# University Of Mumbai
# Institute of Distance & Open Learning

## PRACTICAL JOURNAL IN PAPER-III

**DEEP LEARNING**

**SUBMITTED BY**

**POOJA HEMANT KAPADIA**
**APPLICATION ID: 159646**
**SEAT NO: 0103625**

**MASTER OF SCIENCE IN INFORMATION TECHNOLOGY PART-II**
**SEMESTER IV**

**ACADEMIC YEAR**
**2021-2022**

**INSTITUTE OF DISTANCE AND OPEN LEARNING**
**IDOL BUILDING, VIDYANAGARI,**
**SANTACRUZ (EAST), MUMBAI-400 098**

**CONDUCTED AT**
**RIZVI COLLEGE OF ARTS, SCIENCE AND COMMERCE**
**BANDRA (W), MUMBAI 400050**

# University of Mumbai

# Institute of Distance & Open Learning

Dr.Shankar Dayal Sharama Bhavan, Kalina, Vidanagari,
Santacruz (E), Mumbai-400 098.

# Certificate

This is to certify that

Ms. **POOJA KAPADIA** Application ID: **159646**, Seat No: **0103625** from Rizvi College of Arts, Science and Commerce Bandra(W), Mumbai 400 050 has successfully completed all the practical of Paper III titled **DEEP LEARNING** for M.sc (IT) Part II in the academic year 2021-2022.

Section I ________________

Section II ________________

________________________ ________________________

MSc (IT) Co-ordinator, IDOL External Examiner

# PRACTICAL 1

AIM: Performing matrix multiplication and finding Eigen vectors and Eigen values using Tensor Flow.

CODE:

```
In [2]: 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])
        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\nEigen 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:
[[7.1561775 9.106806 ]
 [6.0239253 9.841141 ]]


Eigen Vectors:
[[-0.7802314 -0.625491 ]
 [ 0.625491  -0.7802314]]

Eigen Values:
[ 2.326955 14.670364]
```

# PRACTICAL 2

AIM: Solving XOR problem using deep feed forward network.

CODE:

jupyter PRAC 2 Last Checkpoint: an hour ago (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help

In [3]:
```
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=['accuracy'])
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=1000,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([[-1.0452268 ,  0.38884377],
       [-1.0456628 , -1.0601878 ]], dtype=float32), array([0., 0.], dtype=float32), array([[ 1.1397151 ],
       [-0.28245735]], dtype=float32), array([0.], dtype=float32)]
Epoch 1/1000
1/1 [==============================] - 3s 3s/step - loss: 0.7073 - accuracy: 0.5000
```

```
None
[array([[-1.0452268 ,  0.38884377],
       [-1.0456628 , -1.0601878 ]], dtype=float32), array([0., 0.], dtype=float32), array([[ 1.1397151 ],
       [-0.28245735]], dtype=float32), array([0.], dtype=float32)]
Epoch 1/1000
1/1 [==============================] - 3s 3s/step - loss: 0.7073 - accuracy: 0.5000
Epoch 2/1000
1/1 [==============================] - 0s 7ms/step - loss: 0.7071 - accuracy: 0.2500
Epoch 3/1000
1/1 [==============================] - 0s 9ms/step - loss: 0.7070 - accuracy: 0.2500
Epoch 4/1000
1/1 [==============================] - 0s 14ms/step - loss: 0.7069 - accuracy: 0.2500
Epoch 5/1000
1/1 [==============================] - 0s 14ms/step - loss: 0.7067 - accuracy: 0.2500
Epoch 6/1000
1/1 [==============================] - 0s 15ms/step - loss: 0.7066 - accuracy: 0.2500
Epoch 7/1000
1/1 [==============================] - 0s 16ms/step - loss: 0.7065 - accuracy: 0.2500
Epoch 8/1000
1/1 [==============================] - 0s 11ms/step - loss: 0.7063 - accuracy: 0.2500
```

```
Epoch 503/1000
1/1 [==============================] - 0s 11ms/step - loss: 0.6931 - accuracy: 0.5000
Epoch 504/1000
1/1 [==============================] - 0s 9ms/step - loss: 0.6931 - accuracy: 0.5000
Epoch 505/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.6931 - accuracy: 0.5000
Epoch 506/1000
1/1 [==============================] - 0s 11ms/step - loss: 0.6931 - accuracy: 0.5000
Epoch 507/1000
1/1 [==============================] - 0s 11ms/step - loss: 0.6931 - accuracy: 0.5000
Epoch 508/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.6931 - accuracy: 0.5000
Epoch 509/1000
1/1 [==============================] - 0s 8ms/step - loss: 0.6931 - accuracy: 0.5000
Epoch 510/1000
1/1 [==============================] - 0s 9ms/step - loss: 0.6931 - accuracy: 0.5000
Epoch 511/1000
1/1 [==============================] - 0s 15ms/step - loss: 0.6931 - accuracy: 0.5000
Epoch 512/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.6931 - accuracy: 0.5000
```

```
1/1 [==============================] - 0s 10ms/step - loss: 0.6931 - accuracy: 0.5000
Epoch 996/1000
1/1 [==============================] - 0s 14ms/step - loss: 0.6931 - accuracy: 0.5000
Epoch 997/1000
1/1 [==============================] - 0s 11ms/step - loss: 0.6931 - accuracy: 0.5000
Epoch 998/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.6931 - accuracy: 0.5000
Epoch 999/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.6931 - accuracy: 0.5000
Epoch 1000/1000
1/1 [==============================] - 0s 11ms/step - loss: 0.6931 - accuracy: 0.5000
[array([[-1.0452268,  0.1875426],
       [-1.0456628, -1.0601878]], dtype=float32), array([ 0.        , -0.20130123], dtype=float32), array([[ 1.1397151 ],
       [-0.13495907]], dtype=float32), array([6.0260376e-08], dtype=float32)]
1/1 [==============================] - 0s 107ms/step
[[0.50000006]
 [0.50000006]
 [0.50000006]
 [0.50000006]]
```

