PRACTICAL JOURNAL

in

Advanced Artificial Intelligence

&

Machine Learning

Submitted to

Laxman Devram Sonawane College, Kalyan (W) 421301

in partial fulfillment for the award of the degree of

Master of Science in Information Technology

(Affiliated to Mumbai University)

Submitted by

KARAN MAHESH KATUDIYA

Under the guidance of

Mrs. Sabina Ansari (Advanced Artificial Intelligence)

&

Dr. Priyanka Pawar (Machine Learning)

Department of Information Technology

Kalyan, Maharashtra

Academic Year 2024-25

PRACTICAL JOURNAL

in

Advance Artificial Intelligence

Submitted to

Laxman Devram Sonawane College, Kalyan (W) 421301

in partial fulfilment for the award of the degree of

Master of Science in Information Technology

(Affiliated to Mumbai University)

Submitted by

Karan Mahesh Katudiya

Under the guidance of

Mrs. Sabina Ansari

Department of Information Technology
Kalyan, Maharashtra

Academic Year 2024-25

The Kalyan Wholesale Merchants Education Society's

Laxman Devram Sonawane College,

Kalyan (W) 421301

Department of Information Technology
Masters of Science – Part II

Certificate

This is to certify that **Mr. Karan Mahesh Katudiya**, Seat number______________, studying in Masters of Science in Information Technology Part II , Semester II has satisfactorily completed the practical of "**Advanced Artificial Intelligence** " as prescribed by University of Mumbai, during the academic year 2024-25.

Subject In-charge	Coordinator In-charge	ExternalExaminer

College Seal

ADVANCED ARTIFICIAL INTELLIGENCE

INDEX

13	Use Python libraries such as GPT-2 or textgenrnn to train generative models on a corpus of text data and generate new text based on the patterns it has learned.	
14	Experiment with neural networks like GANs (Generative Adversarial Networks) using Python libraries like TensorFlow or PyTorch to generate new images based on a dataset of images.	

Practical 1

Aim : Implementing advanced deep learning algorithms such as convolutional neural networks (CNNs) or recurrent neural networks (RNNs) using Python libraries like TensorFlow or PyTorch

Code :

```
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt

# Load and preprocess the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0  # Normalize pixel values to [0, 1]

# Build the 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(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')  # 10 classes for CIFAR-10
])

# Compile the model
```

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

# Train the model
history = model.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))

# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f'\nTest accuracy: {test_acc}')

# Plot training & validation accuracy values
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend()
plt.show()

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

# Define transformations for the training and testing data
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
```

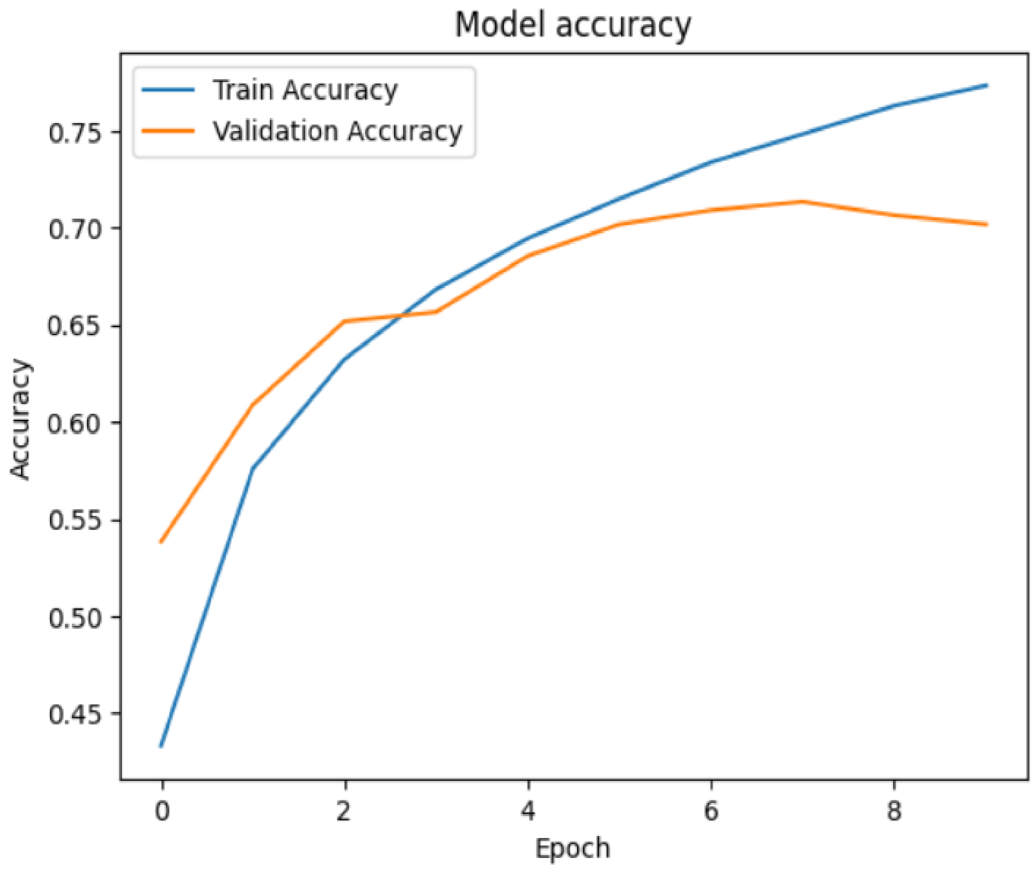
])

# Load the CIFAR-10 dataset

trainset = torchvision

OUTPUT :

```
Test accuracy: 0.7017999887466431
```



Practical 2

Aim : Building a natural language processing (NLP) model for sentiment analysis or text classification.

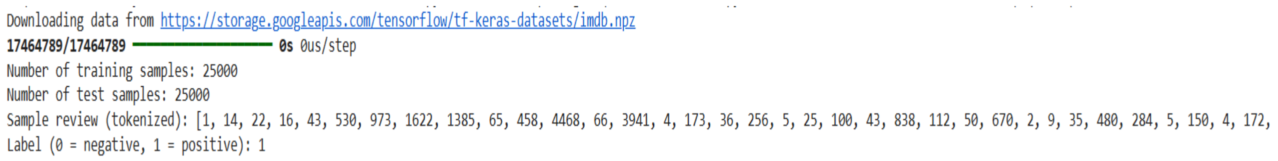
Code :

```
!pip install tensorflow matplotlib


import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.datasets import imdb
import matplotlib.pyplot as plt

# Step 1: Load the IMDB dataset
num_words = 10000  # Use the top 10,000 words in the vocabulary
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=num_words)

# Step 2: Explore the dataset
print(f"Number of training samples: {len(x_train)}")
print(f"Number of test samples: {len(x_test)}")
print(f"Sample review (tokenized): {x_train[0]}")
print(f"Label (0 = negative, 1 = positive): {y_train[0]}")

# Step 3: Decode a sample review
word_index = imdb.get_word_index()
reverse_word_index = {value: key for key, value in word_index.items()}
decoded_review = " ".join([reverse_word_index.get(i - 3, "?") for i in x_train[0]])
```

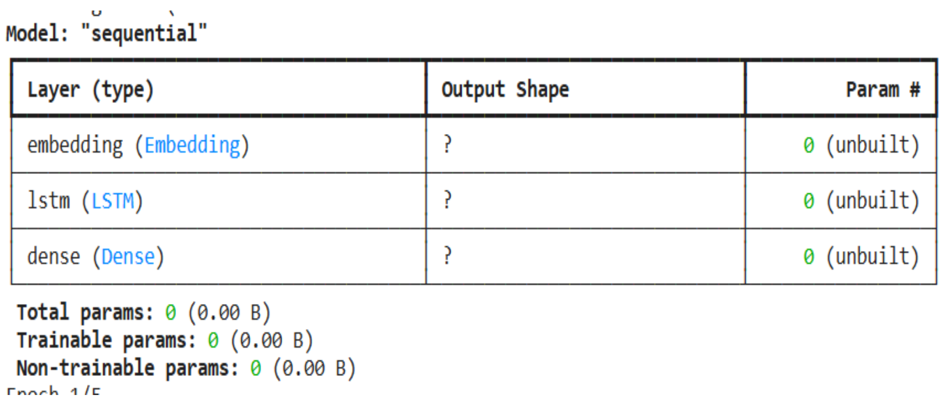
```
print(f"Decoded review: {decoded_review}")

# Step 4: Pad sequences
maxlen = 200  # Limit each review to 200 words
x_train = pad_sequences(x_train, maxlen=maxlen)
x_test = pad_sequences(x_test, maxlen=maxlen)

