Deep Learning

Index

Practical No: 1

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\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:
[[8.920948  9.958236 ]
 [7.6198754 5.5510497]]


Eigen Vectors:
[[-0.62613505  0.7797147 ]
 [ 0.7797147   0.62613505]]

Eigen Values:
[-0.56794643 15.039947  ]
```

Practical No: 2

Aim: Solving XOR problem using deep feed forward network.

Code:

```
import numpy as np


def unitStep(v):
  if v >= 0:
    return 1
  else:
    return 0
def perceptronModel(x, w, b):
  v = np.dot(w, x) + b
  y = unitStep(v)
  return y
def NOT_logicFunction(x):
  wNOT = -1
  bNOT = 0.5
  return perceptronModel(x, wNOT, bNOT)
def AND_logicFunction(x):
  w = np.array([1, 1])
  bAND = -1.5
  return perceptronModel(x, w, bAND)
def OR_logicFunction(x):
  w = np.array([1, 1])
  bOR = -0.5
```

```
   return perceptronModel(x, w, bOR)
def XOR_logicFunction(x):
  y1 = AND_logicFunction(x)
  y2 = OR_logicFunction(x)
  y3 = NOT_logicFunction(y1)
  final_x = np.array([y2, y3])
  finalOutput = AND_logicFunction(final_x)
  y3 = NOT_logicFunction(y1)
  return finalOutput


test1 = np.array([0, 1])
test2 = np.array([1, 1])
test3 = np.array([0, 0])
test4 = np.array([1, 0])


print("XOR({}, {}) = {}".format(0, 1, XOR_logicFunction(test1)))
print("XOR({}, {}) = {}".format(1, 1, XOR_logicFunction(test2)))
print("XOR({}, {}) = {}".format(0, 0, XOR_logicFunction(test3)))
print("XOR({}, {}) = {}".format(1, 0, XOR_logicFunction(test4)))
```

Output:

```
XOR(0, 1) = 1
XOR(1, 1) = 0
XOR(0, 0) = 0
XOR(1, 0) = 1
```

Practical No: 3

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

Code:

```
import pandas as pd
from keras.models import Sequential
from keras.layers import Dense
from scikeras.wrappers import KerasClassifier
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# load dataset
dataframe = pd.read_csv("//content//sonar.all-data", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
# baseline model
def create_baseline():
```

```
    # create model
    model = Sequential()
    model.add(Dense(60, input_dim=60, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

# evaluate model with standardized dataset
estimator = KerasClassifier(build_fn=create_baseline, epochs=100, batch_size=5, verbose=0)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(estimator, X, encoded_Y, cv=kfold)
print("Baseline: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
# evaluate baseline model with standardized dataset
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_baseline, epochs=100, batch_size=5,
verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Standardized: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
def create_smaller():
    # create model
    model = Sequential()
    model.add(Dense(30, input_dim=60, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
```

```
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_smaller, epochs=100, batch_size=5,
verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Smaller: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
# larger model
def create_larger():
    # create model
    model = Sequential()
    model.add(Dense(60, input_dim=60, activation='relu'))
    model.add(Dense(30, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_larger, epochs=100, batch_size=5,
verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
```

```
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)

print("Larger: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Output:

```
Baseline: 82.69% (9.24%)
Standardized: 87.52% (7.73%)
Smaller: 83.12% (5.11%)
Larger: 86.10% (7.49%)
```

Practical No: 4

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

Code:

```
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.summary()
model.fit(X,Y,epochs=200)

Xnew,Yreal=make_blobs(n_samples=3,centers=2,n_features=2,random_state=1)
Xnew=scalar.transform(Xnew)
Yclass=model.predict(Xnew)
```

```
import numpy as np

def predict_prob(number):

  return [number[0],1-number[0]]

y_prob = np.array(list(map(predict_prob, model.predict(Xnew))))

y_prob

for i in range(len(Xnew)):

 print("X=%s,Predicted_probability=%s,Predicted_class=%s"%(Xnew[i],y_prob[i],Yclass[i]))

 #second way

predict_prob=model.predict([Xnew])

predict_classes=np.argmax(predict_prob,axis=1)

predict_classes
```

Output:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 4)                 12

 dense_1 (Dense)             (None, 4)                 20

 dense_2 (Dense)             (None, 1)                 5

=================================================================
Total params: 37 (148.00 Byte)
Trainable params: 37 (148.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________

Epoch 1/200
4/4 [==============================] - 1s 5ms/step - loss: 0.6918

Epoch 200/200
4/4 [==============================] - 0s 4ms/step - loss: 0.1440

1/1 [==============================] - 0s 233ms/step
```

```
1/1 [==============================] - 0s 19ms/step
array([[0.0567151 , 0.9432849 ],
       [0.78821236, 0.21178764],
       [0.05144136, 0.94855864]])

