Deep Learning

INDEX

Practical No	Title
1	Performing matrix multiplication and finding eigen vectors and eigen values using TensorFlow
2	Solving XOR problem using deep feed forward network.
3	Implementing deep neural network for performing binary classification task.
4	Using deep feed forward network with two hidden layers for performing multiclass classification and predicting the class.
5	Using a deep feed forward network with two hidden layers for performing linear regression and predicting values.
6	Evaluating feed forward deep network for multiclass Classification using KFold cross-validation.
7	Implementing regularization to avoid overfitting in binary classification.

Practical No: 1

Aim: Performing matrix multiplication and finding eigen vectors and eigen values using TensorFlow.

```
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\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:
[[6.2221375 3.946991 ]
 [4.0869107 6.401558 ]]


Eigen Vectors:
[[-0.7148235 -0.6993049]
 [ 0.6993049 -0.7148235]]

Eigen Values:
[ 2.223952 10.399741]
```

Practical No: 2

Aim: Solving XOR problem using deep feed forward network.

```
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 (36.00 Byte)
Trainable params: 9 (36.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
[array([[-0.47154486,  0.25770223],
       [ 0.5589845 , -1.145811  ]], dtype=float32), array([0., 0.], dtype=float32), array([[-1.3014339],
       [ 1.4125458]], dtype=float32), array([0.], dtype=float32)]
Epoch 1/1000
1/1 [==============================] - 1s 835ms/step - loss: 0.7451 - accuracy: 0.7500
Epoch 2/1000
1/1 [==============================] - 0s 15ms/step - loss: 0.7445 - accuracy: 0.5000
Epoch 3/1000
1/1 [==============================] - 0s 13ms/step - loss: 0.7440 - accuracy: 0.5000
Epoch 4/1000
1/1 [==============================] - 0s 12ms/step - loss: 0.7434 - accuracy: 0.5000
Epoch 5/1000
1/1 [==============================] - 0s 19ms/step - loss: 0.7429 - accuracy: 0.5000
Epoch 6/1000
1/1 [==============================] - 0s 13ms/step - loss: 0.7423 - accuracy: 0.5000
Epoch 7/1000
1/1 [==============================] - 0s 13ms/step - loss: 0.7417 - accuracy: 0.5000
```

```
1/1 [==============================] - 0s 15ms/step - loss: 0.5068 - accuracy: 0.7500
Epoch 994/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.5067 - accuracy: 0.7500
Epoch 995/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.5067 - accuracy: 0.7500
Epoch 996/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.5066 - accuracy: 0.7500
Epoch 997/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.5066 - accuracy: 0.7500
Epoch 998/1000
1/1 [==============================] - 0s 11ms/step - loss: 0.5065 - accuracy: 0.7500
Epoch 999/1000
1/1 [==============================] - 0s 15ms/step - loss: 0.5065 - accuracy: 0.7500
Epoch 1000/1000
1/1 [==============================] - 0s 14ms/step - loss: 0.5064 - accuracy: 0.7500
[array([[-0.32912844,  1.1115862 ],
       [ 0.26617038, -1.145811  ]], dtype=float32), array([-2.9281399e-01, -2.3616558e-04], dtype=float32), array([[-1.1336356],
       [ 2.4413865]], dtype=float32), array([-0.5318429], dtype=float32)]
1/1 [==============================] - 0s 147ms/step
[[0.37008715]
 [0.37008715]
 [0.898566  ]
 [0.37008715]]
```