# Step 5: Define the model
model = models.Sequential([
    layers.Embedding(input_dim=num_words, output_dim=32, input_length=maxlen),
    layers.LSTM(32),  # Use an LSTM layer for capturing sequential dependencies
    layers.Dense(1, activation='sigmoid')  # Output layer for binary classification
])

# Step 6: Compile the model
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Step 7: Display the model architecture
model.summary()

# Step 8: Train the model
history = model.fit(x_train, y_train, epochs=5, batch_size=64, validation_split=0.2)

# Step 9: Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f"Test Accuracy: {test_acc}")

# Step 10: Plot training history
plt.figure(figsize=(12, 4))
```

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

# Loss plot
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Model Loss')
plt.show()
```

OUTPUT :

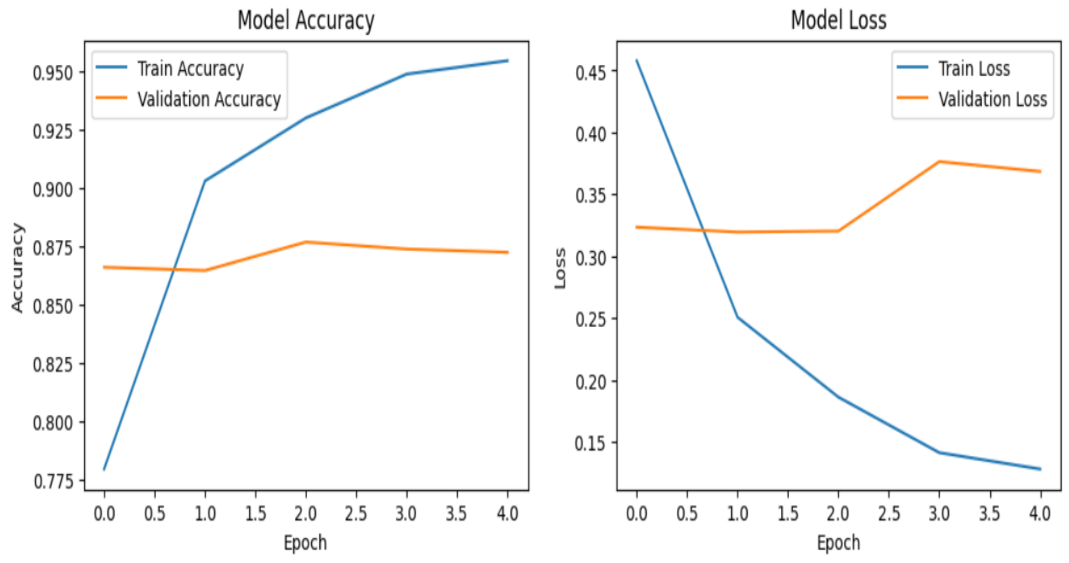


```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz
17464789/17464789 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Number of training samples: 25000
Number of test samples: 25000
Sample review (tokenized): [1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5, 25, 100, 43, 838, 112, 50, 670, 2, 9, 35, 480, 284, 5, 150, 4, 172,
Label (0 = negative, 1 = positive): 1
```



```
Model: "sequential"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ embedding (Embedding)           │ ?                      │   0 (unbuilt) │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ lstm (LSTM)                     │ ?                      │   0 (unbuilt) │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense (Dense)                   │ ?                      │   0 (unbuilt) │
└─────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 0 (0.00 B)
 Trainable params: 0 (0.00 B)
 Non-trainable params: 0 (0.00 B)
Epoch 1/5
```

```
Test Accuracy: 0.8611999750137329
```



Practical 3

Aim : Creating a chatbot using advanced techniques like transformer models.

Code :

```
!pip install transformers torch

from transformers import AutoModelForCausalLM, AutoTokenizer
import torch

# Step 1: Load Pre-trained Model and Tokenizer
print("Loading the DialoGPT model...")
model_name = "microsoft/DialoGPT-medium"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)

# Step 2: Initialize Chat History
chat_history_ids = None
step = 0

# Step 3: Chat with the User
print("Chatbot is ready! Type 'exit' to end the chat.\n")

while True:
    # User input
    user_input = input("You: ")
    if user_input.lower() == 'exit':
        print("Chatbot: Goodbye!")
        break
```

# Encode the user input and add it to the chat history

new_input_ids = tokenizer.encode(user_input + tokenizer.eos_token, return_tensors='pt')

chat_history_ids = torch.cat([chat_history_ids, new_input_ids], dim=-1) if step > 0 else new_input_ids

# Generate a response using the model

response_ids = model.generate(chat_history_ids, max_length=1000, pad_token_id=tokenizer.eos_token_id)

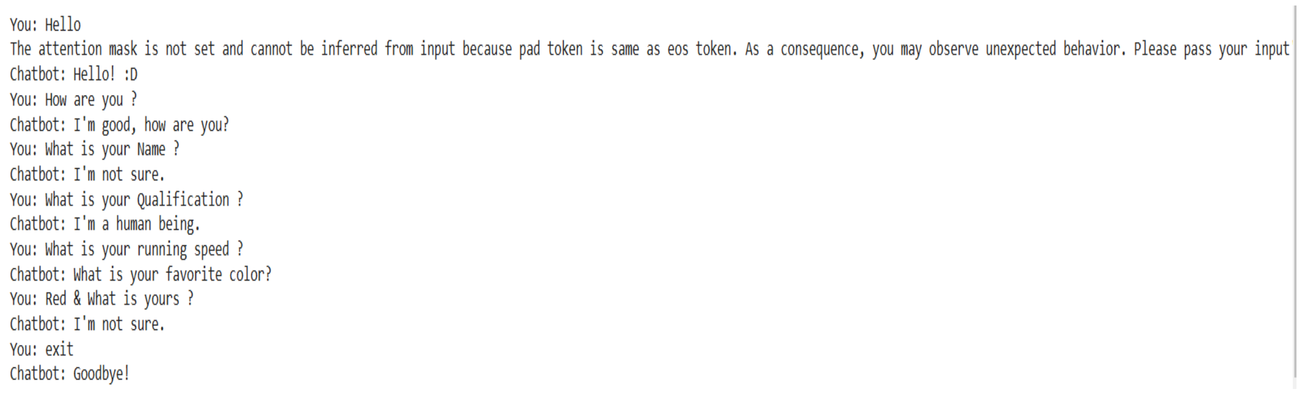
response = tokenizer.decode(response_ids[:, chat_history_ids.shape[-1]:][0], skip_special_tokens=True)

# Display the response

print(f"Chatbot: {response}")

step += 1

Output :



```
You: Hello
The attention mask is not set and cannot be inferred from input because pad token is same as eos token. As a consequence, you may observe unexpected behavior. Please pass your input
Chatbot: Hello! :D
You: How are you ?
Chatbot: I'm good, how are you?
You: What is your Name ?
Chatbot: I'm not sure.
You: What is your Qualification ?
Chatbot: I'm a human being.
You: What is your running speed ?
Chatbot: What is your favorite color?
You: Red & What is yours ?
Chatbot: I'm not sure.
You: exit
Chatbot: Goodbye!
```

Practical 4

Aim : Developing a recommendation system using collaborative filtering or deep learning approaches.

Code :

Step 1: Install Required Libraries

Run the following command to install the necessary libraries:

```
pip install tensorflow numpy pandas matplotlib
```

Step 2: Download the Dataset

Download the MovieLens 100K dataset from grouplens.org/datasets/movielens. Extract the dataset into a folder.

Alternatively, the code below assumes that the u.data file is in the ml-100k folder.

