**Practical 01**

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

**CNN using PyTorch**

import torch

import torch.nn as nn

import torch.optim as optim

import torchvision

import torchvision.transforms as transforms

import torch.nn.functional as F

# Prepare the dataset

transform = transforms.Compose(

[transforms.ToTensor(),

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

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,

download=True, transform=transform)

trainloader = torch.utils.data.DataLoader(trainset, batch\_size=4,

shuffle=True, num\_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,

download=True, transform=transform)

testloader = torch.utils.data.DataLoader(testset, batch\_size=4,

shuffle=False, num\_workers=2)

classes = ('plane', 'car', 'bird', 'cat',

'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

# Define the CNN architecture

class Net(nn.Module):

def \_\_init\_\_(self):

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

self.conv1 = nn.Conv2d(3, 6, 5)

self.pool = nn.MaxPool2d(2, 2)

self.conv2 = nn.Conv2d(6, 16, 5)

self.fc1 = nn.Linear(16 \* 5 \* 5, 120)

self.fc2 = nn.Linear(120, 84)

self.fc3 = nn.Linear(84, 10)

def forward(self, x):

x = self.pool(F.relu(self.conv1(x)))

x = self.pool(F.relu(self.conv2(x)))

x = x.view(-1, 16 \* 5 \* 5)

x = F.relu(self.fc1(x))

x = F.relu(self.fc2(x))

x = self.fc3(x)

return x

net = Net()

# Define loss function and optimizer

criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

# Train the network

for epoch in range(2): # loop over the dataset multiple times

running\_loss = 0.0

for i, data in enumerate(trainloader, 0):

inputs, labels = data

optimizer.zero\_grad()

outputs = net(inputs)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

running\_loss += loss.item()

if i % 2000 == 1999: # print every 2000 mini-batches

print('[%d, %5d] loss: %.3f' %

(epoch + 1, i + 1, running\_loss / 2000))

running\_loss = 0.0

print('Finished Training')

# Test the network

correct = 0

total = 0

with torch.no\_grad():

for data in testloader:

images, labels = data

outputs = net(images)

\_, predicted = torch.max(outputs.data, 1)

total += labels.size(0)

correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (

100 \* correct / total))

**Output:**

[1, 2000] loss: 2.279  
[1, 4000] loss: 1.992  
[1, 6000] loss: 1.718  
[1, 8000] loss: 1.589  
[1, 10000] loss: 1.513  
[1, 12000] loss: 1.492  
[2, 2000] loss: 1.410  
[2, 4000] loss: 1.375  
[2, 6000] loss: 1.366  
[2, 8000] loss: 1.343  
[2, 10000] loss: 1.325  
[2, 12000] loss: 1.263  
Finished Training  
**Accuracy of the network on the 10000 test images: 54 %**

**Recurrent Neural Networks**

[RNNs](https://www.geeksforgeeks.org/introduction-to-recurrent-neural-network/) 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.

**Steps to Build an RNN**

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)

**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:**

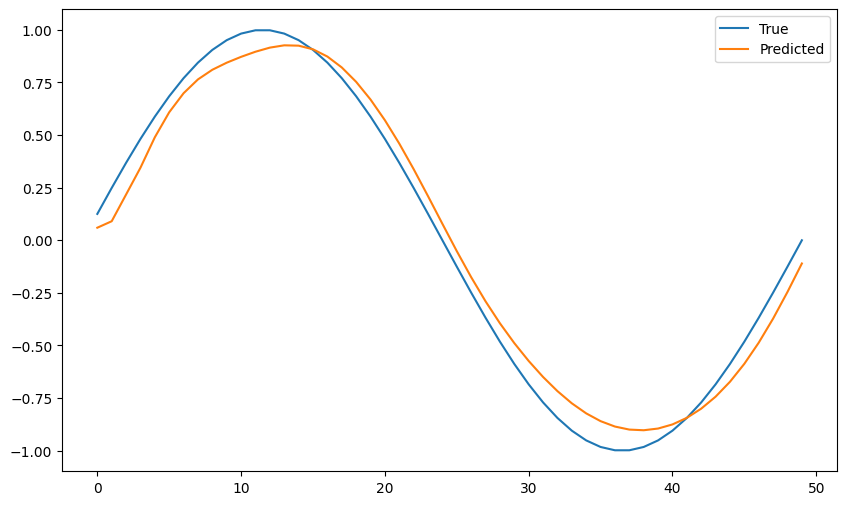
**Step 2**

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

Step 4

Epoch [10/100], Loss: 0.3548  
Epoch [20/100], Loss: 0.2653  
Epoch [30/100], Loss: 0.1757  
Epoch [40/100], Loss: 0.0921  
Epoch [50/100], Loss: 0.0592  
Epoch [60/100], Loss: 0.0421  
Epoch [70/100], Loss: 0.0306  
Epoch [80/100], Loss: 0.0222  
Epoch [90/100], Loss: 0.0151  
Epoch [100/100], Loss: 0.0093

**Step 5**



**Practical 02**

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

Sentiment analysis using NLP is a method that identifies the emotional state or sentiment behind a situation, often using NLP to analyze text data. Language serves as a mediator for human communication, and each statement carries a sentiment, which can be positive, negative, or neutral.

