**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*

**Vrushabh Ravindra pawar**

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*

**Vrushabh Ravindra Pawar**

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. Vrushabh Ravindra Pawar**, Seat number **1313274** , 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**

|  |  |  |
| --- | --- | --- |
| **SR No.** | **Practical** | **Sign** |
| 1 | Implementing advanced deep learning algorithms such as convolutional neural networks (CNNs) or recurrent neural networks (RNNs) using Python libraries like TensorFlow or PyTorch. |  |
| 2 | Building a natural language processing (NLP) model for sentiment analysis or text classification. |  |
| 3 | Creating a chatbot using advanced techniques like transformer models |  |
| 4 | Developing a recommendation system using collaborative filtering or deep learning approaches. |  |
| 5 | Implementing a computer vision project, such as object detection or image segmentation |  |
| 6 | Training a generative adversarial network (GAN) for generating realistic images |  |
| 7 | Applying reinforcement learning algorithms to solve complex decision-making problems. |  |
| 8 | Utilizing transfer learning to improve model performance on limited datasets. |  |
| 9 | Building a deep learning model for time series forecasting or anomaly detection. |  |
| 10 | Implementing a machine learning pipeline for automated feature engineering and model selection. |  |
| 11 | Using advanced optimization techniques like evolutionary algorithms or Bayesian optimization for hyperparameter tuning |  |
| 12 | Deploying a machine learning model in a production environment using containerization and cloud services. |  |
| 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(),

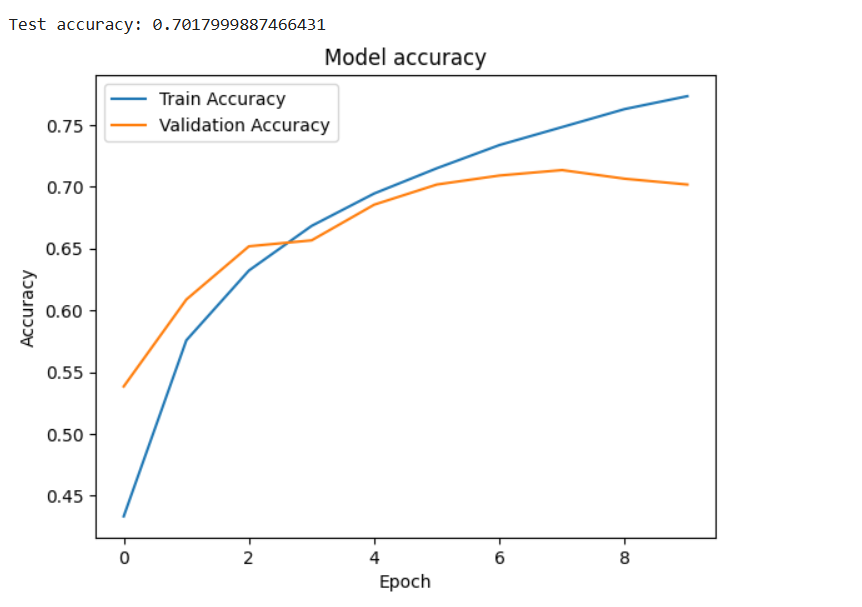
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),

])

**# Load the CIFAR-10 dataset**

trainset = torchvision

**OUTPUT :**

****

**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')

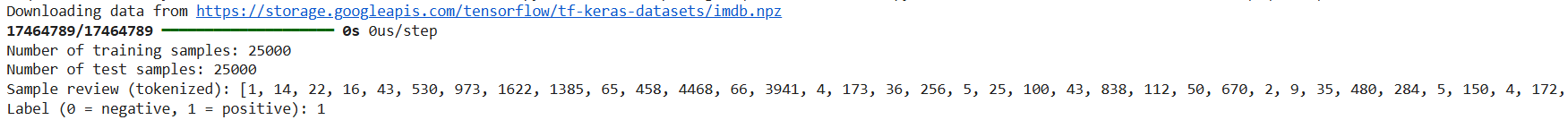
plt.ylabel('Loss')

