**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No** | **Name** | **Date** | **Sign** |
| 1 | Implementing advanced deep learning algorithms such as recurrent neural networks (RNNs) using Python libraries like TensorFlow or PyTorch |  |  |
| 2 | Building a natural language processing (NLP) model for sentiment analysis. |  |  |
| 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 | Building a deep learning model for time series forecasting or anomaly detection |  |  |
| 9 | Using advanced optimization techniques like evolutionary algorithms or Bayesian optimization for hyperparameter tuning |  |  |
| 10 | 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 |  |  |

**PRACTICAL - 1**

**AIM:** Implementing advanced deep learning algorithms such as recurrent neural networks (RNNs) using Python libraries like TensorFlow or PyTorch.

**Recurrent Neural Networks:**

RNNs are designed to recognize patterns in sequences of data, such as time series or text. They achieve this by maintaining a hidden state that is updated at each time step based on the current input and the previous hidden state. This allows RNNs to capture temporal dependencies in the data.The basic structure of an RNN consists of:

* Input Layer: Takes the input data at each time step.
* Hidden Layer: Maintains the hidden state and updates it based on the input and the previous hidden state.
* Output Layer: Produces the output at each time step.

**Building an RNN Using Pytorch:**

1. **Import Libraries:** Bring in the required libraries, torch, torch.nn and torch.optim.
2. **Define the RNN Model:**Create a class for your RNN model by, subclassing [torch.nn.Module](https://www.geeksforgeeks.org/create-model-using-custom-module-in-pytorch/" \t "_blank).
3. **Preparing Data:** Data must be in a sequential format in order for RNNs to function properly. Preprocessing procedures like tokenization for text data, and normalization for time series data are frequently involved in this.
4. **DataLoader in PyTorch:**PyTorch provides the DataLoader class to easily handle batching, shuffling, and loading data in parallel. This is crucial for efficient training of RNNs.
5. **Train the Model:** Use a loss function and an optimizer to train your model on your dataset. Training Loop When training an RNN, the data is iterated over several times, or epochs and the model weights are updated by the use of backpropagation through time (BPTT)
6. **Evaluate the Model:** Test your model to see how well it performs on unseen data.

**Step 1: Import Libraries:**

import torch

import torch.nn as nn

import torch.optim as optim

import numpy as np

import matplotlib.pyplot as plt

**Step 2: Create Synthetic Dataset:**

# Generate sine wave data

def generate\_data(seq\_length, num\_samples):

X = []

y = []

for i in range(num\_samples):

x = np.linspace(i \* 2 \* np.pi, (i + 1) \* 2 \* np.pi, seq\_length + 1)

sine\_wave = np.sin(x)

X.append(sine\_wave[:-1]) # input sequence

y.append(sine\_wave[1:]) # target sequence

return np.array(X), np.array(y)

seq\_length = 50

num\_samples = 1000

X, y = generate\_data(seq\_length, num\_samples)

# Convert to PyTorch tensors

X = torch.tensor(X, dtype=torch.float32)

y = torch.tensor(y, dtype=torch.float32)

print(X.shape, y.shape) # Output: (1000, 50), (1000, 50)

**Output:**

torch.Size([1000, 50]) torch.Size([1000, 50])

**Step 3: Define the RNN Model:**

class SimpleRNN(nn.Module):

def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):

super(SimpleRNN, self).\_\_init\_\_()

self.rnn = nn.RNN(input\_size, hidden\_size, batch\_first=True)

self.fc = nn.Linear(hidden\_size, output\_size)

def forward(self, x):

h0 = torch.zeros(1, x.size(0), hidden\_size).to(x.device)

out, \_ = self.rnn(x, h0)

out = self.fc(out)

return out

input\_size = 1

hidden\_size = 20

output\_size = 1

model = SimpleRNN(input\_size, hidden\_size, output\_size)

**Step 4: Train the Model:**

criterion = nn.MSELoss()

optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop

num\_epochs = 100

for epoch in range(num\_epochs):

model.train()

outputs = model(X.unsqueeze(2)) # Add a dimension for input size

loss = criterion(outputs, y.unsqueeze(2))

optimizer.zero\_grad()

loss.backward()

optimizer.step()

if (epoch + 1) % 10 == 0:

print(f'Epoch [{epoch+1}/{num\_epochs}], Loss: {loss.item():.4f}')