# PRACTICAL 3

AIM: Implementing deep neural network for performing classification task.

PROBLEM STATEMENT: The given dataset comprises of health information about diabetic women patient. We need to create deep feed forward network that will classify women suffering from diabetes mellitus as 1.

CODE & OUTPUT:

jupyter PRAC3 Last Checkpoint: 8 minutes ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Run Code

```
In [1]: from numpy import loadtxt
        from keras.models import Sequential
        from keras.layers import Dense
        dataset=loadtxt('diabetes.csv',delimiter=',')
        dataset

Out[1]: array([[  6.   , 148.   ,  72.   , ...,   0.627,  50.   ,   1.   ],
               [  1.   ,  85.   ,  66.   , ...,   0.351,  31.   ,   0.   ],
               [  8.   , 183.   ,  64.   , ...,   0.672,  32.   ,   1.   ],
               ...,
               [  5.   , 121.   ,  72.   , ...,   0.245,  30.   ,   0.   ],
               [  1.   , 126.   ,  60.   , ...,   0.349,  47.   ,   1.   ],
               [  1.   ,  93.   ,  70.   , ...,   0.315,  23.   ,   0.   ]])
```

```
In [2]: X=dataset[:,0:8]
        Y=dataset[:,8]
        X
        Y

Out[2]: array([1., 0., 1., 0., 1., 0., 1., 0., 1., 1., 0., 1., 0., 1., 1., 1., 1.,
               1., 0., 1., 0., 0., 1., 1., 1., 1., 1., 0., 0., 0., 0., 1., 0., 0.,
               0., 0., 0., 1., 1., 1., 0., 0., 0., 1., 0., 1., 0., 0., 1., 0., 0.,
               0., 0., 1., 0., 0., 1., 0., 0., 0., 0., 1., 0., 0., 1., 0., 1., 0.,
               0., 0., 1., 0., 1., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 1.,
               0., 0., 0., 1., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 1., 1., 0.,
               0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 0., 0., 1., 1., 1., 0., 0.,
               0., 1., 0., 0., 0., 1., 1., 0., 0., 1., 1., 1., 1., 1., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 1.,
               0., 1., 1., 0., 0., 0., 1., 0., 0., 0., 0., 1., 1., 0., 0., 0., 0.,
               1., 1., 0., 0., 0., 1., 0., 1., 0., 1., 0., 0., 0., 0., 0., 1., 1.,
               1., 1., 1., 0., 0., 1., 1., 0., 1., 0., 1., 1., 1., 0., 0., 0., 0.,
               0., 0., 1., 1., 0., 1., 0., 0., 0., 1., 1., 1., 1., 0., 1., 1., 1.,
               1., 0., 0., 0., 0., 0., 1., 0., 0., 1., 1., 0., 0., 0., 1., 1., 1.,
               1., 0., 0., 0., 1., 1., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 1.,
               1., 0., 0., 0., 1., 0., 1., 0., 0., 1., 0., 1., 0., 0., 1., 1., 0.,
               0., 0., 0., 0., 1., 0., 0., 0., 1., 0., 0., 1., 1., 0., 0., 1., 0.,
               0., 0., 1., 1., 1., 0., 0., 1., 0., 1., 0., 1., 1., 0., 1., 0., 0.,
               1., 0., 1., 1., 0., 0., 1., 0., 1., 0., 0., 1., 0., 1., 0., 1., 1.,
               1., 0., 0., 1., 0., 1., 0., 0., 0., 1., 0., 0., 0., 0., 1., 1., 1.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 1., 1.,
               1., 0., 1., 1., 0., 0., 1., 0., 0., 1., 0., 0., 1., 1., 0., 0., 0.,
               0., 1., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 0., 0.,
               1., 0., 0., 1., 0., 0., 1., 0., 1., 1., 0., 1., 0., 1., 0., 1., 0.,
               1., 1., 0., 0., 0., 0., 1., 1., 0., 1., 0., 1., 0., 0., 0., 0., 1.,
               1., 0., 1., 0., 1., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 1., 0.,
               0., 1., 1., 1., 0., 0., 1., 0., 0., 1., 0., 0., 0., 1., 0., 0., 1.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.,
               1., 0., 0., 0., 1., 0., 0., 0., 1., 1., 0., 0., 0., 0., 0., 0., 0.,
```

```
In [3]: model=Sequential()
```

```
In [4]: model.add(Dense(12, input_dim=8,activation='relu' ))
        model.add(Dense(8,activation='relu' ))
        model.add(Dense(1,activation='sigmoid' ))
        model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
        model.fit(X,Y,epochs=150,batch_size=4)
```

```
Epoch 1/150
192/192 [==============================] - 1s 2ms/step - loss: 6.7649 - accuracy: 0.5260
Epoch 2/150
192/192 [==============================] - 0s 2ms/step - loss: 1.8355 - accuracy: 0.5690
Epoch 3/150
192/192 [==============================] - 0s 2ms/step - loss: 1.0783 - accuracy: 0.6185
Epoch 4/150
192/192 [==============================] - 0s 2ms/step - loss: 0.8862 - accuracy: 0.6185
Epoch 5/150
192/192 [==============================] - 0s 2ms/step - loss: 0.8770 - accuracy: 0.6211
Epoch 6/150
192/192 [==============================] - 0s 2ms/step - loss: 0.8434 - accuracy: 0.6289
Epoch 7/150
192/192 [==============================] - 1s 3ms/step - loss: 0.7703 - accuracy: 0.6628
Epoch 8/150
192/192 [==============================] - 0s 2ms/step - loss: 0.7334 - accuracy: 0.6589
Epoch 9/150
192/192 [==============================] - 0s 2ms/step - loss: 0.7118 - accuracy: 0.6445
Epoch 10/150
```

```
In [5]: _,Accuracy=model.evaluate(X,Y)

24/24 [==============================] - 0s 3ms/step - loss: 0.4209 - accuracy: 0.8073

In [6]: print("Äccuracy of Model",(Accuracy*100))

Äccuracy of Model 80.72916865348816

In [7]: prediction=model.predict(X)

24/24 [==============================] - 0s 3ms/step

In [8]: exec("for i in range(5):print(X[i].tolist,prediction[i], Y[i])" )

<built-in method tolist of numpy.ndarray object at 0x000001AF6AF8DDB0> [0.69462395] 1.0
<built-in method tolist of numpy.ndarray object at 0x000001AF6AF8DDB0> [0.06601566] 0.0
<built-in method tolist of numpy.ndarray object at 0x000001AF6AF8DDB0> [0.8118147] 1.0
<built-in method tolist of numpy.ndarray object at 0x000001AF6AF8DDB0> [0.10350533] 0.0
<built-in method tolist of numpy.ndarray object at 0x000001AF6AF8DDB0> [0.73300654] 1.0
```