X=[0.89337759 0.65864154],Predicted_probability=[0.0567151 0.9432849],Predicted_class=[0.0567151]
X=[0.29097707 0.12978982],Predicted_probability=[0.78821236 0.21178764],Predicted_class=[0.78821236]
X=[0.78082614 0.75391697],Predicted_probability=[0.05144136 0.94855864],Predicted_class=[0.05144136]

1/1 [==============================] - 0s 53ms/step
array([0, 0, 0])
```

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

Code:

```
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 [==============================] - 4s 12ms/step - loss: 0.6188
Epoch 2/500
4/4 [==============================] - 0s 7ms/step - loss: 0.6145
Epoch 3/500
4/4 [==============================] - 0s 13ms/step - loss: 0.6101
Epoch 4/500
4/4 [==============================] - 0s 6ms/step - loss: 0.6058

Epoch 495/500
4/4 [==============================] - 0s 4ms/step - loss: 0.0925
Epoch 496/500
4/4 [==============================] - 0s 4ms/step - loss: 0.0923
Epoch 497/500
4/4 [==============================] - 0s 4ms/step - loss: 0.0920
Epoch 498/500
4/4 [==============================] - 0s 4ms/step - loss: 0.0918
Epoch 499/500
4/4 [==============================] - 0s 4ms/step - loss: 0.0916
Epoch 500/500
4/4 [==============================] - 0s 4ms/step - loss: 0.0914
1/1 [==============================] - 0s 84ms/step
X=[0.89337759 0.65864154],Predicted=[0.00614816],Desired=0
X=[0.29097707 0.12978982],Predicted=[0.8343555],Desired=1
X=[0.78082614 0.75391697],Predicted=[0.00339534],Desired=0
```

Aim: C] 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 54ms/step
X=[0.29466096 0.30317302],Predicted=[0.18238887]
X=[0.39445118 0.79390858],Predicted=[0.7612629]
X=[0.02884127 0.6208843 ],Predicted=[0.3965788]
```

Practical No: 5

Aim: Evaluating feed forward deep network for regression using KFold cross validation.

Code:

```
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
from tensorflow.keras.losses import sparse_categorical_crossentropy
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
# Model configuration
batch_size = 50
img_width, img_height, img_num_channels = 32, 32, 3
loss_function = sparse_categorical_crossentropy
no_classes = 100
no_epochs = 10   # you can increase it to 20,50,70, 100
optimizer = Adam()
verbosity = 1
# Load CIFAR-10 data
(input_train, target_train), (input_test, target_test) = cifar10.load_data()
# Determine shape of the data
input_shape = (img_width, img_height, img_num_channels)
# Parse numbers as floats
input_train = input_train.astype('float32')
input_test = input_test.astype('float32')
```

```
# Normalize data
input_train = input_train / 255
input_test = input_test / 255
# Create the model
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(no_classes, activation='softmax'))
model.summary()
# Compile the model
model.compile(loss=loss_function, optimizer=optimizer,metrics=['accuracy'])
# Fit data to model (this will take little time to train)
history = model.fit(input_train, target_train, batch_size=batch_size, epochs=no_epochs,
verbose=verbosity)
# Generate generalization metrics
score = model.evaluate(input_test, target_test, verbose=0)
print(f'Test loss: {score[0]} / Test accuracy: {score[1]}')
# Visualize history
# Plot history: Loss
plt.plot(history.history['loss'])
plt.title('Validation loss history')
plt.ylabel('Loss value')
```

```
plt.xlabel('No. epoch')

plt.show()

# Plot history: Accuracy

plt.plot(history.history['accuracy'])

plt.title('Validation accuracy history')

plt.ylabel('Accuracy value (%)')

plt.xlabel('No. epoch')

plt.show()


# By Adding k fold cross validation

from tensorflow.keras.datasets import cifar10

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D

from tensorflow.keras.losses import sparse_categorical_crossentropy

from tensorflow.keras.optimizers import Adam

from sklearn.model_selection import KFold

KFold

import numpy as np

# Model configuration

batch_size = 50

img_width, img_height, img_num_channels = 32, 32, 3

loss_function = sparse_categorical_crossentropy

no_classes = 100

no_epochs = 10

optimizer = Adam()

verbosity = 1

num_folds = 5
```