Practical No: 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

```
import numpy as np
from keras.layers import Dense
from keras.models import Sequential
dataset =np.loadtxt('pima-indians-diabetes.csv',delimiter=',')
X=dataset[:,0:8]
Y=dataset[:,8]
### Creating model:
model=Sequential()
model.add(Dense(units=12,activation='relu',input_dim=8))
model.add(Dense(units=8,activation='relu'))
model.add(Dense(1,activation='sigmoid'))

### Compiling and fitting model:
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
model.fit(X,Y,epochs=150,batch_size=10)
_,accuracy=model.evaluate(X,Y)
print('Accuracy of model is',(accuracy*100))

### Using model for prediction class:
prediction=model.predict(X)
for i in range(5):print(X[i].tolist(),prediction[i],Y[i])
```

```
Epoch 1/150
77/77 [==============================] - 1s 2ms/step - loss: 14.6357 - accuracy: 0.6562
Epoch 2/150
77/77 [==============================] - 0s 2ms/step - loss: 1.3226 - accuracy: 0.6016
Epoch 3/150
77/77 [==============================] - 0s 2ms/step - loss: 0.9153 - accuracy: 0.6042
Epoch 4/150
77/77 [==============================] - 0s 2ms/step - loss: 0.8085 - accuracy: 0.6471
Epoch 5/150
77/77 [==============================] - 0s 2ms/step - loss: 0.7688 - accuracy: 0.6289
Epoch 6/150
77/77 [==============================] - 0s 2ms/step - loss: 0.7425 - accuracy: 0.6406
Epoch 7/150
77/77 [==============================] - 0s 2ms/step - loss: 0.7239 - accuracy: 0.6432
Epoch 8/150
77/77 [==============================] - 0s 2ms/step - loss: 0.7028 - accuracy: 0.6602
Epoch 9/150
77/77 [==============================] - 0s 2ms/step - loss: 0.7093 - accuracy: 0.6510
Epoch 10/150
77/77 [==============================] - 0s 2ms/step - loss: 0.6897 - accuracy: 0.6432
Epoch 11/150
77/77 [==============================] - 0s 2ms/step - loss: 0.6981 - accuracy: 0.6536
Epoch 12/150
77/77 [==============================] - 0s 2ms/step - loss: 0.6597 - accuracy: 0.6484
Epoch 13/150
77/77 [==============================] - 0s 2ms/step - loss: 0.6620 - accuracy: 0.6589
Epoch 14/150
```

Evaluating the accuracy:

```
Epoch 149/150
77/77 [==============================] - 0s 2ms/step - loss: 0.4919 - accuracy: 0.7656
Epoch 150/150
77/77 [==============================] - 0s 2ms/step - loss: 0.4934 - accuracy: 0.7500
24/24 [==============================] - 0s 2ms/step - loss: 0.4777 - accuracy: 0.7747
Accuracy of model is 77.47395634651184
24/24 [==============================] - 0s 2ms/step
```

Using model for prediction class:

```
[6.0, 148.0, 72.0, 35.0, 0.0, 33.6, 0.627, 50.0] [0.85096246] 1.0
[1.0, 85.0, 66.0, 29.0, 0.0, 26.6, 0.351, 31.0] [0.16352274] 0.0
[8.0, 183.0, 64.0, 0.0, 0.0, 23.3, 0.672, 32.0] [0.7912155] 1.0
[1.0, 89.0, 66.0, 23.0, 94.0, 28.1, 0.167, 21.0] [0.06331065] 0.0
[0.0, 137.0, 40.0, 35.0, 168.0, 43.1, 2.288, 33.0] [0.67320794] 1.0
```

Practical No: 4

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

```
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 489/500
4/4 [==============================] - 0s 5ms/step - loss: 0.1281
Epoch 490/500
4/4 [==============================] - 0s 6ms/step - loss: 0.1278
Epoch 491/500
4/4 [==============================] - 0s 6ms/step - loss: 0.1274
Epoch 492/500
4/4 [==============================] - 0s 6ms/step - loss: 0.1271
Epoch 493/500
4/4 [==============================] - 0s 4ms/step - loss: 0.1267
Epoch 494/500
4/4 [==============================] - 0s 5ms/step - loss: 0.1264
Epoch 495/500
4/4 [==============================] - 0s 4ms/step - loss: 0.1260
Epoch 496/500
4/4 [==============================] - 0s 4ms/step - loss: 0.1257
Epoch 497/500
4/4 [==============================] - 0s 4ms/step - loss: 0.1253
Epoch 498/500
4/4 [==============================] - 0s 6ms/step - loss: 0.1250
Epoch 499/500
4/4 [==============================] - 0s 6ms/step - loss: 0.1246
Epoch 500/500
4/4 [==============================] - 0s 5ms/step - loss: 0.1243
1/1 [==============================] - 0s 116ms/step
X=[0.89337759 0.65864154],Predicted=[0.20667735],Desired=0
X=[0.29097707 0.12978982],Predicted=[0.96429235],Desired=1
X=[0.78082614 0.75391697],Predicted=[0.20667735],Desired=0
```


```
X=[0.89337759 0.65864154],Predicted=[0.20667735],Desired=0
X=[0.29097707 0.12978982],Predicted=[0.96429235],Desired=1
X=[0.78082614 0.75391697],Predicted=[0.20667735],Desired=0
```

