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.
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()
```

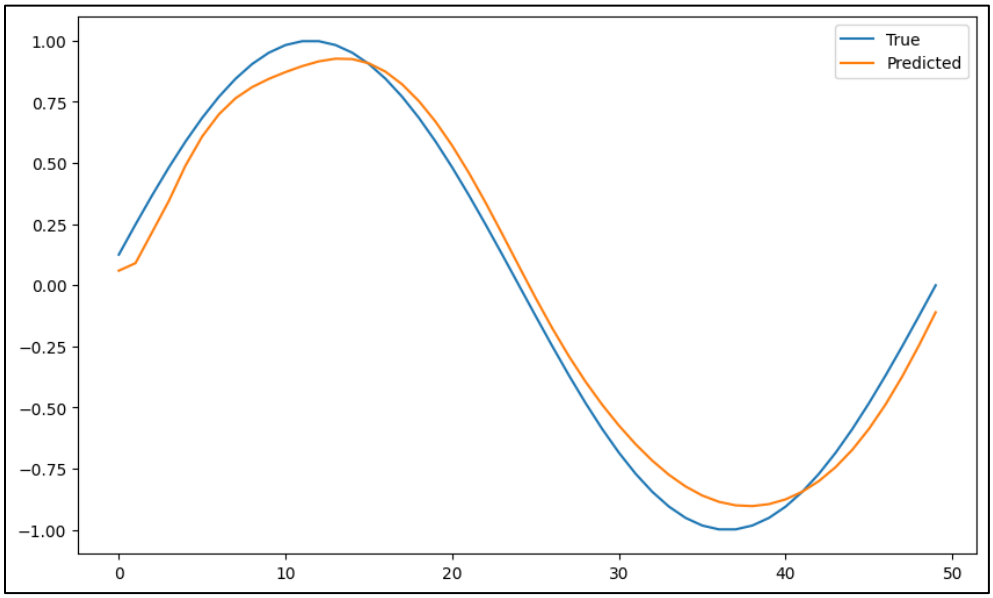
```
optimizer = optim.Adam(model.parameters(), lr=0.001)

# 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:

```
torch.Size([1000, 50]) torch.Size([1000, 50])
Epoch [10/100], Loss: 0.4191
Epoch [20/100], Loss: 0.3348
Epoch [30/100], Loss: 0.2322
Epoch [40/100], Loss: 0.1317
Epoch [50/100], Loss: 0.1005
Epoch [60/100], Loss: 0.0707
Epoch [70/100], Loss: 0.0521
Epoch [80/100], Loss: 0.0360
Epoch [90/100], Loss: 0.0226
Epoch [100/100], Loss: 0.0127
```



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:

```
Accuracuy Score:  0.8588717548311536
```

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(
```

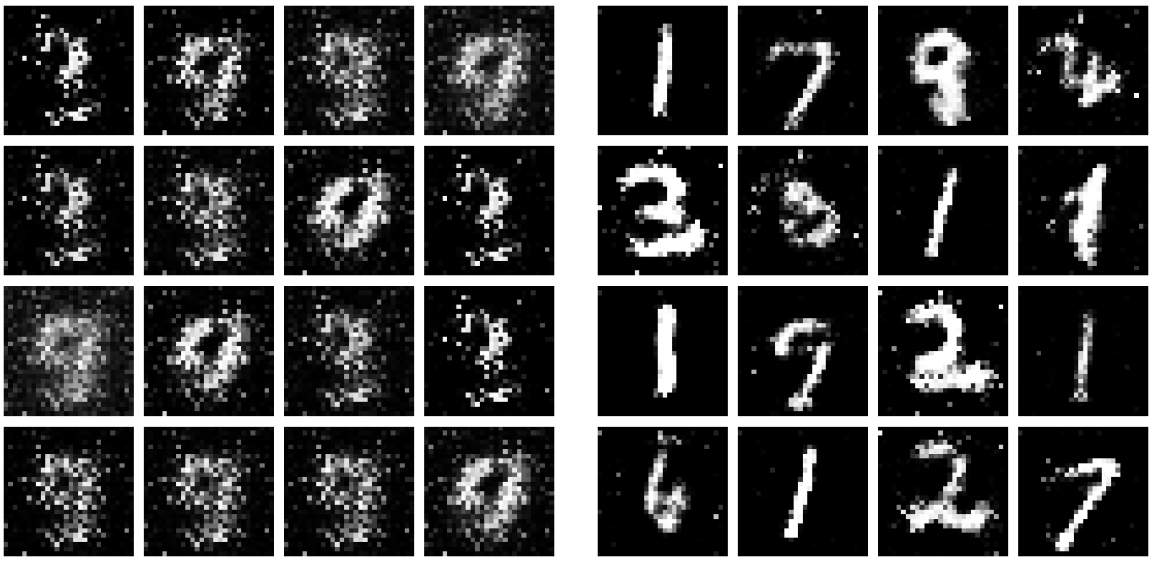
```
            nn.Linear(noise_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, img_dim),
            nn.Tanh(),
        )