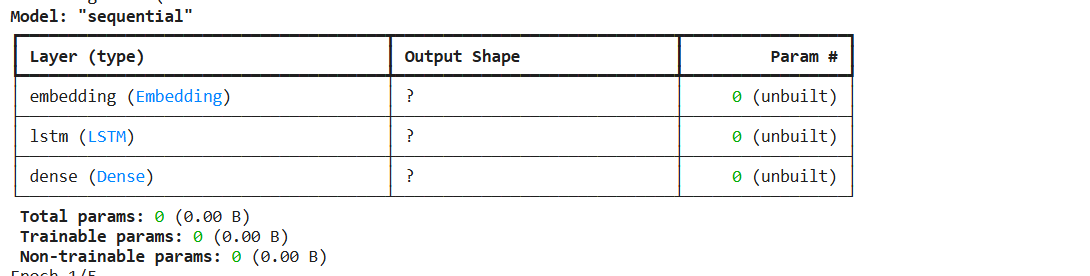
plt.legend()

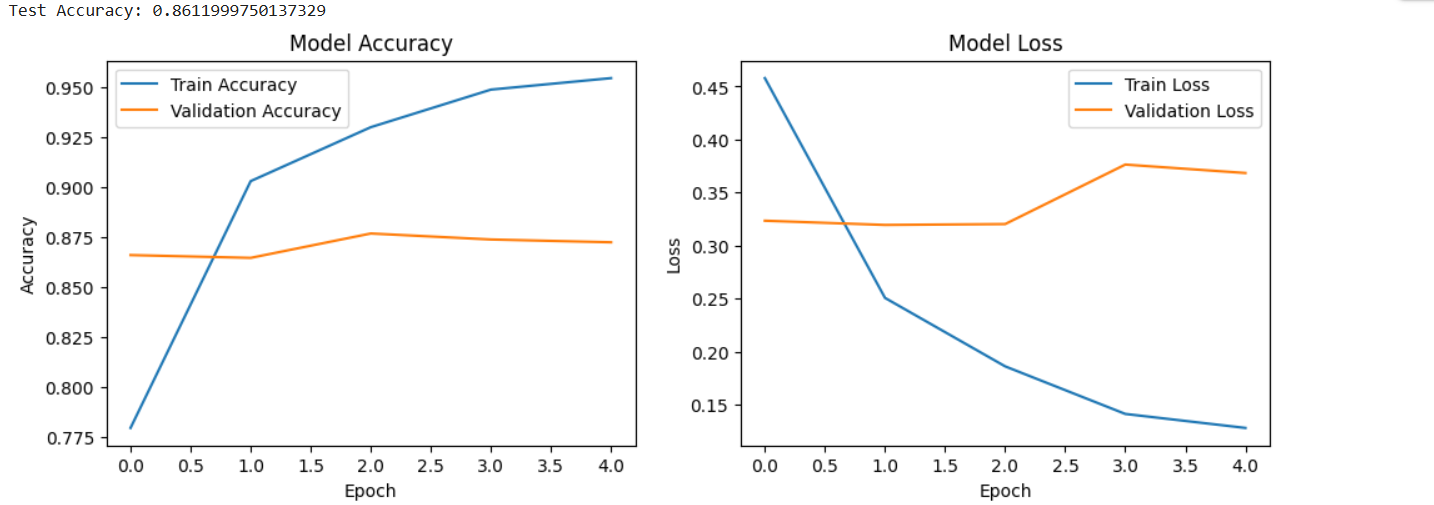
plt.title('Model Loss')

plt.show()