**Step by Step procedure to Implement Sentiment Analysis**

**Step1: Basic Python Libraries**

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from wordcloud import WordCloud

import re

**Step2: Natural Language Processing**

import nltk

from nltk.corpus import stopwords

from nltk.stem import WordNetLemmatizer

### Step3: Scikit-Learn (Machine Learning Library for Python)

from sklearn.feature\_extraction.text import CountVectorizer

from sklearn.model\_selection import GridSearchCV

from sklearn.ensemble import RandomForestClassifier

### Step4: Evaluation Metrics

from sklearn.metrics import accuracy\_score,precision\_score,recall\_score,confusion\_matrix,roc\_curve,classification\_report

from scikitplot.metrics import plot\_confusion\_matrix

### Step5: Evaluate Dataset

df\_train = pd.read\_csv("train.txt",delimiter=';',names=['text','label'])

df\_val = pd.read\_csv("val.txt",delimiter=';',names=['text','label'])

df = pd.concat([df\_train,df\_val])

df.reset\_index(inplace=True,drop=True)

import pandas as pd

df\_train = pd.read\_csv("train.txt",delimiter=';',names=['text','label'])

df\_val = pd.read\_csv("val.txt",delimiter=';',names=['text','label'])

df = pd.concat([df\_train,df\_val])

df.reset\_index(inplace=True,drop=True)

print("Shape of the DataFrame:",df.shape)

print(df.sample(5))

def custom\_encoder(df):

df.replace(to\_replace ="surprise", value =1, inplace=True)

df.replace(to\_replace ="love", value =1, inplace=True)

df.replace(to\_replace ="joy", value =1, inplace=True)

df.replace(to\_replace ="fear", value =0, inplace=True)

df.replace(to\_replace ="anger", value =0, inplace=True)

df.replace(to\_replace ="sadness", value =0, inplace=True)

custom\_encoder(df['label'])

### Step6: Data Pre-processing

#object of WordNetLemmatizer

lm = WordNetLemmatizer()

def text\_transformation(df\_col):

corpus = []

for item in df\_col:

new\_item = re.sub('[^a-zA-Z]',' ',str(item))

new\_item = new\_item.lower()

new\_item = new\_item.split()

new\_item = [lm.lemmatize(word) for word in new\_item if word not in set(stopwords.words('english'))]

corpus.append(' '.join(str(x) for x in new\_item))

return corpus

corpus = text\_transformation(df['text'])

rcParams['figure.figsize'] = 20,8

word\_cloud = ""

for row in corpus:

for word in row:

word\_cloud+=" ".join(word)

wordcloud = WordCloud(width = 1000, height = 500,background\_color ='white',min\_font\_size = 10).generate(word\_cloud)

plt.imshow(wordcloud)

### Step7: Bag of Words

cv = CountVectorizer(ngram\_range=(1,2))

traindata = cv.fit\_transform(corpus)

X = traindata

y = df.label

parameters = {'max\_features': ('auto','sqrt'),

'n\_estimators': [500, 1000, 1500],

'max\_depth': [5, 10, None],

'min\_samples\_split': [5, 10, 15],

'min\_samples\_leaf': [1, 2, 5, 10],

'bootstrap': [True, False]}

grid\_search = GridSearchCV(RandomForestClassifier(),parameters,cv=5,return\_train\_score=True,n\_jobs=-1)

grid\_search.fit(X,y)

grid\_search.best\_params\_

{

'bootstrap': True,

'max\_depth': None,

'max\_features': 'auto',

'min\_samples\_leaf': 1,

'min\_samples\_split': 5,

'n\_estimators': 500

}

for i in range(432):

print('Parameters: ',grid\_search.cv\_results\_['params'][i])

print('Mean Test Score: ',grid\_search.cv\_results\_['mean\_test\_score'][i])

print('Rank: ',grid\_search.cv\_results\_['rank\_test\_score'][i])

rfc = RandomForestClassifier(max\_features=grid\_search.best\_params\_['max\_features'], max\_depth=grid\_search.best\_params\_['max\_depth'],

n\_estimators=grid\_search.best\_params\_['n\_estimators'], min\_samples\_split=grid\_search.best\_params\_['min\_samples\_split'], min\_samples\_leaf=grid\_search.best\_params\_['min\_samples\_leaf'],

bootstrap=grid\_search.best\_params\_['bootstrap'])

rfc.fit(X,y)

### Step8: Test Data Transformation

test\_df = pd.read\_csv('test.txt',delimiter=';',names=['text','label'])

X\_test,y\_test = test\_df.text,test\_df.label

#encode the labels into two classes , 0 and 1

test\_df = custom\_encoder(y\_test)

#pre-processing of text

test\_corpus = text\_transformation(X\_test)

#convert text data into vectors

testdata = cv.transform(test\_corpus)

#predict the target

predictions = rfc.predict(testdata)