Practical No: 5

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

```
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 103ms/step
X=[0.29466096 0.30317302],Predicted=[0.18805341]
X=[0.39445118 0.79390858],Predicted=[0.7581309]
X=[0.02884127 0.6208843 ],Predicted=[0.397914]
```

Practical No: 6

Aim: Evaluating feed forward deep network for multiclass Classification using KFold cross-validation.

```
#loading libraries

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

#loading dataset

df=pandas.read_csv('Flower.csv',header=None)

print(df)

#splitting dataset into input and output variables

X = df.iloc[:,0:4].astype(float)    y=df.iloc[:,4]

#print(X)

#print(y)

#encoding string output into numeric output

encoder=LabelEncoder()

encoder.fit(y)

encoded_y=encoder.transform(y)

print(encoded_y)

dummy_Y=np_utils.to_categorical(encoded_y)

print(dummy_Y)

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=baseline_model()
estimator.fit(X,dummy_Y,epochs=100,shuffle=True)
action=estimator.predict(X)
for i in range(25):
    print(dummy_Y[i])
print('^^^^^^^^^^^^^^^^^^^^^^')
for i in range(25):
print(action[i])
```

OUTPUT:

```
       0    1    2    3               4
0    5.1  3.5  1.4  0.2     Iris-setosa
1    4.9  3.0  1.4  0.2     Iris-setosa
2    4.7  3.2  1.3  0.2     Iris-setosa
3    4.6  3.1  1.5  0.2     Iris-setosa
4    5.0  3.6  1.4  0.2     Iris-setosa
..   ...  ...  ...  ...             ...
145  6.7  3.0  5.2  2.3  Iris-virginica
146  6.3  2.5  5.0  1.9  Iris-virginica
147  6.5  3.0  5.2  2.0  Iris-virginica
148  6.2  3.4  5.4  2.3  Iris-virginica
149  5.9  3.0  5.1  1.8  Iris-virginica

[150 rows x 5 columns]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
[[1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
```

```
Epoch 98/100
5/5 [==============================] - 0s 0s/step - loss: 0.3899 - accuracy: 0.9313
Epoch 99/100
5/5 [==============================] - 0s 0s/step - loss: 0.3896 - accuracy: 0.9230
Epoch 100/100
5/5 [==============================] - 0s 0s/step - loss: 0.3682 - accuracy: 0.9361
[1. 0. 0.]
[1. 0. 0.]
[1. 0. 0.]
[1. 0. 0.]
[1. 0. 0.]
[1. 0. 0.]
[1. 0. 0.]
[1. 0. 0.]
[1. 0. 0.]
[1. 0. 0.]
[1. 0. 0.]
[1. 0. 0.]
[1. 0. 0.]
[1. 0. 0.]
[1. 0. 0.]
[1. 0. 0.]
[1. 0. 0.]
[1. 0. 0.]
[1. 0. 0.]
[1. 0. 0.]
[1. 0. 0.]
```

```
^^^^^^^^^^^^^^^^^^^^^^
[0.9145307  0.08423453 0.00123477]
[0.88751584 0.1100563  0.00242792]
[0.8999843  0.09803853 0.00197715]
[0.858188   0.13759544 0.00421653]
[0.9138275  0.08489472 0.00127787]
[0.8994011  0.09916449 0.0014343 ]
[0.8872866  0.11023647 0.00247695]
[0.89339536 0.10458492 0.00201967]
[0.8545533  0.14064151 0.00480518]
[0.87742513 0.11963753 0.00293737]
[0.9203753  0.07866727 0.00095734]
[0.8665611  0.1300417  0.00339716]
[0.88403696 0.11323617 0.0027269 ]
[0.9008803  0.09682965 0.00229002]
[9.5539063e-01 4.4350266e-02 2.5906262e-04]
[9.4327897e-01 5.6333560e-02 3.8754733e-04]
[9.3672138e-01 6.2714875e-02 5.6370755e-04]
[0.91191673 0.08680107 0.00128225]
[0.9100969  0.08882014 0.00108295]
[0.91078293 0.08794734 0.00126965]
[0.8827079  0.11510085 0.00219123]
[0.9060573  0.09255142 0.00139134]
[9.3434143e-01 6.4821333e-02 8.3730859e-04]
[0.85551745 0.14102885 0.00345369]
[0.80272377 0.1895675  0.00770868]
```