**OUTPUT :**

****

****

****

**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)

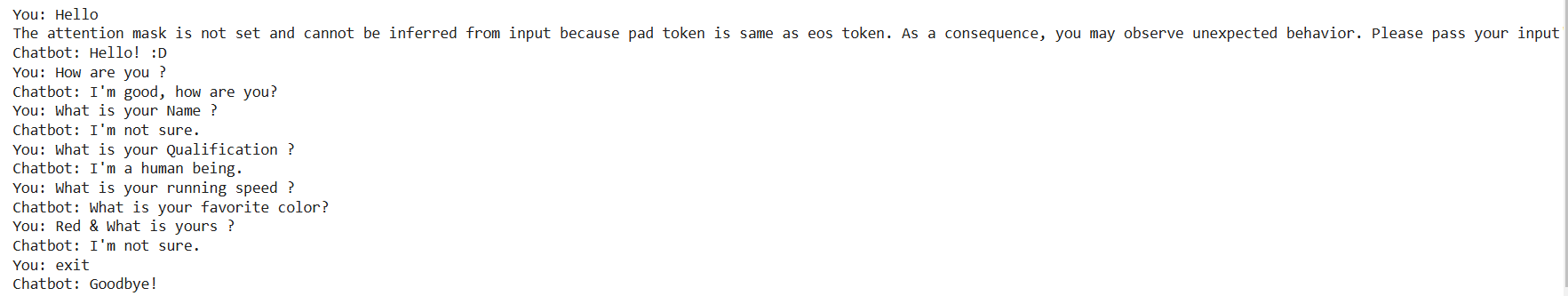
    response = tokenizer.decode(response\_ids[:, chat\_history\_ids.shape[-1]:][0], skip\_special\_tokens=True)

**# Display the response**

    print(f"Chatbot: {response}")

    step += 1

**Output :**

****

**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]

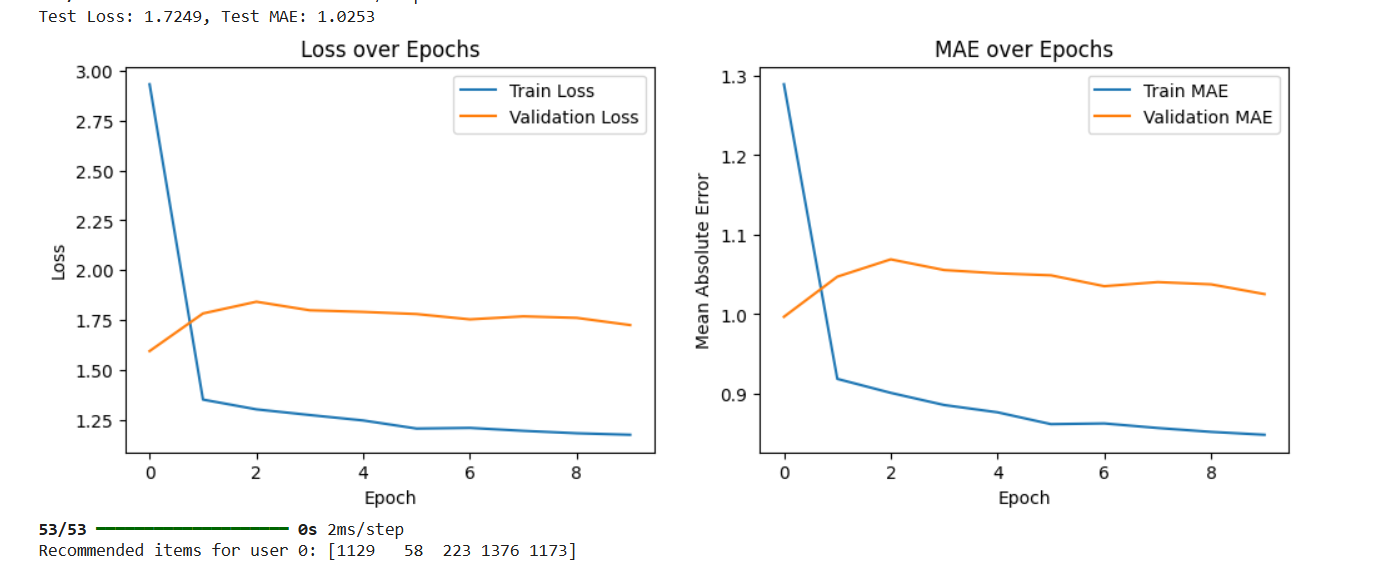
return top\_items

user\_id = 0 # Example user

recommended\_items = recommend(user\_id)

print(f"Recommended items for user {user\_id}: {recommended\_items}"