Step 3: Python Code for the Recommendation System

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# Step 1: Load and preprocess the dataset
file_path = "ml-100k/u.data"
column_names = ['user_id', 'item_id', 'rating', 'timestamp']
data = pd.read_csv(file_path, sep='\t', names=column_names)

# Normalize user IDs and item IDs to start from 0
data['user_id'] -= 1
```

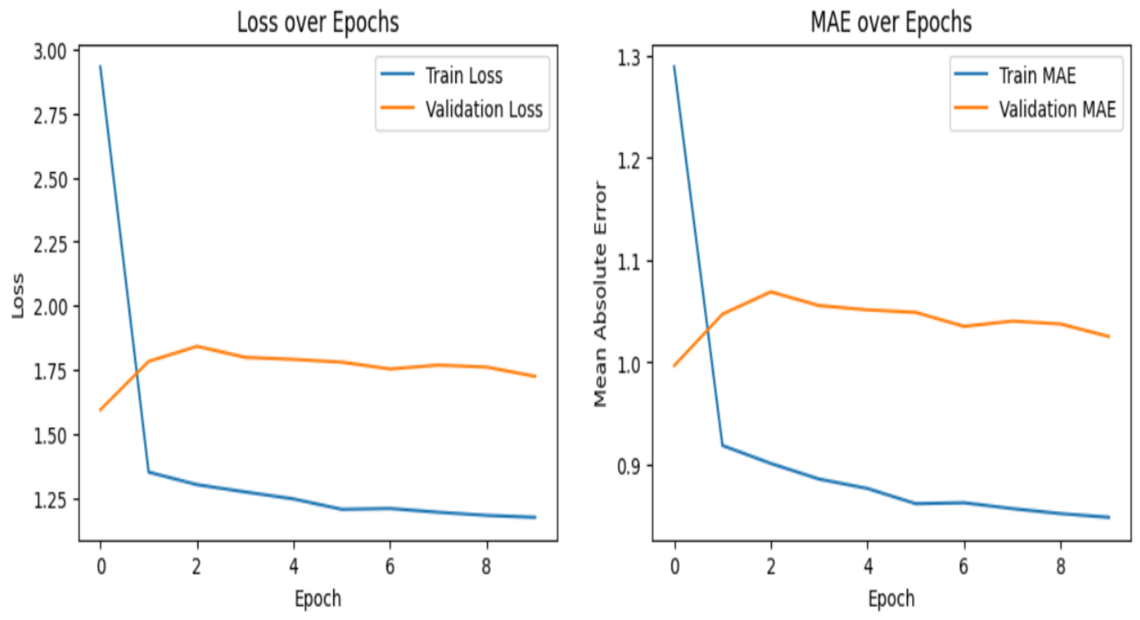
```
data['item_id'] -= 1

# Extract the number of users and items
num_users = data['user_id'].max() + 1
num_items = data['item_id'].max() + 1
print(f"Number of users: {num_users}, Number of items: {num_items}")

# Step 2: Split data into training and testing
train_data, test_data = train_test_split(data, test_size=0.2, random_state=42)

# Step 3: Create TensorFlow datasets
def create_tf_dataset(df):
    users = tf.constant(df['user_id'].values, dtype=tf.int32)
    items = tf.constant(df['item_id'].values, dtype=tf.int32)
    ratings = tf.constant(df['rating'].values, dtype=tf.float32)
    return tf.data.Dataset.from_tensor_slices(((users, items), ratings)).shuffle(1024).batch(32)

train_dataset = create_tf_dataset(train_data)
test_dataset = create_tf_dataset(test_data)

# Step 4: Define the Recommendation Model
class MatrixFactorizationModel(tf.keras.Model):
    def __init__(self, num_users, num_items, embedding_dim=50):
        super().__init__()
        self.user_embedding = tf.keras.layers.Embedding(num_users, embedding_dim)
        self.item_embedding = tf.keras.layers.Embedding(num_items, embedding_dim)

    def call(self, inputs):
        user_vector = self.user_embedding(inputs[0])
        item_vector = self.item_embedding(inputs[1])
        dot_product = tf.reduce_sum(user_vector * item_vector, axis=1)
    return dot_product

model = MatrixFactorizationModel(num_users, num_items)

# Step 5: Compile the model
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.01), loss='mse',
metrics=['mae'])

# Step 6: Train the model
history = model.fit(train_dataset, validation_data=test_dataset, epochs=10)

# Step 7: Evaluate the model
test_loss, test_mae = model.evaluate(test_dataset)
print(f"Test Loss: {test_loss:.4f}, Test MAE: {test_mae:.4f}")

# Step 8: Plot the training history
plt.figure(figsize=(12, 4))

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

plt.subplot(1, 2, 2)
plt.plot(history.history['mae'], label='Train MAE')
plt.plot(history.history['val_mae'], label='Validation MAE')
plt.xlabel('Epoch')
plt.ylabel('Mean Absolute Error')
plt.legend()
plt.title('MAE over Epochs')

plt.show()
```

# Step 9: Make recommendations

```
def recommend(user_id, top_k=5):
    user_vector = tf.constant([user_id] * num_items, dtype=tf.int32)
    item_vector = tf.constant(list(range(num_items)), dtype=tf.int32)
    predictions = model.predict((user_vector, item_vector))
    top_items = np.argsort(-predictions)[:top_k]
    return top_items

user_id = 0  # Example user
recommended_items = recommend(user_id)
print(f"Recommended items for user {user_id}: {recommended_items}"
```

Output :

```
Test Loss: 1.7249, Test MAE: 1.0253
```



```
53/53 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
Recommended items for user 0: [1129   58  223 1376 1173]
```

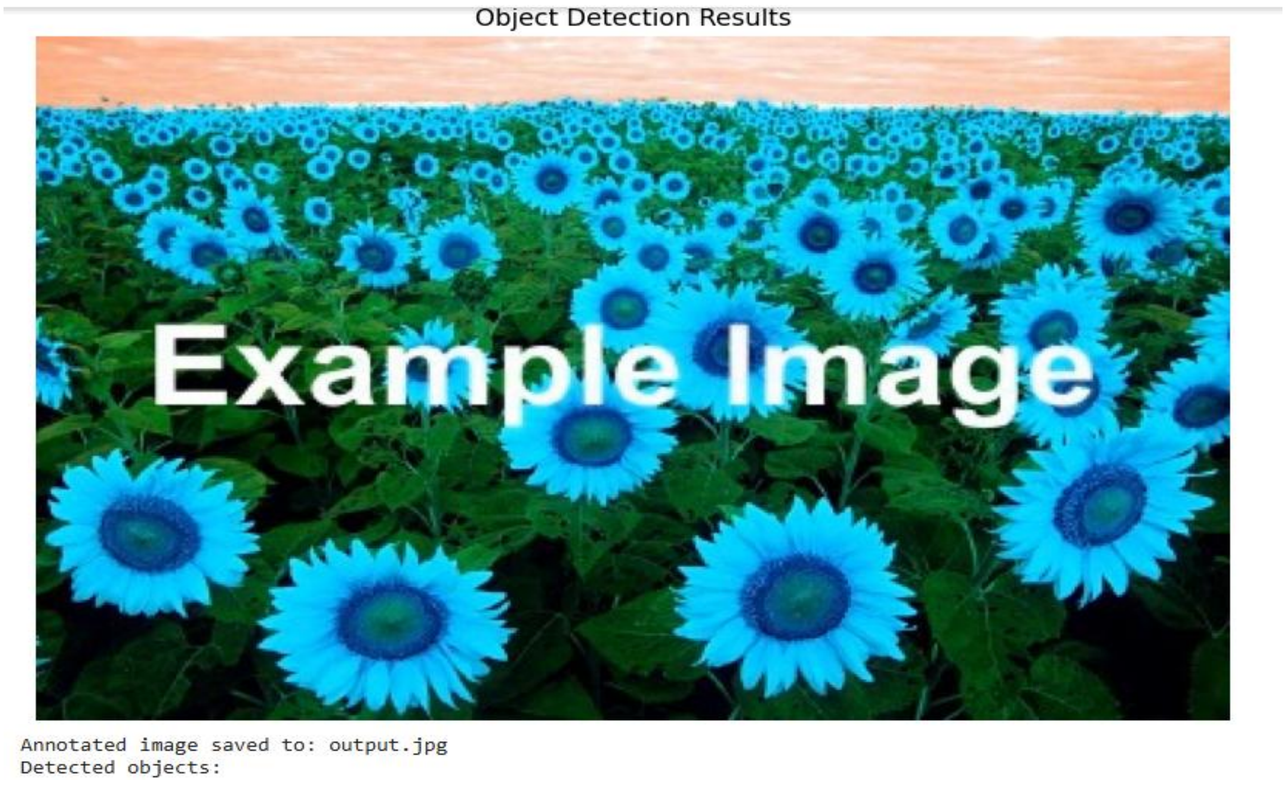
Practical 5

Aim : Implementing a computer vision project, such as object detection or image segmentation.

Code :

```
!pip install torch torchvision numpy matplotlib opencv-python ultralytics

import cv2
import numpy as np
import matplotlib.pyplot as plt
from ultralytics import YOLO  # YOLOv5 library from ultralytics

# Step 1: Load the YOLOv5 Model
print("Loading YOLOv5 model...")
model = YOLO("yolov5s.pt")  # Use the small version of YOLOv5 pre-trained on COCO dataset

# Step 2: Load an Image for Object Detection
image_path = "/example.jpg"  # Replace with your image file path
image = cv2.imread(image_path)
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Step 3: Perform Object Detection
print("Performing object detection...")
results = model.predict(image_rgb)