**Step 5: Visualize the Results:**

# Make predictions

model.eval()

with torch.no\_grad():

predictions = model(X.unsqueeze(2)).squeeze(2).numpy()

# Plot results

plt.figure(figsize=(10, 6))

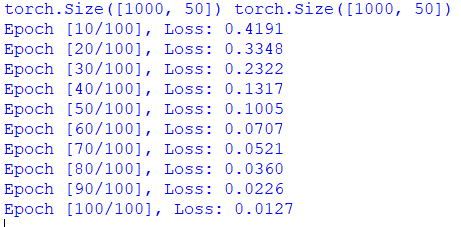
plt.plot(y[0].numpy(), label='True')

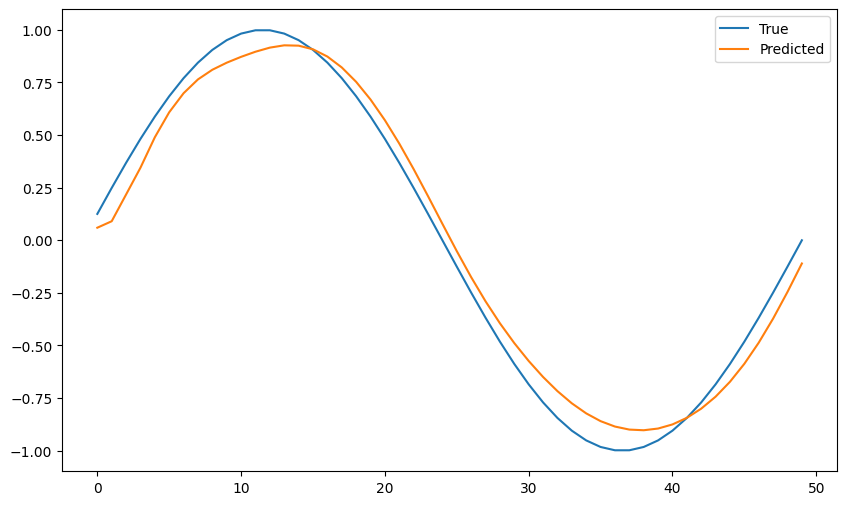
plt.plot(predictions[0], label='Predicted')

plt.legend()

plt.show()

**Output:**





**Predicting Sequential Data**

**PRACTICAL - 2**

**AIM:** Building a natural language processing (NLP) model for sentiment analysis.

**Natural Language Processing:**

NLP, or Natural Language Processing, stands for teaching machines to understand human speech and spoken words. NLP combines computational linguistics, which involves rule-based modeling of human language, with intelligent algorithms like statistical, machine, and deep learning algorithms. Together, these technologies create the smart voice assistants and chatbots we use daily.

In human speech, there are various errors, differences, and unique intonations. NLP technology, including AI chatbots, empowers machines to rapidly understand, process, and respond to large volumes of text in real-time.

**Sentiment analysis:**

Sentiment Analysis is a use case of Natural Language Processing (NLP) and comes under the category of text classification. To put it simply, Sentiment Analysis involves classifying a text into various sentiments, such as positive or negative, Happy, Sad or Neutral, etc. Thus, the ultimate goal of sentiment analysis is to decipher the underlying mood, emotion, or sentiment of a text. This is also referred to as Opinion Mining.

**Sentiment Analysis Working:**

Sentiment analysis in Python typically works by employing natural language processing (NLP) techniques to analyze and understand the sentiment expressed in text. The process involves several steps:

* **Text Preprocessing:** The text cleaning process involves removing irrelevant information, such as special characters, punctuation, and stopwords, from the text data.
* **Tokenization:** The text is divided into individual words or tokens to facilitate analysis.
* **Feature Extraction:** The text extraction process involves extracting relevant features from the text, such as words, n-grams, or even parts of speech.
* **Sentiment Classification:** Machine learning algorithms or pre-trained models are used to classify the sentiment of each text instance. Researchers achieve this through supervised learning, where they train models on labeled data, or through pre-trained models that have learned sentiment patterns from large datasets.
* **Post-processing:** The sentiment analysis results may undergo additional processing, such as aggregating sentiment scores or applying threshold rules to classify sentiments as positive, negative, or neutral.
* **Evaluation:** Researchers assess the performance of the sentiment analysis model using evaluation metrics, such as accuracy, precision, recall, or F1 score.