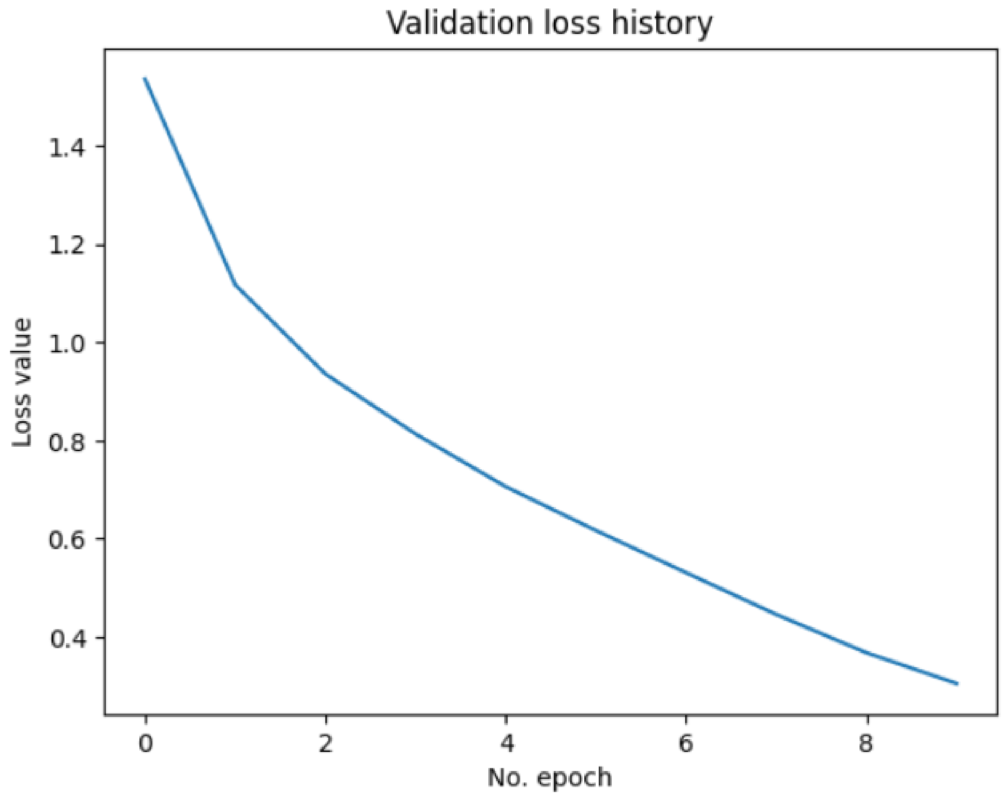
```
# Load CIFAR-10 data
(input_train, target_train), (input_test, target_test) = cifar10.load_data()
# Determine shape of the data
input_shape = (img_width, img_height, img_num_channels)
# Parse numbers as floats
input_train = input_train.astype('float32')
input_test = input_test.astype('float32')
# Normalize data
input_train = input_train / 255
input_test = input_test / 255
# Define per-fold score containers
acc_per_fold = []
loss_per_fold = []
# Merge inputs and targets
inputs = np.concatenate((input_train, input_test), axis=0)
targets = np.concatenate((target_train, target_test), axis=0)
# Define the K-fold Cross Validator
kfold = KFold(n_splits=num_folds, shuffle=True)

# K-fold Cross Validation model evaluation
fold_no = 1
for train, test in kfold.split(inputs, targets):
  # Define the model architecture
  model = Sequential()
  model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
```

