INDEX

PRACTICAL 1

[A] AIM: Perform basic mathematics operations in python.

[I] Addition of matrices.

CODE:

```
import numpy as np
A = np.array([[1, 2], [3, 4], [5, 6]])
A
B = np.array([[2, 5], [7, 4], [4, 3]])
B
# Add matrices A and B
C = A + B
print(C)
```

OUTPUT:

```
[[ 3  4]
 [ 7  8]
 [11 12]]


Process finished with exit code 0
```

[II] Multiplication of matrices.

CODE:

```
import numpy as np
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
print(A)
B = np.array([[2, 7], [1, 2], [3, 6]])
print(B)
C = A.dot(B)
print(C)
```

OUTPUT:

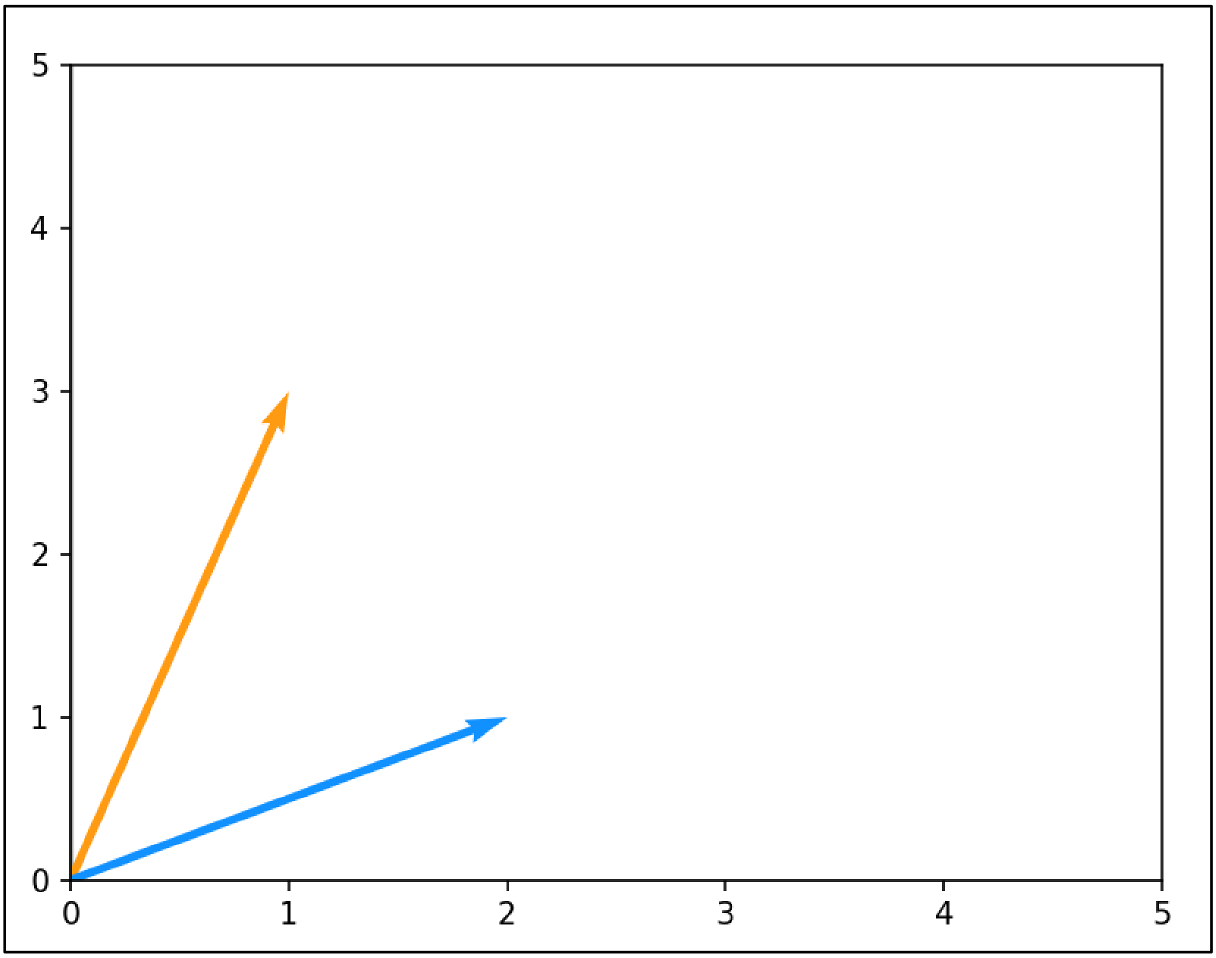
```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
[[2 7]
 [1 2]
 [3 6]]
[[ 13  29]
 [ 31  74]
 [ 49 119]
 [ 67 164]]
```

[III] Linear Combination.

CODE:

```
import numpy as np

import matplotlib.pyplot as plt


def plotVectors(vecs, cols, alpha =1 ):

    plt.figure()

    plt.axvline(x=0, color = '#A9A9A9', zorder = 0)

    for i in range(len(vecs)):

        x = np.concatenate([[0,0],vecs[i]])

        plt.quiver([x[0]],

               [x[1]],

               [x[2]],

               [x[3]],

               angles='xy', scale_units='xy', scale=1,color=cols[i],
alpha=alpha)

orange= '#FF9A13'

blue= '#1190FF'

plotVectors([[1,3],[2,1]],[orange,blue])

plt.xlim(0,5)

plt.ylim(0,5)

plt.show()
```

OUTPUT:

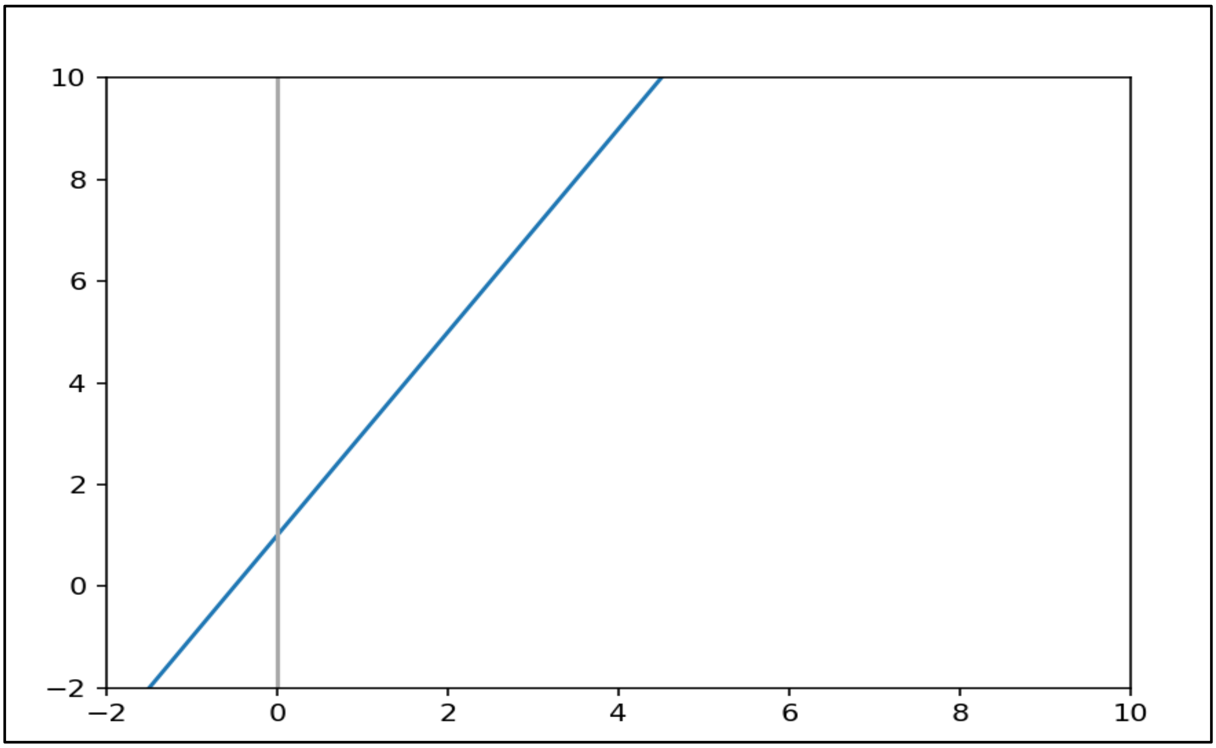


[IVa] Linear Equation.

CODE:

```
import numpy as np

import matplotlib.pyplot as plt


x = np.arange(-10,10)

y = 2*x + 1


plt.figure()

plt.plot(x,y)

plt.xlim(-2,10)

plt.ylim(-2,10)


plt.axvline(x=0, color = '#A9A9A9')

plt.show()
```

OUTPUT:

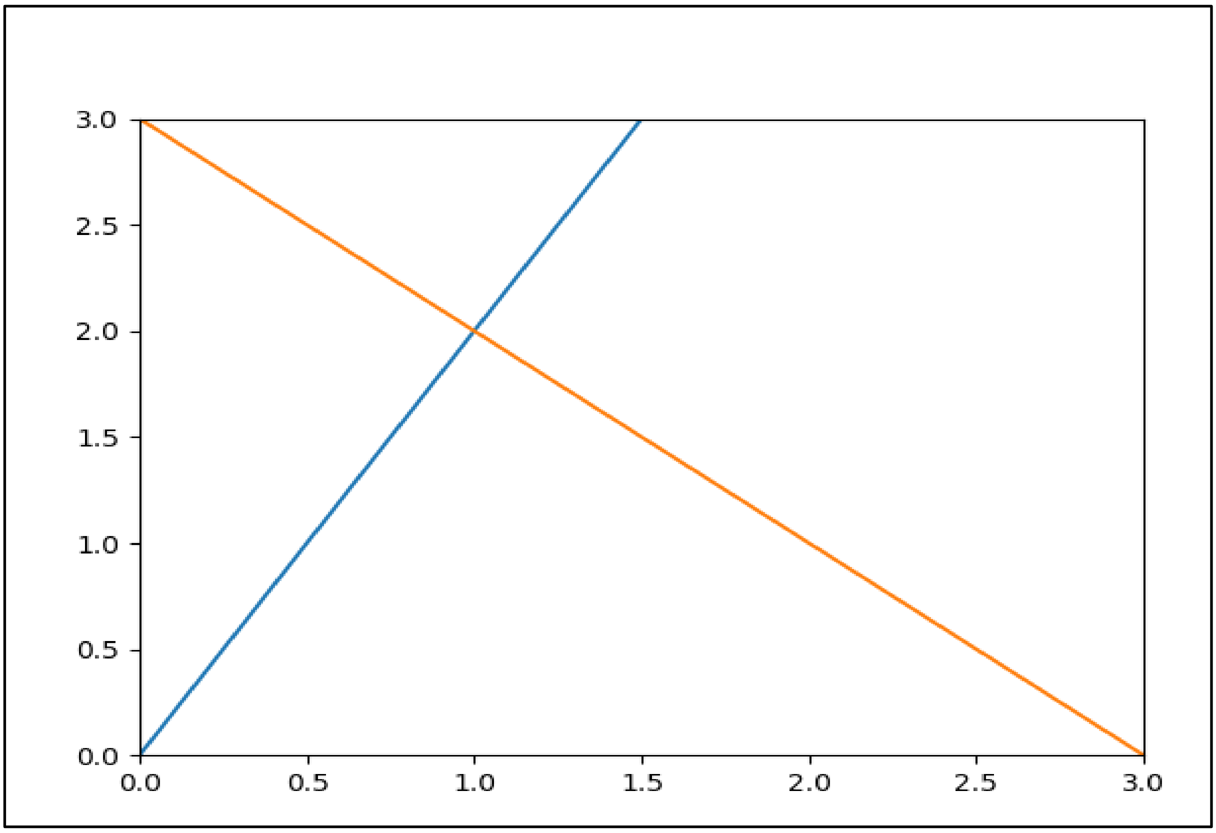


[IVb] Linear Equation.