### Step9: Model Evaluation

rcParams['figure.figsize'] = 10,5

plot\_confusion\_matrix(y\_test,predictions)

acc\_score = accuracy\_score(y\_test,predictions)

pre\_score = precision\_score(y\_test,predictions)

rec\_score = recall\_score(y\_test,predictions)

print('Accuracy\_score: ',acc\_score)

print('Precision\_score: ',pre\_score)

print('Recall\_score: ',rec\_score)

print("-"\*50)

cr = classification\_report(y\_test,predictions)

print(cr)

### Step10: Roc Curve

predictions\_probability = rfc.predict\_proba(testdata)

fpr,tpr,thresholds = roc\_curve(y\_test,predictions\_probability[:,1])

plt.plot(fpr,tpr)

plt.plot([0,1])

plt.title('ROC Curve')

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.show()

Now, we will check for **custom input** as well and let our model identify the sentiment of the input statement.

**Predict for Custom Input:**

def expression\_check(prediction\_input):

if prediction\_input == 0:

print("Input statement has Negative Sentiment.")

elif prediction\_input == 1:

print("Input statement has Positive Sentiment.")

else:

print("Invalid Statement.")

# function to take the input statement and perform the same transformations we did earlier

def sentiment\_predictor(input):

input = text\_transformation(input)

transformed\_input = cv.transform(input)

prediction = rfc.predict(transformed\_input)

expression\_check(prediction)

input1 = ["Sometimes I just want to punch someone in the face."]

input2 = ["I bought a new phone and it's so good."]

sentiment\_predictor(input1)

sentiment\_predictor(input2)

**negetiveOutput:**

## Text Classification

!pip install datasets

import nltk

import pandas as pd

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

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.naive\_bayes import MultinomialNB

from sklearn.metrics import accuracy\_score, confusion\_matrix

from sklearn.datasets import load\_files

from datasets import load\_dataset

nltk.download('stopwords')

nltk.download('punkt')

nltk.download('punkt\_tab')

from datasets import load\_dataset

dataset = load\_dataset('imdb')

print(dataset)

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

stop\_words = set(stopwords.words('english'))

def preprocess\_text(text):

  text = text.lower()

  words = word\_tokenize(text)

  words = [word for word in words if word.isalpha()]

  words = [word for word in words if word not in stop\_words]

  return " ".join(words)

processed\_texts = [preprocess\_text(text) for text in dataset['train']['text']]

vectorizer = TfidfVectorizer(max\_features=1000)

X = vectorizer.fit\_transform(processed\_texts)

y = dataset['train']['label']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

model = MultinomialNB()

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

y\_pred = model.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy \* 100:.2f}%")

cm = confusion\_matrix(y\_test, y\_pred)

print("Confusion Matrix:")

print(cm)

def predict\_sentiment(review\_text):

  processed\_review = preprocess\_text(review\_text)

  review\_vector = vectorizer.transform([processed\_review])

  prediction = model.predict(review\_vector)

  if prediction == 1:

    return "Positive"

  else:

    return "Negative"

new\_review = "I absolutely loved this movie! The plot was amazing and the acting was superb."

predicted\_sentiment = predict\_sentiment(new\_review)

print(f"The sentiment of the review is: {predicted\_sentiment}")

#### Output:

Accuracy: 83.42%

Confusion Matrix:

[[2053 462]

[ 367 2118]]

The sentiment of the review is: Positive

**Practical 03**

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

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)

**Output:**



**Practical 04**

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

!pip install scikit-surprise==1.1.3

import pandas as pd

from surprise import Dataset, Reader

from surprise import KNNBasic, accuracy # Import KNNBasic algorithm

# Assume your data is in a pandas DataFrame named 'df' with columns 'user', 'item', and 'rating'

# Example:

data = {

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

'item': ['A', 'B', 'C', 'B', 'D', 'A', 'C', 'A', 'D', 'B'],

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

}

df = pd.DataFrame(data)

# Define the rating scale

reader = Reader(rating\_scale=(1, 5)) # Adjust the scale if needed

# Load the data into Surprise Dataset format

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

# Create a trainset

trainset = surprise\_data.build\_full\_trainset()

# Define similarity options for item-based collaborative filtering

sim\_options = {

'name': 'cosine', # Use cosine similarity

'user\_based': False # Item-based CF (set to True for user-based CF)

}

# Initialize the KNNBasic algorithm

model = KNNBasic(sim\_options=sim\_options)

# Train the model

model.fit(trainset)

# Get a list of all unique items

all\_items = df['item'].unique()

# Predict ratings for a specific user (e.g., user\_id = 1) for items they haven't rated

user\_id = 1

recommendations = []

for item in all\_items:

if not any((df['user'] == user\_id) & (df['item'] == item)):

pred = model.predict(user\_id, item)

recommendations.append((item, pred.est))

# Sort recommendations by predicted rating

recommendations = sorted(recommendations, key=lambda x: x[1], reverse=True)

# Print top recommendations

print(f"Top 3 recommendations for User {user\_id}:")

for item, rating in recommendations[:3]:

print(f"Item: {item}, Predicted Rating: {rating:.2f}")

# Create a testset (e.g., using trainset.build\_testset())

testset = trainset.build\_testset()