# PRACTICAL 4

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

CODE:

jupyter prac4A Last Checkpoint: Yesterday at 3:20 PM (autosaved)

File Edit View Insert Cell Kernel Widgets Help

```
In [1]: 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)
        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='binary_crossentropy',optimizer='adam')
        model.fit(X,Y,epochs=500)
        Xnew,Yreal=make_blobs(n_samples=3,centers=2,n_features=2,random_state=1)
        Xnew=scalar.transform(Xnew)
        Ynew=model.predict(Xnew)
        for i in range(len(Xnew)):
         print("X=%s,Predicted=%s,Desired=%s"%(Xnew[i],Ynew[i],Yreal[i]))
```

OUTPUT:

```
Epoch 1/500
4/4 [==============================] - 2s 15ms/step - loss: 0.7221
Epoch 2/500
4/4 [==============================] - 0s 3ms/step - loss: 0.7202
Epoch 3/500
4/4 [==============================] - 0s 8ms/step - loss: 0.7188
Epoch 4/500
4/4 [==============================] - 0s 5ms/step - loss: 0.7173
Epoch 5/500
4/4 [==============================] - 0s 5ms/step - loss: 0.7158
Epoch 6/500
4/4 [==============================] - 0s 5ms/step - loss: 0.7145
Epoch 7/500
4/4 [==============================] - 0s 7ms/step - loss: 0.7129
Epoch 8/500
4/4 [==============================] - 0s 5ms/step - loss: 0.7114
Epoch 9/500
4/4 [==============================] - 0s 8ms/step - loss: 0.7101
Epoch 10/500
```

```
4/4 [==============================] - 0s 5ms/step - loss: 0.0033
Epoch 494/500
4/4 [==============================] - 0s 4ms/step - loss: 0.0033
Epoch 495/500
4/4 [==============================] - 0s 5ms/step - loss: 0.0033
Epoch 496/500
4/4 [==============================] - 0s 5ms/step - loss: 0.0032
Epoch 497/500
4/4 [==============================] - 0s 5ms/step - loss: 0.0032
Epoch 498/500
4/4 [==============================] - 0s 3ms/step - loss: 0.0032
Epoch 499/500
4/4 [==============================] - 0s 5ms/step - loss: 0.0032
Epoch 500/500
4/4 [==============================] - 0s 3ms/step - loss: 0.0032
1/1 [==============================] - 0s 149ms/step
X=[0.89337759 0.65864154],Predicted=[0.00576355],Desired=0
X=[0.29097707 0.12978982],Predicted=[0.9970082],Desired=1
X=[0.78082614 0.75391697],Predicted=[0.00585788],Desired=0
```

4B. AIM: Using a deep feed forward network with two hidden layers for performing classification and predicting the probability of class.