**Code:**

import pandas as pd

import numpy as np

from sklearn.feature\_extraction.text import re

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score

df = pd.read\_csv('tripadvisor\_hotel\_reviews.csv')

def create\_sentiment(rating):

res = 0 # neutral sentiment

if rating==1 or rating==2:

res = -1 # negative sentiment

elif rating==4 or rating==5:

res = 1 # positive sentiment

return res

df['Sentiment'] = df['Rating'].apply(create\_sentiment)

def clean\_data(review):

no\_punc = re.sub(r'[^\w\s]', '', review)

no\_digits = ''.join([i for i in no\_punc if not i.isdigit()])

return(no\_digits)

df['Review'] = df['Review'].apply(clean\_data)

tfidf = TfidfVectorizer(strip\_accents=None,

lowercase=False,

preprocessor=None)

X = tfidf.fit\_transform(df['Review'])

y = df['Sentiment']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y)

lr = LogisticRegression(solver='liblinear')

lr.fit(X\_train,y\_train)

preds = lr.predict(X\_test)

print("Accuracuy Score: ",accuracy\_score(preds,y\_test))

**Output:**



**PRACTICAL –3**

**AIM:** Creating a chatbot using advanced techniques like transformer models.

**Transformer Model:**

Transformer is a neural network architecture used for performing machine learning tasks. In 2017 Vaswani et al. published a paper “Attention is All You Need” in which the transformers architecture was introduced. The article explores the architecture, workings, and applications of transformers.

Transformer Architecture is a model that uses self-attention to transform one whole sentence into a single sentence. This is a big shift from how older models work step by step, and it helps overcome the challenges seen in models like RNNs and LSTMs.

**Code:-**

import torch

from transformers import GPT2LMHeadModel, GPT2Tokenizer

# Load the pre-trained GPT-2 model and tokenizer

model\_name = "gpt2"

tokenizer = GPT2Tokenizer.from\_pretrained(model\_name)

model = GPT2LMHeadModel.from\_pretrained(model\_name)

# Set the model to evaluation mode

model.eval()

def generate\_response(prompt, max\_length=50):

input\_ids = tokenizer.encode(prompt, return\_tensors="pt")

# Generate response

with torch.no\_grad():

output = model.generate(input\_ids, max\_length=max\_length, num\_return\_sequences=1, pad\_token\_id=50256)

response = tokenizer.decode(output[0], skip\_special\_tokens=True)

return response

print("Chatbot: Hi there! How can I help you?")

while True:

user\_input = input("You: ")

if user\_input.lower() == "exit":

print("Chatbot: Goodbye!")

break

response = generate\_response(user\_input)

print("Chatbot:", response)

**PRACTICAL - 4**

|  |
| --- |
| **AIM:** Developing a recommendation system using collaborative filtering or deep learning approaches.  **Recommendation system:**  There are a lot of applications where websites collect data from their users and use that data to predict the likes and dislikes of their users. This allows them to recommend the content that they like. Recommender systems are a way of suggesting similar items and ideas to a user’s specific way of thinking.  **Collaborative Filtering:**  In Collaborative Filtering, we tend to find similar users and recommend what similar users like. In this type of recommendation system, we don’t use the features of the item to recommend it, rather we classify the users into clusters of similar types and recommend each user according to the preference of its cluster.  There are two classes of Collaborative Filtering:   * User-based, which measures the similarity between target users and other users. * Item-based, which measures the similarity between the items that target users rate or interact with and other items. |
|  |

**Code:**

# Developing a recommendation system using collaborative filtering.

import pandas as pd

from surprise import Dataset, Reader, SVD

from surprise.model\_selection import train\_test\_split

from surprise import accuracy

# Create a sample dataset

data = {

'user\_id': [1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5],

'item\_id': [101, 102, 103, 101, 103, 104, 101, 102, 104, 102, 103, 104, 101, 103, 104],

'rating': [5, 3, 4, 4, 5, 2, 2, 5, 4, 3, 4, 5, 5, 3, 4]

}

df = pd.DataFrame(data)

# Define a Reader object

reader = Reader(rating\_scale=(1, 5)) # Adjust the rating scale according to your dataset

# Load the dataset into Surprise

data = Dataset.load\_from\_df(df[['user\_id', 'item\_id', 'rating']], reader)

# Split the dataset into training and testing sets

trainset, testset = train\_test\_split(data, test\_size=0.2)