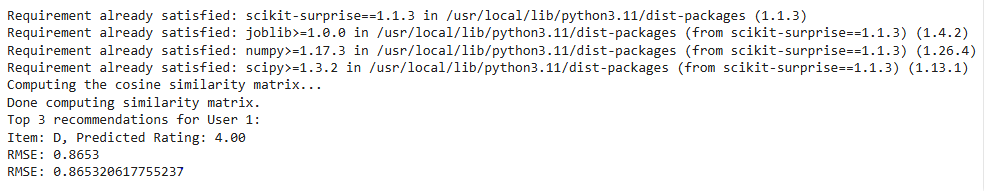
# Make predictions on the testset

predictions = model.test(testset)

# Calculate RMSE to evaluate accuracy

print(f"RMSE: {accuracy.rmse(predictions)}")

**Output:**



**Practical 05**

**Implementing computer vision project such as object detection or image segmentation.**

import cv2

from google.colab import files

import numpy as np

import matplotlib.pyplot as plt

uploaded = files.upload()

image\_path = next(iter(uploaded))

img = cv2.imread(image\_path)

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

face\_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade\_frontalface\_default.xml')

faces = face\_cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5, minSize=(30, 30))

for (x, y, w, h) in faces:

 cv2.rectangle(img, (x, y), (x + w, y + h), (255, 0, 0), 2)

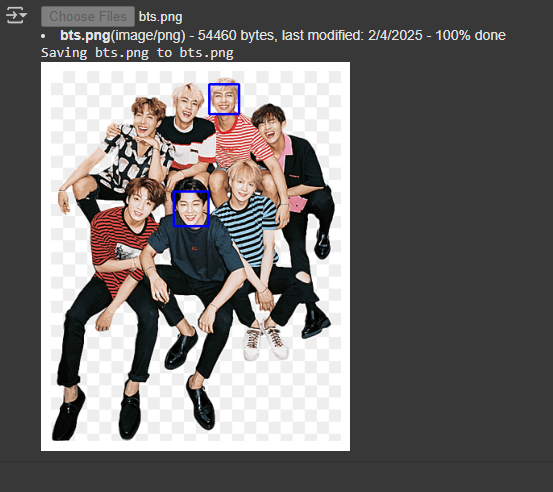
img\_rgb = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)

plt.imshow(img\_rgb)

plt.axis('off')

plt.show()

**Output:**

****

**Practical 06**

**Training a generative adversarial network (GAN) for generating realistic images.**

import tensorflow as tf

from tensorflow.keras import layers

import numpy as np

import matplotlib.pyplot as plt

(train\_images, \_), (\_, \_) = tf.keras.datasets.cifar10.load\_data()

train\_images = train\_images.astype('float32')

train\_images = (train\_images - 127.5) / 127.5

train\_images = train\_images.reshape(train\_images.shape[0], 32, 32, 3)

def build\_generator():

 model = tf.keras.Sequential()

 model.add(layers.Dense(256, input\_dim=100)) # Input: random noise vector of size 100

 model.add(layers.LeakyReLU(alpha=0.2))

 model.add(layers.BatchNormalization(momentum=0.8))

 model.add(layers.Dense(512))

 model.add(layers.LeakyReLU(alpha=0.2))

 model.add(layers.BatchNormalization(momentum=0.8))

 model.add(layers.Dense(1024))

 model.add(layers.LeakyReLU(alpha=0.2))

 model.add(layers.BatchNormalization(momentum=0.8))

 model.add(layers.Dense(np.prod((32, 32, 3)), activation='tanh')) # Output image

 model.add(layers.Reshape((32, 32, 3)))

 return model

def build\_discriminator():

 model = tf.keras.Sequential()

 model.add(layers.Flatten(input\_shape=(32, 32, 3)))

 model.add(layers.Dense(1024))

 model.add(layers.LeakyReLU(alpha=0.2))

 model.add(layers.Dense(512))

 model.add(layers.LeakyReLU(alpha=0.2))

 model.add(layers.Dense(256))

 model.add(layers.LeakyReLU(alpha=0.2))

 model.add(layers.Dense(1, activation='sigmoid')) # Output: real/fake

 return model

generator = build\_generator()

discriminator = build\_discriminator()

discriminator.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

discriminator.trainable = False

gan\_input = layers.Input(shape=(100,))

x = generator(gan\_input)

gan\_output = discriminator(x)

gan = tf.keras.Model(gan\_input, gan\_output)

gan.compile(optimizer='adam', loss='binary\_crossentropy')

epochs = 10000

batch\_size = 64

half\_batch = batch\_size // 2

for epoch in range(epochs):

 idx = np.random.randint(0, train\_images.shape[0], half\_batch)

 real\_images = train\_images[idx]

 noise = np.random.normal(0, 1, (half\_batch, 100))

 fake\_images = generator.predict(noise)

 d\_loss\_real = discriminator.train\_on\_batch(real\_images, np.ones((half\_batch, 1)))

 d\_loss\_fake = discriminator.train\_on\_batch(fake\_images, np.zeros((half\_batch, 1)))