# Step 4: Visualize Results
annotated_image = results[0].plot()  # Annotated image with bounding boxes and labels
plt.figure(figsize=(10, 10))
plt.imshow(cv2.cvtColor(annotated_image, cv2.COLOR_BGR2RGB))
plt.axis("off")
plt.title("Object Detection Results")
plt.show()

# Step 5: Save the Annotated Image
output_path = "output.jpg"
cv2.imwrite(output_path, annotated_image)
print(f"Annotated image saved to: {output_path}")

# Step 6: Print Detected Objects
print("Detected objects:")
for box in results[0].boxes.data.tolist():
    x1, y1, x2, y2, conf, cls = box
    print(f"Class: {results[0].names[int(cls)]}, Confidence: {conf:.2f}, Coordinates: ({x1:.2f}, {y1:.2f}), ({x2:.2f}, {y2:.2f})")
```

Output :



```
Annotated image saved to: output.jpg
Detected objects:
```

Practical 6

Aim : Training a generative adversarial network (GAN) for generating realistic images

Code :

```
!pip install torch torchvision matplotlib numpy

import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

# Step 1: Define Generator and Discriminator
class Generator(nn.Module):
    def __init__(self, noise_dim, img_dim):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(noise_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, img_dim),
            nn.Tanh(),
        )
    def forward(self, x):
        return self.model(x)


class Discriminator(nn.Module):
    def __init__(self, img_dim):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(img_dim, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1),
            nn.Sigmoid(),
        )

    def forward(self, x):
        return self.model(x)
```

# Step 2: Define Constants and Hyperparameters

```
device = "cuda" if torch.cuda.is_available() else "cpu"
img_size = 28
img_dim = img_size * img_size
noise_dim = 100
batch_size = 64
epochs = 50
lr = 0.0002
```

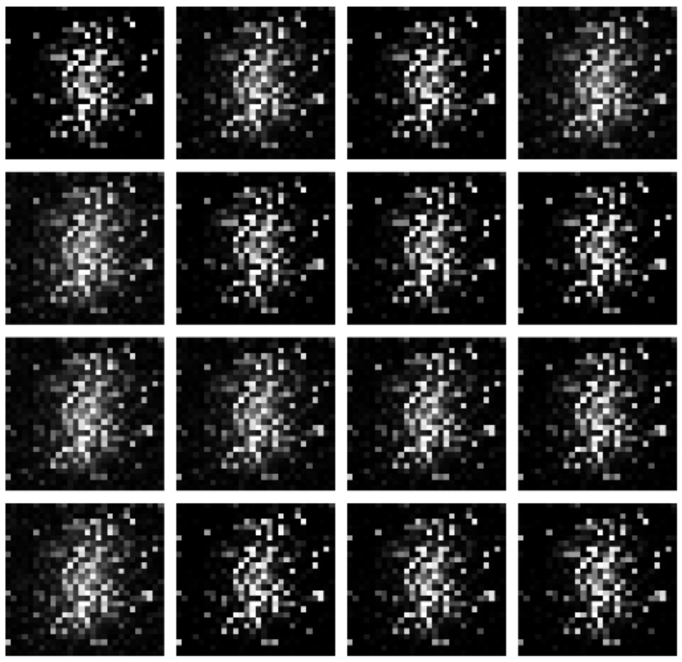
# Step 3: Prepare the MNIST Dataset

```
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,),
(0.5,))])
```

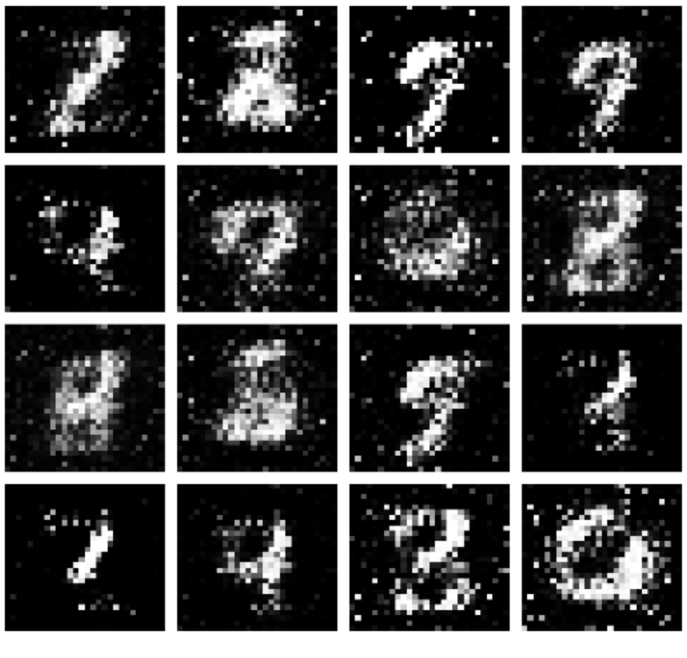
```
dataset = datasets.MNIST(root="data", train=True, transform=transform, download=True)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# Step 4: Initialize Models, Loss, and Optimizers
generator = Generator(noise_dim, img_dim).to(device)
discriminator = Discriminator(img_dim).to(device)

criterion = nn.BCELoss()
optimizer_g = optim.Adam(generator.parameters(), lr=lr)
optimizer_d = optim.Adam(discriminator.parameters(), lr=lr)

# Step 5: Training Loop
for epoch in range(epochs):
    for real_images, _ in dataloader:
        real_images = real_images.view(-1, img_dim).to(device)
        batch_size = real_images.size(0)

        # Labels for real and fake images
        real_labels = torch.ones(batch_size, 1).to(device)
        fake_labels = torch.zeros(batch_size, 1).to(device)

        # Train Discriminator
        noise = torch.randn(batch_size, noise_dim).to(device)
        fake_images = generator(noise)
        real_preds = discriminator(real_images)
        fake_preds = discriminator(fake_images.detach())
        loss_d_real = criterion(real_preds, real_labels)
        loss_d_fake = criterion(fake_preds, fake_labels)
        loss_d = (loss_d_real + loss_d_fake) / 2
        optimizer_d.zero_grad()
        loss_d.backward()
        optimizer_d.step()

        # Train Generator
        noise = torch.randn(batch_size, noise_dim).to(device)
        fake_images = generator(noise)
        fake_preds = discriminator(fake_images)
        loss_g = criterion(fake_preds, real_labels)
        optimizer_g.zero_grad()
        loss_g.backward()
        optimizer_g.step()

    # Print progress
    print(f"Epoch [{epoch+1}/{epochs}] | Loss D: {loss_d:.4f} | Loss G: {loss_g:.4f}")

    # Save generated samples every 10 epochs
    if (epoch + 1) % 10 == 0:
        noise = torch.randn(16, noise_dim).to(device)
        generated_images = generator(noise).view(-1, 1, img_size, img_size).cpu().detach()
        plt.figure(figsize=(8, 8))
        for i in range(16):
            plt.subplot(4, 4, i + 1)
            plt.imshow(generated_images[i].squeeze(), cmap="gray")
            plt.axis("off")
        plt.tight_layout()
        plt.savefig(f"generated_images_epoch_{epoch+1}.png")
        plt.close()

# Step 6: Save the Generator Model
torch.save(generator.state_dict(), "gan_generator.pth")
print("Generator model saved as gan_generator.pth")
```

Output :

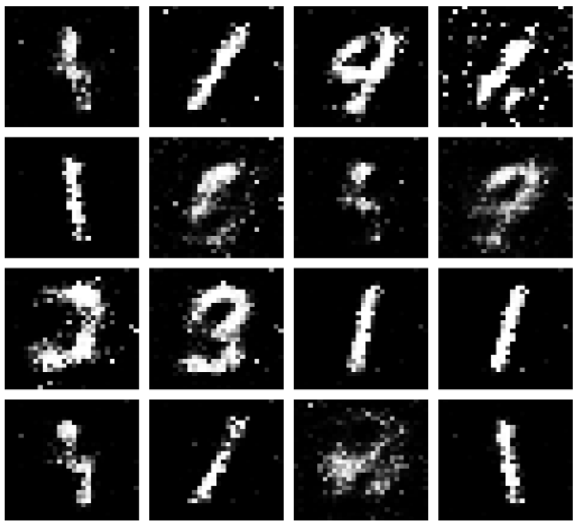
Generate after 10 Epoch



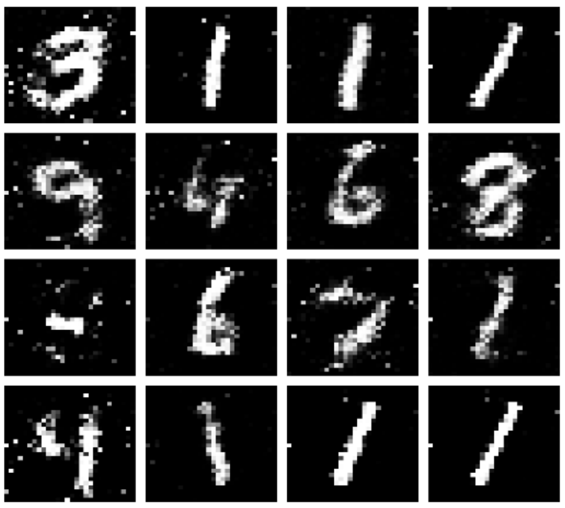
Generate after 20 Epoch



Generate 30 Epoch



Generate after 40 Epoch



Generate after 50 Epoch

Practical 7

Aim : Applying reinforcement learning algorithms to solve complex decision-making problems.

Code :

Step 1: Install Required Libraries

Run the following commands in your terminal to install the necessary libraries:

```
!pip install numpy gym matplotlib
```

Step 2: Python Code for Q-Learning

Save the following code as q_learning_cartpole.py.

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

# Step 1: Initialize Environment and Parameters
env = gym.make("CartPole-v1")
n_actions = env.action_space.n  # Number of actions (2: left, right)
n_states = 20  # Discretize continuous state space
episodes = 500  # Number of episodes
learning_rate = 0.1  # Learning rate (alpha)
discount_factor = 0.99  # Discount factor (gamma)
epsilon = 1.0  # Exploration rate
epsilon_decay = 0.995  # Decay factor for epsilon
min_epsilon = 0.01  # Minimum epsilon

# Step 2: Helper Functions for Discretizing States
```

```
def discretize_state(state, state_bins):
    return tuple(np.digitize(state[i], state_bins[i]) for i in range(len(state)))


def create_bins(n_states, env):
    state_bins = []
    for i in range(env.observation_space.shape[0]):
        low, high = env.observation_space.low[i], env.observation_space.high[i]
        bins = np.linspace(low, high, n_states - 1)
        state_bins.append(bins)
    return state_bins

# Step 3: Initialize Q-Table and State Bins
state_bins = create_bins(n_states, env)
q_table = np.zeros((n_states,) * len(env.observation_space.shape) + (n_actions,))

# Step 4: Training Loop
rewards = []
for episode in range(episodes):
    state = discretize_state(env.reset()[0], state_bins)
    total_reward = 0
    done = False

    while not done:
        # Epsilon-greedy action selection
        if np.random.rand() < epsilon:
            action = np.random.choice(n_actions)  # Explore
        else:
            action = np.argmax(q_table[state])  # Exploit

        # Take action and observe results
        next_state_raw, reward, done, _, _ = env.step(action)
        next_state = discretize_state(next_state_raw, state_bins)
        total_reward += reward

        # Q-Learning update
        q_table[state][action] += learning_rate * (
            reward + discount_factor * np.max(q_table[next_state]) - q_table[state][action]
        )

        state = next_state

    # Decay epsilon
    epsilon = max(min_epsilon, epsilon * epsilon_decay)

    rewards.append(total_reward)
    print(f"Episode {episode + 1}/{episodes}, Reward: {total_reward}, Epsilon: {epsilon:.3f}")

# Step 5: Plot Rewards
plt.plot(rewards)
plt.title("Total Rewards Over Episodes")
plt.xlabel("Episode")
plt.ylabel("Total Reward")
plt.show()

# Step 6: Save the Q-Table
np.save("q_table.npy", q_table)
print("Q-Table saved as 'q_table.npy'.")

# Step 7: Test the Trained Model
state = discretize_state(env.reset()[0], state_bins)
done = False
```

```
total_reward = 0
while not done:
    action = np.argmax(q_table[state])  # Use trained Q-Table
    next_state_raw, reward, done, _, _ = env.step(action)
    state = discretize_state(next_state_raw, state_bins)
    total_reward += reward
    env.render()

env.close()
print(f"Total reward during test: {total_reward}")
```

Practical 8

Aim : Utilizing transfer learning to improve model performance on limited datasets.

Code :

Step 1: Install Required Libraries

Run the following command to install the necessary libraries:

```
!pip install torch torchvision matplotlib numpy
```

Step 2: Python Code for Transfer Learning

Save the following code as transfer_learning.py.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, models, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

# Step 1: Set Device and Hyperparameters
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
num_classes = 10  # CIFAR-10 has 10 classes
batch_size = 32
epochs = 10
learning_rate = 0.001

# Step 2: Define Data Transformations
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # Resize to match ResNet input size
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
])

# Step 3: Load CIFAR-10 Dataset
train_dataset = datasets.CIFAR10(root="data", train=True, transform=transform, download=True)
test_dataset = datasets.CIFAR10(root="data", train=False, transform=transform, download=True)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# Step 4: Load Pre-trained ResNet18 Model
model = models.resnet18(pretrained=True)
for param in model.parameters():
    param.requires_grad = False  # Freeze all layers

# Replace the final fully connected layer for CIFAR-10
model.fc = nn.Linear(model.fc.in_features, num_classes)
model = model.to(device)

# Step 5: Define Loss and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.fc.parameters(), lr=learning_rate)

# Step 6: Training Loop
def train(model, loader, criterion, optimizer):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    for inputs, labels in loader:
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()

    accuracy = 100. * correct / total
    return running_loss / len(loader), accuracy
```

# Step 7: Testing Loop

```
def test(model, loader, criterion):
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in loader:
            inputs, labels = inputs.to(device), labels.to(device)

            outputs = model(inputs)
            loss = criterion(outputs, labels)
            running_loss += loss.item()

            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()

    accuracy = 100. * correct / total
    return running_loss / len(loader), accuracy
```

# Step 8: Train and Evaluate

```
train_losses, test_losses = [], []
train_accuracies, test_accuracies = [], []

for epoch in range(epochs):
    train_loss, train_acc = train(model, train_loader, criterion, optimizer)
    test_loss, test_acc = test(model, test_loader, criterion)

    train_losses.append(train_loss)
    test_losses.append(test_loss)
    train_accuracies.append(train_acc)
    test_accuracies.append(test_acc)

    print(f"Epoch {epoch+1}/{epochs}")
    print(f"Train Loss: {train_loss:.4f}, Train Accuracy: {train_acc:.2f}%")
    print(f"Test Loss: {test_loss:.4f}, Test Accuracy: {test_acc:.2f}%")
```

# Step 9: Plot Training and Testing Metrics

```
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(train_losses, label="Train Loss")
plt.plot(test_losses, label="Test Loss")
plt.legend()
plt.title("Loss")

plt.subplot(1, 2, 2)
plt.plot(train_accuracies, label="Train Accuracy")
plt.plot(test_accuracies, label="Test Accuracy")
plt.legend()
plt.title("Accuracy")
plt.show()