**Output :**

****

**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 :**

****

**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()

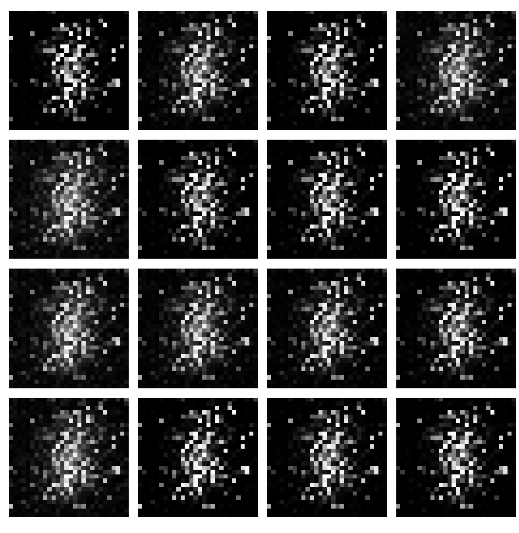
**# Step 6: Save the Generator Model**

torch.save(generator.state\_dict(), "gan\_generator.pth")

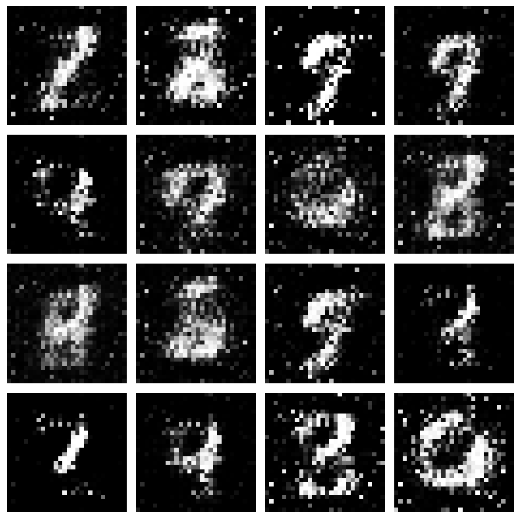
print("Generator model saved as gan\_generator.pth")