CODE:

```
import numpy as np

import matplotlib.pyplot as plt


x = np.arange(-10,10)

y = 2*x

y1 = -x + 3

plt.figure()

plt.plot(x,y)

plt.plot(x,y1)

plt.xlim(0,3)

plt.ylim(0,3)

plt.axvline(x=0, color='grey')

plt.show()
```

OUTPUT:

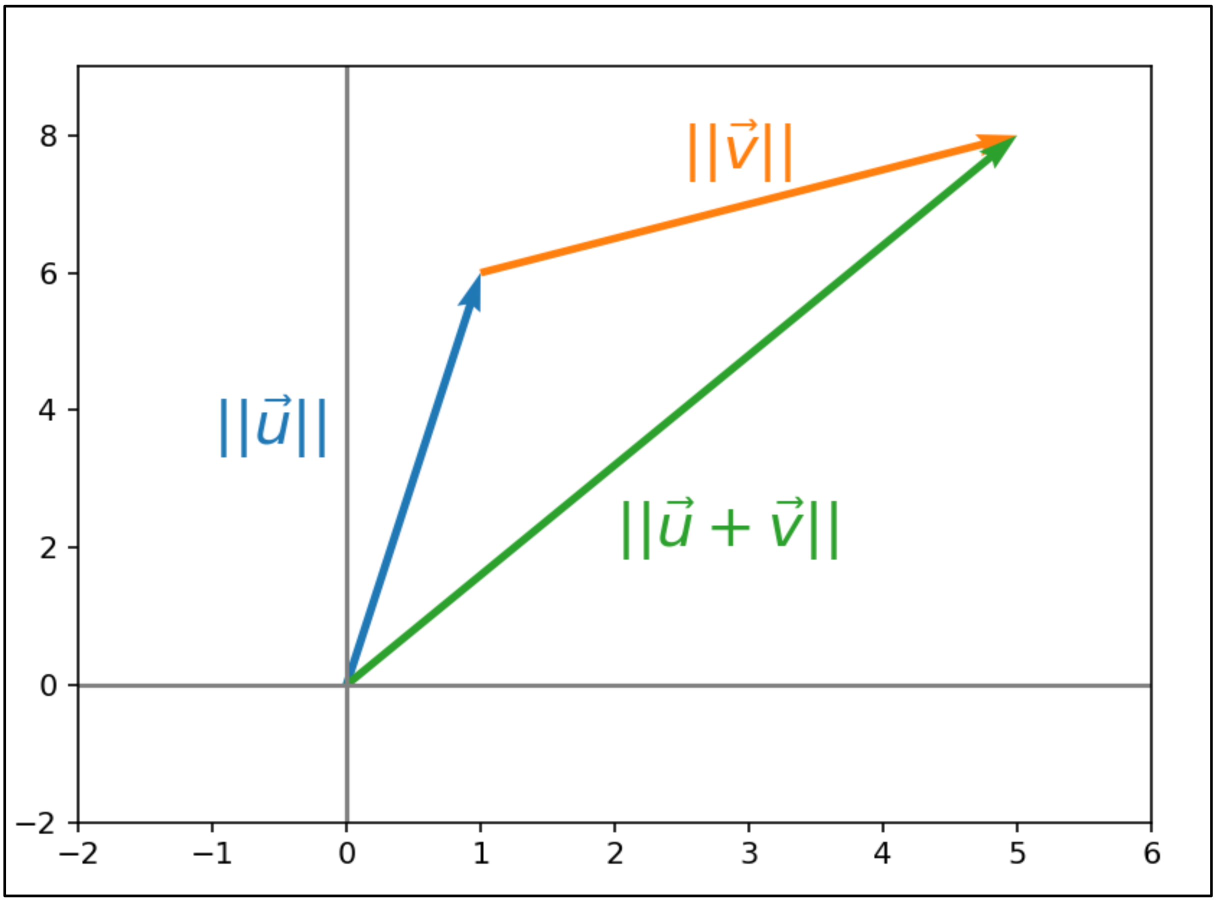


[V] Norm.

CODE:

```
import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns


u = [0,0,1,6]

v = [0,0,4,2]

u_bis = [1,6,v[2],v[3]]

w = [0,0,5,8]

plt.quiver([u[0], u_bis[0], w[0]],

           [u[1], u_bis[1], w[1]],

           [u[2], u_bis[2], w[2]],

           [u[3], u_bis[3], w[3]],

           angles = 'xy', scale_units = 'xy', scale = 1, color =
sns.color_palette())


plt.xlim(-2,6)

plt.ylim(-2,9)

plt.axvline(x=0, color='grey')

plt.axhline(y=0, color='grey')

plt.text(-1, 3.5, r'$||\vec{u}||$', color = sns.color_palette()[0], size
=20)

plt.text(2.5, 7.5, r'$||\vec{v}||$', color = sns.color_palette()[1], size
=20)

plt.text(2, 2, r'$||\vec{u}+\vec{v}||$', color = sns.color_palette()[2],
size =20)

plt.show()
```

OUTPUT:



[VI] Symmetric Matrices.

CODE:

```
import numpy as np


A = np.array([[2,4,-1],[4,-8,0],[-1,0,3]])
print(A)
print(A.T)
```

OUTPUT:

```
C:\Users\Admin\PycharmProjects\DLPracs
[[ 2  4 -1]
 [ 4 -8  0]
 [-1  0  3]]
[[ 2  4 -1]
 [ 4 -8  0]
 [-1  0  3]]
```

[B] AIM: Performing matrix multiplication and finding eigen vectors and eigen values using TensorFlow.

CODE:

```
import tensorflow as tf

print("Matrix Multiplication Demo")

x=tf.constant([1,2,3,4,5,6],shape=[2,3])

print(x)

y=tf.constant([7,8,9,10,11,12],shape=[3,2])

print(y)

z=tf.matmul(x,y)

print("Product:",z)

e_matrix_A=tf.random.uniform([2,2],minval=3,maxval=10,dtype=tf.float32,name
="matrixA")

print("Matrix A:\n{}\n\n".format(e_matrix_A))

eigen_values_A,eigen_vectors_A=tf.linalg.eigh(e_matrix_A)

print("Eigen Vectors:\n{}\n\n Eigen  Values:

\n{}\n".format(eigen_vectors_A,eigen_values_A))
```

OUTPUT:

```
Matrix Multiplication Demo
tf.Tensor(
[[1 2 3]
 [4 5 6]], shape=(2, 3), dtype=int32)
tf.Tensor(
[[ 7  8]
 [ 9 10]
 [11 12]], shape=(3, 2), dtype=int32)
Product: tf.Tensor(
[[ 58  64]
 [139 154]], shape=(2, 2), dtype=int32)
Matrix A:
[[9.669554  9.363649 ]
 [6.5489373 6.7324896]]


Eigen Vectors:
[[-0.6249776   0.78064275]
 [ 0.78064275  0.6249776 ]]

 Eigen   Values:
[ 1.4894521 14.912593 ]


Process finished with exit code 0
```

PRACTICAL 2

AIM: Solving XOR problem using deep feed forward network.

CODE:

```
import numpy as np
from keras.layers import Dense
from keras.models import Sequential
model=Sequential()
model.add(Dense(units=2,activation='relu',input_dim=2))
model.add(Dense(units=1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accurac
y'])
print(model.summary())
print(model.get_weights())
X=np.array([[0.,0.],[0.,1.],[1.,0.],[1.,1.]])
Y=np.array([0.,1.,1.,0.])
model.fit(X,Y,epochs=900,batch_size=4)
print(model.get_weights())
print(model.predict(X,batch_size=4))
```

OUTPUT:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 2)                 6

 dense_1 (Dense)             (None, 1)                 3

=================================================================
Total params: 9
Trainable params: 9
Non-trainable params: 0
_________________________________________________________________
None
[array([[ 0.36801147, -1.188614  ],
       [ 0.7940141 ,  0.95810044]], dtype=float32), array([0., 0.], dtype=float32), array([[-0.81275207],
       [-0.59155834]], dtype=float32), array([0.], dtype=float32)]
```

```
Epoch 1/1000
1/1 [==============================] - 2s 2s/step - loss: 0.8370 - accuracy: 0.5000
Epoch 2/1000
1/1 [==============================] - 0s 16ms/step - loss: 0.8360 - accuracy: 0.2500
Epoch 3/1000
1/1 [==============================] - 0s 16ms/step - loss: 0.8349 - accuracy: 0.2500
Epoch 4/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.8339 - accuracy: 0.2500
Epoch 5/1000
1/1 [==============================] - 0s 15ms/step - loss: 0.8328 - accuracy: 0.2500
Epoch 6/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.8318 - accuracy: 0.2500
Epoch 7/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.8308 - accuracy: 0.2500
Epoch 8/1000
1/1 [==============================] - 0s 18ms/step - loss: 0.8298 - accuracy: 0.2500
Epoch 9/1000
1/1 [==============================] - 0s 8ms/step - loss: 0.8288 - accuracy: 0.2500
Epoch 10/1000
1/1 [==============================] - 0s 8ms/step - loss: 0.8277 - accuracy: 0.2500
```

```
Epoch 11/1000
1/1 [==============================] - 0s 6ms/step - loss: 0.8267 - accuracy: 0.2500
Epoch 12/1000
1/1 [==============================] - 0s 7ms/step - loss: 0.8257 - accuracy: 0.2500
Epoch 13/1000
1/1 [==============================] - 0s 17ms/step - loss: 0.8247 - accuracy: 0.2500
Epoch 14/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.8238 - accuracy: 0.2500
Epoch 15/1000
1/1 [==============================] - 0s 106ms/step - loss: 0.8228 - accuracy: 0.2500
Epoch 16/1000
1/1 [==============================] - 0s 13ms/step - loss: 0.8218 - accuracy: 0.2500
Epoch 17/1000
1/1 [==============================] - 0s 14ms/step - loss: 0.8208 - accuracy: 0.2500
Epoch 18/1000
1/1 [==============================] - 0s 30ms/step - loss: 0.8199 - accuracy: 0.2500
Epoch 19/1000
1/1 [==============================] - 0s 32ms/step - loss: 0.8189 - accuracy: 0.2500
Epoch 20/1000
1/1 [==============================] - 0s 19ms/step - loss: 0.8179 - accuracy: 0.2500
```

```
Epoch 995/1000
1/1 [==============================] - 0s 5ms/step - loss: 0.5997 - accuracy: 0.7500
Epoch 996/1000
1/1 [==============================] - 0s 5ms/step - loss: 0.5996 - accuracy: 0.7500
Epoch 997/1000
1/1 [==============================] - 0s 6ms/step - loss: 0.5994 - accuracy: 0.7500
Epoch 998/1000
1/1 [==============================] - 0s 5ms/step - loss: 0.5993 - accuracy: 0.7500
Epoch 999/1000
1/1 [==============================] - 0s 5ms/step - loss: 0.5992 - accuracy: 0.7500
Epoch 1000/1000
1/1 [==============================] - 0s 6ms/step - loss: 0.5991 - accuracy: 0.7500
[array([[ 0.6527725 , -1.188614  ],
       [ 0.65223974,  0.47420728]], dtype=float32), array([-0.6530008 , -0.48389307], dtype=float32), array([[-1.389874 ],
       [-0.2258762]], dtype=float32), array([0.18355492], dtype=float32)]
1/1 [==============================] - 0s 176ms/step
[[0.5457603 ]
 [0.5457603 ]
 [0.5457603 ]
 [0.32680774]]
```

PRACTICAL 3

AIM: Implementing deep neural network for performing binary classification task.

CODE:

```
from numpy import loadtxt
from keras.models import Sequential
from keras.layers import Dense
dataset = loadtxt('diabetes.csv',delimiter= ',',skiprows = 1)
X = dataset[:,0:8]
Y = dataset[:,8]
print(X)
print(Y)
model =Sequential()
model.add(Dense(12,input_dim=8,activation='relu'))
model.add(Dense(8,input_dim=8,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accurac
y'])
model.fit(X,Y,batch_size=12,epochs=500)
Accuracy=model.evaluate(X,Y)
print('Accuracy of model is ',(Accuracy*100))
prediction=model.predict(X)
exec('for i in range(5):print(X[i].tolist(),prediction[i],Y[i])')
```

OUTPUT:

```
Epoch 1/150
77/77 [==============================] - 2s 3ms/step - loss: 2.5263 - accuracy: 0.5586
Epoch 2/150
77/77 [==============================] - 0s 3ms/step - loss: 1.1873 - accuracy: 0.5885
Epoch 3/150
77/77 [==============================] - 0s 3ms/step - loss: 1.0318 - accuracy: 0.5443
Epoch 4/150
77/77 [==============================] - 0s 3ms/step - loss: 0.9486 - accuracy: 0.5404
Epoch 5/150
77/77 [==============================] - 0s 3ms/step - loss: 0.8373 - accuracy: 0.5651
Epoch 6/150
77/77 [==============================] - 0s 3ms/step - loss: 0.7979 - accuracy: 0.5794
Epoch 7/150
77/77 [==============================] - 0s 3ms/step - loss: 0.7693 - accuracy: 0.5951
Epoch 8/150
77/77 [==============================] - 0s 3ms/step - loss: 0.7384 - accuracy: 0.6094
Epoch 9/150
77/77 [==============================] - 0s 3ms/step - loss: 0.7010 - accuracy: 0.6289
Epoch 10/150
77/77 [==============================] - 0s 4ms/step - loss: 0.6959 - accuracy: 0.6328
Epoch 11/150
77/77 [==============================] - 0s 5ms/step - loss: 0.6670 - accuracy: 0.6523
```

```
Accuracy of model is [0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375,
0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125,
0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0
.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125,
0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0
.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125,
0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0
.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125,
0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0
.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125,
0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0
.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125,
0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0
.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125,
0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0
.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125,
0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0
.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125,
0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0
.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125, 0.7734375, 0.479278564453125,
```

```
24/24 [==============================] - 0s 2ms/step
[6.0, 148.0, 72.0, 35.0, 0.0, 33.6, 0.627, 50.0] [0.73097056] 1.0
[1.0, 85.0, 66.0, 29.0, 0.0, 26.6, 0.351, 31.0] [0.11377989] 0.0
[8.0, 183.0, 64.0, 0.0, 0.0, 23.3, 0.672, 32.0] [0.87028414] 1.0
[1.0, 89.0, 66.0, 23.0, 94.0, 28.1, 0.167, 21.0] [0.08588263] 0.0
[0.0, 137.0, 40.0, 35.0, 168.0, 43.1, 2.288, 33.0] [0.66002005] 1.0
```

PRACTICAL 4

[A] AIM: Using deep feed forward network with two hidden layers for performing multiclass classification and predicting the class.

CODE:

```
import numpy as np
from sklearn.datasets import load_iris
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import to_categorical
from sklearn.model_selection import train_test_split

# Load the iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Convert target variable to one-hot encoded format
y = to_categorical(y)

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Define the model
model = Sequential()
model.add(Dense(64, input_dim=X_train.shape[1], activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(y_train.shape[1], activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=100, batch_size=32, validation_split=0.2)
```

```
# Evaluate the model on the test set
loss, accuracy = model.evaluate(X_test, y_test)
print('Test accuracy:', accuracy)
```

OUTPUT:

```
Epoch 1/100
3/3 [==============================] - 1s 73ms/step - loss: 1.2578 - accuracy: 0.3646 - val_loss: 1.2153 - val_accuracy: 0.2083
Epoch 2/100
3/3 [==============================] - 0s 11ms/step - loss: 1.0729 - accuracy: 0.3750 - val_loss: 1.0499 - val_accuracy: 0.5000
Epoch 3/100
3/3 [==============================] - 0s 12ms/step - loss: 0.9695 - accuracy: 0.6667 - val_loss: 0.9646 - val_accuracy: 0.4167
Epoch 4/100
3/3 [==============================] - 0s 12ms/step - loss: 0.9031 - accuracy: 0.6875 - val_loss: 0.9082 - val_accuracy: 0.5000
Epoch 5/100
3/3 [==============================] - 0s 12ms/step - loss: 0.8462 - accuracy: 0.6979 - val_loss: 0.8486 - val_accuracy: 0.5000
Epoch 6/100
3/3 [==============================] - 0s 12ms/step - loss: 0.7968 - accuracy: 0.7188 - val_loss: 0.7993 - val_accuracy: 0.5833
Epoch 7/100
3/3 [==============================] - 0s 12ms/step - loss: 0.7542 - accuracy: 0.8021 - val_loss: 0.7638 - val_accuracy: 0.7083
Epoch 8/100
3/3 [==============================] - 0s 12ms/step - loss: 0.7192 - accuracy: 0.9271 - val_loss: 0.7258 - val_accuracy: 0.9583
Epoch 9/100
3/3 [==============================] - 0s 12ms/step - loss: 0.6898 - accuracy: 0.9479 - val_loss: 0.6879 - val_accuracy: 0.9583
Epoch 10/100
3/3 [==============================] - 0s 13ms/step - loss: 0.6613 - accuracy: 0.8646 - val_loss: 0.6570 - val_accuracy: 0.9583
Epoch 11/100
3/3 [==============================] - 0s 11ms/step - loss: 0.6327 - accuracy: 0.8229 - val_loss: 0.6339 - val_accuracy: 0.9583
Epoch 12/100
3/3 [==============================] - 0s 11ms/step - loss: 0.6055 - accuracy: 0.8750 - val_loss: 0.6152 - val_accuracy: 0.9583
Epoch 13/100
3/3 [==============================] - 0s 13ms/step - loss: 0.5821 - accuracy: 0.9479 - val_loss: 0.6024 - val_accuracy: 1.0000
Epoch 14/100
3/3 [==============================] - 0s 13ms/step - loss: 0.5601 - accuracy: 0.9583 - val_loss: 0.5797 - val_accuracy: 1.0000
Epoch 15/100
3/3 [==============================] - 0s 13ms/step - loss: 0.5368 - accuracy: 0.9688 - val_loss: 0.5649 - val_accuracy: 1.0000
Epoch 16/100
3/3 [==============================] - 0s 13ms/step - loss: 0.5164 - accuracy: 0.9688 - val_loss: 0.5517 - val_accuracy: 0.9583
Epoch 17/100
3/3 [==============================] - 0s 12ms/step - loss: 0.4991 - accuracy: 0.9792 - val_loss: 0.5404 - val_accuracy: 0.9583
Epoch 18/100
3/3 [==============================] - 0s 11ms/step - loss: 0.4832 - accuracy: 0.9792 - val_loss: 0.5241 - val_accuracy: 0.9583
Epoch 19/100
3/3 [==============================] - 0s 10ms/step - loss: 0.4679 - accuracy: 0.9688 - val_loss: 0.5122 - val_accuracy: 0.9583
```

```
3/3 [==============================] - 0s 14ms/step - loss: 0.1055 - accuracy: 0.9688 - val_loss: 0.0829 - val_accuracy: 1.0000
Epoch 99/100
3/3 [==============================] - 0s 14ms/step - loss: 0.1041 - accuracy: 0.9688 - val_loss: 0.0790 - val_accuracy: 1.0000
Epoch 100/100
3/3 [==============================] - 0s 15ms/step - loss: 0.1025 - accuracy: 0.9688 - val_loss: 0.0853 - val_accuracy: 1.0000
1/1 [==============================] - 0s 20ms/step - loss: 0.1196 - accuracy: 0.9333
Test accuracy: 0.9333333373069763
```

[B] AIM: Using a deep feed forward network with two hidden layers for performing classification and predicting the probability of class.

CODE:

```
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense
from sklearn.datasets import make_blobs
from sklearn.preprocessing import MinMaxScaler

X, Y = make_blobs(n_samples=100, centers=2, n_features=2, random_state=1)
scalar = MinMaxScaler()
scalar.fit(X)
X = scalar.transform(X)
models = keras.Sequential()
models.add(Dense(4, input_dim=2, activation='relu'))
models.add(Dense(4, activation='relu'))
models.add(Dense(1, activation='sigmoid'))
models.compile(loss='binary_crossentropy', optimizer='adam')
models.fit(X, Y, epochs=500)
import numpy as np

Xnew, Yreal = make_blobs(n_samples=3, centers=2, n_features=2,
random_state=1)
Xnew = scalar.transform(Xnew)
Yclass = np.argmax(models.predict(Xnew), axis=-1)
Ynew = models.predict(Xnew)
for i in range(len(Xnew)):
print("X=%s,Predicted_probability=%s,Predicted_class=%s" % (Xnew[i],
Ynew[i], Yclass[i]))
```

OUTPUT:

```
Epoch 1/500
4/4 [==============================] - 1s 2ms/step - loss: 1.0409
Epoch 2/500
4/4 [==============================] - 0s 1ms/step - loss: 1.0221
Epoch 3/500
4/4 [==============================] - 0s 1ms/step - loss: 1.0026
Epoch 4/500
4/4 [==============================] - 0s 990us/step - loss: 0.9839
Epoch 5/500
4/4 [==============================] - 0s 999us/step - loss: 0.9667
Epoch 6/500
4/4 [==============================] - 0s 1ms/step - loss: 0.9513
Epoch 7/500
4/4 [==============================] - 0s 999us/step - loss: 0.9345
Epoch 8/500
4/4 [==============================] - 0s 1ms/step - loss: 0.9191
Epoch 9/500
4/4 [==============================] - 0s 661us/step - loss: 0.9056
Epoch 10/500
4/4 [==============================] - 0s 1ms/step - loss: 0.8912
Epoch 11/500
4/4 [==============================] - 0s 1ms/step - loss: 0.8772
Epoch 12/500
4/4 [==============================] - 0s 667us/step - loss: 0.8656
Epoch 13/500
4/4 [==============================] - 0s 1ms/step - loss: 0.8546
Epoch 14/500
4/4 [==============================] - 0s 1ms/step - loss: 0.8423
Epoch 15/500
4/4 [==============================] - 0s 1ms/step - loss: 0.8290
Epoch 16/500
4/4 [==============================] - 0s 667us/step - loss: 0.8169
Epoch 17/500
4/4 [==============================] - 0s 666us/step - loss: 0.8043
Epoch 18/500
4/4 [==============================] - 0s 1ms/step - loss: 0.7907
Epoch 19/500
4/4 [==============================] - 0s 1ms/step - loss: 0.7784
```