 d\_loss = 0.5 \* np.add(d\_loss\_real, d\_loss\_fake)

 noise = np.random.normal(0, 1, (batch\_size, 100))

 g\_loss = gan.train\_on\_batch(noise, np.ones((batch\_size, 1)))

 if epoch % 1000 == 0:

  print(f"{epoch} [D loss: {d\_loss[0]} | D accuracy: {100\*d\_loss[1]}] [G loss: {g\_loss}]")

 if epoch % 1000 == 0:

  noise = np.random.normal(0, 1, (25, 100))

 generated\_images = generator.predict(noise)

 generated\_images = 0.5 \* generated\_images + 0.5 # Rescale to [0,1]

 fig, axs = plt.subplots(5, 5)

 count = 0

 for i in range(5):

  for j in range(5):

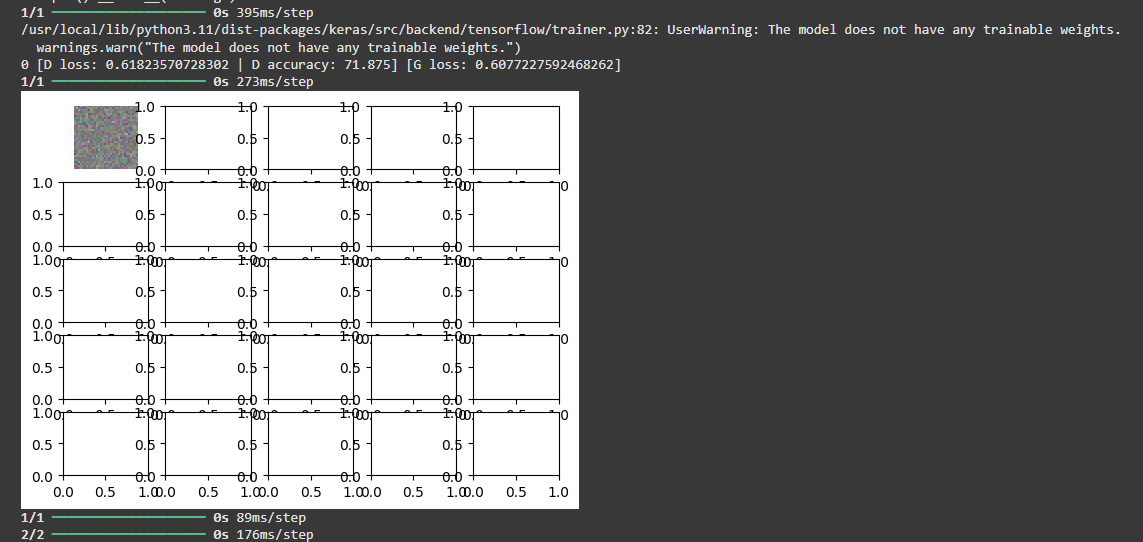
    axs[i, j].imshow(generated\_images[count])

    axs[i, j].axis('off')

    count += 1

    plt.show()

**Output:**



**Practical 07**

**Applying reinforcement learning algorithms to solve the complex decision making problems.**

import gym

import numpy as np

import warnings

warnings.filterwarnings("ignore", category=DeprecationWarning)

env = gym.make('CartPole-v1', render\_mode="human")

state = env.reset()

print("State space:", env.observation\_space)

print("Action space:", env.action\_space)

for \_ in range(10):

    env.render

    action = env.action\_space.sample()

    step\_result = env.step(action)

    if len(step\_result) == 4:

        next\_state, reward, done, info = step\_result

        terminated = False

    else:

        next\_state, reward, done, truncated, info = step\_result

        terminated = done or truncated

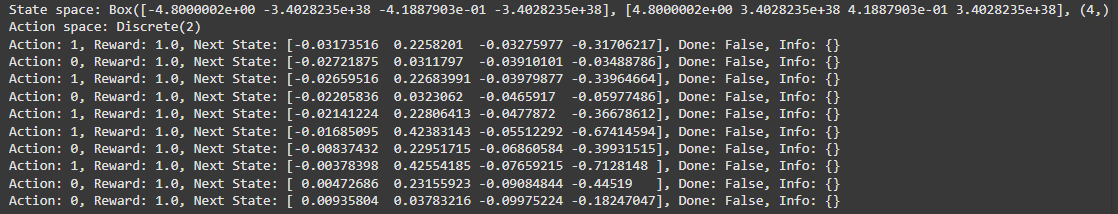
    print(f"Action: {action}, Reward: {reward}, Next State: {next\_state}, Done: {done}, Info: {info}")

    if terminated:

        state = env.reset()

env.close()

**Output:**

****

**Practical 08**

Utilising transfer learning to improve model performance on limited datasets.