**Output :**

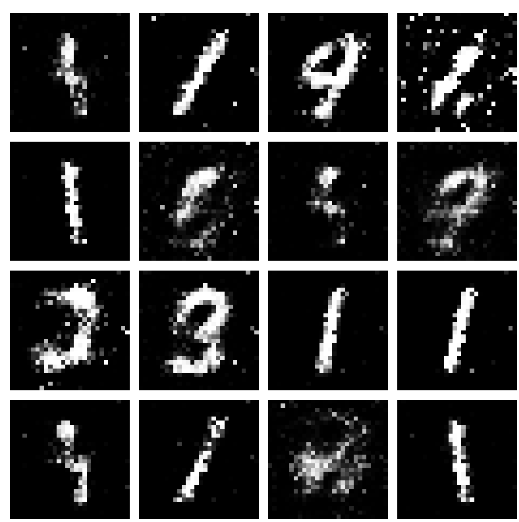
**Generate after 10 Epoch**

****

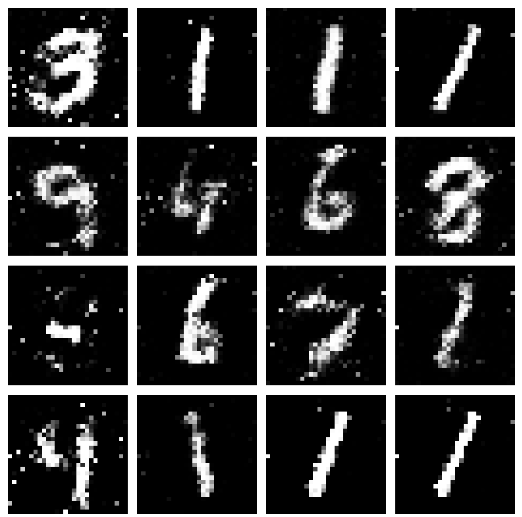
**Generate after 20 Epoch**

****

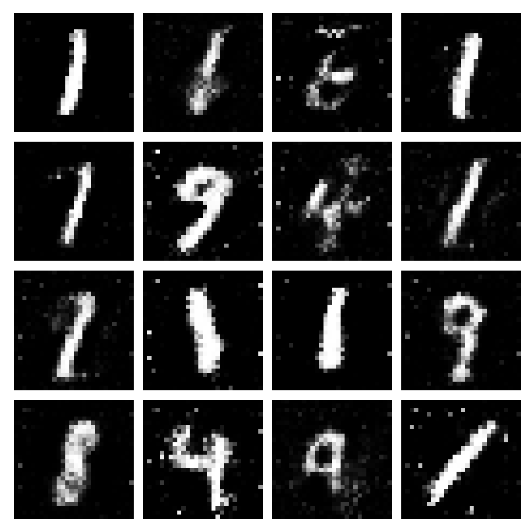
**Generate 30 Epoch**

****

**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()

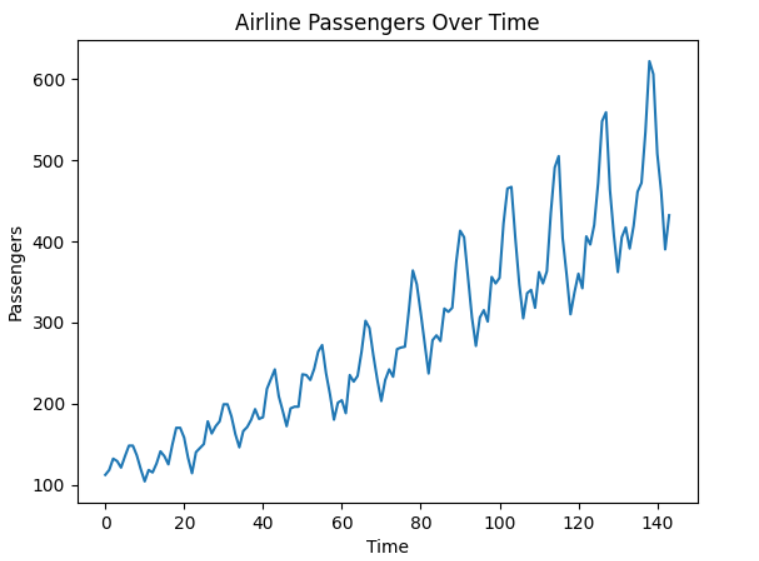
plt.show()