CODE:

jupyter PRAC4B Last Checkpoint: 8 minutes ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Run Code

```
In [1]: 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)
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='binary_crossentropy',optimizer='adam')
model.fit(X,Y,epochs=500)
Xnew,Yreal=make_blobs(n_samples=3,centers=2,n_features=2,random_state=1)
Xnew=scalar.transform(Xnew)
Yclass=model.predict_classes(Xnew)
Ynew=model.predict_proba(Xnew)
for i in range(len(Xnew)):
 print("X=%s,Predicted_probability=%s,Predicted_class=%s"%(Xnew[i],Ynew[i],Yclass[i]))
```

OUTPUT:

```
Epoch 491/500
4/4 [==============================] - 0s 5ms/step - loss: 0.0020
Epoch 492/500
4/4 [==============================] - 0s 5ms/step - loss: 0.0020
Epoch 493/500
4/4 [==============================] - 0s 3ms/step - loss: 0.0020
Epoch 494/500
4/4 [==============================] - 0s 5ms/step - loss: 0.0020
Epoch 495/500
4/4 [==============================] - 0s 5ms/step - loss: 0.0020
Epoch 496/500
4/4 [==============================] - 0s 3ms/step - loss: 0.0019
Epoch 497/500
4/4 [==============================] - 0s 3ms/step - loss: 0.0019
Epoch 498/500
4/4 [==============================] - 0s 5ms/step - loss: 0.0019
Epoch 499/500
4/4 [==============================] - 0s 5ms/step - loss: 0.0019
Epoch 500/500
4/4 [==============================] - 0s 3ms/step - loss: 0.0019
```

4C. AIM: Using a deep field forward network with two hidden layers for performing linear regression and predicting values.

CODE:

```
In [1]: from keras.models import Sequential
        from keras.layers import Dense
        from sklearn.datasets import make_regression
        from sklearn.preprocessing import MinMaxScaler

In [2]: 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 125ms/step
X=[0.29466096 0.30317302],Predicted=[0.18164389]
X=[0.39445118 0.79390858],Predicted=[0.76110995]
X=[0.02884127 0.6208843 ],Predicted=[0.39497763]
```

# PRACTICAL 5

A. AIM: Evaluating feed forward deep network for regression using K Fold cross validation

CODE AND OUTPUT:

jupyter PRAC5 Last Checkpoint: 3 minutes ago (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help

Run Code

In [ ]:
```
import pandas as pd
from keras.models import Sequential
from keras.layers import Dense
#from keras.wrappers.scikit_learn import KerasRegressor
from scikeras.wrappers import KerasClassifier, KerasRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
```

In [ ]:
```
dataframe=pd.read_csv("housing (1).csv",delim_whitespace=True,header=None)
dataset=dataframe.values
```

In [ ]:
```
X=dataset[:,0:13]
Y=dataset[:,:13]
```

In [2]:
```
def wider_model(my_param):
    model=Sequential()
    model.add(Dense(15,input_dim=13,kernel_initializer='normal',activation='relu'))
    model.add(Dense(13,kernel_initializer='normal',activation='relu'))
    model.add(Dense(1,kernel_initializer='normal'))
    model.compile(loss='mean_squared_error',optimizer='adam')
    return model
```

In [3]:
```
estimators=[]
estimators.append(('standardize',StandardScaler()))
estimators.append(('mlp',KerasClassifier(model=wider_model,my_param=123)))
pipeline=Pipeline(estimators)
kfold=KFold(n_splits=10)
results=cross_val_score(pipeline,X,Y,cv=kfold)
print("Wider: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

(After changing neuron)

model.add(Dense(20, input_dim=13,kernel_initializer='normal',activation='relu'))

5B. AIM: Evaluating feed forward deep network for multiclass Classification using K Fold cross-validation.

CODE AND OUTPUT:

```
File  Edit  View  Insert  Cell  Kernel  Widgets  Help
Run  Code

In [6]: import pandas
        from keras.models import Sequential
        from keras.layers import Dense
        from keras.wrappers.scikit_learn import KerasClassifier
        from keras.utils import np_utils
        from sklearn.model_selection import cross_val_score
        from sklearn.model_selection import KFold
        from sklearn.preprocessing import LabelEncoder

In [7]: dataset=pandas.read_csv('flower.csv',sep=',',header=1
                        )
        print(dataset)