import tensorflow as tf

from tensorflow.keras.applications import MobileNetV2

from tensorflow.keras.layers import Input, Lambda, Dense, GlobalAveragePooling2D

from tensorflow.keras.models import Model

from tensorflow.keras.datasets import mnist

from tensorflow.keras.utils import to\_categorical

import numpy as np

(train\_images, train\_labels), (test\_images, test\_labels) = mnist.load\_data()

train\_images = np.stack([train\_images]\*3, axis=-1) / 255.0

test\_images = np.stack([test\_images]\*3, axis=-1) / 255.0

train\_images = tf.image.resize(train\_images, [32, 32])

test\_images = tf.image.resize(test\_images, [32, 32])

train\_labels = to\_categorical(train\_labels, 10)

test\_labels = to\_categorical(test\_labels, 10)

base\_model = MobileNetV2(weights='imagenet', include\_top=False, input\_shape=(32, 32, 3))

base\_model.trainable = False

inputs = Input(shape=(32, 32, 3))

x = base\_model(inputs, training=False)

x = GlobalAveragePooling2D()(x)

outputs = Dense(10, activation='softmax')(x)

model = Model(inputs, outputs)

model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

model.fit(train\_images, train\_labels, epochs=10, validation\_split=0.2)

base\_model.trainable = True

for layer in base\_model.layers[:100]:

layer.trainable = False

model.compile(optimizer=tf.keras.optimizers.Adam(1e-5), # Lower lr for fine-tuning

loss='categorical\_crossentropy',

metrics=['accuracy'])

model.fit(train\_images, train\_labels, epochs=5, validation\_split=0.2)

loss, accuracy = model.evaluate(test\_images, test\_labels)

print(f"Test loss: {loss}")

print(f"Test accuracy: {accuracy}")

import matplotlib.pyplot as plt

import numpy as np

from sklearn.metrics import confusion\_matrix

import seaborn as sns

test\_predictions = model.predict(test\_images)

test\_predictions\_classes = np.argmax(test\_predictions, axis=1)

test\_true\_classes = np.argmax(test\_labels, axis=1)

cm = confusion\_matrix(test\_true\_classes, test\_predictions\_classes)

plt.figure(figsize=(10, 8))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False)

plt.xlabel('Predicted Label')

plt.ylabel('True Label')

plt.title('Confusion Matrix')

plt.show()

def display\_sample(sample\_images, sample\_labels, sample\_predictions):

fig, axes = plt.subplots(3, 3, figsize=(12, 12))

fig.subplots\_adjust(hspace=0.5, wspace=0.5)

for i, ax in enumerate(axes.flat):

ax.imshow(sample\_images[i].reshape(32, 32), cmap='gray') # Change to 32x32

ax.set\_xlabel(f"True: {sample\_labels[i]}\nPredicted: {sample\_predictions[i]}")

ax.set\_xticks([])

ax.set\_yticks([])

plt.show()

test\_images\_gray = np.dot(test\_images[...,:3], [0.2989, 0.5870, 0.1140])

random\_indices = np.random.choice(len(test\_images\_gray), 9, replace=False)