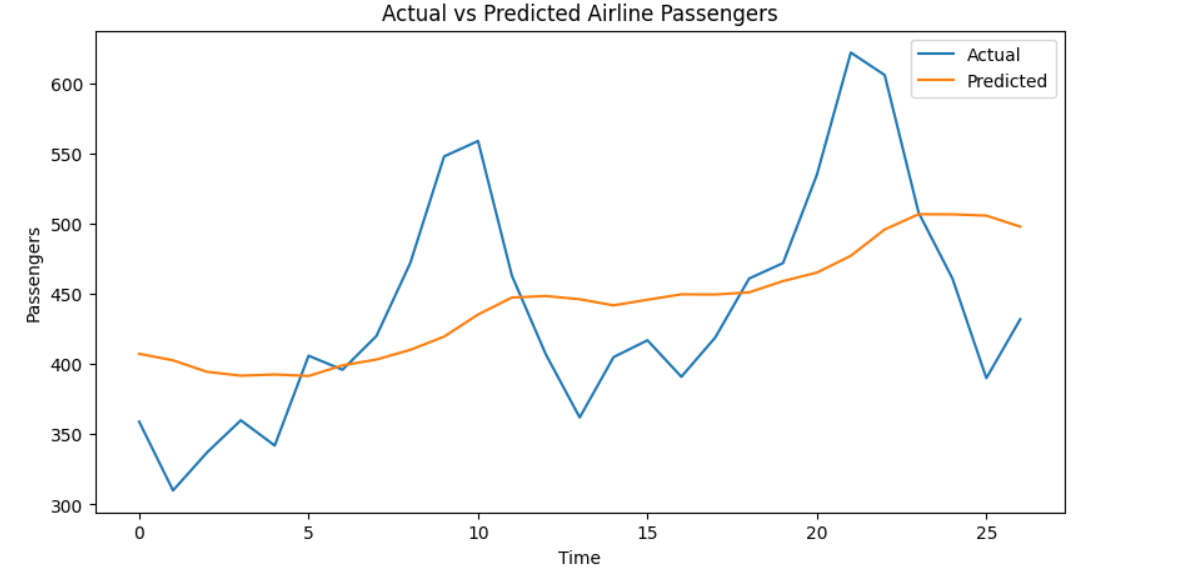
**# Step 11: Save the Model**

model.save("lstm\_time\_series.h5")

print("Model saved as 'lstm\_time\_series.h5'.")

**Output :**

****

****

**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))

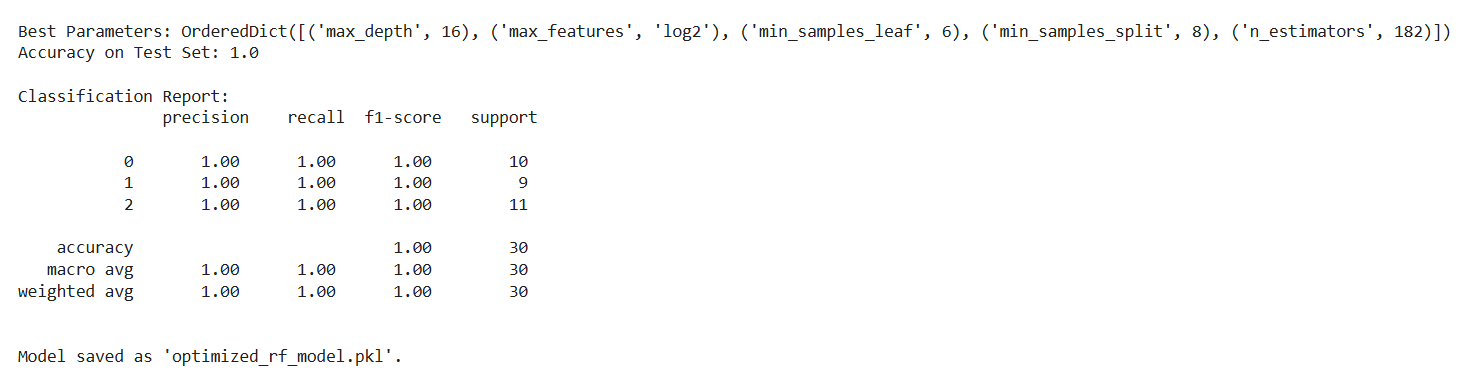
**# Optional: Save the Best Model**

import joblib

joblib.dump(best\_model, "optimized\_rf\_model.pkl")

print("\nModel saved as 'optimized\_rf\_model.pkl'.")

**Output :**

****

**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

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

{

"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

# 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 .

1. **Run the Docker container**:

bash

Copy code

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

1. **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

1. **Deploy to AWS ECS (Example)**:
   * Create an ECS cluster.
   * Use the Docker image in a task definition.
   * Deploy the task to the cluster.
2. **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)

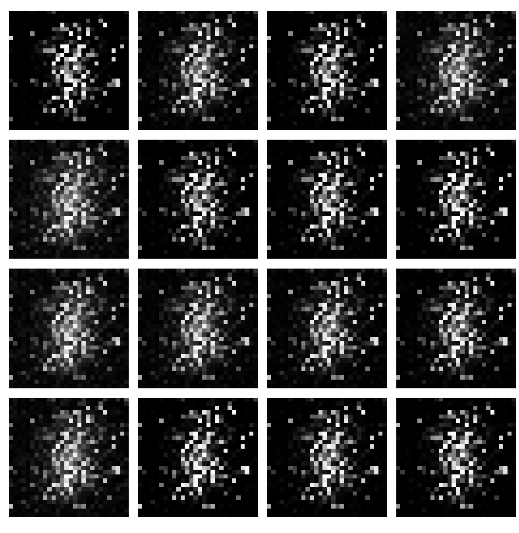
        plt.savefig(f"generated\_images/epoch\_{epoch+1}.png")

        plt.close()