```
Epoch 485/500
4/4 [==============================] - 0s 1ms/step - loss: 0.0257
Epoch 486/500
4/4 [==============================] - 0s 1ms/step - loss: 0.0255
Epoch 487/500
4/4 [==============================] - 0s 1ms/step - loss: 0.0253
Epoch 488/500
4/4 [==============================] - 0s 999us/step - loss: 0.0252
Epoch 489/500
4/4 [==============================] - 0s 1ms/step - loss: 0.0250
Epoch 490/500
4/4 [==============================] - 0s 989us/step - loss: 0.0248
Epoch 491/500
4/4 [==============================] - 0s 0s/step - loss: 0.0247
Epoch 492/500
4/4 [==============================] - 0s 0s/step - loss: 0.0245
Epoch 493/500
4/4 [==============================] - 0s 0s/step - loss: 0.0243
Epoch 494/500
4/4 [==============================] - 0s 0s/step - loss: 0.0242
Epoch 495/500
4/4 [==============================] - 0s 6ms/step - loss: 0.0240
Epoch 496/500
4/4 [==============================] - 0s 665us/step - loss: 0.0239
Epoch 497/500
4/4 [==============================] - 0s 1ms/step - loss: 0.0237
Epoch 498/500
4/4 [==============================] - 0s 667us/step - loss: 0.0236
Epoch 499/500
4/4 [==============================] - 0s 1ms/step - loss: 0.0234
Epoch 500/500
4/4 [==============================] - 0s 2ms/step - loss: 0.0233
1/1 [==============================] - 0s 68ms/step
1/1 [==============================] - 0s 18ms/step
X=[0.89337759 0.65864154],Predicted_probability=[0.01474354],Predicted_class=0
X=[0.29097707 0.12978982],Predicted_probability=[0.9631602],Predicted_class=0
X=[0.78082614 0.75391697],Predicted_probability=[0.012025],Predicted_class=0
```

[C] AIM: Using a deep feed forward network with two hidden layers for performing linear regression and predicting values.

CODE:

```
from keras.models import Sequential
from keras.layers import Dense
from sklearn.datasets import make_regression
from sklearn.preprocessing import MinMaxScaler

X,Y=make_regression(n_samples=100,n_features=2,noise=0.1,random_state=1)
scalarX,scalarY=MinMaxScaler(),MinMaxScaler()

scalarX.fit(X)
scalarY.fit(Y.reshape(100,1))
X=scalarX.transform(X)
Y=scalarY.transform(Y.reshape(100,1))

model=Sequential()
model.add(Dense(4,input_dim=2,activation='relu'))
model.add(Dense(4,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='mse',optimizer='adam')
model.fit(X,Y,epochs=1000,verbose=0)

Xnew,a=make_regression(n_samples=3,n_features=2,noise=0.1,random_state=1)
Xnew=scalarX.transform(Xnew)

Ynew=model.predict(Xnew)

for i in range(len(Xnew)):
    print("X=%s,Predicted=%s"%(Xnew[i],Ynew[i]))
```

OUTPUT:

```
1/1 [==============================] - 0s 126ms/step
X=[0.29466096 0.30317302],Predicted=[0.18755545]
X=[0.39445118 0.79390858],Predicted=[0.7539139]
X=[0.02884127 0.6208843 ],Predicted=[0.3981339]
```

PRACTICAL 5

[A] AIM: Evaluating feed forward deep network for regression using KFold cross validation.

CODE:

```
import numpy as np
from sklearn.model_selection import KFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Generate sample data
X = np.random.rand(1000, 10)
y = np.sum(X, axis=1)

# Define KFold cross-validation
kfold = KFold(n_splits=5, shuffle=True, random_state=42)

# Initialize list to store evaluation metrics
eval_metrics = []

# Iterate through each fold
for train_index, test_index in kfold.split(X):
   # Split data into training and testing sets
   X_train, X_test = X[train_index], X[test_index]
   y_train, y_test = y[train_index], y[test_index]

   # Define and compile model
   model = Sequential()
   model.add(Dense(64, activation='relu', input_dim=10))
   model.add(Dense(1))
   model.compile(optimizer='adam', loss='mse', metrics=['mae'])

   # Fit model to training data
   model.fit(X_train, y_train, epochs=100, batch_size=32, verbose=0)

   # Evaluate model on testing data
   eval_metrics.append(model.evaluate(X_test, y_test))
```

```
# Print average evaluation metrics across all folds
print("Average evaluation metrics:")
print("Loss:", np.mean([m[0] for m in eval_metrics]))
print("MAE:", np.mean([m[1] for m in eval_metrics]))
```

OUTPUT:

```
7/7 [==============================] - 0s 1ms/step - loss: 3.7221e-04 - mae: 0.0139
7/7 [==============================] - 0s 1ms/step - loss: 1.7676e-04 - mae: 0.0104
7/7 [==============================] - 0s 837us/step - loss: 5.5438e-04 - mae: 0.0157
7/7 [==============================] - 0s 843us/step - loss: 4.1214e-04 - mae: 0.0163
7/7 [==============================] - 0s 845us/step - loss: 3.4817e-04 - mae: 0.0105
Average evaluation metrics:
Loss: 0.00037273057969287036
MAE: 0.013363478891551494
```

[B] AIM: Evaluating feed forward deep network for multiclass Classification using KFold cross-validation.

CODE:

```
from keras.models import Sequential
from keras.layers import Dense
from scikeras.wrappers import KerasClassifier
from keras.utils import to_categorical
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import LabelEncoder
from sklearn import datasets
from sklearn.pipeline import Pipeline

# load dataset
dataset = datasets.load_iris()
X = dataset.data[:, 0:4].astype(float)
Y = dataset.target

# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)

# convert integers to dummy variables (i.e. one hot encoded)
dummy_y = to_categorical(encoded_Y)

# define baseline model
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(8, input_dim=4, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
    return model
```

```
estimator = KerasClassifier(model=baseline_model, epochs=200, batch_size=5,
verbose=0)
kfold = KFold(n_splits=10, shuffle=True)
results = cross_val_score(estimator, X, dummy_y, cv=kfold)
print("Baseline: %.2f%% (%.2f%%)" % (results.mean() * 100, results.std() *
100))
```

OUTPUT:

```
Baseline: 96.67% (3.33%)

Process finished with exit code 0
```

PRACTICAL 6

AIM: Implementing regularization to avoid overfitting in binary classification.

[A] Without regularization.

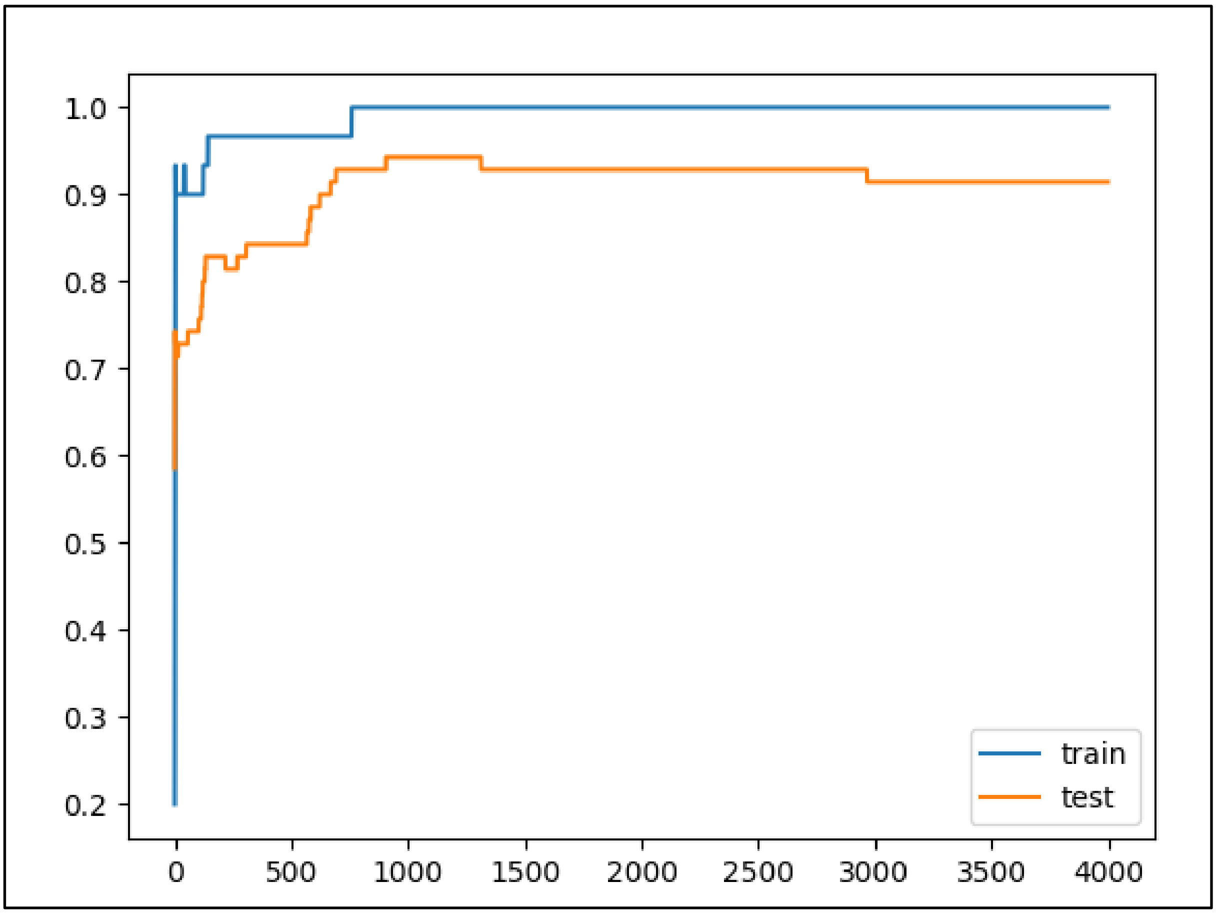
CODE:

```
from matplotlib import pyplot
from sklearn.datasets import make_moons
from keras.models import Sequential
from keras.layers import Dense
X,Y=make_moons(n_samples=100,noise=0.2,random_state=1)
n_train=30
trainX,testX=X[:n_train,:],X[n_train:]
trainY,testY=Y[:n_train],Y[n_train:]
model= Sequential()
model.add(Dense(500,input_dim=2,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accurac
y'])
history=model.fit(trainX,trainY,validation_data=(testX,testY),epochs=4000)
pyplot.plot(history.history['accuracy'],label='train')
pyplot.plot(history.history['val_accuracy'],label='test')
pyplot.legend()
pyplot.show()
```

OUTPUT:

```
Epoch 1/4000
1/1 [==============================] - 1s 726ms/step - loss: 0.7134 - accuracy: 0.2000 - val_loss: 0.6934 - val_accuracy: 0.5857
Epoch 2/4000
1/1 [==============================] - 0s 24ms/step - loss: 0.6956 - accuracy: 0.5000 - val_loss: 0.6818 - val_accuracy: 0.7429
Epoch 3/4000
1/1 [==============================] - 0s 25ms/step - loss: 0.6783 - accuracy: 0.8333 - val_loss: 0.6706 - val_accuracy: 0.7429
Epoch 4/4000
1/1 [==============================] - 0s 23ms/step - loss: 0.6616 - accuracy: 0.9333 - val_loss: 0.6598 - val_accuracy: 0.7286
Epoch 5/4000
1/1 [==============================] - 0s 25ms/step - loss: 0.6453 - accuracy: 0.9000 - val_loss: 0.6493 - val_accuracy: 0.7286
Epoch 6/4000
1/1 [==============================] - 0s 23ms/step - loss: 0.6295 - accuracy: 0.9000 - val_loss: 0.6393 - val_accuracy: 0.7286
Epoch 7/4000
1/1 [==============================] - 0s 26ms/step - loss: 0.6141 - accuracy: 0.9000 - val_loss: 0.6296 - val_accuracy: 0.7286
Epoch 8/4000
1/1 [==============================] - 0s 24ms/step - loss: 0.5992 - accuracy: 0.9000 - val_loss: 0.6204 - val_accuracy: 0.7286
Epoch 9/4000
1/1 [==============================] - 0s 25ms/step - loss: 0.5848 - accuracy: 0.9000 - val_loss: 0.6114 - val_accuracy: 0.7286
Epoch 10/4000
1/1 [==============================] - 0s 24ms/step - loss: 0.5708 - accuracy: 0.9000 - val_loss: 0.6029 - val_accuracy: 0.7286
Epoch 11/4000
1/1 [==============================] - 0s 24ms/step - loss: 0.5572 - accuracy: 0.9000 - val_loss: 0.5946 - val_accuracy: 0.7143
Epoch 12/4000
1/1 [==============================] - 0s 23ms/step - loss: 0.5440 - accuracy: 0.9000 - val_loss: 0.5867 - val_accuracy: 0.7143
Epoch 13/4000
1/1 [==============================] - 0s 25ms/step - loss: 0.5311 - accuracy: 0.9000 - val_loss: 0.5792 - val_accuracy: 0.7143
Epoch 14/4000
1/1 [==============================] - 0s 27ms/step - loss: 0.5187 - accuracy: 0.9000 - val_loss: 0.5719 - val_accuracy: 0.7143
Epoch 15/4000
1/1 [==============================] - 0s 28ms/step - loss: 0.5066 - accuracy: 0.9000 - val_loss: 0.5649 - val_accuracy: 0.7143
Epoch 16/4000
1/1 [==============================] - 0s 27ms/step - loss: 0.4949 - accuracy: 0.9000 - val_loss: 0.5583 - val_accuracy: 0.7286
```