# Step 10: Save the Model
torch.save(model.state_dict(), "resnet18_cifar10.pth")
print("Model saved as 'resnet18_cifar10.pth'.")
```

Practical 9

Aim : Building a deep learning model for time series forecasting or anomaly detection

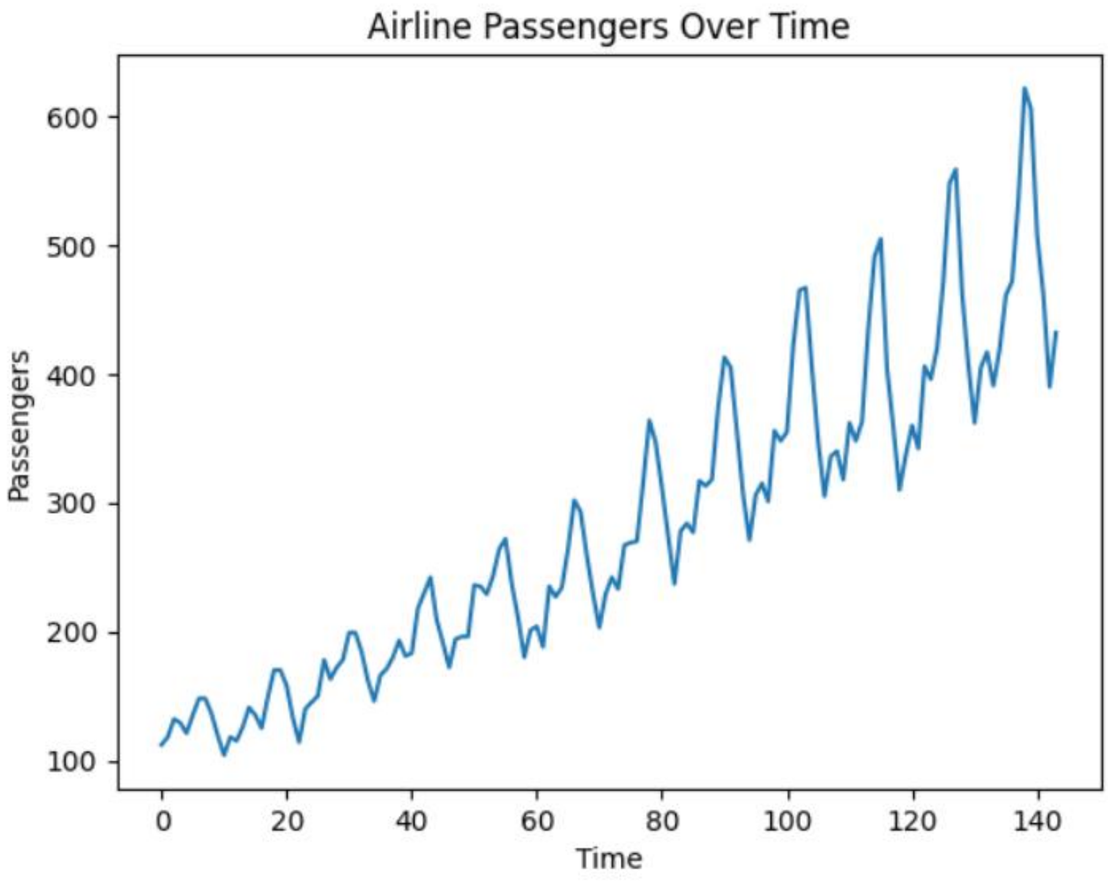
Code :

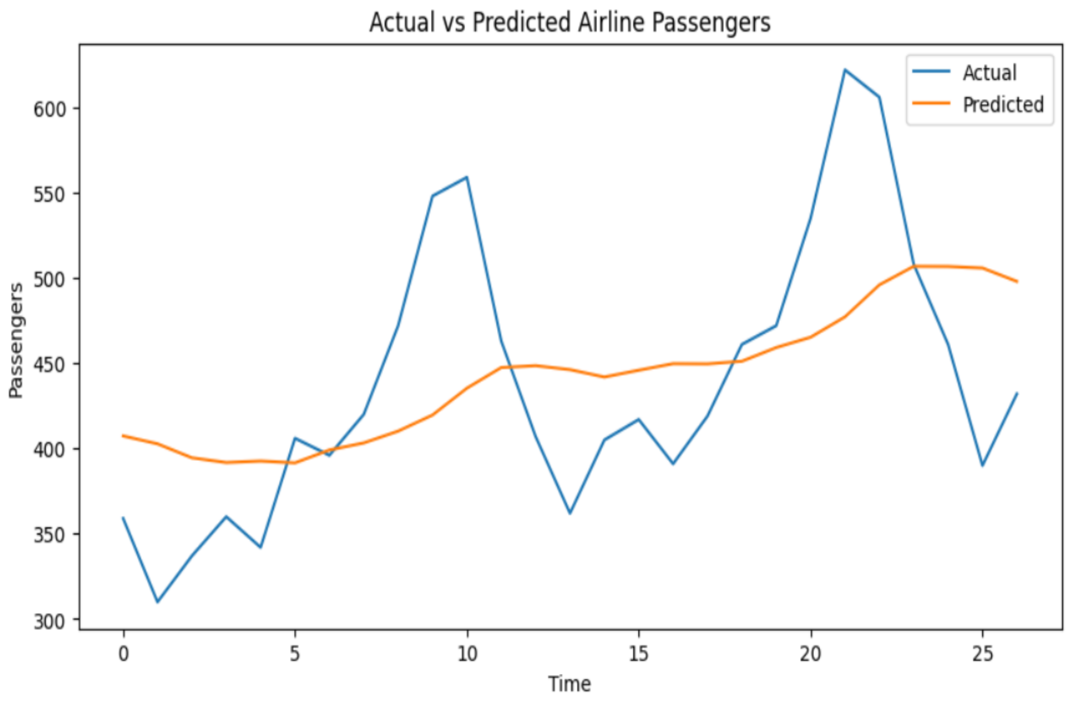
```
!pip install numpy pandas matplotlib scikit-learn tensorflow

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

# Step 1: Load the Dataset
url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passengers.csv"
data = pd.read_csv(url, usecols=[1], header=0)
data = data.values.astype("float32")  # Ensure the data is float

# Step 2: Visualize the Data
plt.plot(data)
plt.title("Airline Passengers Over Time")
plt.xlabel("Time")
plt.ylabel("Passengers")
plt.show()

# Step 3: Normalize the Data
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(data)
```

```
# Step 4: Prepare the Data for LSTM
def create_dataset(dataset, look_back=1):
    X, y = [], []
    for i in range(len(dataset) - look_back):
        X.append(dataset[i:(i + look_back), 0])
        y.append(dataset[i + look_back, 0])
    return np.array(X), np.array(y)

look_back = 12  # Use 12 months (1 year) as input to predict the next value
X, y = create_dataset(scaled_data, look_back)
X = X.reshape((X.shape[0], X.shape[1], 1))  # Reshape for LSTM [samples, time_steps, features]

# Step 5: Split Data into Training and Testing Sets
train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# Step 6: Build the LSTM Model
model = Sequential([
    LSTM(50, activation="relu", input_shape=(look_back, 1)),
    Dense(1)
])
model.compile(optimizer="adam", loss="mean_squared_error")

# Step 7: Train the Model
history = model.fit(X_train, y_train, epochs=50, batch_size=32, validation_data=(X_test, y_test), verbose=1)

# Step 8: Evaluate the Model
```

```
loss = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Loss: {loss:.4f}")

# Step 9: Predict and Inverse Transform
y_pred = model.predict(X_test)
y_pred = scaler.inverse_transform(y_pred)
y_test_actual = scaler.inverse_transform(y_test.reshape(-1, 1))

# Step 10: Plot Actual vs Predicted
plt.figure(figsize=(10, 5))
plt.plot(y_test_actual, label="Actual")
plt.plot(y_pred, label="Predicted")
plt.title("Actual vs Predicted Airline Passengers")
plt.xlabel("Time")
plt.ylabel("Passengers")
plt.legend()
plt.show()

# Step 11: Save the Model
model.save("lstm_time_series.h5")
print("Model saved as 'lstm_time_series.h5'.")
```

Output :


Airline Passengers Over Time
Passengers
600
500
400
300
200
100
0
20
40
60
80
100
120
140
Time


Actual vs Predicted Airline Passengers
Actual
Predicted
Passengers
600
550
500
450
400
350
300
0
5
10
15
20
25
Time

Practical 10

Aim : Implementing a machine learning pipeline for automated feature engineering and model selection.

Code :

Step 1: Install Required Libraries

Run the following command to install the required libraries

```
pip install pandas numpy scikit-learn
```

Step 2: Python Code for Machine Learning Pipeline

Save the following code as ml_pipeline.py.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.metrics import classification_report, accuracy_score

# Step 1: Load Dataset
# Using the Titanic dataset for demonstration
url = "https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv"
data = pd.read_csv(url)
```

```
# Step 2: Basic Preprocessing
# Drop unnecessary columns
data = data.drop(["PassengerId", "Name", "Ticket", "Cabin"], axis=1)

# Handle missing values
data["Age"].fillna(data["Age"].median(), inplace=True)
data["Embarked"].fillna(data["Embarked"].mode()[0], inplace=True)

# Separate features and target
X = data.drop("Survived", axis=1)
y = data["Survived"]

# Step 3: Split Data into Train and Test Sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 4: Define Preprocessing Steps
# Numerical features: Scale values
numerical_features = ["Age", "Fare"]
numerical_transformer = Pipeline(steps=[
    ("scaler", StandardScaler())
])

# Categorical features: One-hot encode
categorical_features = ["Sex", "Embarked", "Pclass"]
categorical_transformer = Pipeline(steps=[
    ("onehot", OneHotEncoder(handle_unknown="ignore"))
])

# Combine preprocessors into a column transformer
preprocessor = ColumnTransformer(
    transformers=[
        ("num", numerical_transformer, numerical_features),
        ("cat", categorical_transformer, categorical_features)
    ]
)

# Step 5: Define Feature Selection and Model Options
feature_selection = SelectKBest(score_func=f_classif)

# Define candidate models
models = {
    "RandomForest": RandomForestClassifier(random_state=42),
    "SVC": SVC(probability=True, random_state=42)
}

# Step 6: Create the Pipeline
pipeline = Pipeline(steps=[
    ("preprocessor", preprocessor),
    ("feature_selection", feature_selection),
    ("classifier", RandomForestClassifier())
])

# Step 7: Define Grid Search for Hyperparameter Tuning
param_grid = {
    "feature_selection__k": [5, 6, 7],
    "classifier": [models["RandomForest"], models["SVC"]],
    "classifier__n_estimators": [100, 200] if "n_estimators" in
RandomForestClassifier().get_params() else [None],
    "classifier__C": [0.1, 1, 10] if "C" in SVC().get_params() else [None]
}

grid_search = GridSearchCV(pipeline, param_grid, cv=3, scoring="accuracy", verbose=2)
```

```
# Step 8: Train the Model
grid_search.fit(X_train, y_train)

# Step 9: Evaluate the Model
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)

print("Best Parameters:", grid_search.best_params_)
print("Accuracy on Test Set:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

Practical 11

Aim : Using advanced optimization techniques like evolutionary algorithms or Bayesian optimization for hyperparameter tuning.

Code :