# Create an SVD model

model = SVD()

# Train the model on the training set

model.fit(trainset)

# Make predictions on the test set

predictions = model.test(testset)

# Calculate and print RMSE

rmse = accuracy.rmse(predictions)

print(f'RMSE: {rmse}')

def get\_top\_n\_recommendations(predictions, n=10):

# Convert the predictions into a DataFrame

top\_n = {}

for uid, iid, true\_r, est, \_ in predictions:

if not uid in top\_n:

top\_n[uid] = []

top\_n[uid].append((iid, est))

# Sort the predictions for each user and retrieve the n highest ones

for uid, user\_ratings in top\_n.items():

user\_ratings.sort(key=lambda x: x[1], reverse=True)

top\_n[uid] = user\_ratings[:n]

return top\_n

# Get top 10 recommendations for each user

top\_n\_recommendations = get\_top\_n\_recommendations(predictions, n=10)

# Display recommendations for a specific user

user\_id = 1 # Replace with an actual user ID

print(f"Top 10 recommendations for user {user\_id}: {top\_n\_recommendations.get(user\_id, [])}")

**PRACTICAL -5**

|  |
| --- |
| **AIM:** Implementing a computer vision project, such as object detection or image segmentation. |

**Computer Vision:**

* Computer vision is a field of artificial intelligence (AI) that uses machine learning and neural networks to teach computers and systems to derive meaningful information from digital images, videos and other visual inputs—and to make recommendations or take actions when they see defects or issues.
* if AI enables computers to think, computer vision enables them to see, observe and understand.
* Computer vision works much the same as human vision, except humans have a head start. Human sight has the advantage of lifetimes of context to train how to tell objects apart, how far away they are, whether they are moving or something is wrong with an image.

**Object Detection:**

Object detection is a technique that uses neural networks to localize and classify objects in images. This computer vision task has a wide range of applications, from medical imaging to self-driving cars.

Object detection is a computer vision task that aims to locate objects in digital images. As such, it is an instance of artificial intelligence that consists of training computers to see as humans do, specifically by recognizing and classifying objects according to semantic categories.

**Code:**

from math import sqrt

import pandas as pd

import numpy as np

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics import mean\_squared\_error

from skforecast.datasets import fetch\_dataset

data = fetch\_dataset('bike\_sharing', raw=True)

dataset = data['users'].values.reshape(-1, 1)

# scale down the values between 0 and 1

scaler = MinMaxScaler(feature\_range=(0, 1))

dataset = scaler.fit\_transform(dataset)

# train-test split

train\_size = int(len(dataset) \* 0.67)

test\_size = len(dataset) - train\_size

train, test = dataset[0:train\_size,:], dataset[train\_size:len(dataset),:]

def create\_dataset(dataset, look\_back=1):

dataX, dataY = [], []

for i in range(len(dataset)-look\_back-1):

a = dataset[i:(i+look\_back), 0]

dataX.append(a)

dataY.append(dataset[i + look\_back, 0])

return np.array(dataX), np.array(dataY)

look\_back = 1

# separate the ylabel

trainX, trainY = create\_dataset(train, look\_back)

testX, testY = create\_dataset(test, look\_back)

trainX = np.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))

testX = np.reshape(testX, (testX.shape[0], 1, testX.shape[1]))

model = Sequential()

model.add(LSTM(4, input\_shape=(1, look\_back)))

model.add(Dense(1))

model.compile(loss='mean\_squared\_error', optimizer='adam')

model.fit(trainX, trainY, epochs=3, batch\_size=1, verbose=2)