```
Epoch 3987/4000
1/1 [==============================] - 0s 26ms/step - loss: 2.1828e-04 - accuracy: 1.0000 - val_loss: 0.4914 - val_accuracy: 0.9143
Epoch 3988/4000
1/1 [==============================] - 0s 24ms/step - loss: 2.1811e-04 - accuracy: 1.0000 - val_loss: 0.4914 - val_accuracy: 0.9143
Epoch 3989/4000
1/1 [==============================] - 0s 26ms/step - loss: 2.1793e-04 - accuracy: 1.0000 - val_loss: 0.4915 - val_accuracy: 0.9143
Epoch 3990/4000
1/1 [==============================] - 0s 23ms/step - loss: 2.1775e-04 - accuracy: 1.0000 - val_loss: 0.4915 - val_accuracy: 0.9143
Epoch 3991/4000
1/1 [==============================] - 0s 23ms/step - loss: 2.1757e-04 - accuracy: 1.0000 - val_loss: 0.4916 - val_accuracy: 0.9143
Epoch 3992/4000
1/1 [==============================] - 0s 21ms/step - loss: 2.1739e-04 - accuracy: 1.0000 - val_loss: 0.4917 - val_accuracy: 0.9143
Epoch 3993/4000
1/1 [==============================] - 0s 23ms/step - loss: 2.1723e-04 - accuracy: 1.0000 - val_loss: 0.4917 - val_accuracy: 0.9143
Epoch 3994/4000
1/1 [==============================] - 0s 21ms/step - loss: 2.1706e-04 - accuracy: 1.0000 - val_loss: 0.4918 - val_accuracy: 0.9143
Epoch 3995/4000
1/1 [==============================] - 0s 22ms/step - loss: 2.1687e-04 - accuracy: 1.0000 - val_loss: 0.4918 - val_accuracy: 0.9143
Epoch 3996/4000
1/1 [==============================] - 0s 26ms/step - loss: 2.1669e-04 - accuracy: 1.0000 - val_loss: 0.4918 - val_accuracy: 0.9143
Epoch 3997/4000
1/1 [==============================] - 0s 25ms/step - loss: 2.1651e-04 - accuracy: 1.0000 - val_loss: 0.4919 - val_accuracy: 0.9143
Epoch 3998/4000
1/1 [==============================] - 0s 24ms/step - loss: 2.1633e-04 - accuracy: 1.0000 - val_loss: 0.4920 - val_accuracy: 0.9143
Epoch 3999/4000
1/1 [==============================] - 0s 24ms/step - loss: 2.1615e-04 - accuracy: 1.0000 - val_loss: 0.4920 - val_accuracy: 0.9143
Epoch 4000/4000
1/1 [==============================] - 0s 24ms/step - loss: 2.1598e-04 - accuracy: 1.0000 - val_loss: 0.4921 - val_accuracy: 0.9143
```


1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
500
1000
1500
2000
2500
3000
3500
4000
train
test

[B] Implementing L2 regularization.

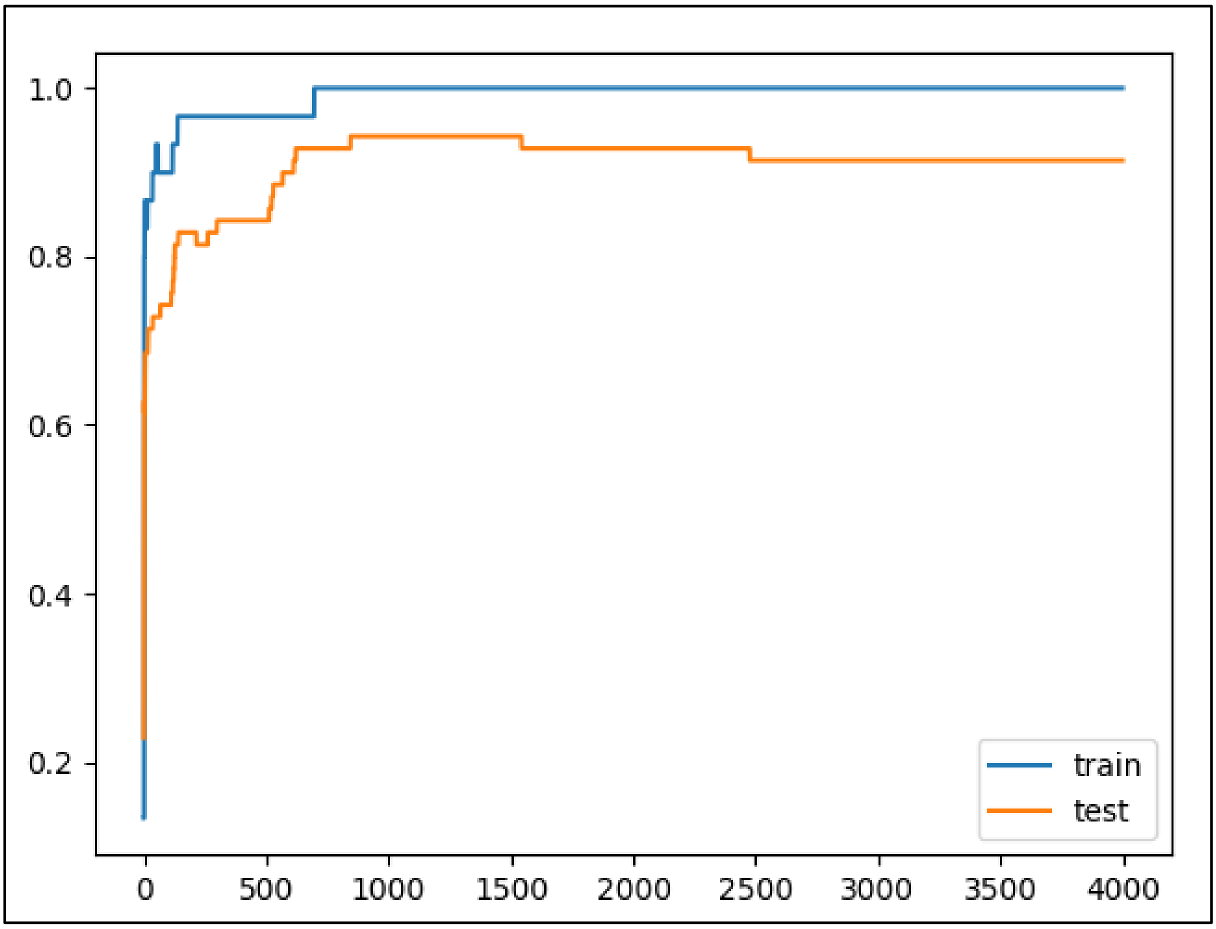
CODE:

```
from matplotlib import pyplot
from sklearn.datasets import make_moons
from keras.models import Sequential
from keras.layers import Dense
from keras.regularizers import l1_l2
X,Y=make_moons(n_samples=100,noise=0.2,random_state=1)
n_train=30
trainX,testX=X[:n_train,:],X[n_train:]
trainY,testY=Y[:n_train],Y[n_train:]
model= Sequential()
model.add(Dense(500,input_dim=2,activation='relu',kernel_regularizer=l1_l2(
l1=0.001,l2=0.001)))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accurac
y'])
history=model.fit(trainX,trainY,validation_data=(testX,testY),epochs=4000)
pyplot.plot(history.history['accuracy'],label='train')
pyplot.plot(history.history['val_accuracy'],label='test')
pyplot.legend()
pyplot.show()
```

OUTPUT:

```
Epoch 1/4000
1/1 [==============================] - 1s 514ms/step - loss: 0.7230 - accuracy: 0.1333 - val_loss: 0.7073 - val_accuracy: 0.2286
Epoch 2/4000
1/1 [==============================] - 0s 23ms/step - loss: 0.7060 - accuracy: 0.1333 - val_loss: 0.6964 - val_accuracy: 0.6000
Epoch 3/4000
1/1 [==============================] - 0s 21ms/step - loss: 0.6894 - accuracy: 0.8000 - val_loss: 0.6858 - val_accuracy: 0.6286
Epoch 4/4000
1/1 [==============================] - 0s 23ms/step - loss: 0.6733 - accuracy: 0.8000 - val_loss: 0.6756 - val_accuracy: 0.6143
Epoch 5/4000
1/1 [==============================] - 0s 23ms/step - loss: 0.6576 - accuracy: 0.8667 - val_loss: 0.6657 - val_accuracy: 0.6286
Epoch 6/4000
1/1 [==============================] - 0s 22ms/step - loss: 0.6424 - accuracy: 0.8667 - val_loss: 0.6562 - val_accuracy: 0.6571
Epoch 7/4000
1/1 [==============================] - 0s 24ms/step - loss: 0.6276 - accuracy: 0.8333 - val_loss: 0.6470 - val_accuracy: 0.6857
Epoch 8/4000
1/1 [==============================] - 0s 22ms/step - loss: 0.6133 - accuracy: 0.8333 - val_loss: 0.6382 - val_accuracy: 0.6857
Epoch 9/4000
1/1 [==============================] - 0s 21ms/step - loss: 0.5993 - accuracy: 0.8333 - val_loss: 0.6297 - val_accuracy: 0.6857
Epoch 10/4000
1/1 [==============================] - 0s 21ms/step - loss: 0.5858 - accuracy: 0.8333 - val_loss: 0.6214 - val_accuracy: 0.6857
Epoch 11/4000
1/1 [==============================] - 0s 22ms/step - loss: 0.5726 - accuracy: 0.8333 - val_loss: 0.6135 - val_accuracy: 0.6857
Epoch 12/4000
1/1 [==============================] - 0s 23ms/step - loss: 0.5598 - accuracy: 0.8333 - val_loss: 0.6059 - val_accuracy: 0.6857
```