    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:

```
Epoch [1/50] | Loss D: 0.3080 | Loss G: 3.6539
Epoch [2/50] | Loss D: 0.5798 | Loss G: 2.0501
Epoch [3/50] | Loss D: 0.0928 | Loss G: 3.2668
Epoch [4/50] | Loss D: 0.1919 | Loss G: 2.8874
Epoch [5/50] | Loss D: 0.1597 | Loss G: 4.7366
Epoch [6/50] | Loss D: 0.3886 | Loss G: 5.6996
Epoch [7/50] | Loss D: 0.1156 | Loss G: 5.9367
Epoch [8/50] | Loss D: 0.2688 | Loss G: 6.0902
Epoch [9/50] | Loss D: 0.2411 | Loss G: 4.5448
Epoch [10/50] | Loss D: 0.0440 | Loss G: 4.6899
Epoch [11/50] | Loss D: 0.2975 | Loss G: 4.1305
Epoch [12/50] | Loss D: 0.1045 | Loss G: 3.4638
Epoch [13/50] | Loss D: 0.1078 | Loss G: 3.2226
Epoch [14/50] | Loss D: 0.1241 | Loss G: 3.5415
Epoch [15/50] | Loss D: 0.2899 | Loss G: 3.3469
Epoch [16/50] | Loss D: 0.3462 | Loss G: 2.9702
Epoch [17/50] | Loss D: 0.2164 | Loss G: 3.2247
Epoch [18/50] | Loss D: 0.1157 | Loss G: 3.7014
Epoch [19/50] | Loss D: 0.1698 | Loss G: 3.4677
Epoch [20/50] | Loss D: 0.0605 | Loss G: 4.3014
Epoch [21/50] | Loss D: 0.1138 | Loss G: 2.8076
Epoch [22/50] | Loss D: 0.2044 | Loss G: 3.3167
Epoch [23/50] | Loss D: 0.1925 | Loss G: 3.8898
Epoch [24/50] | Loss D: 0.3526 | Loss G: 2.9908
Epoch [25/50] | Loss D: 0.1817 | Loss G: 3.0945
Epoch [26/50] | Loss D: 0.2835 | Loss G: 3.0077
Epoch [27/50] | Loss D: 0.3144 | Loss G: 3.0285
Epoch [28/50] | Loss D: 0.3102 | Loss G: 3.2433
```