Code 2:

```
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
dataset=pandas.read_csv("Flower.csv",header=None)
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)
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))
```

```
3/3 [==============================] - 0s 2ms/step - loss: 0.2491 - accuracy: 0.9333
Baseline: 96.00% (4.42%)
```

(Changing neuron)

```
model.add(Dense(10,input_dim=4,activation='relu'))
```

```
3/3 [==============================] - 0s 999us/step - loss: 0.1436 - accuracy: 1.0000
Baseline: 98.67% (2.67%)
```

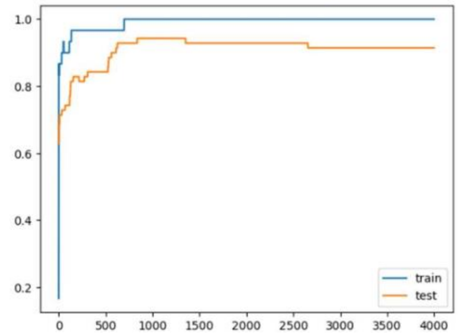
Practical No : 7

Aim: implementing regularization to avoid overfitting in binary classification.

```
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()
```

OUTPUT:

```
1/1 [==============================] - 0s 69ms/step - loss: 1.8653e-04 - accuracy: 1.0000 - val_loss: 0.4948 - val_accuracy: 0.9143
Epoch 3991/4000
1/1 [==============================] - 0s 65ms/step - loss: 1.8638e-04 - accuracy: 1.0000 - val_loss: 0.4948 - val_accuracy: 0.9143
Epoch 3992/4000
1/1 [==============================] - 0s 67ms/step - loss: 1.8623e-04 - accuracy: 1.0000 - val_loss: 0.4949 - val_accuracy: 0.9143
Epoch 3993/4000
1/1 [==============================] - 0s 59ms/step - loss: 1.8608e-04 - accuracy: 1.0000 - val_loss: 0.4949 - val_accuracy: 0.9143
Epoch 3994/4000
1/1 [==============================] - 0s 59ms/step - loss: 1.8593e-04 - accuracy: 1.0000 - val_loss: 0.4950 - val_accuracy: 0.9143
Epoch 3995/4000
1/1 [==============================] - 0s 74ms/step - loss: 1.8579e-04 - accuracy: 1.0000 - val_loss: 0.4950 - val_accuracy: 0.9143
Epoch 3996/4000
1/1 [==============================] - 0s 66ms/step - loss: 1.8564e-04 - accuracy: 1.0000 - val_loss: 0.4951 - val_accuracy: 0.9143
Epoch 3997/4000
1/1 [==============================] - 0s 68ms/step - loss: 1.8550e-04 - accuracy: 1.0000 - val_loss: 0.4951 - val_accuracy: 0.9143
Epoch 3998/4000
1/1 [==============================] - 0s 72ms/step - loss: 1.8535e-04 - accuracy: 1.0000 - val_loss: 0.4952 - val_accuracy: 0.9143
Epoch 3999/4000
1/1 [==============================] - 0s 52ms/step - loss: 1.8521e-04 - accuracy: 1.0000 - val_loss: 0.4952 - val_accuracy: 0.9143
Epoch 4000/4000
1/1 [==============================] - 0s 56ms/step - loss: 1.8506e-04 - accuracy: 1.0000 - val_loss: 0.4953 - val_accuracy: 0.9143
```



