**Practical - 01**

**Aim: Implementing advanced deep learning algorithms such as convolutional neuralnetworks (CNNs) or recurrent neural networks (RNNs) using Python librarieslike TensorFlow or PyTorch.**

from keras.datasets import mnist

from keras.layers import Dense, Conv2D,Flatten

from keras.models import Sequential

import matplotlib.pyplot as plt

from google.colab import drive

drive.mount('/content/drive')

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

train\_images.shape

test\_images.shape

plt.imshow(train\_images[1])

plt.imshow(test\_images[1])

test\_labels[1]

train\_image,test\_images=train\_images/255.0,test\_images/255.0

import numpy as np

train\_images=train\_images.reshape(60000,28,28,1)

test\_images=test\_images.reshape(10000,28,28,1)

from keras.utils import to\_categorical

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

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

test\_labels[1]

train\_labels[1]

cnnmodel=Sequential()

cnnmodel.add(Conv2D(32,(3,3),activation='relu',input\_shape=(28,28,1)))

cnnmodel.add(Conv2D(64,(3,3,),activation='relu'))

cnnmodel.add(Flatten())

cnnmodel.add(Dense(10,activation='softmax'))

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

cnnmodel.fit(train\_images,train\_labels,epochs=3)

predictions=cnnmodel.predict(test\_images)

predictions[1]

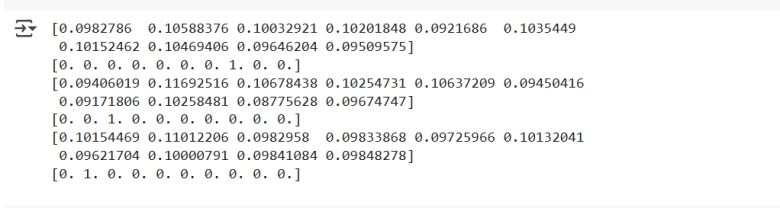
#import numpy as np

for i in range(3):

print(predictions[i])

print(test\_labels[i])

**OUTPUT**



**Practical - 02**

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

from sklearn.datasets import fetch\_20newsgroups

from sklearn.feature\_extraction.text import CountVectorizer

from sklearn.naive\_bayes import BernoulliNB,MultinomialNB

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

from sklearn.metrics import accuracy\_score

newsgroup=fetch\_20newsgroups(subset='all')

newsgroup.target\_names

vectorizer1=CountVectorizer(binary=True)

vectorizer2=CountVectorizer(binary=False)

x1=vectorizer1.fit\_transform(newsgroup.data)

x2=vectorizer2.fit\_transform(newsgroup.data)

y=newsgroup.target

print(y)

xtrain1,xtest1,ytrain,ytest=train\_test\_split(x1,y,test\_size=0.25,random\_state=42)

xtrain2,xtest2,ytrain,ytest=train\_test\_split(x2,y,test\_size=0.25,random\_state=42)

bnb=BernoulliNB()

mnb=MultinomialNB()

bnb.fit(xtrain1,ytrain)

mnb.fit(xtrain2,ytrain)

y\_pred1=bnb.predict(xtest1)

y\_pred2=mnb.predict(xtest2)

accuracy\_score(ytest,y\_pred1)

accuracy\_score(ytest,y\_pred2)

OUTPUT



**Practical - 03**

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

import pandas as pd

ratings = pd.read\_csv('BX-Book-Ratings.csv',encoding="latin-1",delimiter=";")

ratings

books=pd.read\_csv('BX-Books (2).csv', encoding="latin-1")

books

ratings\_books=pd.merge(ratings,books,on="ISBN")

ratings\_books

ratings\_books\_sample=ratings\_books.sample(frac=0.5,random\_state=1)

ratings\_books\_sample

ratings\_books\_sample.to\_csv('ravi.csv')

ratings\_books\_sample.columns

ratings\_books\_pivot=ratings\_books\_sample.pivot\_table(index='Book-Title',columns='User-ID',values='Book-Rating').fillna(0)

ratings\_books\_pivot

from sklearn.neighbors import NearestNeighbors

model=NearestNeighbors(metric='cosine',algorithm='brute',n\_neighbors=7, n\_jobs=-1)

model.fit(ratings\_books\_pivot)

ratings\_books\_pivot.loc[["Cracking the Da Vinci Code : The Unauthorized Guide to the Facts Behind Dan Brown's Bestselling Novel"]]