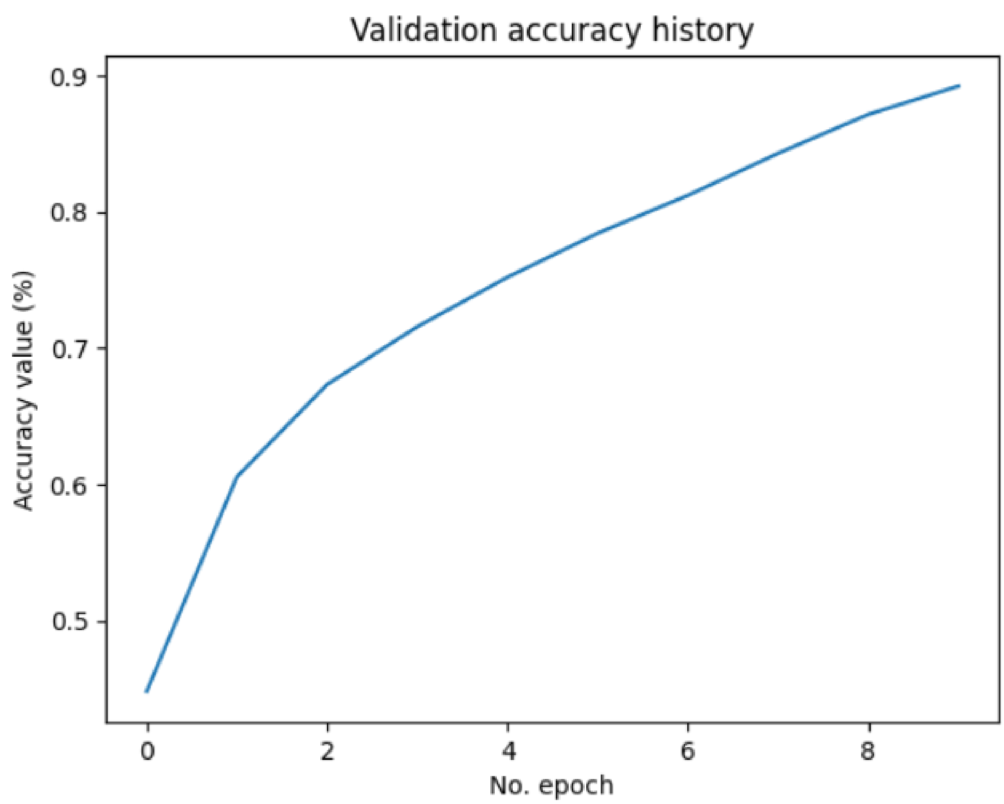
```
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Flatten())
  model.add(Dense(256, activation='relu'))
  model.add(Dense(128, activation='relu'))
  model.add(Dense(no_classes, activation='softmax'))
  # Compile the model
  model.compile(loss=loss_function,
                optimizer=optimizer,
                metrics=['accuracy'])
  # Generate a print
  print('------------------------------------------------------------------------')
  print(f'Training for fold {fold_no} ...')
  # Fit data to model
  history = model.fit(inputs[train], targets[train],
              batch_size=batch_size,
              epochs=no_epochs,
              verbose=verbosity)
  # Generate generalization metrics
  scores = model.evaluate(inputs[test], targets[test], verbose=0)
  print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]};
{model.metrics_names[1]} of {scores[1]*100}%')
  acc_per_fold.append(scores[1] * 100)
  loss_per_fold.append(scores[0])
  # Increase fold number
  fold_no = fold_no + 1

# == Provide average scores ==
```

```
print('------------------------------------------------------------------------')
print('Score per fold')
for i in range(0, len(acc_per_fold)):
  print('------------------------------------------------------------------------')
  print(f'> Fold {i+1} - Loss: {loss_per_fold[i]} - Accuracy: {acc_per_fold[i]}%')
print('------------------------------------------------------------------------')
print('Average scores for all folds:')
print(f'> Accuracy: {np.mean(acc_per_fold)} (+- {np.std(acc_per_fold)})')
print(f'> Loss: {np.mean(loss_per_fold)}')
print('------------------------------------------------------------------------')
```

Output:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 30, 30, 32)        896

 max_pooling2d (MaxPooling2  (None, 15, 15, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 13, 13, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 6, 6, 64)          0
 g2D)

 flatten (Flatten)           (None, 2304)              0

 dense (Dense)               (None, 256)               590080

 dense_1 (Dense)             (None, 128)               32896

 dense_2 (Dense)             (None, 100)               12900

=================================================================
Total params: 655268 (2.50 MB)
Trainable params: 655268 (2.50 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________

Test loss: 1.137102484703064 / Test accuracy: 0.6966999769210815
```


Validation loss history
1.4
1.2
1.0
0.8
0.6
0.4
Loss value
0
2
4
6
8
No. epoch


Validation accuracy history
0.9
0.8
0.7
0.6
0.5
Accuracy value (%)
0
2
4
6
8
No. epoch

```
Training for fold 1 ...
Epoch 1/10
960/960 [==============================] - 71s 73ms/step - loss: 1.5497 - accuracy: 0.4419
Epoch 2/10
960/960 [==============================] - 71s 74ms/step - loss: 1.1217 - accuracy: 0.6035
Epoch 3/10
960/960 [==============================] - 72s 75ms/step - loss: 0.9531 - accuracy: 0.6672
Epoch 4/10
960/960 [==============================] - 70s 73ms/step - loss: 0.8358 - accuracy: 0.7075
Epoch 5/10
960/960 [==============================] - 69s 72ms/step - loss: 0.7359 - accuracy: 0.7411
Epoch 6/10
960/960 [==============================] - 73s 76ms/step - loss: 0.6461 - accuracy: 0.7727
Epoch 7/10
960/960 [==============================] - 68s 70ms/step - loss: 0.5593 - accuracy: 0.8024
Epoch 8/10
960/960 [==============================] - 69s 72ms/step - loss: 0.4746 - accuracy: 0.8320
Epoch 9/10
960/960 [==============================] - 69s 72ms/step - loss: 0.4027 - accuracy: 0.8556
Epoch 10/10
960/960 [==============================] - 68s 71ms/step - loss: 0.3315 - accuracy: 0.8819
Score for fold 1: loss of 1.1447182893753052; accuracy of 69.01666522026062%
------------------------------------------------------------------------