# get the final rmse after prediction

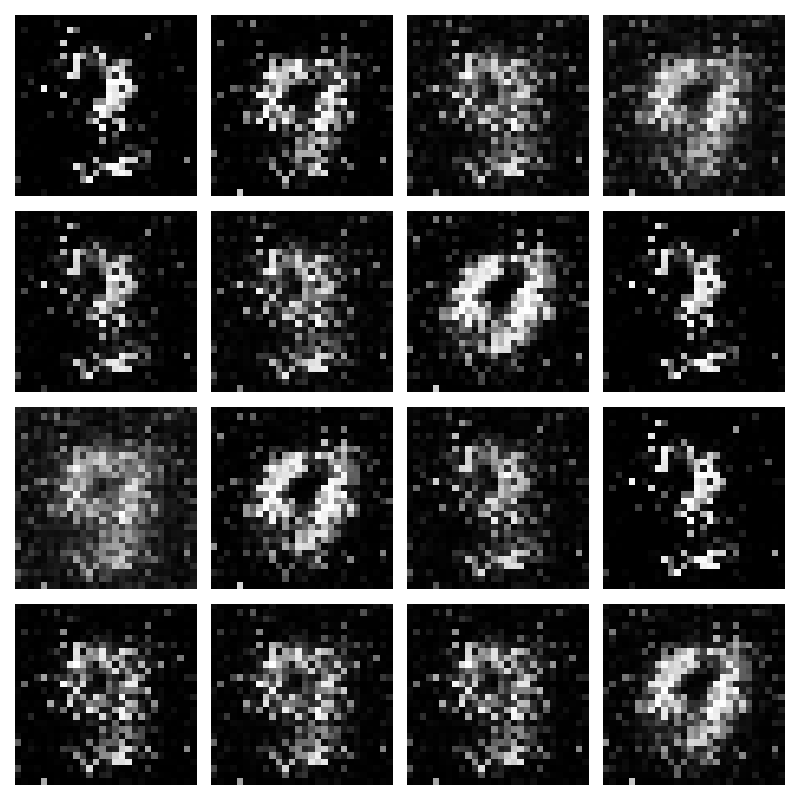
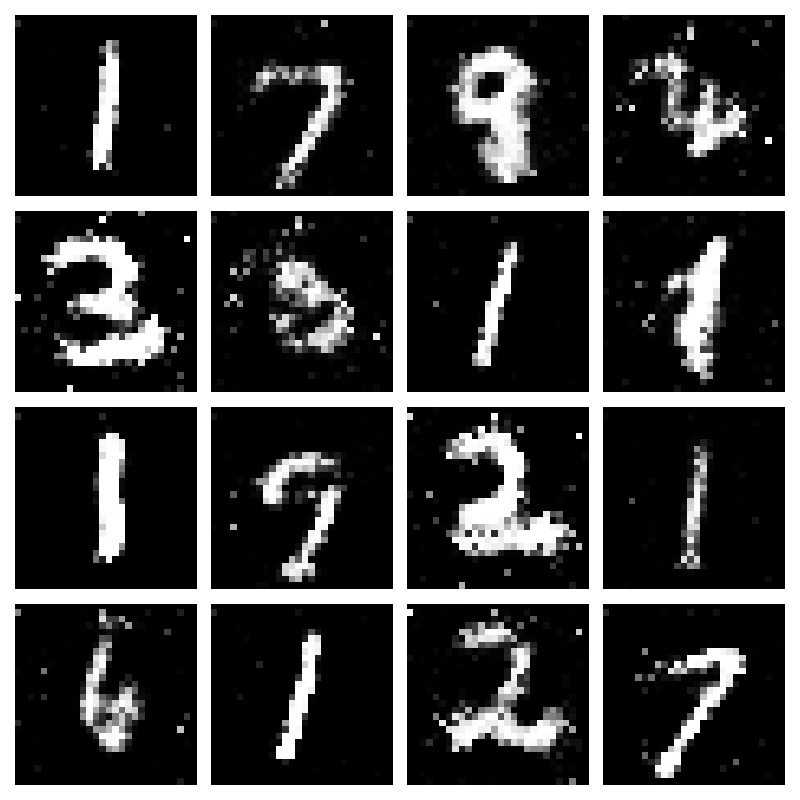
testPredict = model.predict(testX)

testScore = sqrt(mean\_squared\_error(testY, testPredict[:,0]))

print('Test Score: %.2f RMSE' % (testScore))

**PRACTICAL -6**

|  |
| --- |
| **AIM:** Training a generative adversarial network (GAN) for generating realistic images.  **Generative Adversarial Network (GAN):**  Generative Adversarial Networks (GANs) are a powerful class of neural networks that are  used for an unsupervised learning. GANs are made up of two neural networks, a discriminator  and a generator. They use adversarial training to produce artificial data that is identical to  actual data.   * The Generator attempts to fool the Discriminator, which is tasked with accurately   distinguishing between produced and genuine data, by producing random noise samples.   * Realistic, high-quality samples are produced as a result of this competitive interaction,   which drives both networks toward advancement.   * GANs are proving to be highly versatile artificial intelligence tools, as evidenced by   their extensive use in image synthesis, style transfer, and text-to-image synthesis.   * They have also revolutionized generative modeling.   Through adversarial training, these models engage in a competitive interplay until the  generator becomes adept at creating realistic samples, fooling the discriminator approximately  half the time.  **Code:**  **Step 1: Install Required Libraries**  pip install torch torchvision matplotlib numpy  **Step 2: Python Code for GAN Training**  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:** |