#indices=model.kneighbors(ratings\_books\_pivot.loc[['Flesh Tones: A Novel']])

#indices

indices=model.kneighbors(ratings\_books\_pivot.loc[["Crossing Over"]])

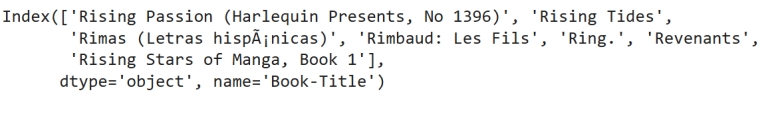
indices

neighbor\_indices = indices[1][0]

recommendations = ratings\_books\_pivot.index[neighbor\_indices]

print(recommendations)

**OUTPUT**



**Practical - 04**

**AIM: Applying reinforcement learning algorithms to solve complex decision-making problems**

Code:

import numpy as np

import random

# Define the environment

grid\_size = 3 # Smaller grid

goal\_state = (2, 2)

obstacle\_state = (1, 1) # Single obstacle

actions = ['up', 'down', 'left', 'right']

action\_to\_delta = {

'up': (-1, 0),

'down': (1, 0),

'left': (0, -1),

'right': (0, 1)

}

# Initialize Q-table (simple 3D array for states and actions)

q\_table = np.zeros((grid\_size, grid\_size, len(actions)))

# Parameters

alpha = 0.1 # Learning rate

gamma = 0.9 # Discount factor

epsilon = 1.0 # Exploration rate

epsilon\_decay = 0.99

min\_epsilon = 0.1

episodes = 200 # Fewer episodes

# Reward function

def get\_reward(state):

if state == goal\_state:

return 10 # Reward for reaching the goal

elif state == obstacle\_state:

return -10 # Penalty for hitting the obstacle

return -1 # Step penalty

# Check if the new state is valid

def is\_valid\_state(state):

return 0 <= state[0] < grid\_size and 0 <= state[1] < grid\_size and state != obstacle\_state

# Main Q-learning loop

for episode in range(episodes):

state = (0, 0) # Start at the top-left corner

total\_reward = 0

while state != goal\_state:

# Choose an action (epsilon-greedy

strategy) if random.uniform(0, 1) < epsilon:

action = random.choice(actions) # Explore

else:

action = actions[np.argmax(q\_table[state[0], state[1]])] # Exploit best action

# Perform the action

delta = action\_to\_delta[action]

next\_state = (state[0] + delta[0], state[1] + delta[1])

# Stay in the same state if the move is invalid

if not is\_valid\_state(next\_state):

next\_state = state

# Get reward and update Q-table

reward = get\_reward(next\_state)

total\_reward += reward

best\_next\_action = np.max(q\_table[next\_state[0], next\_state[1]])

q\_table[state[0], state[1], actions.index(action)] += alpha \* (

reward + gamma \* best\_next\_action - q\_table[state[0], state[1], actions.index(action)]

)

# Update state

state = next\_state

# Decay epsilon

epsilon = max(min\_epsilon, epsilon \* epsilon\_decay)

print(f"Episode {episode + 1}: Total Reward = {total\_reward}")

# Display the learned policy

policy = np.full((grid\_size, grid\_size), ' ')

for i in range(grid\_size):

for j in range(grid\_size):

if (i, j) == goal\_state:

policy[i, j] = 'G' # Goal

elif (i, j) == obstacle\_state:

policy[i, j] = 'X' # Obstacle

else:

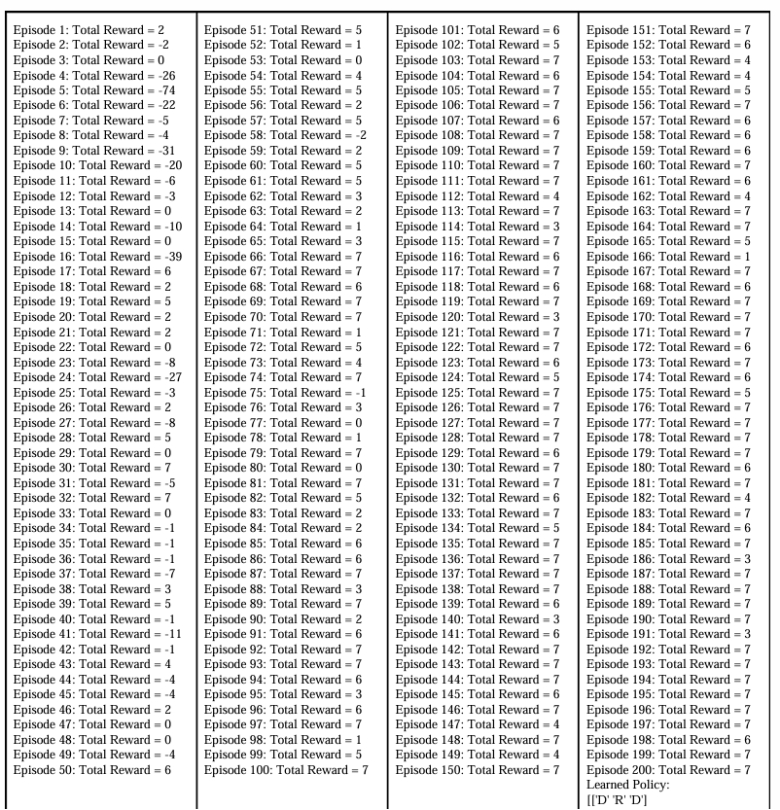
best\_action = np.argmax(q\_table[i, j])

policy[i, j] = actions[best\_action][0].upper() # First letter of the best action

print("Learned Policy:")

print(policy)

**OUTPUT**



**Practical - 05**

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

!pip install pycaret &>/dev/null

print("pycaret install sucessful")

from pycaret.utils import version

version()

from pycaret.datasets import get\_data

from pycaret.classification import setup,compare\_models,finalize\_model,predict\_model

data=get\_data("diamond")

data.shape

data.info()

dataset=data.sample(frac=0.9,random\_state=786)

dataset

data\_unseen=data.drop(dataset.index)

dataset.shape,data\_unseen.shape

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

dataset

data\_unseen.reset\_index(drop=True,inplace=True)

data\_unseen

from pycaret.regression import\*

s=setup(data=dataset,target='Price',session\_id=120)

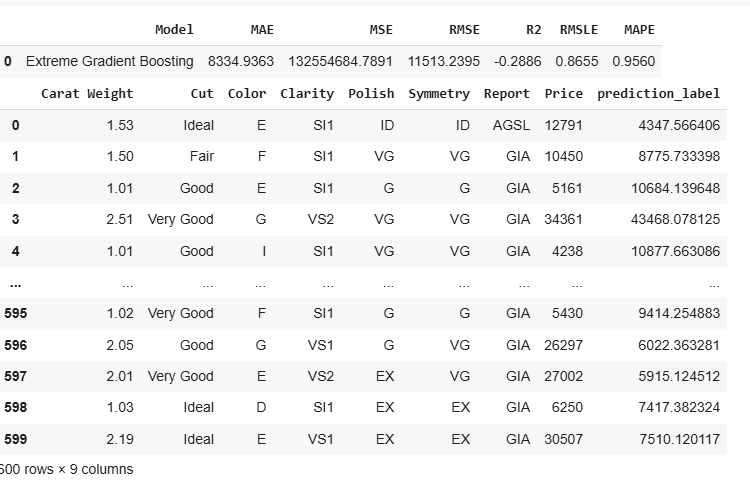
best=compare\_models()

final\_model=create\_model('xgboost')

print(final\_model)

predict\_model(final\_model,data=data\_unseen)