Training for fold 2 ...
Epoch 1/10
960/960 [==============================] - 67s 69ms/step - loss: 1.4946 - accuracy: 0.4574
Epoch 2/10
960/960 [==============================] - 66s 69ms/step - loss: 1.0824 - accuracy: 0.6143
Epoch 3/10
960/960 [==============================] - 67s 70ms/step - loss: 0.9355 - accuracy: 0.6705
Epoch 4/10
960/960 [==============================] - 65s 68ms/step - loss: 0.8318 - accuracy: 0.7074
Epoch 5/10
960/960 [==============================] - 71s 74ms/step - loss: 0.7579 - accuracy: 0.7330
Epoch 6/10
960/960 [==============================] - 66s 69ms/step - loss: 0.6879 - accuracy: 0.7597
Epoch 7/10
960/960 [==============================] - 68s 71ms/step - loss: 0.6271 - accuracy: 0.7802
Epoch 8/10
960/960 [==============================] - 69s 72ms/step - loss: 0.5677 - accuracy: 0.8015
Epoch 9/10
960/960 [==============================] - 68s 71ms/step - loss: 0.5144 - accuracy: 0.8185
Epoch 10/10
960/960 [==============================] - 66s 69ms/step - loss: 0.4610 - accuracy: 0.8378
Score for fold 2: loss of 0.9815402030944824; accuracy of 70.333331823349%
------------------------------------------------------------------------
```

```
Training for fold 3 ...
Epoch 1/10
960/960 [==============================] - 68s 70ms/step - loss: 1.7482 - accuracy: 0.3514
Epoch 2/10
960/960 [==============================] - 67s 70ms/step - loss: 1.3000 - accuracy: 0.5319
Epoch 3/10
960/960 [==============================] - 68s 71ms/step - loss: 1.1235 - accuracy: 0.6001
Epoch 4/10
960/960 [==============================] - 70s 73ms/step - loss: 1.0199 - accuracy: 0.6381
Epoch 5/10
960/960 [==============================] - 69s 72ms/step - loss: 0.9427 - accuracy: 0.6674
Epoch 6/10
960/960 [==============================] - 66s 69ms/step - loss: 0.8764 - accuracy: 0.6916
Epoch 7/10
960/960 [==============================] - 66s 69ms/step - loss: 0.8204 - accuracy: 0.7114
Epoch 8/10
960/960 [==============================] - 68s 70ms/step - loss: 0.7683 - accuracy: 0.7293
Epoch 9/10
960/960 [==============================] - 66s 69ms/step - loss: 0.7147 - accuracy: 0.7476
Epoch 10/10
960/960 [==============================] - 65s 68ms/step - loss: 0.6659 - accuracy: 0.7670
Score for fold 3: loss of 0.9777575135231018; accuracy of 67.64166951179504%
------------------------------------------------------------------------

Training for fold 4 ...
Epoch 1/10
960/960 [==============================] - 66s 68ms/step - loss: 1.5605 - accuracy: 0.4300
Epoch 2/10
960/960 [==============================] - 67s 69ms/step - loss: 1.1548 - accuracy: 0.5909
Epoch 3/10
960/960 [==============================] - 72s 75ms/step - loss: 1.0053 - accuracy: 0.6465
Epoch 4/10
960/960 [==============================] - 68s 71ms/step - loss: 0.9051 - accuracy: 0.6813
Epoch 5/10
960/960 [==============================] - 67s 70ms/step - loss: 0.8169 - accuracy: 0.7136
Epoch 6/10
960/960 [==============================] - 69s 71ms/step - loss: 0.7510 - accuracy: 0.7357
Epoch 7/10
960/960 [==============================] - 66s 69ms/step - loss: 0.6871 - accuracy: 0.7579
Epoch 8/10
960/960 [==============================] - 68s 71ms/step - loss: 0.6304 - accuracy: 0.7791
Epoch 9/10
960/960 [==============================] - 67s 70ms/step - loss: 0.5766 - accuracy: 0.7966
Epoch 10/10
960/960 [==============================] - 68s 71ms/step - loss: 0.5261 - accuracy: 0.8164
Score for fold 4: loss of 0.9539607167243958; accuracy of 69.25833225250244%
------------------------------------------------------------------------
```

```
Training for fold 5 ...
Epoch 1/10
960/960 [==============================] - 67s 69ms/step - loss: 1.5813 - accuracy: 0.4209
Epoch 2/10
960/960 [==============================] - 73s 76ms/step - loss: 1.2035 - accuracy: 0.5685
Epoch 3/10
960/960 [==============================] - 68s 70ms/step - loss: 1.0382 - accuracy: 0.6306
Epoch 4/10
960/960 [==============================] - 68s 71ms/step - loss: 0.9473 - accuracy: 0.6646
Epoch 5/10
960/960 [==============================] - 66s 68ms/step - loss: 0.8742 - accuracy: 0.6905
Epoch 6/10
960/960 [==============================] - 66s 69ms/step - loss: 0.8167 - accuracy: 0.7120
Epoch 7/10
960/960 [==============================] - 69s 71ms/step - loss: 0.7613 - accuracy: 0.7337
Epoch 8/10
960/960 [==============================] - 66s 69ms/step - loss: 0.7087 - accuracy: 0.7507
Epoch 9/10
960/960 [==============================] - 68s 71ms/step - loss: 0.6611 - accuracy: 0.7652
Epoch 10/10
960/960 [==============================] - 67s 70ms/step - loss: 0.6132 - accuracy: 0.7837
Score for fold 5: loss of 0.9543024301528931; accuracy of 68.57500076293945%