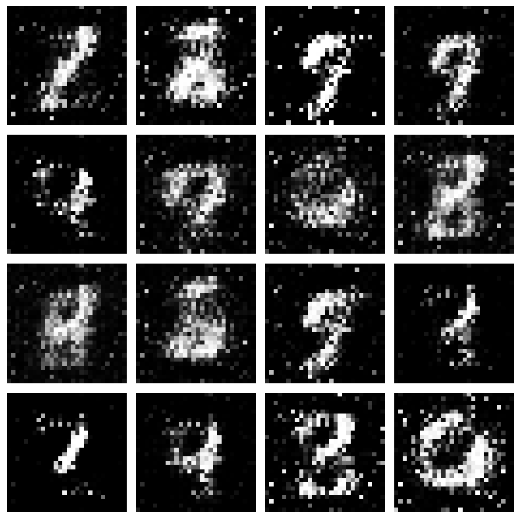
print("Training Complete. Generated images are saved in 'generated\_images' folder.")

**Output :**

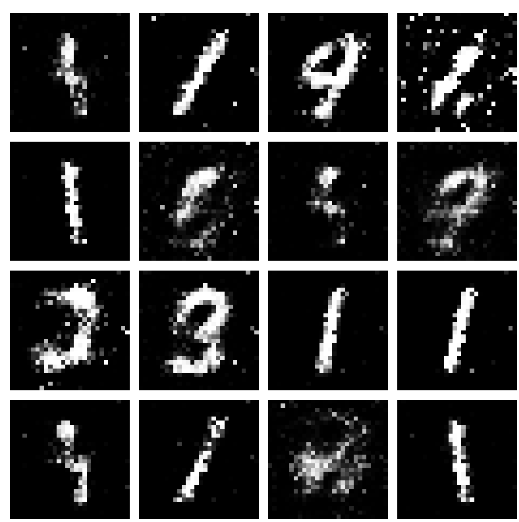
**Generate after 10 Epoch**

****

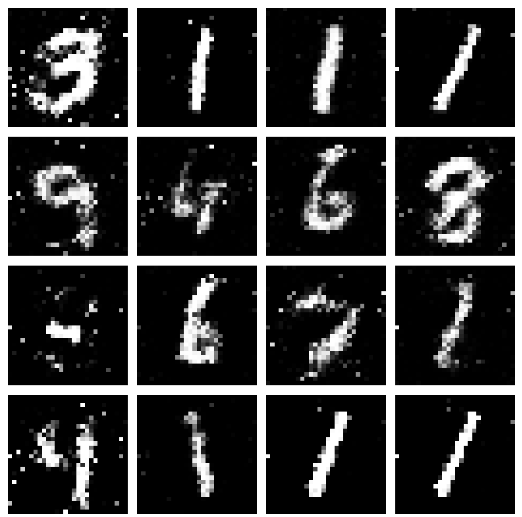
**Generate after 20 Epoch**

****

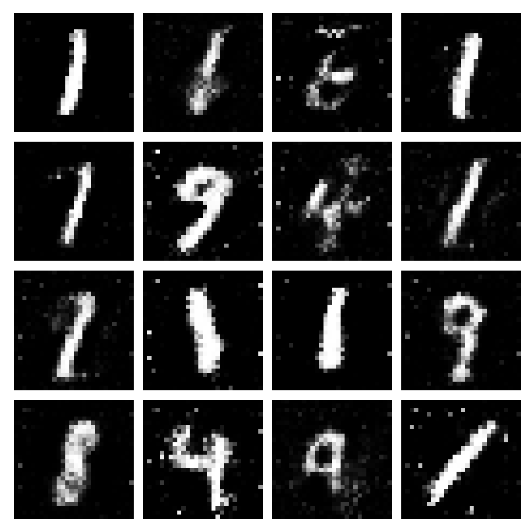
**Generate 30 Epoch**

****

**Generate after 40 Epoch**

****

**Generate after 50 Epoch**

****