The above code and resultant graph demonstrate overfitting with accuracy of testing data less than accuracy of training data also the accuracy of testing data increases once and then start decreases gradually.to solve this problem we can use regularization

Hence, we will add two lines in the above code as highlighted below to implement l2 regularization with alpha=0.001

```
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)

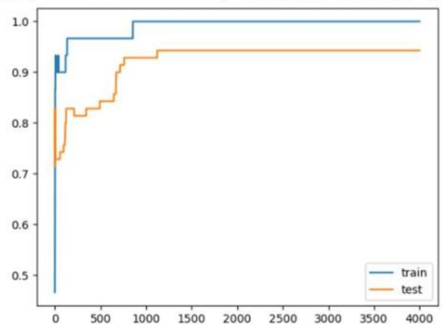
pyplot.plot(history.history['accuracy'],label='train')

pyplot.plot(history.history['val_accuracy'],label='test')

pyplot.legend()

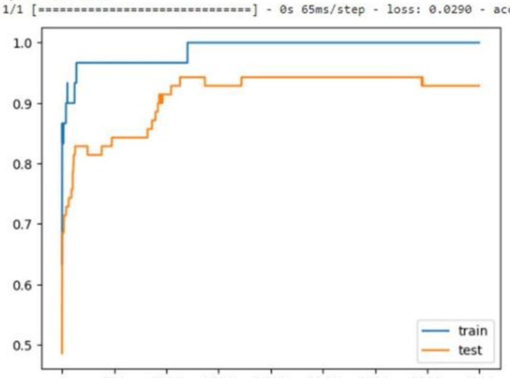
pyplot.show()

```
Epoch 3990/4000
1/1 [==============================] - 0s 59ms/step - loss: 0.0157 - accuracy: 1.0000 - val_loss: 0.2736 - val_accuracy: 0.9429
Epoch 3991/4000
1/1 [==============================] - 0s 74ms/step - loss: 0.0157 - accuracy: 1.0000 - val_loss: 0.2735 - val_accuracy: 0.9429
Epoch 3992/4000
1/1 [==============================] - 0s 66ms/step - loss: 0.0157 - accuracy: 1.0000 - val_loss: 0.2735 - val_accuracy: 0.9429
Epoch 3993/4000
1/1 [==============================] - 0s 78ms/step - loss: 0.0157 - accuracy: 1.0000 - val_loss: 0.2736 - val_accuracy: 0.9429
Epoch 3994/4000
1/1 [==============================] - 0s 74ms/step - loss: 0.0157 - accuracy: 1.0000 - val_loss: 0.2736 - val_accuracy: 0.9429
Epoch 3995/4000
1/1 [==============================] - 0s 76ms/step - loss: 0.0157 - accuracy: 1.0000 - val_loss: 0.2735 - val_accuracy: 0.9429
Epoch 3996/4000
1/1 [==============================] - 0s 72ms/step - loss: 0.0157 - accuracy: 1.0000 - val_loss: 0.2736 - val_accuracy: 0.9429
Epoch 3997/4000
1/1 [==============================] - 0s 57ms/step - loss: 0.0157 - accuracy: 1.0000 - val_loss: 0.2736 - val_accuracy: 0.9429
Epoch 3998/4000
1/1 [==============================] - 0s 77ms/step - loss: 0.0157 - accuracy: 1.0000 - val_loss: 0.2735 - val_accuracy: 0.9429
Epoch 3999/4000
1/1 [==============================] - 0s 65ms/step - loss: 0.0157 - accuracy: 1.0000 - val_loss: 0.2735 - val_accuracy: 0.9429
Epoch 4000/4000
1/1 [==============================] - 0s 74ms/step - loss: 0.0157 - accuracy: 1.0000 - val_loss: 0.2736 - val_accuracy: 0.9429
```



By replacing l2 regularizer with l1 regularizer at the same learning rate 0.001 we get the following output.