------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 1.1447182893753052 - Accuracy: 69.01666522026062%
------------------------------------------------------------------------
> Fold 2 - Loss: 0.9815402030944824 - Accuracy: 70.333331823349%
------------------------------------------------------------------------
> Fold 3 - Loss: 0.9777575135231018 - Accuracy: 67.64166951179504%
------------------------------------------------------------------------
> Fold 4 - Loss: 0.9539607167243958 - Accuracy: 69.25833225250244%
------------------------------------------------------------------------
> Fold 5 - Loss: 0.9543024301528931 - Accuracy: 68.57500076293945%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 68.96499991416931 (+- 0.8791300324813014)
> Loss: 1.0024558305740356
------------------------------------------------------------------------
```

Practical No: 6 new

Aim: Implementing regularization to avoid overfitting in binary classification.

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:]
print(trainX.shape)
print(trainY.shape)
print(testX.shape)
print(testY.shape)

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'])
model.summary()
history=model.fit(trainX,trainY,validation_data=(testX,testY),epochs=100)

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

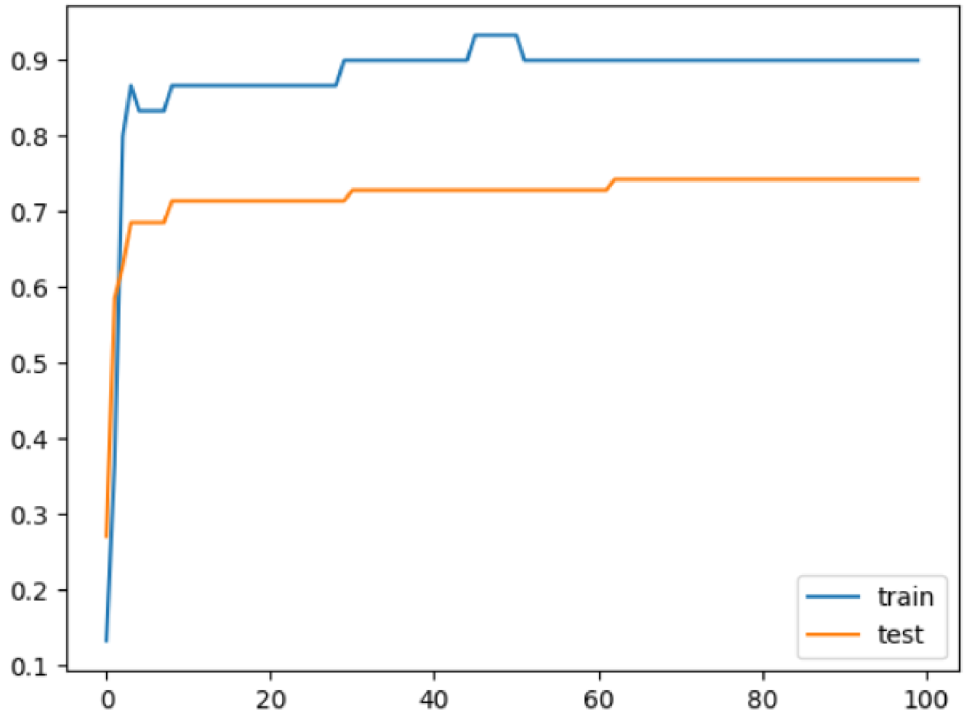
```
pyplot.plot(history.history['val_accuracy'],label='test')
pyplot.legend()
pyplot.show()

from keras.regularizers import l2
model=Sequential()
model.add(Dense(500,input_dim=2,activation='relu',kernel_regularizer=l2(0.001)))
model.add(Dense(1,activation='sigmoid'))
model.summary()

model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
history=model.fit(trainX,trainY,validation_data=(testX,testY),epochs=100)

pyplot.plot(history.history['accuracy'],label='train')
pyplot.plot(history.history['val_accuracy'],label='test')
pyplot.legend()
pyplot.show()

from keras.regularizers import l1_l2
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'])
model.summary()
history=model.fit(trainX,trainY,validation_data=(testX,testY),epochs=100)
```

```
pyplot.plot(history.history['accuracy'],label='train')
pyplot.plot(history.history['val_accuracy'],label='test')
pyplot.legend()
pyplot.show()
```

Output:

```
(30, 2)
(30,)
(70, 2)
(70,)

Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 500)               1500

 dense_1 (Dense)             (None, 1)                 501

=================================================================
Total params: 2001 (7.82 KB)
Trainable params: 2001 (7.82 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