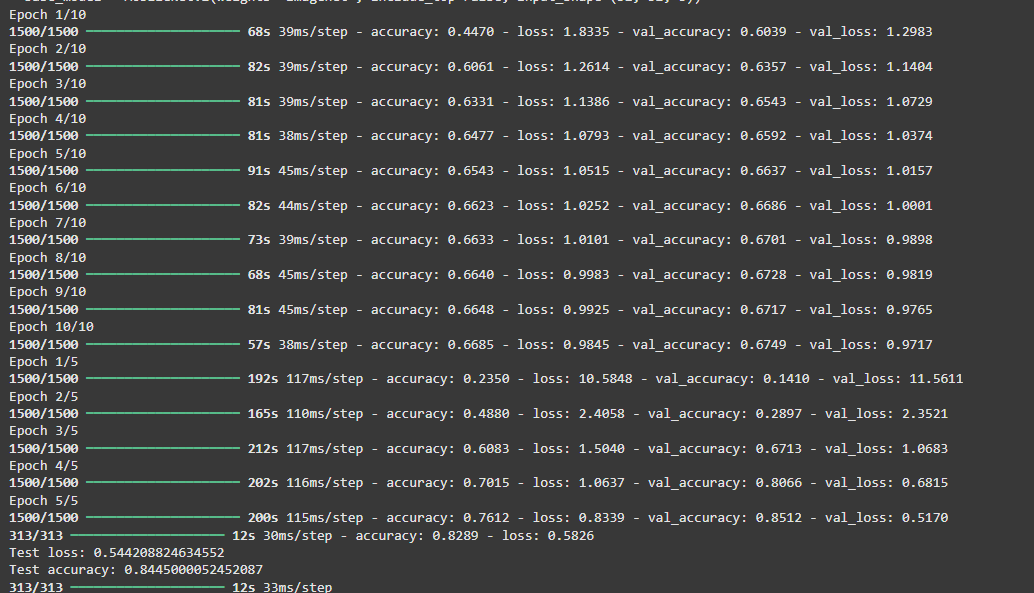
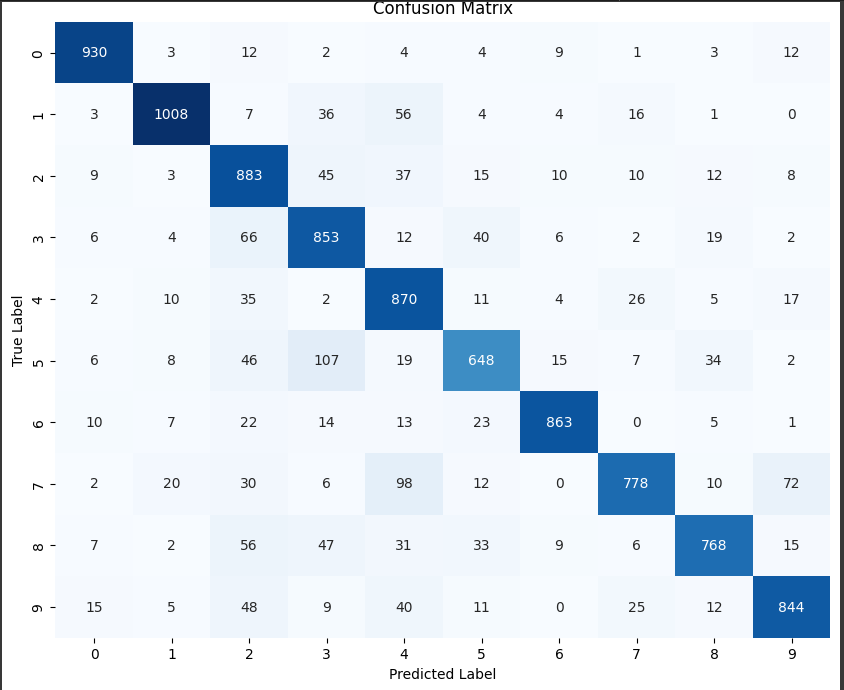
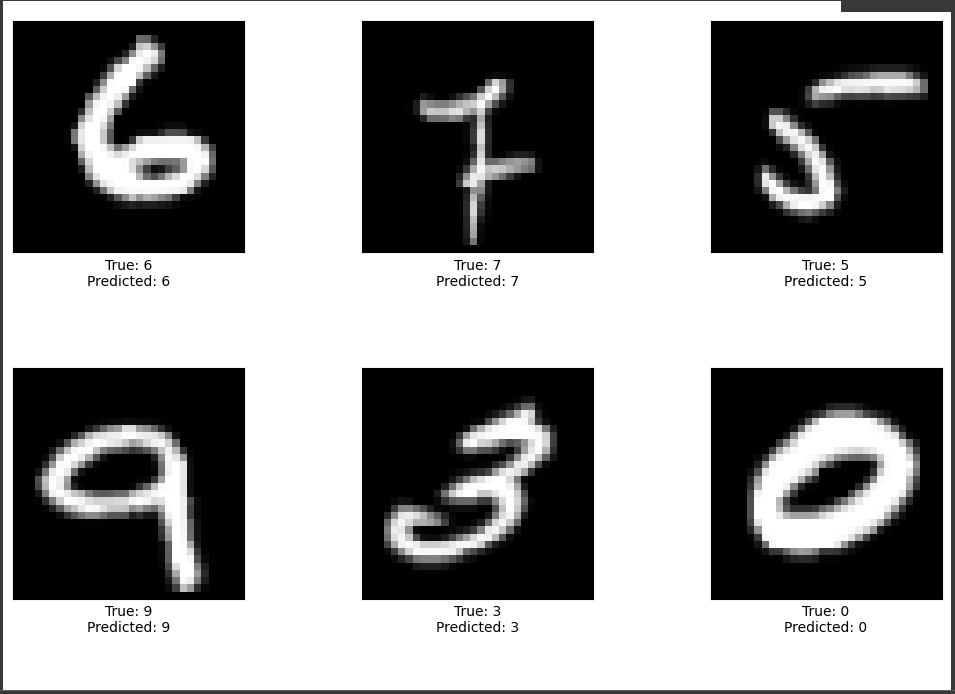
sample\_images = test\_images\_gray[random\_indices]

sample\_labels = test\_true\_classes[random\_indices]

sample\_predictions = test\_predictions\_classes[random\_indices]

display\_sample(sample\_images, sample\_labels, sample\_predictions)

Output:



**Practical 09**

**Building a deep learning model for time series forecasting or anomaly detection.**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.preprocessing import MinMaxScaler

url ="https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passengers.csv"

data = pd.read\_csv(url, usecols=[1], engine='python', header=0)

plt.plot(data)

plt.title('Monthly Airline Passengers')

plt.xlabel('Month')

plt.ylabel('Passengers')

plt.show()

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

data\_scaled = scaler.fit\_transform(data)

def create\_dataset(data, time\_step=1):

  X, y = [], []

  for i in range(len(data)-time\_step-1):

    X.append(data[i:(i+time\_step), 0])

    y.append(data[i + time\_step, 0])

    return np.array(X), np.array(y)

    time\_step = 12

    X, y = create\_dataset(data\_scaled, time\_step)

    X = X.reshape(X.shape[0], X.shape[1], 1)

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense, Dropout

model = Sequential()

model.add(LSTM(units=50, return\_sequences=False, input\_shape=(X.shape[1], 1)))

model.add(Dropout(0.2))

model.add(Dense(units=1))