             1  0  1.1  1.2  4  3  15   25  15.1
        0    2  1    0    0  2  1   3  150    50
        1    3  0    1    0  3  3   1  150    50
        2    4  0    0    1  4  2  16  125    50
        3    5  0    1    0  5  2   2   20    15
        4    6  0    1    0  4  3  12   50    40
        5    7  0    0    0  4  3  13   40    20
        6    8  0    0    1  2  2   7  100    15
        7    9  1    1    0  3  1   4   25    15
        8   10  1    1    0  5  2  14  100    60
        9   11  1    1    1  5  3   8   45    10
        10  12  1    1    1  1  2   9   90    25
        11  13  1    1    0  1  2   6   20    10
        12  14  1    1    1  4  2  11   80    30
        13  15  1    0    0  3  2  10   40    20
        14  16  1    0    0  4  2  18  200    60
        15  17  1    0    0  2  2  17  150    60
        16  18  0    0    1  2  1   5   25    10
```

```
File  Edit  View  Insert  Cell  Kernel  Widgets  Help
Run  Code

In [8]: dataset1=dataset.values
        X=dataset1[:,0:4].astype(float)
        Y=dataset1[:,4]
        print(Y)
        encoder=LabelEncoder()
        encoder.fit(Y)
        encoder_Y=encoder.transform(Y)
        print(encoder_Y)
        dummy_Y=np_utils.to_categorical(encoder_Y)

        [2 3 4 5 4 4 2 3 5 5 1 1 4 3 4 2 2]
        [1 2 3 4 3 3 1 2 4 4 0 0 3 2 3 1 1]

In [9]: print(dummy_Y)
        def baseline_model():
            model=Sequential()
            model.add(Dense(8,input_dim=4,activation='relu'))
            model.add(Dense(3,activation='softmax'))
            model.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
            return model
        estimator=KerasClassifier(build_fn=baseline_model,epochs=100,batch_size=5)
        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))
        #(Changing neuron)
        model=Sequential()
        model.add(Dense(10,input_dim=4,activation='relu'))
```

```
        [[0. 1. 0. 0. 0.]
         [0. 0. 1. 0. 0.]
         [0. 0. 0. 1. 0.]
         [0. 0. 0. 0. 1.]
         [0. 0. 0. 1. 0.]
         [0. 0. 0. 1. 0.]
         [0. 1. 0. 0. 0.]
         [0. 0. 1. 0. 0.]
         [0. 0. 0. 0. 1.]
         [0. 0. 0. 0. 1.]
         [1. 0. 0. 0. 0.]
         [1. 0. 0. 0. 0.]
         [0. 0. 0. 1. 0.]
         [0. 0. 1. 0. 0.]
         [0. 0. 0. 1. 0.]
         [0. 1. 0. 0. 0.]
         [0. 1. 0. 0. 0.]]
        Epoch 1/100

        <ipython-input-9-ec470be275a4>:8: DeprecationWarning: KerasClassifier is deprecated, use Sci-Keras (https://github.com/adrian