** **

**PRACTICAL -7**

|  |
| --- |
| **AIM:** Applying reinforcement learning algorithms to solve complex decision-making problems.  **Reinforcement Learning:**   * Reinforcement learning can be defined as a machine learning technique involving an agent who needs to decide which actions it needs to do to perform a task that has been assigned to it most effectively. * For this, rewards are assigned to the different actions that the agent can take at different situations or states of the environment. Initially, the agent has no idea about the best or correct actions. * Using reinforcement learning, it explores its action choices via trial and error and figures out the best set of actions for completing its assigned task. * The basic idea behind a reinforcement learning agent is to learn from experience. Just like humans learn lessons from their past successes and mistakes, reinforcement learning agents do the same – when they do something “good” they get a reward, but, if they do something “bad”, they get penalized. The reward reinforces the good actions while the penalty avoids the bad ones.   **Types of Reinforcement Learning:**  There are two types of reinforcement learning: model-based and model-free.   * Model-Based Reinforcement Learning:   With model-based reinforcement learning (RL), there’s a model that an agent uses to create additional experiences. Think of this model as a mental image that the agent can analyze to assess whether particular strategies could work.   * Model-Free Reinforcement Learning:   In this case, an agent doesn’t rely on a model. Instead, the basis for its actions lies in direct interactions with the environment. An agent tries different scenarios and tests whether they’re successful. If yes, the agent will keep repeating them. If not, it will try another scenario until it finds the right one.  **Code:**  import numpy as np  import gym  import random  # Create the Taxi environment  env = gym.make("Taxi-v3")  # Initialize the Q-table  q\_table = np.zeros((env.observation\_space.n, env.action\_space.n))  # Hyperparameters  alpha = 0.1 # Learning rate  gamma = 0.6 # Discount factor  epsilon = 0.1 # Exploration rate  num\_episodes = 10000 # Number of episodes  # Training the agent  for i in range(num\_episodes):  state = env.reset()  done = False  while not done:  # Choose action using epsilon-greedy policy  if random.uniform(0, 1) < epsilon:  action = env.action\_space.sample() # Explore action space  else:  action = np.argmax(q\_table[state]) # Exploit learned values  # Take action and observe the result  next\_state, reward, done, \_ = env.step(action)  # Update Q-value using the Bellman equation  old\_value = q\_table[state, action]  next\_max = np.max(q\_table[next\_state])  new\_value = (1 - alpha) \* old\_value + alpha \* (reward + gamma \* next\_max)  q\_table[state, action] = new\_value  # Transition to the next state  state = next\_state  # Evaluate the agent's performance  total\_epochs, total\_penalties = 0, 0  episodes = 100  for \_ in range(episodes):  state = env.reset()  epochs, penalties, reward = 0, 0, 0  done = False  while not done:  action = np.argmax(q\_table[state]) # Choose the best action  state, reward, done, \_ = env.step(action)  if reward == -10: # Penalty for illegal actions  penalties += 1  epochs += 1  total\_penalties += penalties  total\_epochs += epochs  print(f"Results after {episodes} episodes:")  print(f"Average timesteps per episode: {total\_epochs / episodes}")  print(f"Average penalties per episode: {total\_penalties / episodes}") |

**PRACTICAL - 8**

|  |
| --- |
| **AIM:** Building a deep learning model for time series forecasting or anomaly detection.  **Time Series Forecasting:**   * A time series is a collection of data points recorded at regular intervals over time. Time series forecasting refers to the process of using historical data to predict future values in a sequence of observations. * In finance, for example, forecasting models help predict stock prices and market trends, enabling investors to make informed, timely decisions. * Retail businesses rely on time series forecasting to anticipate customer demand, ensuring that inventory is aligned with expected sales. Similarly, energy companies use these models to forecast consumption patterns, allowing them to manage resources effectively and optimize energy distribution.   **Anomaly Detection:**  Anomaly detection is the process of finding outlier values in a series of data. That process assumes you have data that falls within a certain understood range (based on historical data, for example), and that occasional values outside that range happen fairly infrequently.  Supervised anomaly detection allows historical data to be labeled as “normal” and “abnormal,” in order to develop models to apply those labels to new data. Anomaly detection can be applied to unlabeled data in unsupervised machine learning, using the historical data to analyze the probability distribution of values that can then determine if a new value is unlikely and therefore an anomaly. Anomaly detection can be performed on a single variable or on a combination of variables. Some examples of multivariate anomaly detection methods include cluster-based local outlier factor, histogram-based outlier detection, isolation forest, and k-nearest neighbors. |