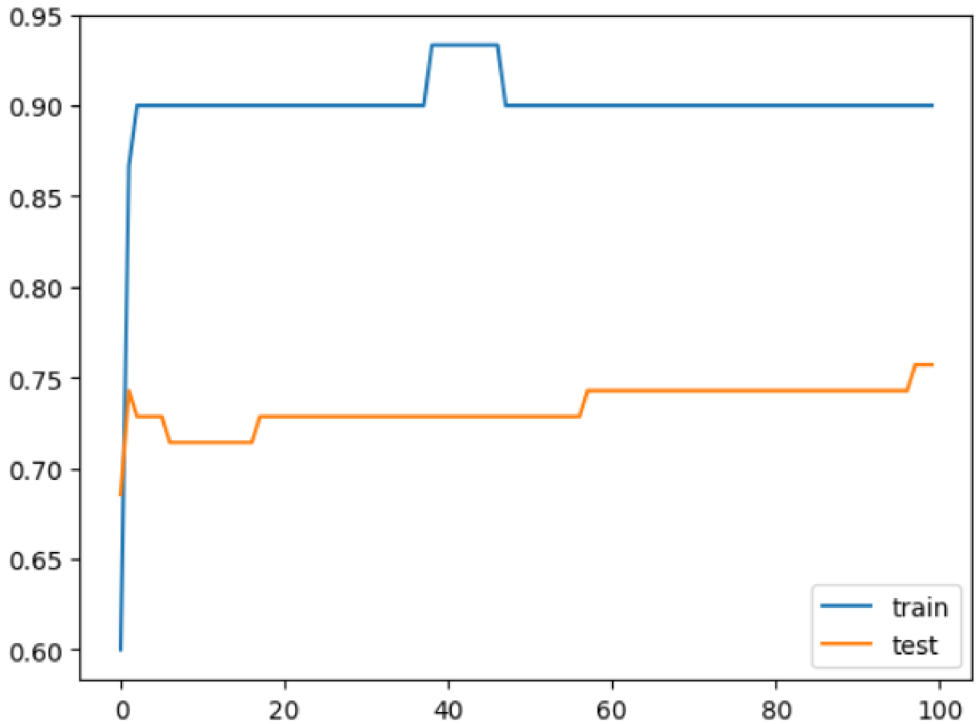
```
Epoch 1/100
1/1 [==============================] - 1s 1s/step - loss: 0.7125 - accuracy: 0.1333 - val_loss: 0.6994 - val_accuracy: 0.2714
Epoch 2/100
1/1 [==============================] - 0s 47ms/step - loss: 0.6957 - accuracy: 0.3667 - val_loss: 0.6884 - val_accuracy: 0.5857
Epoch 3/100
1/1 [==============================] - 0s 39ms/step - loss: 0.6794 - accuracy: 0.8000 - val_loss: 0.6778 - val_accuracy: 0.6286
Epoch 4/100
1/1 [==============================] - 0s 42ms/step - loss: 0.6635 - accuracy: 0.8667 - val_loss: 0.6675 - val_accuracy: 0.6857
Epoch 5/100
1/1 [==============================] - 0s 43ms/step - loss: 0.6481 - accuracy: 0.8333 - val_loss: 0.6577 - val_accuracy: 0.6857

Epoch 95/100
1/1 [==============================] - 0s 76ms/step - loss: 0.1927 - accuracy: 0.9000 - val_loss: 0.4286 - val_accuracy: 0.7429
Epoch 96/100
1/1 [==============================] - 0s 68ms/step - loss: 0.1919 - accuracy: 0.9000 - val_loss: 0.4279 - val_accuracy: 0.7429
Epoch 97/100
1/1 [==============================] - 0s 78ms/step - loss: 0.1911 - accuracy: 0.9000 - val_loss: 0.4271 - val_accuracy: 0.7429
Epoch 98/100
1/1 [==============================] - 0s 69ms/step - loss: 0.1903 - accuracy: 0.9000 - val_loss: 0.4264 - val_accuracy: 0.7429
Epoch 99/100
1/1 [==============================] - 0s 67ms/step - loss: 0.1896 - accuracy: 0.9000 - val_loss: 0.4256 - val_accuracy: 0.7429
Epoch 100/100
1/1 [==============================] - 0s 67ms/step - loss: 0.1888 - accuracy: 0.9000 - val_loss: 0.4249 - val_accuracy: 0.7429
```




```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_2 (Dense)             (None, 500)               1500

 dense_3 (Dense)             (None, 1)                 501

=================================================================
Total params: 2001 (7.82 KB)
Trainable params: 2001 (7.82 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

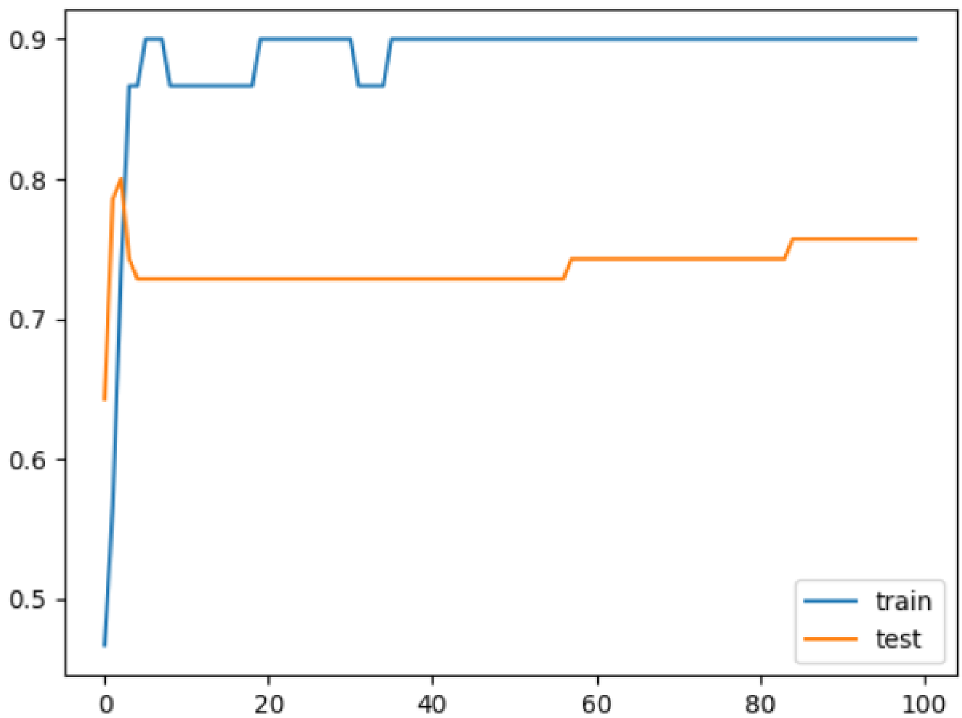
```
Epoch 1/100
1/1 [==============================] - 1s 1s/step - loss: 0.6955 - accuracy: 0.6000 - val_loss: 0.6837 - val_accuracy: 0.6857
Epoch 2/100
1/1 [==============================] - 0s 43ms/step - loss: 0.6789 - accuracy: 0.8667 - val_loss: 0.6729 - val_accuracy: 0.7429
Epoch 3/100
1/1 [==============================] - 0s 47ms/step - loss: 0.6627 - accuracy: 0.9000 - val_loss: 0.6625 - val_accuracy: 0.7286
Epoch 4/100
1/1 [==============================] - 0s 57ms/step - loss: 0.6470 - accuracy: 0.9000 - val_loss: 0.6525 - val_accuracy: 0.7286
Epoch 5/100
1/1 [==============================] - 0s 55ms/step - loss: 0.6317 - accuracy: 0.9000 - val_loss: 0.6429 - val_accuracy: 0.7286
```