```
!pip install numpy pandas scikit-learn scikit-optimize


import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.ensemble import RandomForestClassifier
from skopt import BayesSearchCV
from sklearn.metrics import accuracy_score, classification_report

# Step 1: Load the Dataset
data = load_iris()
X, y = data.data, data.target

# Step 2: Split the Data into Training and Testing Sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 3: Define the Model
model = RandomForestClassifier(random_state=42)

# Step 4: Define the Search Space for Hyperparameters
param_space = {
    "n_estimators": (10, 200),          # Number of trees in the forest
    "max_depth": (1, 20),               # Maximum depth of each tree
    "min_samples_split": (2, 10),       # Minimum samples to split a node
    "min_samples_leaf": (1, 10),       # Minimum samples at each leaf
    "max_features": ["sqrt", "log2", None]  # Number of features considered for split
}

# Step 5: Use Bayesian Optimization for Hyperparameter Tuning
optimizer = BayesSearchCV(
    estimator=model,
    search_spaces=param_space,
    n_iter=30,  # Number of iterations to search
    cv=3,       # 3-fold cross-validation
    random_state=42,
    n_jobs=-1
)

# Step 6: Train the Optimized Model
print("Starting Bayesian Optimization...")
optimizer.fit(X_train, y_train)

# Step 7: Evaluate the Best Model
best_model = optimizer.best_estimator_
y_pred = best_model.predict(X_test)

print("\nBest Parameters:", optimizer.best_params_)
print("Accuracy on Test Set:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))

# Optional: Save the Best Model
import joblib
joblib.dump(best_model, "optimized_rf_model.pkl")
print("\nModel saved as 'optimized_rf_model.pkl'.")
```

Output :

```
Best Parameters: OrderedDict([('max_depth', 16), ('max_features', 'log2'), ('min_samples_leaf', 6), ('min_samples_split', 8), ('n_estimators', 182)])
Accuracy on Test Set: 1.0

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      1.00      1.00         9
           2       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30


Model saved as 'optimized_rf_model.pkl'.
```

Practical 12

Aim : Deploying a machine learning model in a production environment using containerization and cloud services

Code :

Step 1: Install Required Libraries

```
!pip install scikit-learn pandas fastapi uvicorn
```

Step 2: Build the Machine Learning Model

Save the following Python script as train_model.py. This script trains the model and saves it.

```
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
import joblib

# Step 1: Load Dataset
data = load_iris()
X, y = data.data, data.target

# Step 2: Train Model
model = RandomForestClassifier(random_state=42)
model.fit(X, y)

# Step 3: Save Model
joblib.dump(model, "iris_model.pkl")
print("Model saved as iris_model.pkl")
```

Run the script to save the trained model:

bash

Copy code

```
python train_model.py
```

Step 3: Create the API

Save the following Python script as app.py.

python

Copy code

```
from fastapi import FastAPI
from pydantic import BaseModel
import joblib
import numpy as np

# Step 1: Load the trained model
model = joblib.load("iris_model.pkl")

# Step 2: Initialize FastAPI
app = FastAPI()

# Step 3: Define input schema
class IrisRequest(BaseModel):
    sepal_length: float
    sepal_width: float
    petal_length: float
    petal_width: float

# Step 4: Define prediction endpoint
@app.post("/predict/")
def predict(iris: IrisRequest):
    features = np.array([[iris.sepal_length, iris.sepal_width, iris.petal_length, iris.petal_width]])
    prediction = model.predict(features)
    species = ["setosa", "versicolor", "virginica"]
    return {"prediction": species[prediction[0]]}
```

Run the API locally for testing:

bash

Copy code

```bash
uvicorn app:app --host 0.0.0.0 --port 8000
```

Test the API in your browser or with a tool like **Postman**:

- URL: http://127.0.0.1:8000/predict/
- Example Request Body:

json

Copy code

```json
{
    "sepal_length": 5.1,
    "sepal_width": 3.5,
    "petal_length": 1.4,
    "petal_width": 0.2
}
```

Step 4: Create a Dockerfile

Save the following as Dockerfile.

dockerfile

Copy code

```dockerfile
# Use an official Python runtime as the base image
FROM python:3.9-slim

# Set the working directory in the container
WORKDIR /app

# Copy the current directory contents into the container
COPY . /app
```

```
# Install required Python libraries
RUN pip install --no-cache-dir fastapi uvicorn scikit-learn joblib

# Expose the API port
EXPOSE 8000

# Command to run the application
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]
```

Step 5: Build and Run the Docker Container

1. **Build the Docker image**:

bash

Copy code

```
docker build -t iris-api .
```

2. **Run the Docker container**:

bash

Copy code

```
docker run -d -p 8000:8000 iris-api
```

3. **Test the API**:
   - URL: http://localhost:8000/predict/
   - Use the same JSON request as earlier.

Step 6: Deploy to a Cloud Service (Optional)

1. **Prepare the Docker Image**:
   - Tag the image for a container registry (e.g., Docker Hub, AWS ECR, or GCP Artifact Registry):

bash

Copy code

```
docker tag iris-api <your_dockerhub_username>/iris-api
docker push <your_dockerhub_username>/iris-api
```

2. **Deploy to AWS ECS (Example)**:

    - Create an ECS cluster.
    - Use the Docker image in a task definition.
    - Deploy the task to the cluster.
3. **Other Options**:
    - Use **AWS Lambda** with **API Gateway**.
    - Deploy on **Google Cloud Run** or **Azure App Service** for managed hosting.

Step 7: Dataset

- **Iris Dataset**:
    - Included with sklearn.datasets for demonstration purposes.
    - Automatically loaded in the train_model.py script.

Practical 13

Aim : Use Python libraries such as GPT-2 or textgenrnn to train generative models on a corpus of text data and generate new text based on the patterns it has learned.

Code :

Step 1: Install Required Libraries

Run the following commands to install necessary libraries:

```
!pip install transformers datasets torch
```

Step 2: Prepare a Text Dataset

For demonstration, we'll use the Tiny Shakespeare Corpus available via Hugging Face's datasets library. Alternatively, you can use your own dataset.

Step 3: Python Code for Training and Generating Text

Save the following code as train_gpt2.py.

```
import os
from datasets import load_dataset
from transformers import GPT2LMHeadModel, GPT2Tokenizer, Trainer, TrainingArguments

# Step 1: Load the Dataset
print("Loading dataset...")
dataset = load_dataset("tiny_shakespeare")

# Split into train and test sets
train_data = dataset["train"]
test_data = dataset["test"]

# Step 2: Load Pre-trained GPT-2 Tokenizer and Model
```

```
print("Loading GPT-2 tokenizer and model...")
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
model = GPT2LMHeadModel.from_pretrained("gpt2")

# Step 3: Tokenize the Dataset
def tokenize_function(examples):
    return tokenizer(examples["text"], truncation=True, padding="max_length",
max_length=512)


print("Tokenizing dataset...")
tokenized_train = train_data.map(tokenize_function, batched=True)
tokenized_test = test_data.map(tokenize_function, batched=True)

# Step 4: Define Training Arguments
training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    learning_rate=5e-5,
    weight_decay=0.01,
    per_device_train_batch_size=4,
    num_train_epochs=3,
    save_total_limit=2,
    logging_dir="./logs",
    logging_steps=10,
)

# Step 5: Initialize Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_train,
    eval_dataset=tokenized_test,
)

# Step 6: Train the Model
print("Starting training...")
trainer.train()

# Step 7: Save the Fine-Tuned Model
model.save_pretrained("./fine_tuned_gpt2")
tokenizer.save_pretrained("./fine_tuned_gpt2")
print("Model saved to './fine_tuned_gpt2'.")