In [10]: #(Changing neuron)
         model.add(Dense(10,input_dim=4,activation='relu'))
```

# PRACTICAL 6

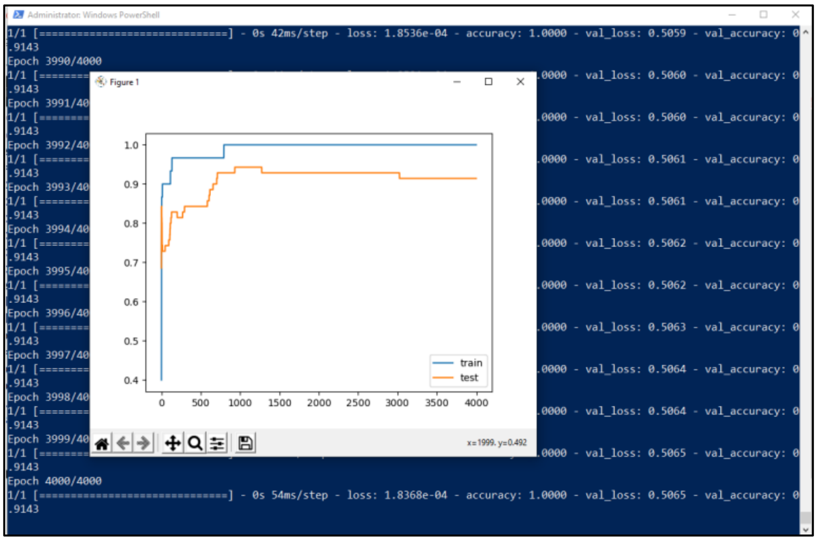
AIM: Implementing regularization to avoid overfitting in binary classification.

CODE & OUTPUT:

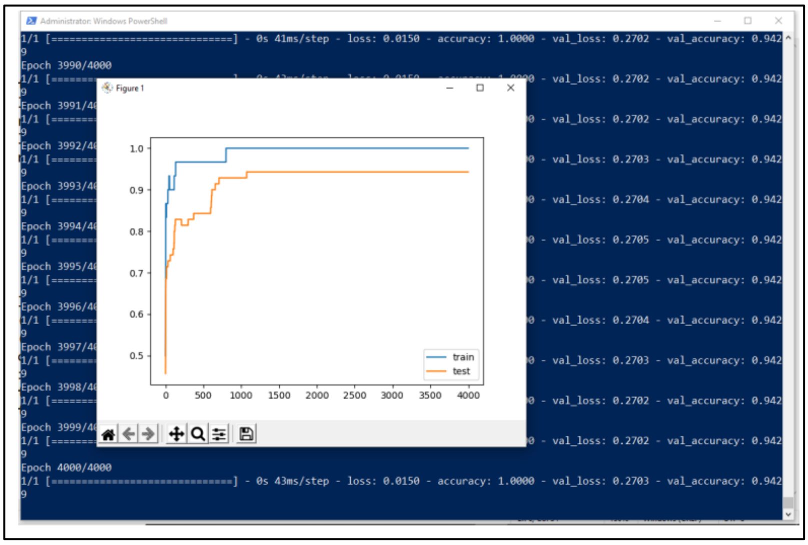
jupyter Untitled6 Last Checkpoint: 06/08/2022 (autosaved)

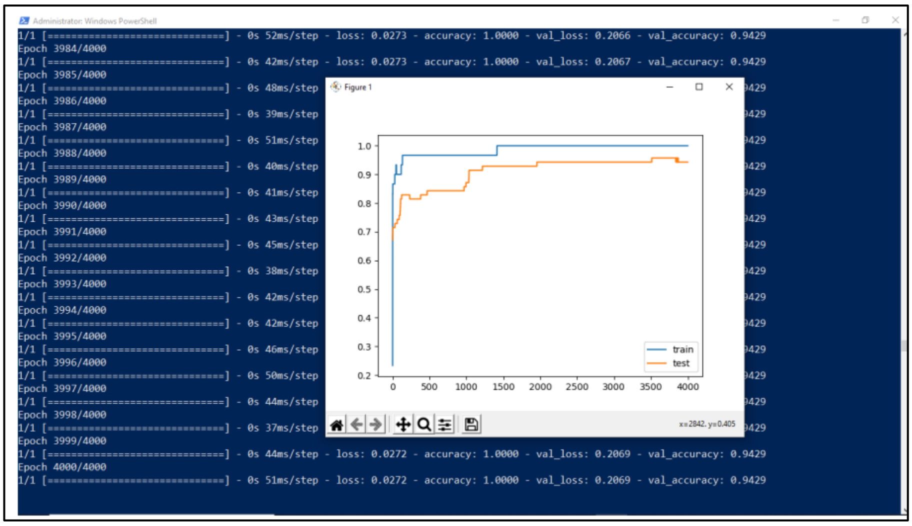
In [ ]:
```python
from matplotlib import pyplot
from sklearn.datasets import make_moons
from keras.models import Sequential
from keras.layers import Dense
X,Y=make_moons(n_samples=100,noise=0.2,random_state=1)
n_train=30
trainX,testX=X[:n_train,:],X[n_train:]
trainY,testY=Y[:n_train],Y[n_train:]
#print(trainX)
#print(trainY)
#print(testX)
#print(testY)
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=['accuracy'])
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()
```

```
Epoch 1/4000
1/1 [==============================] - 1s 900ms/step - loss: 0.6947 - accuracy: 0.4667 - val_loss: 0.6843 - val_accuracy: 0.6
857
Epoch 2/4000
1/1 [==============================] - 0s 47ms/step - loss: 0.6777 - accuracy: 0.8000 - val_loss: 0.6735 - val_accuracy: 0.68
57
Epoch 3/4000
1/1 [==============================] - 0s 57ms/step - loss: 0.6612 - accuracy: 0.8333 - val_loss: 0.6631 - val_accuracy: 0.68
57
Epoch 4/4000
1/1 [==============================] - 0s 80ms/step - loss: 0.6452 - accuracy: 0.8333 - val_loss: 0.6531 - val_accuracy: 0.68
57
Epoch 5/4000
1/1 [==============================] - 0s 44ms/step - loss: 0.6296 - accuracy: 0.8667 - val_loss: 0.6434 - val_accuracy: 0.71
43
Epoch 6/4000
1/1 [==============================] - 0s 47ms/step - loss: 0.6146 - accuracy: 0.8667 - val_loss: 0.6341 - val_accuracy: 0.71
43
Epoch 7/4000
```



