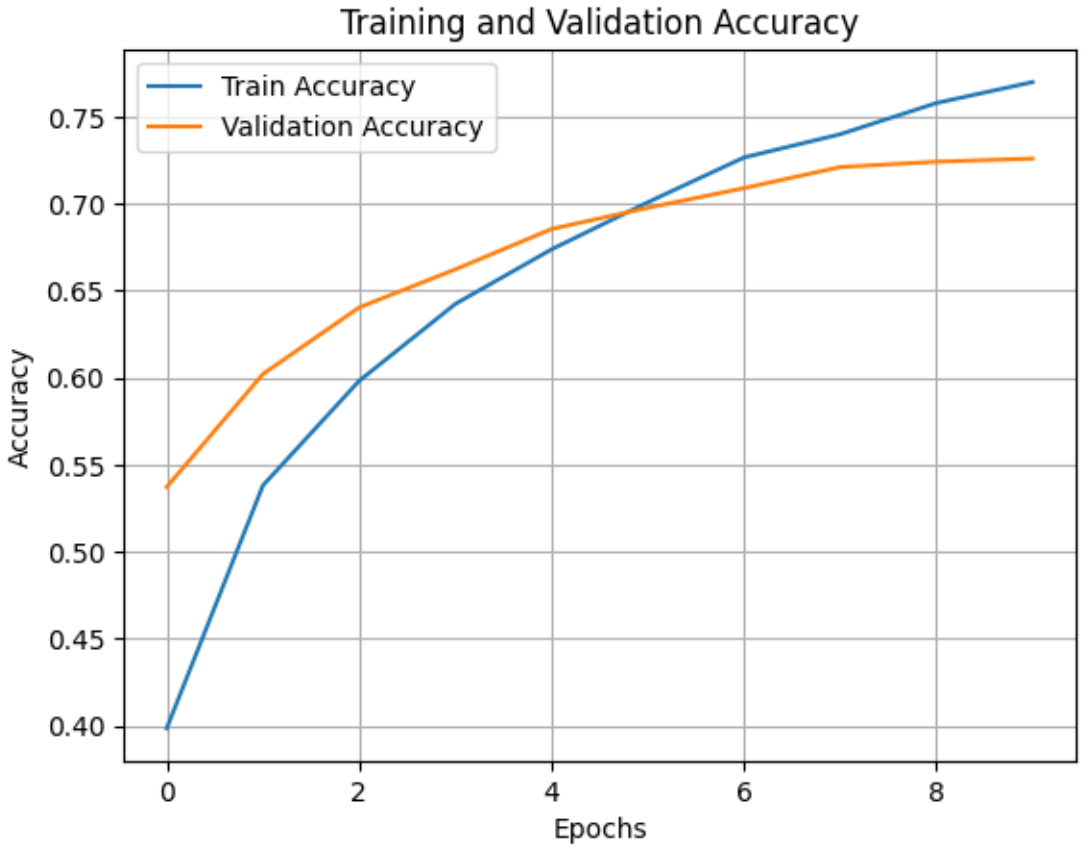
CODING:

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
#Load CIFAR-10 dataset
(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 for CIFAR-10
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']
#Build CNN Model
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(128, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(10)  # 10 classes for CIFAR-10
])
#Compile model
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
#Train model
history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels),
                    batch_size=64)
```

```
#Evaluate model
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print(f"\nTest Accuracy: {test_acc * 100:.2f}%")
#Plot Training & Validation Accuracy
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:





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

RESULT:

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

EX.NO:06

DATE:

RNN ARCHITECTURE FOR TIME SERIES PREDICTION

AIM:

To Demonstrate the RNN architecture for time series data.

ALGORITHM:

STEP 1: Start the process.

STEP 2: Import required libraries: numpy for data handling, matplotlib.pyplot for plotting, and tensorflow.keras for building the RNN.

STEP 3: Generate synthetic sine wave data using a function:

- For each sample, randomly select a start point.
- Create a sequence of length seq_length + 1.
- Use the first seq_length values as input X and the last value as target y.

STEP 4: Reshape input X to 3D shape (samples, timesteps, features) to make it compatible with RNN input.

STEP 5: Build a Simple RNN model using Sequential:

- RNN layer with 64 units and tanh activation
- Dense layer with 1 neuron to predict the next value in the sequence

STEP 6: Compile the model using:

- Optimizer: Adam
- Loss function: Mean Squared Error (MSE)

STEP 7: Train the model on the generated data using fit(), with 20% validation split, batch size of 32, and a set number of epochs (10).