```
Epoch 3989/4000
1/1 [==============================] - 0s 23ms/step - loss: 1.8910e-04 - accuracy: 1.0000 - val_loss: 0.4993 - val_accuracy: 0.9143
Epoch 3990/4000
1/1 [==============================] - 0s 23ms/step - loss: 1.8894e-04 - accuracy: 1.0000 - val_loss: 0.4993 - val_accuracy: 0.9143
Epoch 3991/4000
1/1 [==============================] - 0s 23ms/step - loss: 1.8880e-04 - accuracy: 1.0000 - val_loss: 0.4994 - val_accuracy: 0.9143
Epoch 3992/4000
1/1 [==============================] - 0s 30ms/step - loss: 1.8866e-04 - accuracy: 1.0000 - val_loss: 0.4994 - val_accuracy: 0.9143
Epoch 3993/4000
1/1 [==============================] - 0s 29ms/step - loss: 1.8851e-04 - accuracy: 1.0000 - val_loss: 0.4995 - val_accuracy: 0.9143
Epoch 3994/4000
1/1 [==============================] - 0s 29ms/step - loss: 1.8836e-04 - accuracy: 1.0000 - val_loss: 0.4995 - val_accuracy: 0.9143
Epoch 3995/4000
1/1 [==============================] - 0s 26ms/step - loss: 1.8821e-04 - accuracy: 1.0000 - val_loss: 0.4996 - val_accuracy: 0.9143
Epoch 3996/4000
1/1 [==============================] - 0s 24ms/step - loss: 1.8806e-04 - accuracy: 1.0000 - val_loss: 0.4996 - val_accuracy: 0.9143
Epoch 3997/4000
1/1 [==============================] - 0s 21ms/step - loss: 1.8792e-04 - accuracy: 1.0000 - val_loss: 0.4997 - val_accuracy: 0.9143
Epoch 3998/4000
1/1 [==============================] - 0s 22ms/step - loss: 1.8777e-04 - accuracy: 1.0000 - val_loss: 0.4997 - val_accuracy: 0.9143
Epoch 3999/4000
1/1 [==============================] - 0s 22ms/step - loss: 1.8762e-04 - accuracy: 1.0000 - val_loss: 0.4998 - val_accuracy: 0.9143
Epoch 4000/4000
1/1 [==============================] - 0s 28ms/step - loss: 1.8747e-04 - accuracy: 1.0000 - val_loss: 0.4998 - val_accuracy: 0.9143
```


1.0
0.8
0.6
0.4
0.2
0
500
1000
1500
2000
2500
3000
3500
4000
train
test

[C] Replacing L2 regularizer with L1 regularizer.

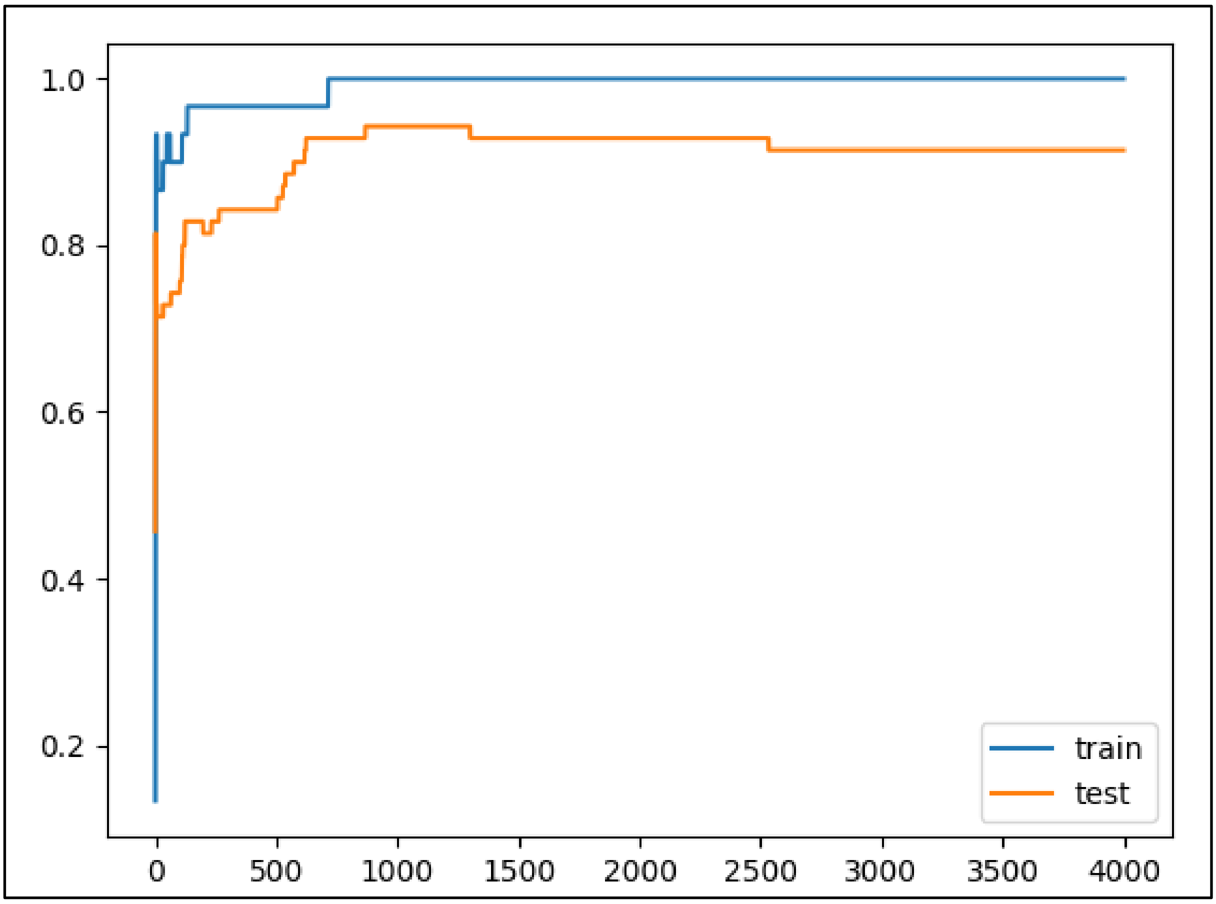
CODE:

```
from matplotlib import pyplot
from sklearn.datasets import make_moons
from keras.models import Sequential
from keras.layers import Dense
from keras.regularizers import l1_l2
X,Y=make_moons(n_samples=100,noise=0.2,random_state=1)
n_train=30
trainX,testX=X[:n_train,:],X[n_train:]
trainY,testY=Y[:n_train],Y[n_train:]
model= Sequential()
model.add(Dense(500,input_dim=2,activation='relu',kernel_regularizer=l1_l2(
l1=0.001,l2=0.001)))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accurac
y'])
history=model.fit(trainX,trainY,validation_data=(testX,testY),epochs=4000)
pyplot.plot(history.history['accuracy'],label='train')
pyplot.plot(history.history['val_accuracy'],label='test')
pyplot.legend()
pyplot.show()
```

OUTPUT:

```
Epoch 1/4000
1/1 [==============================] - 1s 513ms/step - loss: 0.7198 - accuracy: 0.1333 - val_loss: 0.6967 - val_accuracy: 0.4571
Epoch 2/4000
1/1 [==============================] - 0s 25ms/step - loss: 0.7030 - accuracy: 0.4333 - val_loss: 0.6858 - val_accuracy: 0.7286
Epoch 3/4000
1/1 [==============================] - 0s 25ms/step - loss: 0.6865 - accuracy: 0.6000 - val_loss: 0.6754 - val_accuracy: 0.8143
Epoch 4/4000
1/1 [==============================] - 0s 24ms/step - loss: 0.6705 - accuracy: 0.9000 - val_loss: 0.6653 - val_accuracy: 0.7429
Epoch 5/4000
1/1 [==============================] - 0s 23ms/step - loss: 0.6550 - accuracy: 0.9333 - val_loss: 0.6557 - val_accuracy: 0.7286
Epoch 6/4000
1/1 [==============================] - 0s 25ms/step - loss: 0.6400 - accuracy: 0.9333 - val_loss: 0.6464 - val_accuracy: 0.7286
Epoch 7/4000
1/1 [==============================] - 0s 21ms/step - loss: 0.6254 - accuracy: 0.9333 - val_loss: 0.6375 - val_accuracy: 0.7286
Epoch 8/4000
1/1 [==============================] - 0s 23ms/step - loss: 0.6113 - accuracy: 0.9000 - val_loss: 0.6289 - val_accuracy: 0.7286
Epoch 9/4000
1/1 [==============================] - 0s 22ms/step - loss: 0.5975 - accuracy: 0.9000 - val_loss: 0.6206 - val_accuracy: 0.7143
Epoch 10/4000
1/1 [==============================] - 0s 23ms/step - loss: 0.5841 - accuracy: 0.9000 - val_loss: 0.6127 - val_accuracy: 0.7143
Epoch 11/4000
1/1 [==============================] - 0s 21ms/step - loss: 0.5711 - accuracy: 0.8667 - val_loss: 0.6050 - val_accuracy: 0.7143
```

```
Epoch 3990/4000
1/1 [==============================] - 0s 21ms/step - loss: 2.0657e-04 - accuracy: 1.0000 - val_loss: 0.4963 - val_accuracy: 0.9143
Epoch 3991/4000
1/1 [==============================] - 0s 28ms/step - loss: 2.0639e-04 - accuracy: 1.0000 - val_loss: 0.4963 - val_accuracy: 0.9143
Epoch 3992/4000
1/1 [==============================] - 0s 29ms/step - loss: 2.0623e-04 - accuracy: 1.0000 - val_loss: 0.4964 - val_accuracy: 0.9143
Epoch 3993/4000
1/1 [==============================] - 0s 31ms/step - loss: 2.0606e-04 - accuracy: 1.0000 - val_loss: 0.4964 - val_accuracy: 0.9143
Epoch 3994/4000
1/1 [==============================] - 0s 27ms/step - loss: 2.0589e-04 - accuracy: 1.0000 - val_loss: 0.4965 - val_accuracy: 0.9143
Epoch 3995/4000
1/1 [==============================] - 0s 23ms/step - loss: 2.0572e-04 - accuracy: 1.0000 - val_loss: 0.4965 - val_accuracy: 0.9143
Epoch 3996/4000
1/1 [==============================] - 0s 22ms/step - loss: 2.0557e-04 - accuracy: 1.0000 - val_loss: 0.4966 - val_accuracy: 0.9143
Epoch 3997/4000
1/1 [==============================] - 0s 21ms/step - loss: 2.0540e-04 - accuracy: 1.0000 - val_loss: 0.4966 - val_accuracy: 0.9143
Epoch 3998/4000
1/1 [==============================] - 0s 22ms/step - loss: 2.0523e-04 - accuracy: 1.0000 - val_loss: 0.4967 - val_accuracy: 0.9143
Epoch 3999/4000
1/1 [==============================] - 0s 29ms/step - loss: 2.0507e-04 - accuracy: 1.0000 - val_loss: 0.4968 - val_accuracy: 0.9143
Epoch 4000/4000
1/1 [==============================] - 0s 29ms/step - loss: 2.0489e-04 - accuracy: 1.0000 - val_loss: 0.4968 - val_accuracy: 0.9143
```


1.0
0.8
0.6
0.4
0.2
0
500
1000
1500
2000
2500
3000
3500
4000
train
test

PRACTICAL 7

AIM: Demonstrate recurrent neural network that learns to perform sequence analysis for stock price.

CODE:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout
from sklearn.preprocessing import MinMaxScaler
dataset_train=pd.read_csv('Google_Stock_Price_Train.csv')
#print(dataset_train)
training_set=dataset_train.iloc[:,1:2].values
print(training_set)
sc=MinMaxScaler(feature_range=(0,1))
training_set_scaled=sc.fit_transform(training_set)
print(training_set_scaled)
X_train=[]
Y_train=[]
for i in range(60,1258):
    X_train.append(training_set_scaled[i-60:i,0])
    Y_train.append(training_set_scaled[i,0])
X_train,Y_train=np.array(X_train),np.array(Y_train)
print(X_train)
print('*********************************************')
print(Y_train)
X_train=np.reshape(X_train,(X_train.shape[0],X_train.shape[1],1))
print('*********************************************')
print(X_train)
regressor=Sequential()
regressor.add(LSTM(units=50,return_sequences=True,input_shape=(X_train.shap
e[1],1)))
regressor.add(Dropout(0.2))
```

```
regressor.add(LSTM(units=50,return_sequences=True))
regressor.add(Dropout(0.2))
regressor.add(LSTM(units=50,return_sequences=True))
regressor.add(Dropout(0.2))
regressor.add(LSTM(units=50))
regressor.add(Dropout(0.2))
regressor.add(Dense(units=1))
regressor.compile(optimizer='adam',loss='mean_squared_error')
regressor.fit(X_train,Y_train,epochs=100,batch_size=32)
dataset_test=pd.read_csv('Google_Stock_Price_Test.csv')
real_stock_price=dataset_test.iloc[:,1:2].values
dataset_total=pd.concat((dataset_train['Open'],dataset_test['Open']),axis=0
)
inputs=dataset_total[len(dataset_total)-len(dataset_test)-60:].values
inputs=inputs.reshape(-1,1)
inputs=sc.transform(inputs)
X_test=[]
for i in range(60,80):
    X_test.append(inputs[i-60:i,0])
X_test=np.array(X_test)
X_test=np.reshape(X_test,(X_test.shape[0],X_test.shape[1],1))
predicted_stock_price=regressor.predict(X_test)
predicted_stock_price=sc.inverse_transform(predicted_stock_price)
plt.plot(real_stock_price,color='red',label='real google stock price')
plt.plot(predicted_stock_price,color='blue',label='predicted stock price')
plt.xlabel('time')
plt.ylabel('google stock price')
plt.legend()
plt.show()
```