```
In [*]: from matplotlib import pyplot
        from sklearn.datasets import make_moons
        from keras.models import Sequential
        from keras.layers import Dense
        from keras.regularizers import 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:]
        #print(trainX)
        #print(trainY)
        #print(testX)
        #print(testY)
        model=Sequential()
        model.add(Dense(500,input_dim=2,activation='relu',kernel_regularizer=l2(0.001)))
        model.add(Dense(1,activation='sigmoid'))
        model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
        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()
```



# Practical No: 7

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

CODE & OUTPUT:

jupyter prac 7 Last Checkpoint: 23 minutes ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Run Code

In [2]:
```
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.csv')
#print(dataset_train)
training_set=dataset_train.iloc[:,1:2].values
```

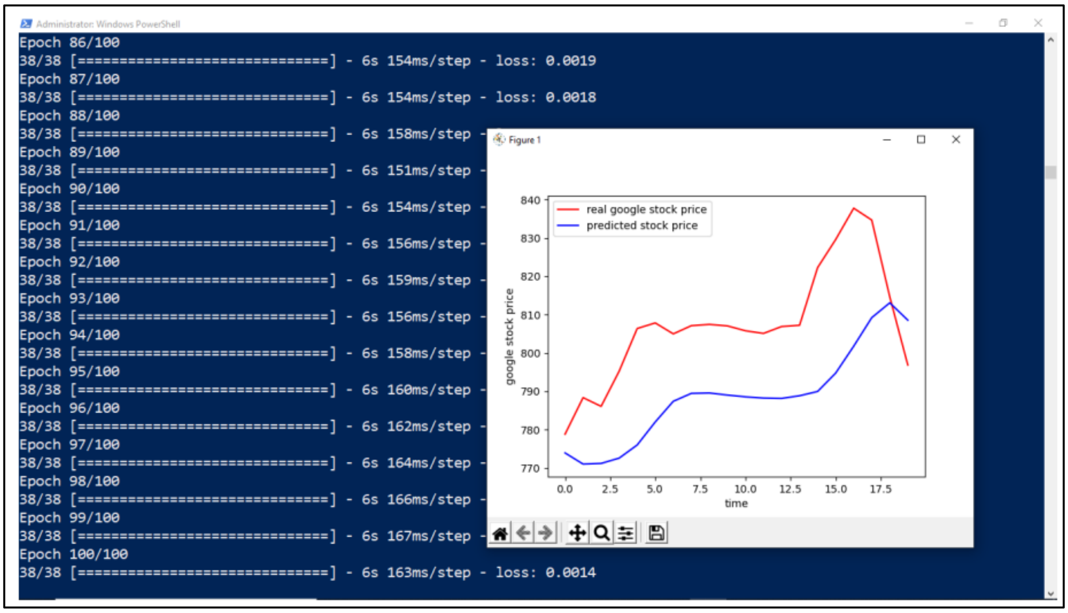
In [3]:
```
#print(training_set)
sc=MinMaxScaler(feature_range=(0,1))
training_set_scaled=sc.fit_transform(training_set)
#print(training_set_scaled)
X_train=[]
Y_train=[]
for i in range(60,1258):
 X_train.append(training_set_scaled[i-60:i,0])
 Y_train.append(training_set_scaled[i,0])
X_train,Y_train=np.array(X_train),np.array(Y_train)
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.shape[1],1)))
regressor.add(Dropout(0.2))
regressor.add(LSTM(units=50,return_sequences=True))
regressor.add(Dropout(0.2))
```

```
regressor=Sequential()
regressor.add(LSTM(units=50,return_sequences=True,input_shape=(X_train.shape[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))
```

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

 [[0.09701243]
  [0.09433366]
  [0.09156187]
```

```
[[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]]]
```

```
In [ ]: 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.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()
```



# PRACTICAL 8

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

CODE:

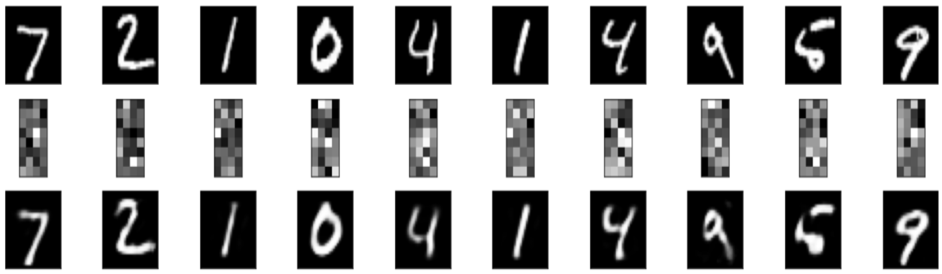
```
In [1]: import keras
        from keras import layers
        from keras.datasets import mnist
        import numpy as np
        encoding_dim=32
        #this is our input image
        input_img=keras.Input(shape=(784,))
        #"encoded" is the encoded representation of the input
        encoded=layers.Dense(encoding_dim, 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,
```

```
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:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 43s 4us/step
(60000, 784)
(10000, 784)
Epoch 1/50
235/235 [==============================] - 4s 12ms/step - loss: 0.2756 - val_loss: 0.1899
Epoch 2/50
235/235 [==============================] - 4s 16ms/step - loss: 0.1716 - val_loss: 0.1545
Epoch 3/50
235/235 [==============================] - 3s 14ms/step - loss: 0.1447 - val_loss: 0.1326
Epoch 4/50
235/235 [==============================] - 3s 11ms/step - loss: 0.1274 - val_loss: 0.1202
Epoch 5/50
235/235 [==============================] - 3s 12ms/step - loss: 0.1173 - val_loss: 0.1121
Epoch 6/50
235/235 [==============================] - 3s 12ms/step - loss: 0.1104 - val_loss: 0.1065
Epoch 7/50
235/235 [==============================] - 3s 15ms/step - loss: 0.1054 - val_loss: 0.1021
Epoch 8/50
```

```
235/235 [==============================] - 3s 12ms/step - loss: 0.0926 - val_loss: 0.0915
Epoch 49/50
235/235 [==============================] - 3s 13ms/step - loss: 0.0926 - val_loss: 0.0915
Epoch 50/50
235/235 [==============================] - 3s 13ms/step - loss: 0.0926 - val_loss: 0.0915
313/313 [==============================] - 1s 2ms/step
313/313 [==============================] - 1s 2ms/step
```