# Step 8: Generate Text Using the Fine-Tuned Model
print("Generating new text...")
model.eval()

input_text = "To be or not to be, that is the"
inputs = tokenizer.encode(input_text, return_tensors="pt")
outputs = model.generate(inputs, max_length=100, num_return_sequences=1,
temperature=0.7)

generated_text = tokenizer.decode(outputs[0], skip_special_tokens=True)
print("\nGenerated Text:\n")
print(generated_text)
```

Practical 14

Aim : Experiment with neural networks like GANs (Generative Adversarial Networks) using Python libraries like TensorFlow or PyTorch to generate new images based on a dataset of images.

Code :

```
!pip install torch torchvision matplotlib


import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import os

# Step 1: Set Device Configuration
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Step 2: Define Generator
class Generator(nn.Module):
    def __init__(self, noise_dim, img_dim):
        super(Generator, self).__init__()
        self.gen = nn.Sequential(
            nn.Linear(noise_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, img_dim),
            nn.Tanh()
        )

    def forward(self, x):
        return self.gen(x)
```

# Step 3: Define Discriminator

```
class Discriminator(nn.Module):
    def __init__(self, img_dim):
        super(Discriminator, self).__init__()
        self.disc = nn.Sequential(
            nn.Linear(img_dim, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.disc(x)
```

# Step 4: Define Hyperparameters

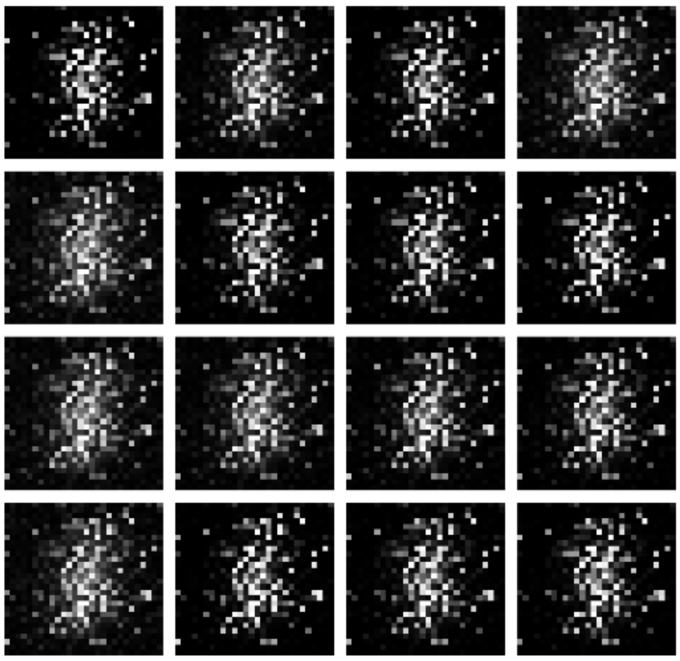
```
noise_dim = 100
img_dim = 28 * 28  # 28x28 images flattened
batch_size = 64
lr = 0.0002
epochs = 50
```

# Step 5: Load Dataset

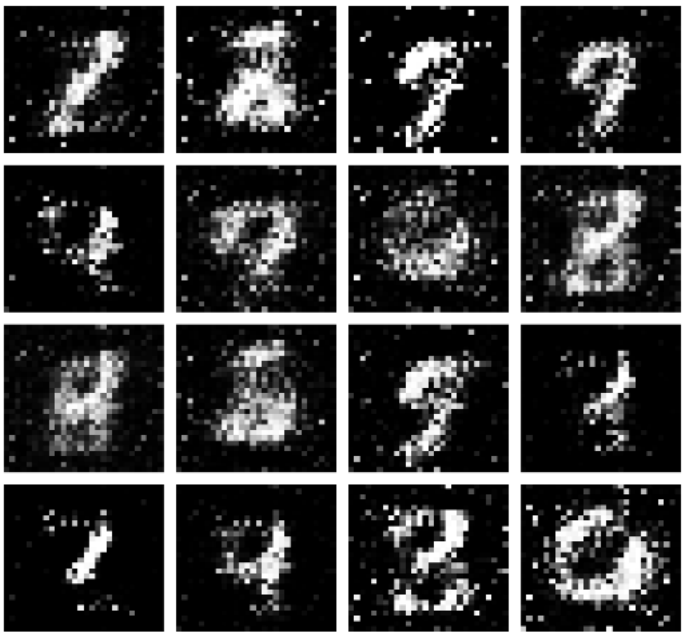
```
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
dataset = datasets.MNIST(root="data", train=True, transform=transform, download=True)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# Step 6: Initialize Models, Optimizers, and Loss Function
gen = Generator(noise_dim, img_dim).to(device)
disc = Discriminator(img_dim).to(device)
criterion = nn.BCELoss()
opt_gen = optim.Adam(gen.parameters(), lr=lr)
opt_disc = optim.Adam(disc.parameters(), lr=lr)

# Step 7: Training Loop
print("Starting Training...")
for epoch in range(epochs):
    for batch_idx, (real, _) in enumerate(dataloader):
        real = real.view(-1, img_dim).to(device)
        batch_size = real.size(0)

        # Train Discriminator
        noise = torch.randn(batch_size, noise_dim).to(device)
        fake = gen(noise)
        disc_real = disc(real).view(-1)
        disc_fake = disc(fake.detach()).view(-1)
        loss_disc = criterion(disc_real, torch.ones_like(disc_real)) + \
                    criterion(disc_fake, torch.zeros_like(disc_fake))
        opt_disc.zero_grad()
        loss_disc.backward()
        opt_disc.step()

        # Train Generator
        output = disc(fake).view(-1)
        loss_gen = criterion(output, torch.ones_like(output))
        opt_gen.zero_grad()
        loss_gen.backward()
        opt_gen.step()

    print(f"Epoch [{epoch+1}/{epochs}] | Loss D: {loss_disc:.4f}, Loss G: {loss_gen:.4f}")

    # Save and Display Sample Images
    if (epoch + 1) % 10 == 0:
        with torch.no_grad():
            fake_images = gen(torch.randn(16, noise_dim).to(device)).view(-1, 1, 28, 28)
        plt.figure(figsize=(4, 4))
        for i in range(16):
            plt.subplot(4, 4, i+1)
            plt.imshow(fake_images[i][0].cpu(), cmap="gray")
            plt.axis("off")
        plt.tight_layout()
        os.makedirs("generated_images", exist_ok=True)
        plt.savefig(f"generated_images/epoch_{epoch+1}.png")
        plt.close()

print("Training Complete. Generated images are saved in 'generated_images' folder.")
```

Output :

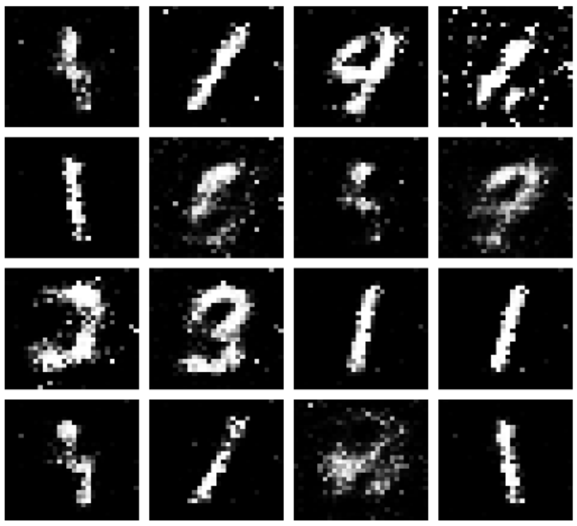
Generate after 10 Epoch



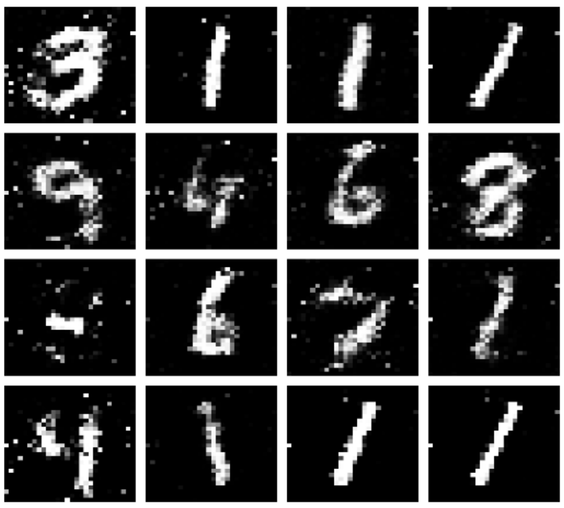
Generate after 20 Epoch



Generate 30 Epoch



Generate after 40 Epoch



Generate after 50 Epoch