STEP 8: Use the trained model to predict the next value for the first 10 samples.

STEP 9: Print a few true vs predicted values to check performance.

STEP 10: Plot training and validation loss per epoch using matplotlib to visualize model convergence.

STEP 11: Analyze the plot to assess learning, overfitting, or underfitting trends.

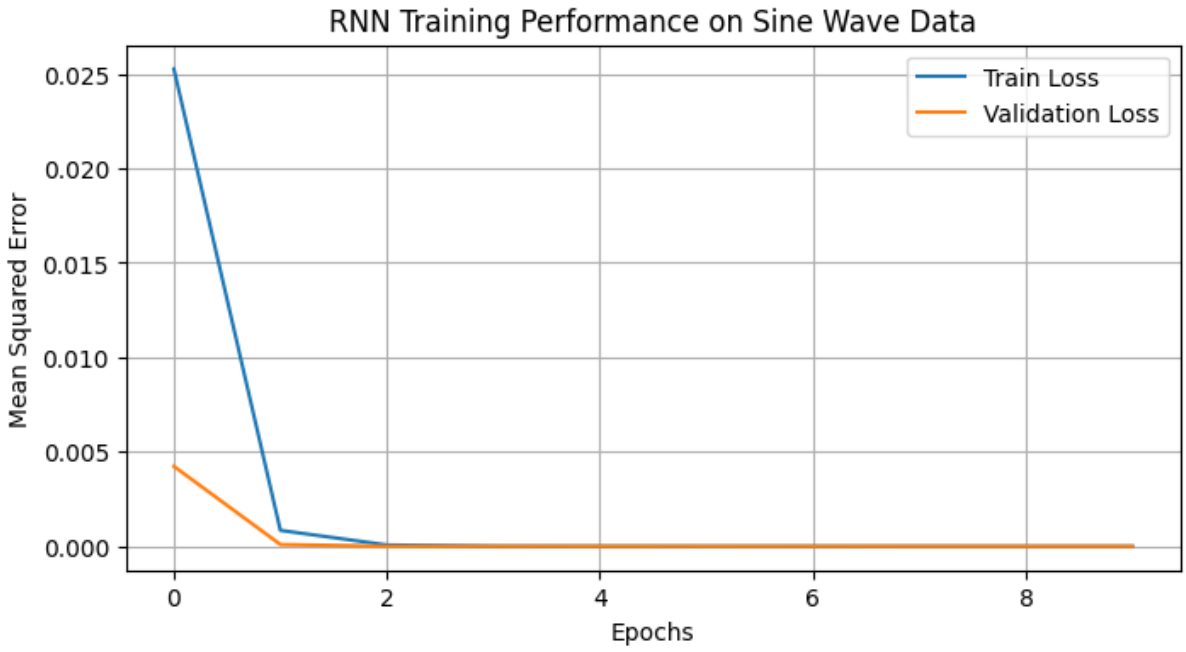
STEP 12: Stop the Process.

CODING:

```
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
#Generate synthetic sine wave data
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)
# Reshape for RNN: (samples, timesteps, features)
X = X.reshape((num_samples, seq_length, 1))
#Build RNN model
model = Sequential([
    SimpleRNN(64, activation='tanh', input_shape=(seq_length, 1)),
    Dense(1)  # Predict next value in the sequence])
model.compile(optimizer='adam', loss='mse')
#Train model
history = model.fit(X, y,
    epochs=10,          # slightly more epochs
    batch_size=32,
```

```
    validation_split=0.2,
    verbose=1)
#Predict on first 10 samples
predictions = model.predict(X[:10])
print("\nSample Predictions:")
for i in range(5):
    print(f"True: {y[i]:.3f}, Predicted: {predictions[i][0]:.3f}")
#Plot training & validation loss
plt.figure(figsize=(8,4))
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel("Epochs")
plt.ylabel("Mean Squared Error")
plt.legend()
plt.title("RNN Training Performance on Sine Wave Data")
plt.grid(True)
plt.show()
```