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
```

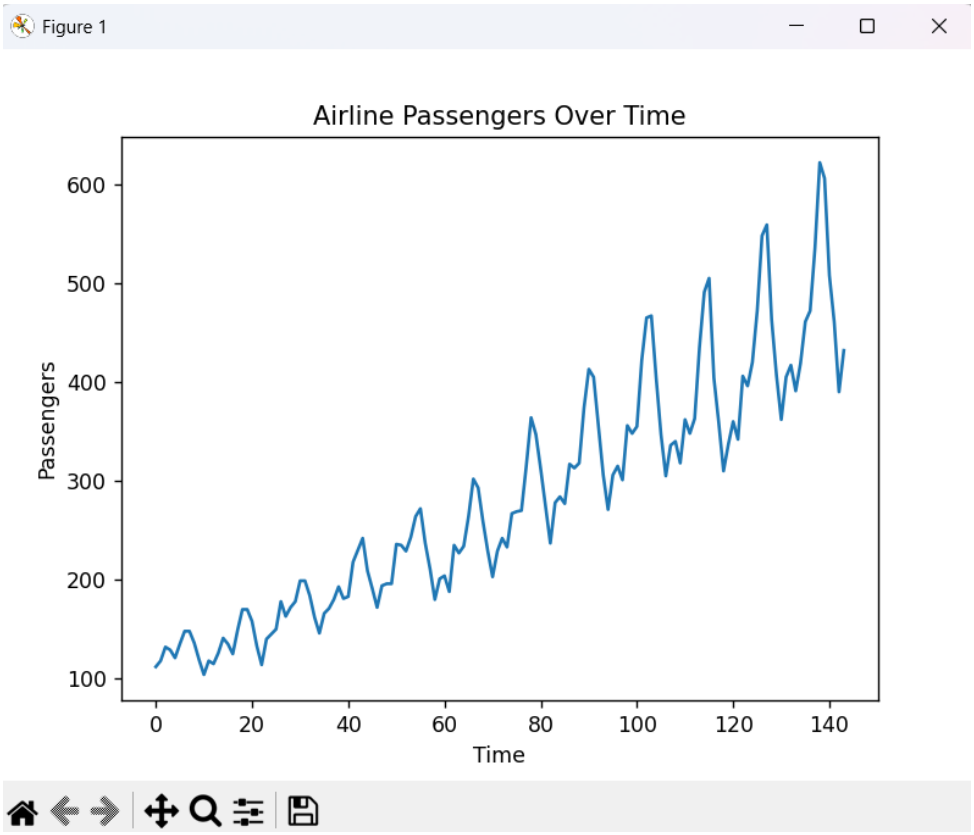
Step 2: Python Code for Time Series Forecasting

```
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:

```
Starting Bayesian Optimization...

Best Parameters: OrderedDict({'max_depth': 16, 'max_features': 'log2', 'min_s
amples_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 - 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:

```
Epoch [1/50] | Loss D: 0.3080 | Loss G: 3.6539
Epoch [2/50] | Loss D: 0.5798 | Loss G: 2.0501
Epoch [3/50] | Loss D: 0.0928 | Loss G: 3.2668
Epoch [4/50] | Loss D: 0.1919 | Loss G: 2.8874
Epoch [5/50] | Loss D: 0.1597 | Loss G: 4.7366
Epoch [6/50] | Loss D: 0.3886 | Loss G: 5.6996
Epoch [7/50] | Loss D: 0.1156 | Loss G: 5.9367
Epoch [8/50] | Loss D: 0.2688 | Loss G: 6.0902
Epoch [9/50] | Loss D: 0.2411 | Loss G: 4.5448
Epoch [10/50] | Loss D: 0.0440 | Loss G: 4.6899
Epoch [11/50] | Loss D: 0.2975 | Loss G: 4.1305
Epoch [12/50] | Loss D: 0.1045 | Loss G: 3.4638
Epoch [13/50] | Loss D: 0.1078 | Loss G: 3.2226
Epoch [14/50] | Loss D: 0.1241 | Loss G: 3.5415
Epoch [15/50] | Loss D: 0.2899 | Loss G: 3.3469
Epoch [16/50] | Loss D: 0.3462 | Loss G: 2.9702
Epoch [17/50] | Loss D: 0.2164 | Loss G: 3.2247
Epoch [18/50] | Loss D: 0.1157 | Loss G: 3.7014
Epoch [19/50] | Loss D: 0.1698 | Loss G: 3.4677
Epoch [20/50] | Loss D: 0.0605 | Loss G: 4.3014
Epoch [21/50] | Loss D: 0.1138 | Loss G: 2.8076
Epoch [22/50] | Loss D: 0.2044 | Loss G: 3.3167
Epoch [23/50] | Loss D: 0.1925 | Loss G: 3.8898
```