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

model.fit(X, y, epochs=20, batch\_size=32)

test\_input = data\_scaled[-time\_step:]

test\_input = test\_input.reshape(1, -1, 1)

predicted = model.predict(test\_input)

predicted = scaler.inverse\_transform(predicted)

print(f'Predicted passengers for the next month: {predicted[0][0]}')

train\_data = data[:len(data) - 12]

plt.plot(train\_data, label='Training Data')

plt.plot(range(len(train\_data), len(train\_data) + 12), predicted,

label='Forecast', color='red')

plt.legend()

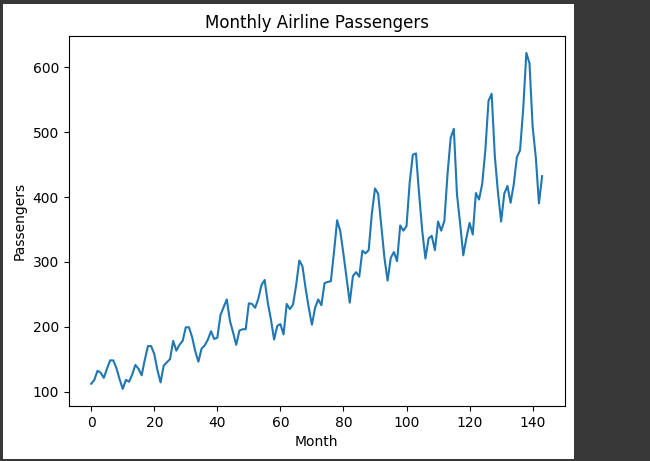
plt.title('Time Series Forecasting')

plt.xlabel('Month')

plt.ylabel('Passengers')

plt.show()

Output:

****

**Practical 10**

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

pip install scikit-learn tpot optuna pandas numpy matplotlib

import pandas as pd

from sklearn import datasets

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

from sklearn.preprocessing import StandardScaler

iris = datasets.load\_iris()

X = iris.data

y = iris.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

from tpot import TPOTClassifier

tpot = TPOTClassifier( generations=5, population\_size=20, random\_state=42, cv=5, verbosity=2)

tpot.fit(X\_train\_scaled, y\_train)

accuracy = tpot.score(X\_test\_scaled, y\_test)

print(f"Test accuracy: {accuracy:.4f}")

tpot.export('best\_model\_pipeline.py')

import optuna

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy\_score

def objective(trial):

n\_estimators = trial.suggest\_int('n\_estimators', 50, 200)

max\_depth = trial.suggest\_int('max\_depth', 5, 20)

min\_samples\_split = trial.suggest\_int('min\_samples\_split', 2, 10)

model = RandomForestClassifier(n\_estimators=n\_estimators, max\_depth=max\_depth, min\_samples\_split=min\_samples\_split, random\_state=42)

model.fit(X\_train\_scaled, y\_train)

y\_pred = model.predict(X\_test\_scaled)

accuracy = accuracy\_score(y\_test, y\_pred)

return accuracy

study = optuna.create\_study(direction="maximize")

study.optimize(objective, n\_trials=20)

print("Best hyperparameters:", study.best\_params)

best\_params = study.best\_params

best\_model = RandomForestClassifier(\*\*best\_params, random\_state=42)

best\_model.fit(X\_train\_scaled, y\_train)

best\_accuracy = best\_model.score(X\_test\_scaled, y\_test)

print(f"Best Model Test Accuracy: {best\_accuracy:.4f}")

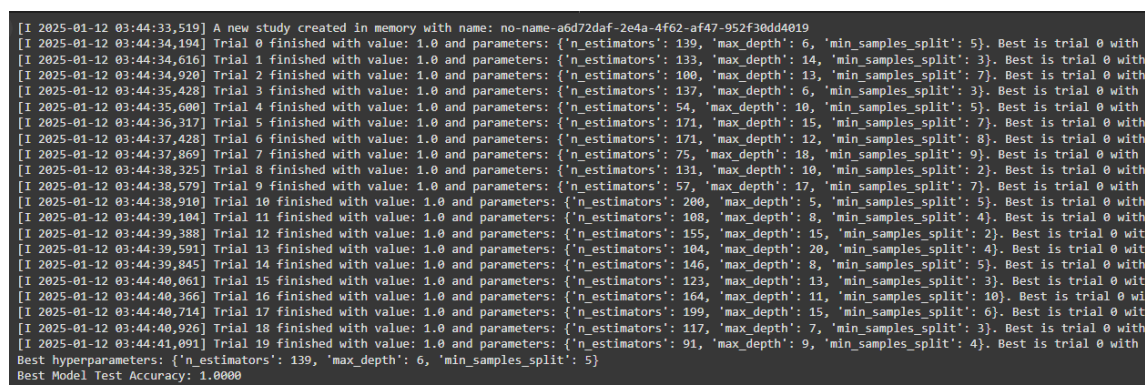
y\_pred = best\_model.predict(X\_test\_scaled)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Test Accuracy of Final Model: {accuracy:.4f}")

import joblib

joblib.dump(best\_model, 'best\_model.pkl')

**Output:**