**OUTPUT**



**Practical - 06**

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

import numpy as np

import matplotlib.pyplot as plt

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input,Dense

normal\_data = np.random.normal(0,1,(1000,2))

anomalous\_data = np.random.uniform(5,10,(100,2))

normal\_data[:3]

anomalous\_data[:3]

len(normal\_data),len(anomalous\_data)

data=np.vstack([normal\_data,anomalous\_data])

labels=np.array([0] \* len(normal\_data) + [1] \* len(anomalous\_data))

data.shape

labels

labels.shape

def build\_autoencoder(input\_dim):

input\_layer=Input(shape=(input\_dim,))

encoded=Dense(4,activation='relu')(input\_layer)

encoded=Dense(2,activation='relu')(encoded)

decoded=Dense(4,activation='relu')(encoded)

output\_layer=Dense(input\_dim,activation='linear')(decoded)

autoencoder=Model(inputs=input\_layer,outputs=output\_layer)

autoencoder.compile(optimizer='adam',loss='mse')

return autoencoder

autoencoder = build\_autoencoder(input\_dim=2)

autoencoder.fit(normal\_data,normal\_data,epochs=20)

reconstructed=autoencoder.predict(data)

reconstructed\_error=np.mean((data-reconstructed)\*\*2,axis=1)

reconstructed\_error

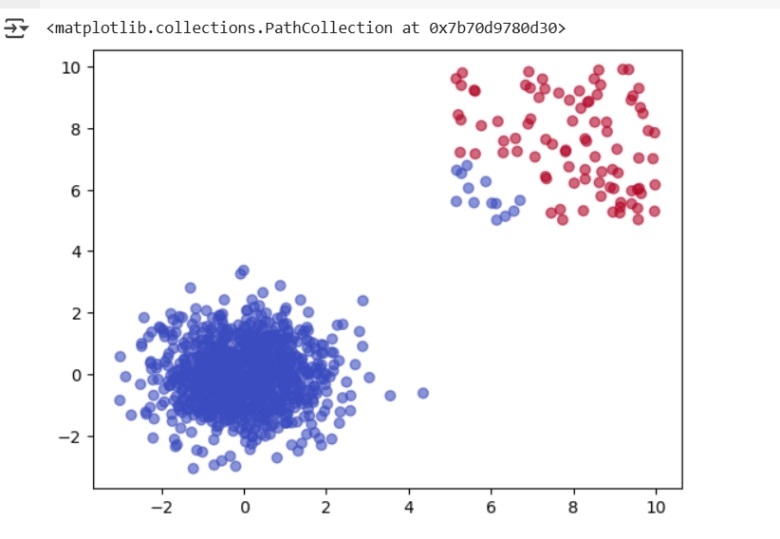
predicted\_labels=(reconstructed\_error>40)

plt.scatter(data[:,0],data[:,1],

c=predicted\_labels,cmap='coolwarm',alpha=0.6)

#plt.colorbar(label='Anomaly')

**OUTPUT**



**Practical - 07**

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

! pip install pygad

import numpy as np

from sklearn.datasets import load\_iris

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

from sklearn.svm import SVC

from sklearn.metrics import accuracy\_score

import pygad

data = load\_iris()

X,y = data.data,data.target

xtrain,xtest,ytrain,ytest = train\_test\_split(X,y,test\_size=0.2,random\_state=42)

def fitness\_func(ga\_instance,solution,solution\_idx):

C=solution[0]

gamma=solution[1]

kernel = ['linear', 'poly','rbf','sigmoid'][int(solution[2])]

if kernel not in ['linear','poly','rbf','sigmoid']:

return 0

model= SVC(C=C,gamma=gamma,kernel=kernel, random\_state=42)

model.fit(xtrain,ytrain)

predictions=model.predict(xtest)

accuracy = accuracy\_score(ytest,predictions)

return accuracy

ga\_instance = pygad.GA(num\_generations=50,num\_parents\_mating=5,fitness\_func=fitness\_func,sol\_per\_pop=10,num\_genes=3,gene\_space=[