OUTPUT:



```
Epoch 1/10
/usr/local/lib/python3.12/dist-packages/keras/src/layers/rnn/rnn.py:199: UserWarning: Do not pass an `inpu
  super().__init__(**kwargs)
25/25 ━━━━━━━━━━━━━━━━━━━━ 2s 23ms/step - loss: 0.0617 - val_loss: 0.0042
Epoch 2/10
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - loss: 0.0015 - val_loss: 9.7148e-05
Epoch 3/10
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 1.0834e-04 - val_loss: 6.7934e-06
Epoch 4/10
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - loss: 1.0450e-05 - val_loss: 4.4337e-06
Epoch 5/10
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 5.0254e-06 - val_loss: 5.0871e-06
Epoch 6/10
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - loss: 3.0573e-06 - val_loss: 1.8856e-06
Epoch 7/10
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step - loss: 2.2178e-06 - val_loss: 3.1610e-06
Epoch 8/10
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 2.0146e-06 - val_loss: 1.0110e-06
Epoch 9/10
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 1.0331e-06 - val_loss: 7.5627e-07
Epoch 10/10
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 8.0946e-07 - val_loss: 1.2079e-06
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 154ms/step

Sample Predictions:
True: -0.468, Predicted: -0.468
True: 0.777, Predicted: 0.778
True: 0.520, Predicted: 0.521
True: 0.340, Predicted: 0.342
```

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

RESULT:

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

EX.NO:07

DATE:

TEXT ANALYSIS USING NATURAL LANGUAGE PROCESSING

AIM:

To Construct the steps to deal with text analysis using NLP.

ALGORITHM:

STEP 1: Start the process.

STEP 2: Import required libraries: pandas for data handling,re for regex text cleaning, nltk for stopwords removal, sklearn.feature_extraction.text.TfidfVectorizer for feature extraction, matplotlib.pyplot and seaborn for plotting, wordcloud for visualizing important terms.

STEP 3: Load or create the text dataset as a Pandas DataFrame.

STEP 4: Download NLTK stopwords and define a preprocessing function:

- Convert text to lowercase
- Remove non-alphabetic characters using regex
- Remove stopwords
- Tokenize and join back into cleaned text

STEP 5: Apply the preprocessing function to all documents to create a cleaned text column.

STEP 6: Initialize a TF-IDF Vectorizer with:

- ngram_range = (1,2) to include unigrams and bigrams
- max_features =50 to limit number of features

STEP 7: Fit the vectorizer on cleaned text and transform the documents into TF-IDF feature vectors.

STEP 8: Extract top TF-IDF terms for each document by sorting the TF-IDF scores and printing the top 5 terms per document.

STEP 9: Create a heatmap of TF-IDF scores (documents vs features) using Seaborn to visualize term importance across documents.

STEP 10: Compute the aggregate TF-IDF scores across all documents to get global term importance.

STEP 11: Generate a WordCloud from TF-IDF weighted terms to visualize the most significant words across the dataset.

STEP 12: Stop the Process.

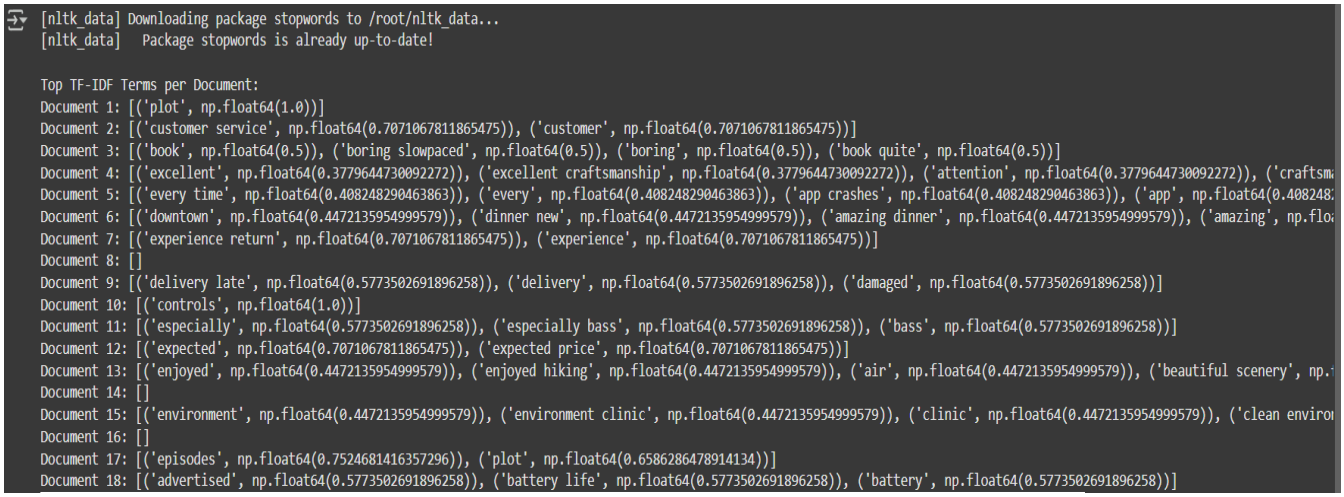
CODING:

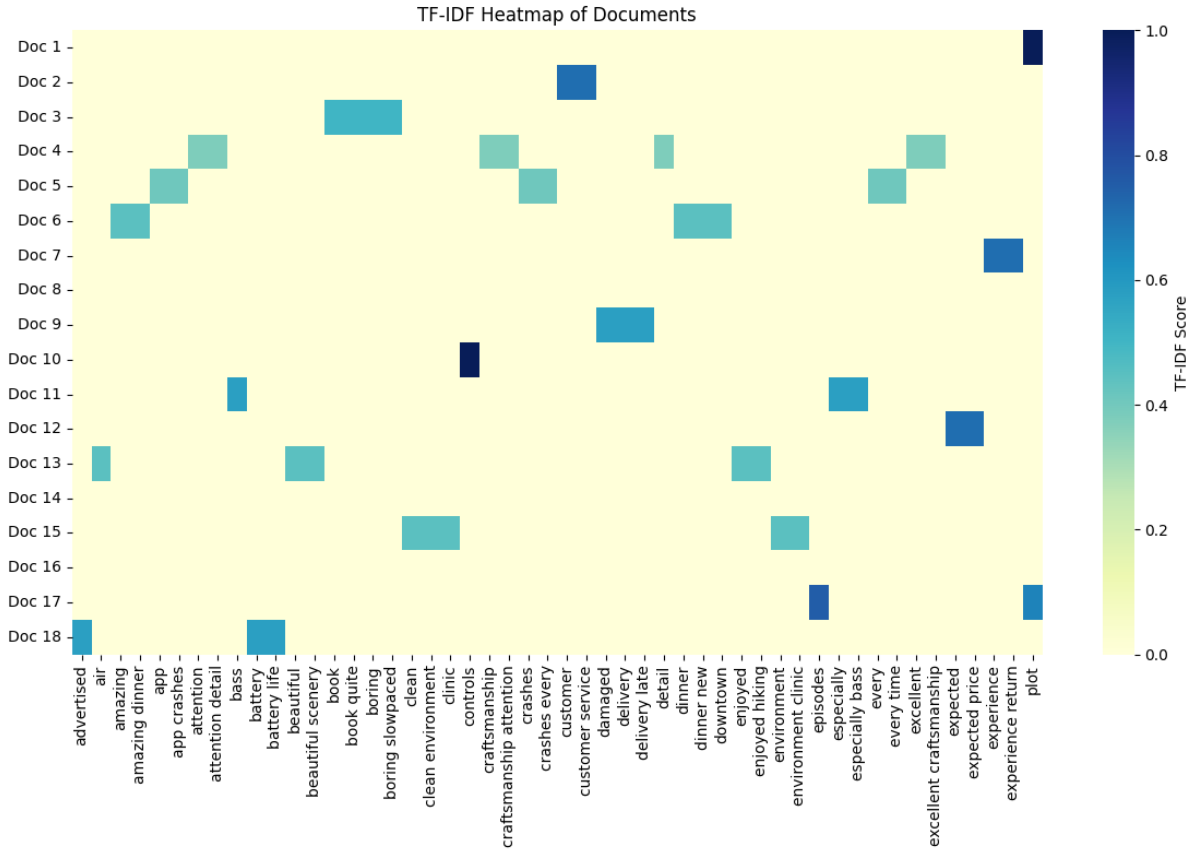
```
import pandas as pd
import re
import nltk
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
import matplotlib.pyplot as plt
import seaborn as sns
nltk.download('stopwords')
#Sample Data
data = {
    'text': [
        "The movie had stunning visuals but 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!",
```