**Code:**

**Step 1: Install Required Libraries**

pip install numpy pandas matplotlib scikit-learn tensorflow

**Step 2: Python Code for Time Series Forecasting**

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()

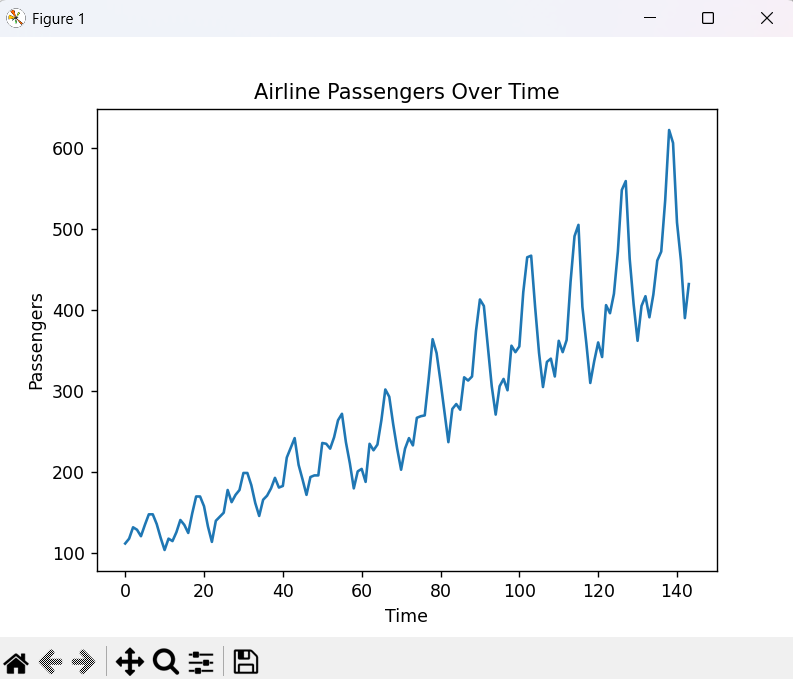
plt.show()

# Step 11: Save the Model

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

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

**Output:**



**PRACTICAL – 9**

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

Bayesian Optimization for hyperparameter tuning in Python. We'll use the scikit-optimize library to optimize hyperparameters for a Random Forest Classifier trained on the Iris dataset.

**Step 1: Install Required Libraries**

pip install numpy pandas scikit-learn scikit-optimize

**Step 2: Python Code for Bayesian Optimization**

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))

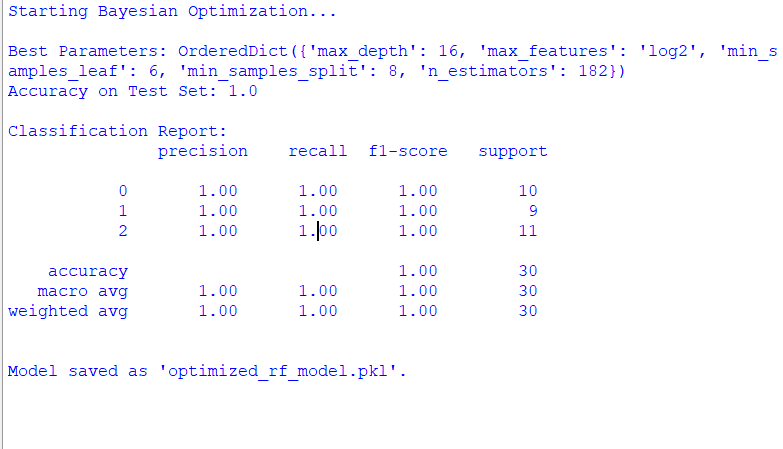
# Optional: Save the Best Model

import joblib

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

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

**Output:**



**PRACTICAL - 10**

**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**

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**

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)