```
Epoch 3992/4000
1/1 [==============================] - 0s 71ms/step - loss: 0.0291 - accuracy: 1.0000 - val_loss: 0.2236 - val_accuracy: 0.9286
Epoch 3993/4000
1/1 [==============================] - 0s 66ms/step - loss: 0.0291 - accuracy: 1.0000 - val_loss: 0.2236 - val_accuracy: 0.9286
Epoch 3994/4000
1/1 [==============================] - 0s 92ms/step - loss: 0.0291 - accuracy: 1.0000 - val_loss: 0.2235 - val_accuracy: 0.9286
Epoch 3995/4000
1/1 [==============================] - 0s 74ms/step - loss: 0.0290 - accuracy: 1.0000 - val_loss: 0.2235 - val_accuracy: 0.9286
Epoch 3996/4000
1/1 [==============================] - 0s 83ms/step - loss: 0.0290 - accuracy: 1.0000 - val_loss: 0.2234 - val_accuracy: 0.9286
Epoch 3997/4000
1/1 [==============================] - 0s 72ms/step - loss: 0.0290 - accuracy: 1.0000 - val_loss: 0.2234 - val_accuracy: 0.9286
Epoch 3998/4000
1/1 [==============================] - 0s 75ms/step - loss: 0.0290 - accuracy: 1.0000 - val_loss: 0.2234 - val_accuracy: 0.9286
Epoch 3999/4000
1/1 [==============================] - 0s 73ms/step - loss: 0.0290 - accuracy: 1.0000 - val_loss: 0.2235 - val_accuracy: 0.9286
Epoch 4000/4000
1/1 [==============================] - 0s 65ms/step - loss: 0.0290 - accuracy: 1.0000 - val_loss: 0.2235 - val_accuracy: 0.9286
```



By applying l1 and l2 regularizer we can observe the following changes in accuracy of both trainig and testing data. The changes in code are also highlighted.

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:]

#print(trainX)
#print(trainY)
#print(testX) #print(testY)

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=['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()

OUTPUT:

```
:poch 3992/4000
l/1 [==============================] - 0s 76ms/step - loss: 0.0431 - accuracy: 1.0000 - val_loss: 0.2088 - val_accuracy: 0.9571
:poch 3993/4000
l/1 [==============================] - 0s 90ms/step - loss: 0.0431 - accuracy: 1.0000 - val_loss: 0.2088 - val_accuracy: 0.9571
:poch 3994/4000
l/1 [==============================] - 0s 85ms/step - loss: 0.0431 - accuracy: 1.0000 - val_loss: 0.2087 - val_accuracy: 0.9571
:poch 3995/4000
l/1 [==============================] - 0s 86ms/step - loss: 0.0431 - accuracy: 1.0000 - val_loss: 0.2087 - val_accuracy: 0.9571
:poch 3996/4000
l/1 [==============================] - 0s 63ms/step - loss: 0.0431 - accuracy: 1.0000 - val_loss: 0.2086 - val_accuracy: 0.9571
:poch 3997/4000
l/1 [==============================] - 0s 91ms/step - loss: 0.0431 - accuracy: 1.0000 - val_loss: 0.2085 - val_accuracy: 0.9571
:poch 3998/4000
l/1 [==============================] - 0s 80ms/step - loss: 0.0431 - accuracy: 1.0000 - val_loss: 0.2084 - val_accuracy: 0.9571
:poch 3999/4000
l/1 [==============================] - 0s 64ms/step - loss: 0.0431 - accuracy: 1.0000 - val_loss: 0.2083 - val_accuracy: 0.9571
:poch 4000/4000
l/1 [==============================] - 0s 77ms/step - loss: 0.0431 - accuracy: 1.0000 - val_loss: 0.2083 - val_accuracy: 0.9571
```