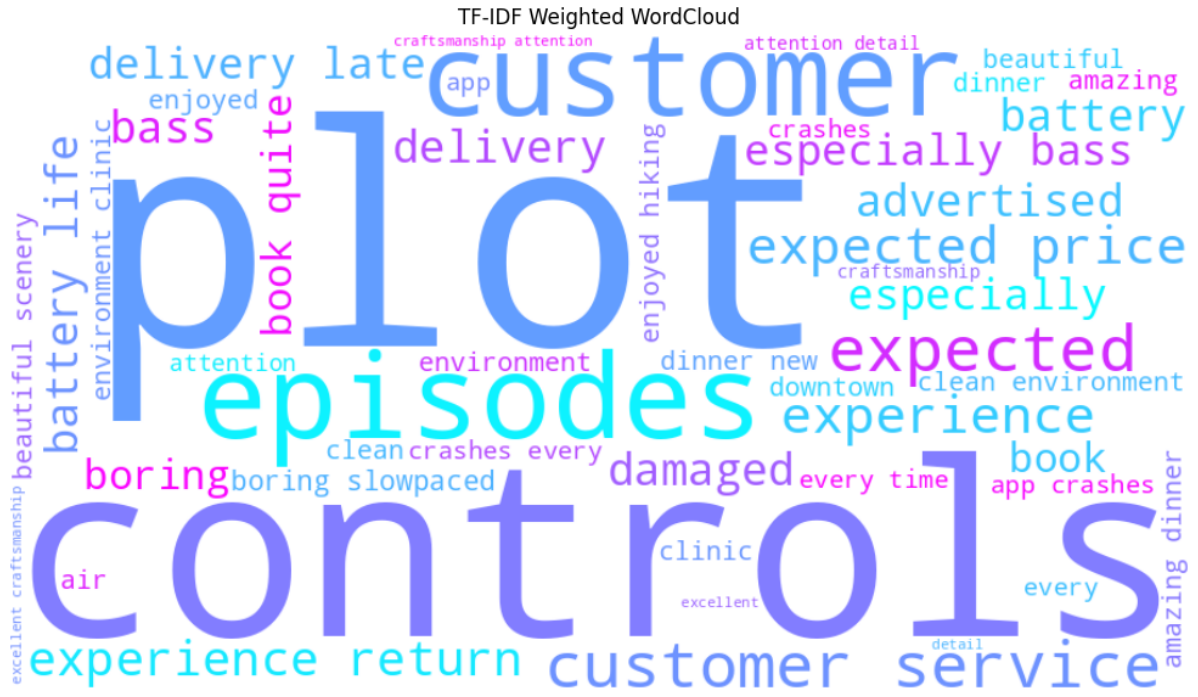
```
        "Battery life is shorter than advertised."
    ]
}
df = pd.DataFrame(data)
#Text Preprocessing
stop_words = set(stopwords.words('english'))
def preprocess(text):
    text = text.lower()
    text = re.sub(r'[^a-z\s]', '', text)
    tokens = [w for w in text.split() if w not in stop_words]
    return ' '.join(tokens)
df['clean_text'] = df['text'].apply(preprocess)
#TF-IDF with unigrams and bigrams
vectorizer = TfidfVectorizer(ngram_range=(1,2), max_features=50)
X = vectorizer.fit_transform(df['clean_text'])
feature_names = vectorizer.get_feature_names_out()
#Show top TF-IDF terms for each document
print("\nTop TF-IDF Terms per Document:")
for i, doc in enumerate(df['clean_text']):
    scores = X[i].toarray()[0]
    top_idx = scores.argsort()[-5:][::-1]
    top_terms = [(feature_names[idx], scores[idx]) for idx in top_idx if scores[idx] > 0]
    print(f"Document {i+1}: {top_terms}")
#Heatmap of TF-IDF (documents vs top features)
plt.figure(figsize=(12,8))
sns.heatmap(X.toarray(), cmap='YlGnBu', yticklabels=[f'Doc {i+1}' for i in
range(X.shape[0])],
            xticklabels=feature_names, cbar_kws={'label':'TF-IDF Score'})
plt.xticks(rotation=90)
plt.title("TF-IDF Heatmap of Documents")
plt.tight_layout()
```

```
plt.show()
#WordCloud of most important terms across all documents
from wordcloud import WordCloud
import numpy as np
tfidf_sum = np.sum(X.toarray(), axis=0)
tfidf_dict = {feature_names[i]: tfidf_sum[i] for i in range(len(feature_names))}
wordcloud = WordCloud(width=900, height=500, background_color='white',
                      colormap='cool', max_words=50).generate_from_frequencies(tfidf_dict)
plt.figure(figsize=(14,7))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title("TF-IDF Weighted WordCloud")
plt.show()
```