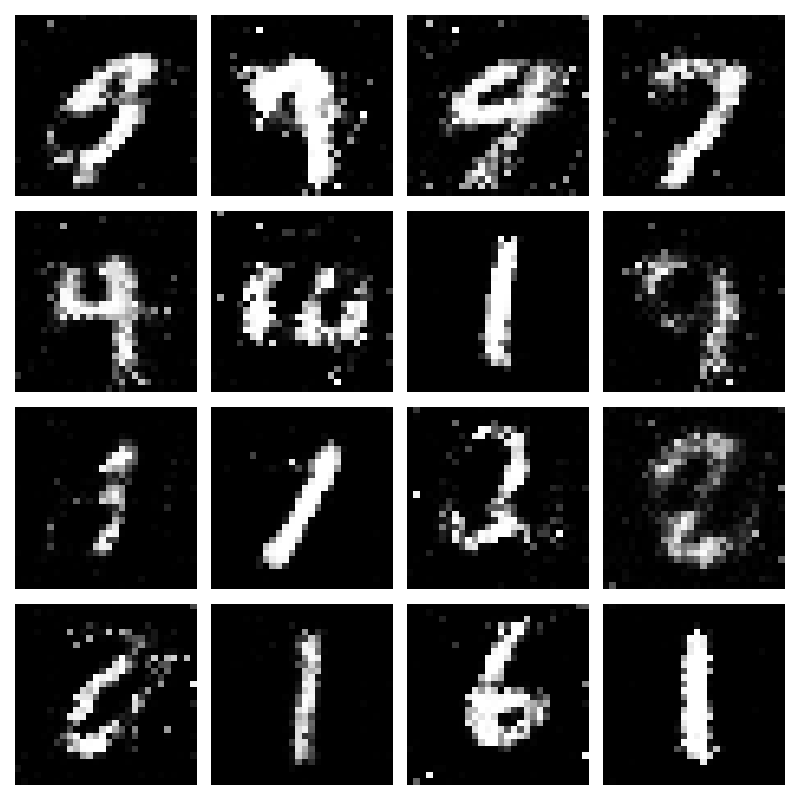
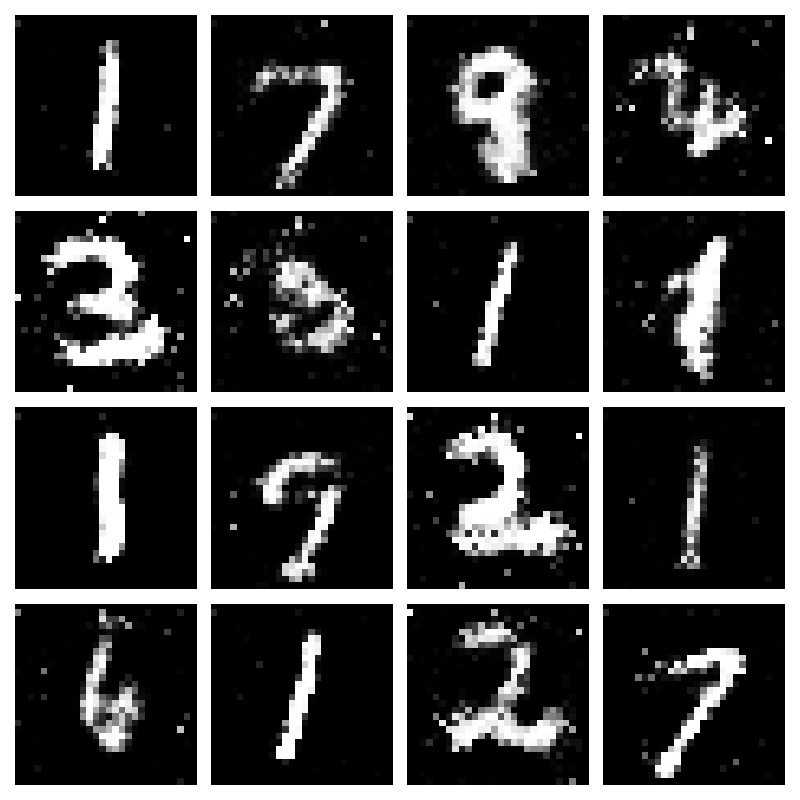
generated\_text = tokenizer.decode(outputs[0], skip\_special\_tokens=True)

print("\nGenerated Text:\n")

print(generated\_text)

**Output:**

****

** **