# PRACTICAL 9

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

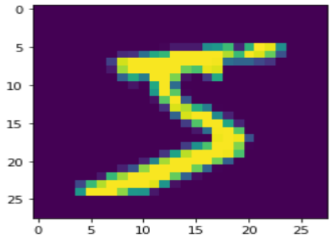
CODE & OUTPUT:

jupyter PRAC9 Last Checkpoint: 3 minutes ago (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help

Run Code

In [1]:
```
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()
```




In [2]:
```
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])
```

```
(28, 28)
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

```
In [3]: 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=['accuracy'])
        #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])

        Epoch 1/3
        1875/1875 [==============================] - 201s 107ms/step - loss: 0.2541 - accuracy: 0.9520 - val_loss: 0.0963 - val_accurac
        y: 0.9714
        Epoch 2/3
        1875/1875 [==============================] - 173s 92ms/step - loss: 0.0684 - accuracy: 0.9796 - val_loss: 0.0816 - val_accurac
        y: 0.9753
        Epoch 3/3
        1875/1875 [==============================] - 180s 96ms/step - loss: 0.0479 - accuracy: 0.9849 - val_loss: 0.1011 - val_accurac
        y: 0.9743
```


```
Epoch 1/3
1875/1875 [==============================] - 201s 107ms/step - loss: 0.2541 - accuracy: 0.9520 - val_loss: 0.0963 - val_accurac
y: 0.9714
Epoch 2/3
1875/1875 [==============================] - 173s 92ms/step - loss: 0.0684 - accuracy: 0.9796 - val_loss: 0.0816 - val_accurac
y: 0.9753
Epoch 3/3
1875/1875 [==============================] - 180s 96ms/step - loss: 0.0479 - accuracy: 0.9849 - val_loss: 0.1011 - val_accurac
y: 0.9743
```


```
1/1 [==============================] - 0s 187ms/step
[[1.76193229e-08 5.17589769e-13 1.27019305e-07 2.36613255e-06
  4.52036629e-13 1.80279767e-11 4.82169312e-15 9.99997497e-01
  2.48748737e-08 9.88452098e-10]
 [2.94664765e-10 6.09432573e-05 9.99938965e-01 9.68984781e-10
  4.67801145e-12 2.16221369e-13 9.06896886e-08 4.22226781e-15
  3.89350374e-09 6.33098090e-15]
 [1.30127512e-06 9.99911308e-01 7.55180736e-07 6.27240269e-08
  1.10290584e-05 1.53752826e-05 8.20892467e-07 6.14862665e-06
  5.28885648e-05 2.15647262e-07]
 [9.99999404e-01 3.44014366e-11 2.22936958e-08 4.20287589e-12
  1.45322955e-11 4.09832479e-09 5.80229084e-07 4.25992158e-11
  4.54949661e-10 4.17157553e-09]]
[[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 auto encoder.

CODE & OUTPUT:

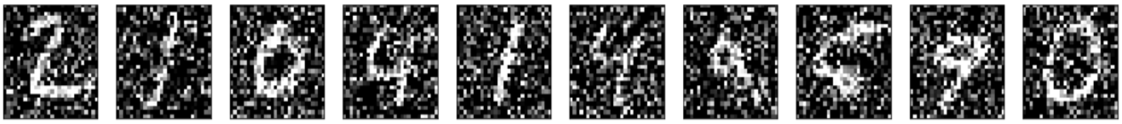
jupyter PRAC10 Last Checkpoint: 12 minutes ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Run Code

In [1]:
```
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.s
X_test_noisy=X_test+noise_factor*np.random.normal(loc=0.0,scale=1.0,size=X_test.shap
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.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()
```



```
Epoch 1/3
469/469 [==============================] - 155s 328ms/step - loss: 0.1734 - val_loss: 0.1182
Epoch 2/3
469/469 [==============================] - 149s 318ms/step - loss: 0.1149 - val_loss: 0.1100
Epoch 3/3
469/469 [==============================] - 146s 311ms/step - loss: 0.1088 - val_loss: 0.1056
313/313 [==============================] - 7s 23ms/step
```