OUTPUT:




TF-IDF Heatmap of Documents
Doc 1
Doc 2
Doc 3
Doc 4
Doc 5
Doc 6
Doc 7
Doc 8
Doc 9
Doc 10
Doc 11
Doc 12
Doc 13
Doc 14
Doc 15
Doc 16
Doc 17
Doc 18
advertised
air
amazing
amazing dinner
app
app crashes
attention
attention detail
bass
battery
battery life
beautiful
beautiful scenery
book
book quite
boring
boring slowpaced
clean
clean environment
clinic
controls
craftsmanship
craftsmanship attention
crashes
crashes every
customer
customer service
damaged
delivery
delivery late
detail
dinner
dinner new
downtown
enjoyed
enjoyed hiking
environment
environment clinic
episodes
especially
especially bass
every
every time
excellent
excellent craftsmanship
expected
expected price
experience
experience return
plot
TF-IDF Score
1.0
0.8
0.6
0.4
0.2
0.0


TF-IDF Weighted WordCloud
craftsmanship attention
attention detail
delivery late
customer
app
beautiful
dinner
amazing
enjoyed
battery
bass
crashes
delivery
especially bass
book quite
enjoyed hiking
advertised
plot
expected price
craftsmanship
especially
environment clinic
battery life
beautiful scenery
attention
environment
dinner new
expected
episodes
downtown
clean environment
experience
clean
crashes every
book
boring
boring slowpaced
damaged
every time
app crashes
controls
clinic
amazing dinner
air
every
excellent
excellent craftsmanship
detail
experience return
customer service

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

RESULT:

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

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.

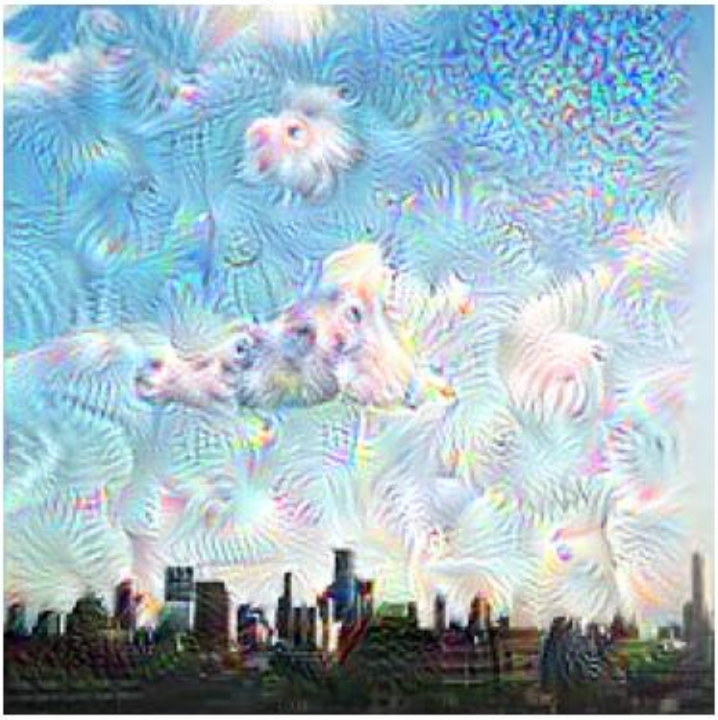
CODING:

```
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)
```