OUTPUT:

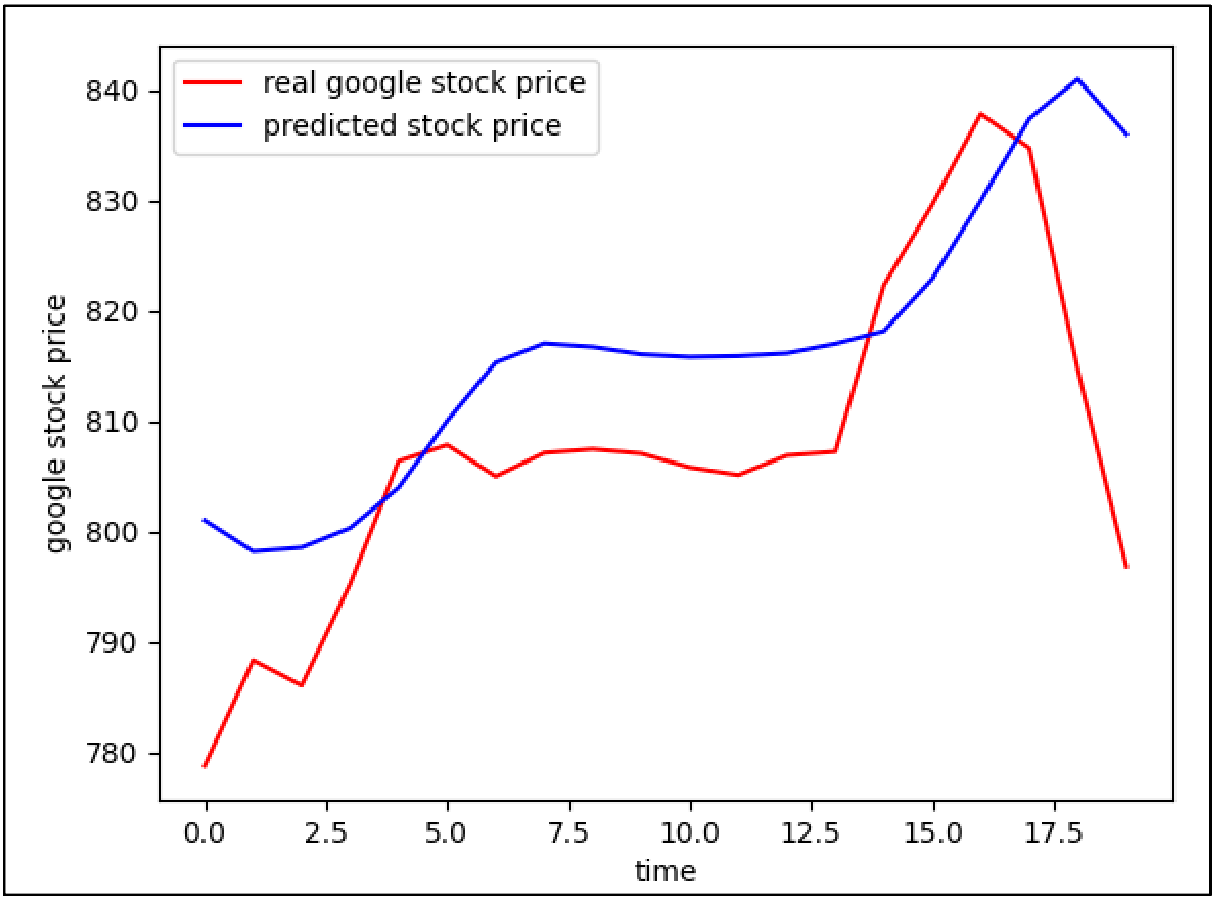
```
[[[0.08581368]
  [0.09701243]
  [0.09433366]
  ...
  [0.07846566]
  [0.08034452]
  [0.08497656]]

 [[0.09701243]
  [0.09433366]
  [0.09156187]
  ...
  [0.08034452]
  [0.08497656]
  [0.08627874]]

 [[0.09433366]
  [0.09156187]
  [0.07984225]
  ...
  [0.08497656]
  [0.08627874]
  [0.08471612]]

 ...
```

```
 [[0.92106928]
  [0.92438053]
  [0.93048218]
  ...
  [0.95475854]
  [0.95204256]
  [0.95163331]]

 [[0.92438053]
  [0.93048218]
  [0.9299055 ]
  ...
  [0.95204256]
  [0.95163331]
  [0.95725128]]

 [[0.93048218]
  [0.9299055 ]
  [0.93113327]
  ...
  [0.95163331]
  [0.95725128]
  [0.93796041]]]
```

```
[[325.25]
 [331.27]
 [329.83]
 ...
 [793.7 ]
 [783.33]
 [782.75]]
[[0.08581368]
 [0.09701243]
 [0.09433366]
 ...
 [0.95725128]
 [0.93796041]
 [0.93688146]]
[[0.08581368 0.09701243 0.09433366 ... 0.07846566 0.08034452 0.08497656]
 [0.09701243 0.09433366 0.09156187 ... 0.08034452 0.08497656 0.08627874]
 [0.09433366 0.09156187 0.07984225 ... 0.08497656 0.08627874 0.08471612]
 ...
 [0.92106928 0.92438053 0.93048218 ... 0.95475854 0.95204256 0.95163331]
 [0.92438053 0.93048218 0.9299055  ... 0.95204256 0.95163331 0.95725128]
 [0.93048218 0.9299055  0.93113327 ... 0.95163331 0.95725128 0.93796041]]
*********************************************
[0.08627874 0.08471612 0.07454052 ... 0.95725128 0.93796041 0.93688146]
**********************************************
```

```
Epoch 1/100
38/38 [==============================] - 6s 48ms/step - loss: 0.0517
Epoch 2/100
38/38 [==============================] - 2s 48ms/step - loss: 0.0068
Epoch 3/100
38/38 [==============================] - 2s 48ms/step - loss: 0.0058
Epoch 4/100
38/38 [==============================] - 2s 48ms/step - loss: 0.0050
Epoch 5/100
38/38 [==============================] - 2s 47ms/step - loss: 0.0046
Epoch 6/100
38/38 [==============================] - 2s 47ms/step - loss: 0.0048
```

```
38/38 [==============================] - 2s 48ms/step - loss: 0.0016
Epoch 97/100
38/38 [==============================] - 2s 49ms/step - loss: 0.0015
Epoch 98/100
38/38 [==============================] - 2s 48ms/step - loss: 0.0016
Epoch 99/100
38/38 [==============================] - 2s 49ms/step - loss: 0.0015
Epoch 100/100
38/38 [==============================] - 2s 48ms/step - loss: 0.0015
1/1 [==============================] - 1s 983ms/step
```



PRACTICAL 8

AIM: Performing encoding and decoding of images using deep autoencoder.

CODE:

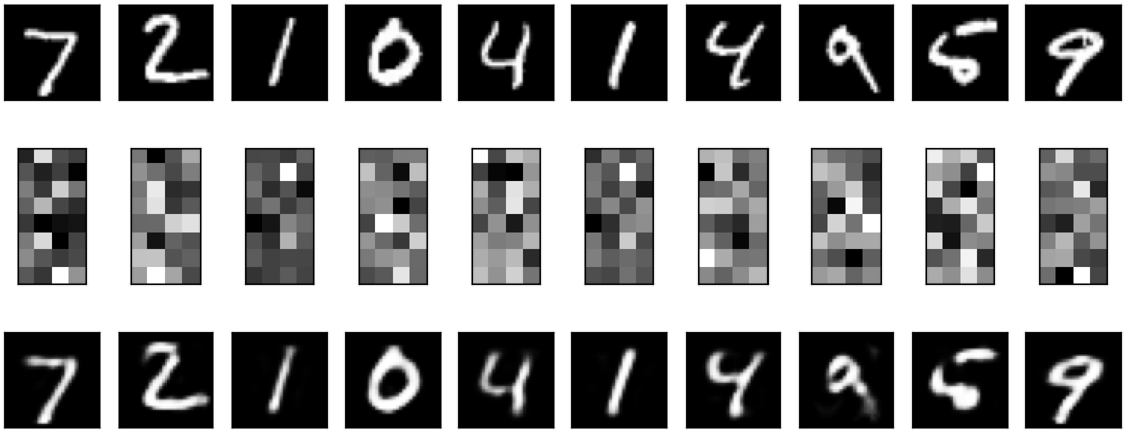
```
import keras
from keras import layers
from keras.datasets import mnist
import numpy as np
encoding_dim=32
#this is our input image
input_img=keras.Input(shape=(784,))
#"encoded" is the encoded representation of the input
encoded=layers.Dense(encoding_dim, activation='relu')(input_img)
#"decoded" is the lossy reconstruction of the input
decoded=layers.Dense(784, activation='sigmoid')(encoded)
#creating autoencoder model
autoencoder=keras.Model(input_img,decoded)
#create the encoder model
encoder=keras.Model(input_img,encoded)
encoded_input=keras.Input(shape=(encoding_dim,))
#Retrive the last layer of the autoencoder model
decoder_layer=autoencoder.layers[-1]
#create the decoder model
decoder=keras.Model(encoded_input,decoder_layer(encoded_input))
autoencoder.compile(optimizer='adam',loss='binary_crossentropy')
#scale and make train and test dataset
(X_train,_),(X_test,_)=mnist.load_data()
X_train=X_train.astype('float32')/255.
X_test=X_test.astype('float32')/255.
X_train=X_train.reshape((len(X_train),np.prod(X_train.shape[1:])))
X_test=X_test.reshape((len(X_test),np.prod(X_test.shape[1:])))
print(X_train.shape)
print(X_test.shape)
#train autoencoder with training dataset
autoencoder.fit(X_train,X_train,  epochs=50,  batch_size=256,  shuffle=True,
validation_data=(X_test,X_test))
```

```
encoded_imgs=encoder.predict(X_test)
decoded_imgs=decoder.predict(encoded_imgs)
import matplotlib.pyplot as plt
n = 10 # How many digits we will display
plt.figure(figsize=(40, 4))
for i in range(10):
# display original
    ax = plt.subplot(3, 20, i + 1)
    plt.imshow(X_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    # display encoded image
    ax = plt.subplot(3, 20, i + 1 + 20)
    plt.imshow(encoded_imgs[i].reshape(8, 4))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    # display reconstruction
    ax = plt.subplot(3, 20, 2 * 20 + i + 1)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

OUTPUT:

```
11490434/11490434 [==============================] - 1s 0us/step
(60000, 784)
(10000, 784)
Epoch 1/50
235/235 [==============================] - 1s 4ms/step - loss: 0.2784 - val_loss: 0.1932
Epoch 2/50
235/235 [==============================] - 1s 3ms/step - loss: 0.1722 - val_loss: 0.1533
Epoch 3/50
235/235 [==============================] - 1s 3ms/step - loss: 0.1434 - val_loss: 0.1330
Epoch 4/50
235/235 [==============================] - 1s 3ms/step - loss: 0.1274 - val_loss: 0.1199
Epoch 5/50
235/235 [==============================] - 1s 3ms/step - loss: 0.1170 - val_loss: 0.1114
Epoch 6/50
235/235 [==============================] - 1s 3ms/step - loss: 0.1101 - val_loss: 0.1059
Epoch 7/50
235/235 [==============================] - 1s 3ms/step - loss: 0.1054 - val_loss: 0.1020
```

```
235/235 [==============================] - 1s 3ms/step - loss: 0.0927 - val_loss: 0.0916
Epoch 48/50
235/235 [==============================] - 1s 3ms/step - loss: 0.0927 - val_loss: 0.0916
Epoch 49/50
235/235 [==============================] - 1s 3ms/step - loss: 0.0927 - val_loss: 0.0916
Epoch 50/50
235/235 [==============================] - 1s 3ms/step - loss: 0.0927 - val_loss: 0.0916
313/313 [==============================] - 0s 551us/step
313/313 [==============================] - 0s 601us/step
```



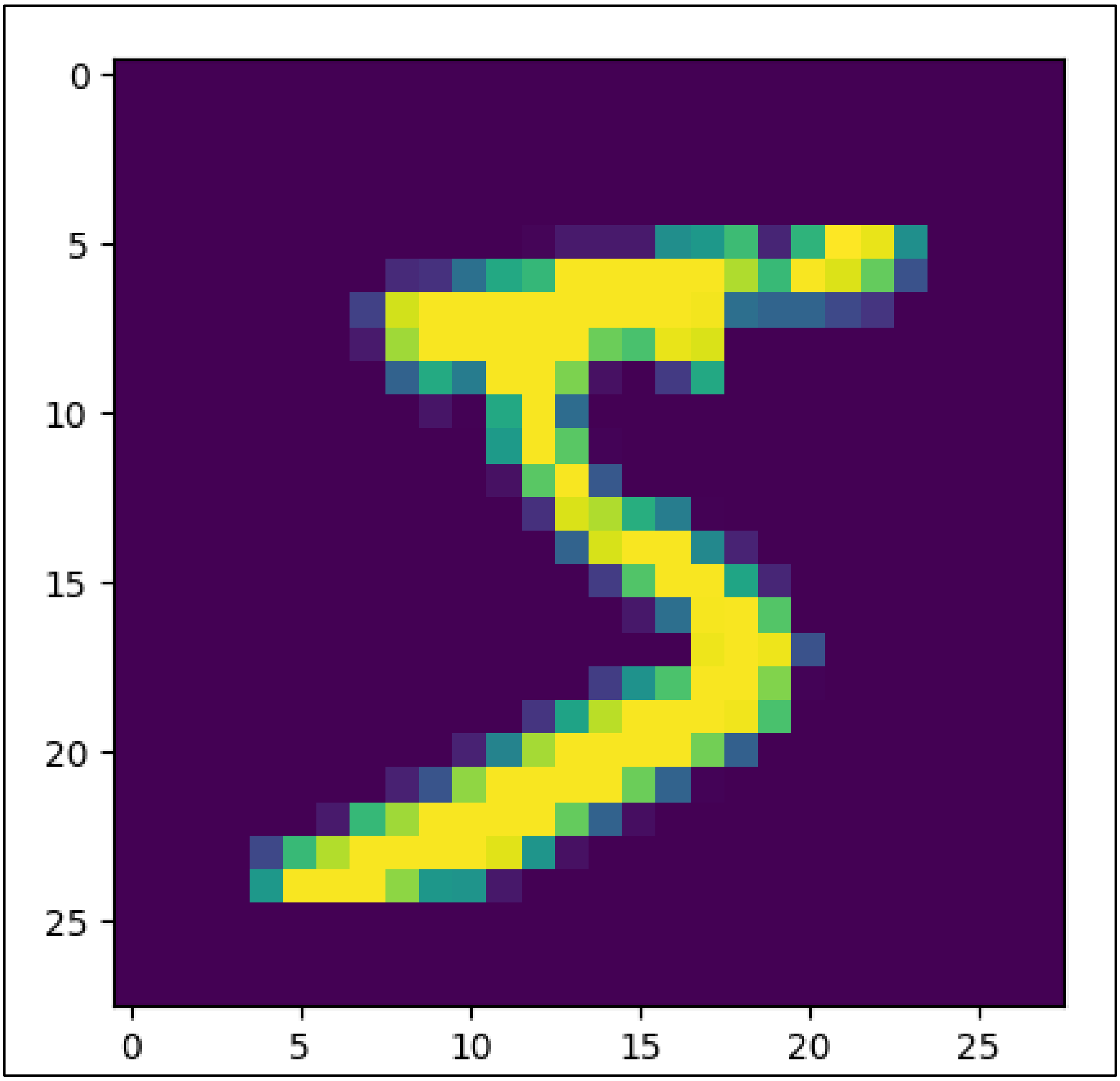
PRACTICAL 9

AIM: Implementation of convolutional neural network to predict numbers from number images.

CODE:

```
from keras.datasets import mnist
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense,Conv2D,Flatten
import matplotlib.pyplot as plt
#download mnist data and split into train and test sets
(X_train,Y_train),(X_test,Y_test)=mnist.load_data()
#plot the first image in the dataset
plt.imshow(X_train[0])
plt.show()
print(X_train[0].shape)
X_train=X_train.reshape(60000,28,28,1)
X_test=X_test.reshape(10000,28,28,1)
Y_train=to_categorical(Y_train)
Y_test=to_categorical(Y_test)
Y_train[0]
print(Y_train[0])
model=Sequential()
#add model layers
#learn image features
model.add(Conv2D(64,kernel_size=3,activation='relu',input_shape=(28,28,1)))
model.add(Conv2D(32,kernel_size=3,activation='relu'))
model.add(Flatten())
model.add(Dense(10,activation='softmax'))
model.compile(optimizer='adam',loss='categorical_crossentropy',metrics=['ac
curacy'])
#train
model.fit(X_train,Y_train,validation_data=(X_test,Y_test),epochs=3)
print(model.predict(X_test[:4]))
#actual results for 1st 4 images in the test set
print(Y_test[:4])
```

OUTPUT:




```
(28, 28)
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
Epoch 1/3
1875/1875 [==============================] - 54s 29ms/step - loss: 0.2302 - accuracy: 0.9516 - val_loss: 0.1051 - val_accuracy: 0.9669
Epoch 2/3
1875/1875 [==============================] - 48s 26ms/step - loss: 0.0699 - accuracy: 0.9786 - val_loss: 0.0744 - val_accuracy: 0.9782
Epoch 3/3
1875/1875 [==============================] - 48s 26ms/step - loss: 0.0501 - accuracy: 0.9841 - val_loss: 0.0953 - val_accuracy: 0.9729
1/1 [==============================] - 0s 54ms/step
```

```
[[1.28678295e-08 1.60042327e-11 4.08504593e-06 1.85614681e-05
  4.91933751e-11 1.62152708e-10 5.10676401e-14 9.99967337e-01
  8.30323734e-06 1.70347710e-06]
 [8.40842915e-07 2.47974286e-09 9.99998212e-01 2.44631337e-09
  5.72847195e-12 3.67103084e-11 9.84666372e-07 7.65504793e-13
  5.66962228e-08 5.52745458e-13]
 [2.66522898e-06 9.97940004e-01 3.96974838e-06 1.76509811e-08
  1.96918356e-03 2.62945696e-06 1.83274778e-06 5.80798678e-06
  7.38622475e-05 6.91403415e-08]
 [9.99975204e-01 2.32659465e-12 2.21851951e-05 8.21377133e-11
  1.66345146e-10 2.17179590e-08 3.32290639e-08 4.78619835e-12
  1.18910428e-08 2.57370039e-06]]
[[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

PRACTICAL 10

AIM: Denoising of images using autoencoder.

CODE:

```
import keras
from keras.datasets import mnist
from keras import layers
import numpy as np
from keras.callbacks import TensorBoard
import matplotlib.pyplot as plt
(X_train,_),(X_test,_)=mnist.load_data()
X_train=X_train.astype('float32')/255.
X_test=X_test.astype('float32')/255.
X_train=np.reshape(X_train,(len(X_train),28,28,1))
X_test=np.reshape(X_test,(len(X_test),28,28,1))
noise_factor=0.5
X_train_noisy=X_train+noise_factor*np.random.normal(loc=0.0,scale=1.0,size=
X_train.shape)
X_test_noisy=X_test+noise_factor*np.random.normal(loc=0.0,scale=1.0,size=X_
test.shape)
X_train_noisy=np.clip(X_train_noisy,0.,1.)
X_test_noisy=np.clip(X_test_noisy,0.,1.)
n=10
plt.figure(figsize=(20,2))
for i in range(1,n+1):
 ax=plt.subplot(1,n,i)
 plt.imshow(X_test_noisy[i].reshape(28,28))
 plt.gray()
 ax.get_xaxis().set_visible(False)
 ax.get_yaxis().set_visible(False)
plt.show()
input_img=keras.Input(shape=(28,28,1))
x=layers.Conv2D(32,(3,3),activation='relu',padding='same')(input_img)
x=layers.MaxPooling2D((2,2),padding='same')(x)
x=layers.Conv2D(32,(3,3),activation='relu',padding='same')(x)
encoded=layers.MaxPooling2D((2,2),padding='same')(x)
```

```
x=layers.Conv2D(32,(3,3),activation='relu',padding='same')(encoded)
x=layers.UpSampling2D((2,2))(x)
x=layers.Conv2D(32,(3,3),activation='relu',padding='same')(x)
x=layers.UpSampling2D((2,2))(x)
decoded=layers.Conv2D(1,(3,3),activation='sigmoid',padding='same')(x)
autoencoder=keras.Model(input_img,decoded)
autoencoder.compile(optimizer='adam',loss='binary_crossentropy')
autoencoder.fit(X_train_noisy,X_train,
 epochs=3,
 batch_size=128,
 shuffle=True,
 validation_data=(X_test_noisy,X_test),
callbacks=[TensorBoard(log_dir='/tmo/tb',histogram_freq=0,write_graph=False
)])
predictions=autoencoder.predict(X_test_noisy)
m=10
plt.figure(figsize=(20,2))
for i in range(1,m+1):
 ax=plt.subplot(1,m,i)
 plt.imshow(predictions[i].reshape(28,28))
 plt.gray()
 ax.get_xaxis().set_visible(False)
 ax.get_yaxis().set_visible(False)
plt.show()
```

OUTPUT:

```
Epoch 1/3
469/469 [==============================] - 48s 101ms/step - loss: 0.1625 - val_loss: 0.1183
Epoch 2/3
469/469 [==============================] - 48s 101ms/step - loss: 0.1152 - val_loss: 0.1102
Epoch 3/3
469/469 [==============================] - 48s 102ms/step - loss: 0.1094 - val_loss: 0.1064
313/313 [==============================] - 2s 6ms/step
```