{ 'low': 0.1, 'high': 10.0},

{ 'low': 0.0001, 'high': 1.0},

{ 'low': 0, 'high': 3, 'step': 1},

parent\_selection\_type="rank", keep\_parents=2, crossover\_type="single\_point",

mutation\_type="random",

mutation\_percent\_genes=10 )

ga\_instance.run()

solution,solution\_fitness,solution\_idx = ga\_instance.best\_solution()

print (solution,solution\_fitness,solution\_idx)

C\_best,gamma\_best,kernel\_best = solution

kernel\_best = ['linear', 'poly','rbf','sigmoid'][int(kernel\_best)]

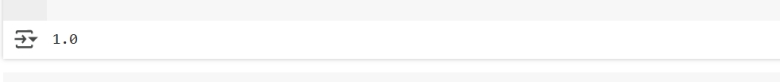
kernel\_best

best\_model = SVC(C=C\_best,gamma=gamma\_best,kernel=kernel\_best, random\_state=42)

best\_model.fit(xtrain,ytrain)

final\_accuracy = accuracy\_score(ytest,best\_model.predict(xtest))

final\_accuracy

**OUTPUT**

**Practical – 08**

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

import torch

import torch.nn as nn

import torch.optim as optim

import torchvision

import torchvision.transforms as transforms

import matplotlib.pyplot as plt

from torch.utils.data import DataLoader

# Generator Model

class Generator(nn.Module):

def \_\_init\_\_(self, z\_dim=100):

super().\_\_init\_\_()

self.model = nn.Sequential(

nn.Linear(z\_dim, 256), nn.LeakyReLU(0.2),

nn.Linear(256, 512), nn.LeakyReLU(0.2),

nn.Linear(512, 1024), nn.LeakyReLU(0.2),

nn.Linear(1024, 28\*28), nn.Tanh()

def forward(self, z):

return self.model(z).view(z.size(0), 1, 28, 28)

# Discriminator Model

class Discriminator(nn.Module):

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

super().\_\_init\_\_()

self.model = nn.Sequential(

nn.Flatten(),

nn.Linear(28\*28, 1024), nn.LeakyReLU(0.2),

nn.Linear(1024, 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)

# Data loading

transform = transforms.Compose([

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

])

train\_loader = DataLoader(

torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform),

batch\_size=64, shuffle=True

)

# Model, optimizer and loss function

z\_dim = 100

generator, discriminator = Generator(z\_dim), Discriminator()

criterion = nn.BCELoss()

optimizer\_gen = optim.Adam(generator.parameters(), lr=0.0002, betas=(0.5, 0.999))

optimizer\_disc = optim.Adam(discriminator.parameters(), lr=0.0002, betas=(0.5, 0.999))

# Training loop

for epoch in range(20):

for real\_images, \_ in train\_loader:

batch\_size = real\_images.size(0)

real\_images = real\_images.view(batch\_size, -1)

# Labels

real\_labels, fake\_labels = torch.ones(batch\_size, 1), torch.zeros(batch\_size, 1)

# Train Discriminator

optimizer\_disc.zero\_grad()

d\_loss\_real = criterion(discriminator(real\_images), real\_labels)

d\_loss\_fake = criterion(discriminator(generator(torch.randn(batch\_size, z\_dim)).view(batch\_size, -1)), fake\_labels)

d\_loss = d\_loss\_real + d\_loss\_fake

d\_loss.backward()

optimizer\_disc.step()

# Train Generator

optimizer\_gen.zero\_grad()

g\_loss = criterion(discriminator(generator(torch.randn(batch\_size, z\_dim)).view(batch\_size, -1)), real\_labels)

g\_loss.backward()

optimizer\_gen.step()

# Print loss and save images

if (epoch+1) % 5 == 0:

print(f"Epoch [{epoch+1}/20], D Loss: {d\_loss.item():.4f}, G Loss: {g\_loss.item():.4f}")

with torch.no\_grad():

fake\_images = generator(torch.randn(64, z\_dim)).view(64, 1, 28, 28)

fake\_images = (fake\_images + 1) / 2 # Rescale to [0, 1]

grid\_img = torchvision.utils.make\_grid(fake\_images)

plt.imshow(grid\_img.permute(1, 2, 0).cpu().numpy())

plt.show()

**Output**