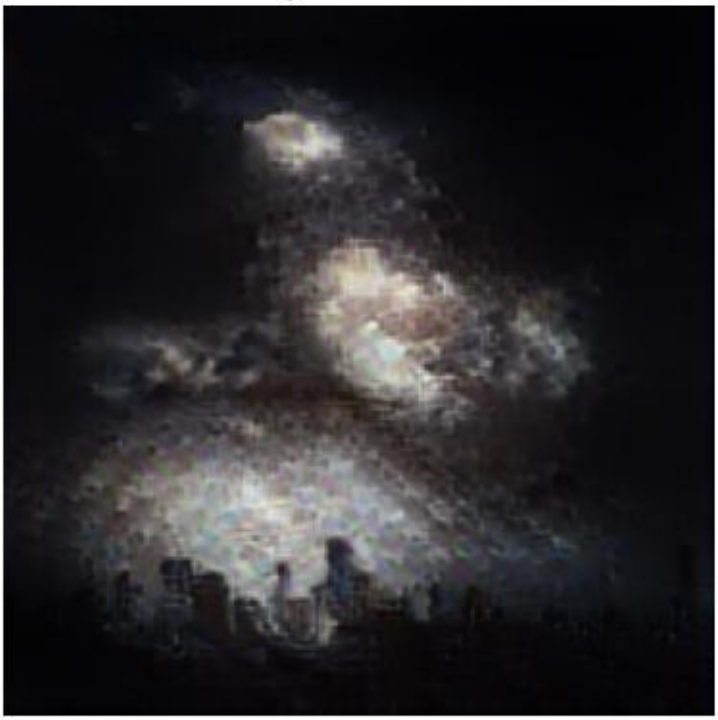
```
        grads = grads / (tf.math.reduce_std(grads) + 1e-8)
        image = image + grads * step_size
        image = tf.clip_by_value(image, 0.0, 1.0)
    return image
# Load InceptionV3 for Deep Dream
base_model = tf.keras.applications.InceptionV3(include_top=False, weights="imagenet")
dream_layers = ['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")
```

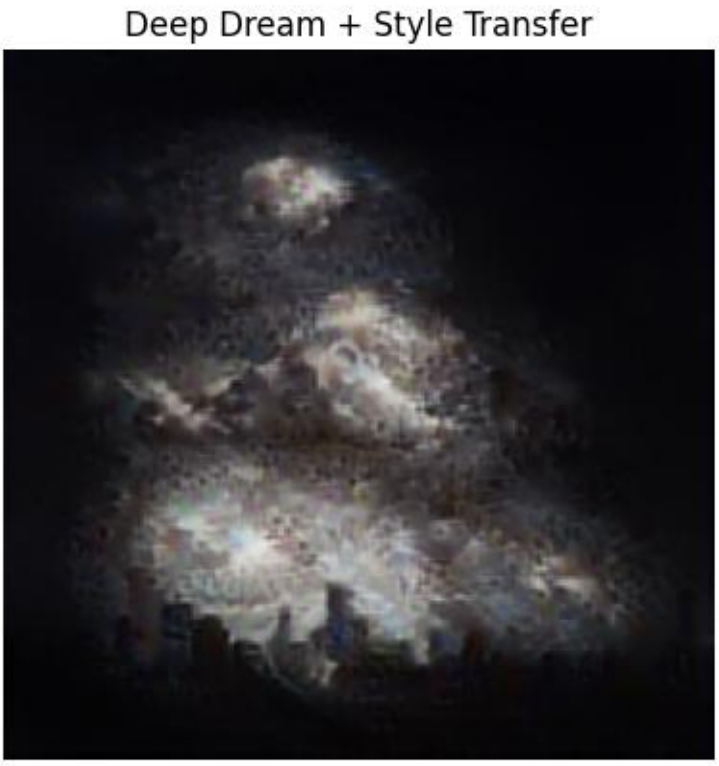
OUTPUT:

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:

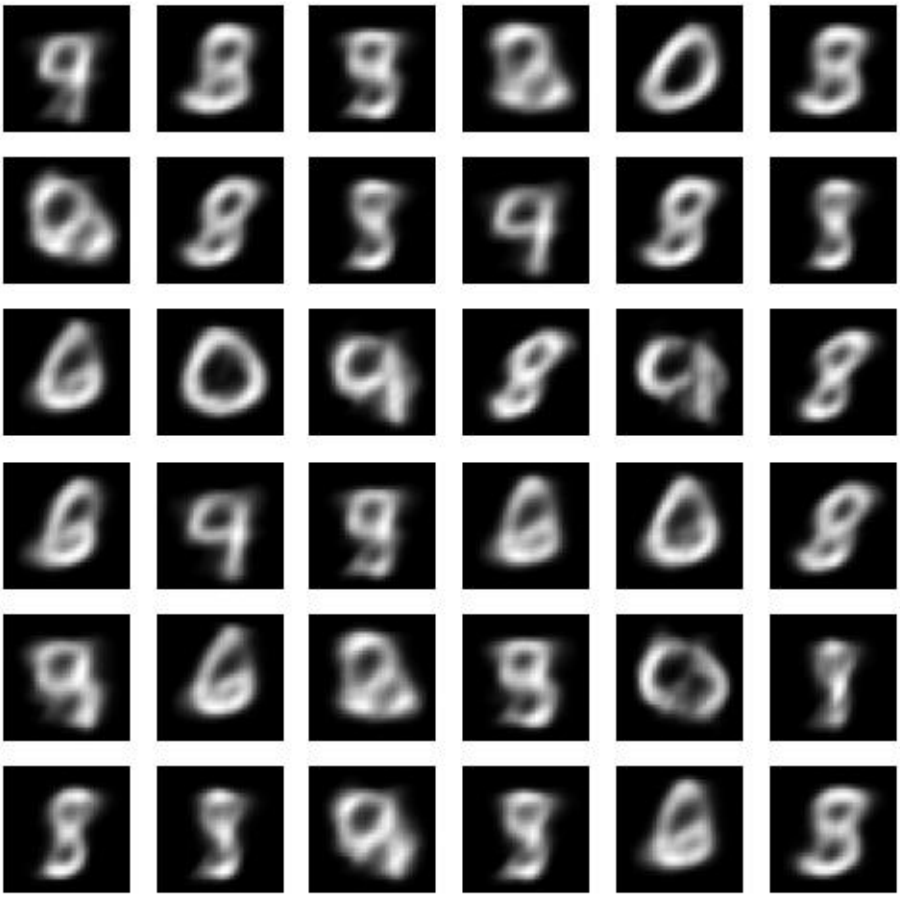
```
import numpy as np

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.

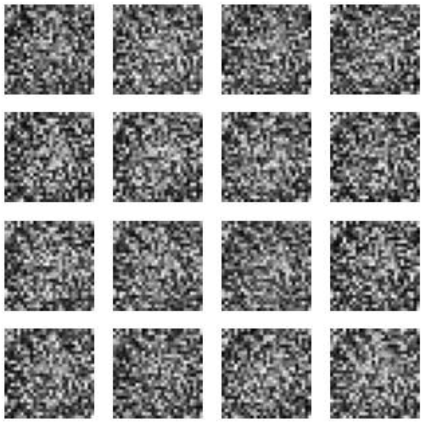
CODING:

```
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
```

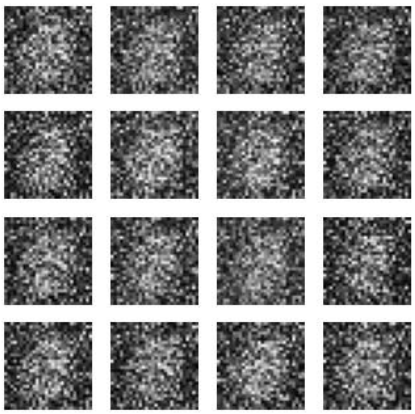
```
def build_discriminator():
    model = tf.keras.Sequential([
        layers.Flatten(input_shape=(28,28,1)),
        layers.Dense(128, activation="relu"),
        layers.Dense(1, activation="sigmoid")
    ])
    return model
generator = build_generator()
discriminator = build_discriminator()
#Loss & 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)
```

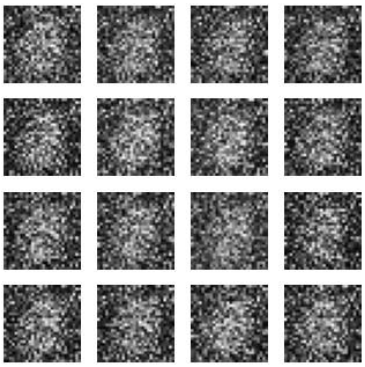
OUTPUT:



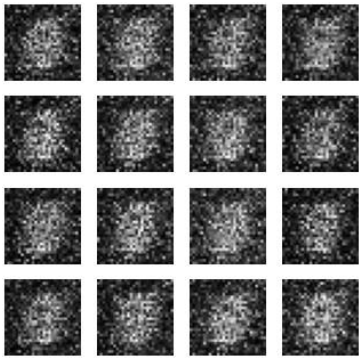
Epoch 1/10 | Gen Loss: 1.1682 | Disc Loss: 0.2248



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.