```
Epoch 95/100
1/1 [==============================] - 0s 43ms/step - loss: 0.1986 - accuracy: 0.9000 - val_loss: 0.4315 - val_accuracy: 0.7429
Epoch 96/100
1/1 [==============================] - 0s 42ms/step - loss: 0.1979 - accuracy: 0.9000 - val_loss: 0.4309 - val_accuracy: 0.7429
Epoch 97/100
1/1 [==============================] - 0s 56ms/step - loss: 0.1972 - accuracy: 0.9000 - val_loss: 0.4303 - val_accuracy: 0.7429
Epoch 98/100
1/1 [==============================] - 0s 58ms/step - loss: 0.1965 - accuracy: 0.9000 - val_loss: 0.4297 - val_accuracy: 0.7571
Epoch 99/100
1/1 [==============================] - 0s 57ms/step - loss: 0.1958 - accuracy: 0.9000 - val_loss: 0.4290 - val_accuracy: 0.7571
Epoch 100/100
1/1 [==============================] - 0s 56ms/step - loss: 0.1951 - accuracy: 0.9000 - val_loss: 0.4284 - val_accuracy: 0.7571
```




```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_4 (Dense)             (None, 500)               1500

 dense_5 (Dense)             (None, 1)                 501

=================================================================
Total params: 2001 (7.82 KB)
Trainable params: 2001 (7.82 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
Epoch 1/100
1/1 [==============================] - 1s 1s/step - loss: 0.7606 - accuracy: 0.4667 - val_loss: 0.7435 - val_accuracy: 0.6429
Epoch 2/100
1/1 [==============================] - 0s 58ms/step - loss: 0.7445 - accuracy: 0.5667 - val_loss: 0.7329 - val_accuracy: 0.7857
Epoch 3/100
1/1 [==============================] - 0s 57ms/step - loss: 0.7288 - accuracy: 0.7333 - val_loss: 0.7225 - val_accuracy: 0.8000
Epoch 4/100
1/1 [==============================] - 0s 58ms/step - loss: 0.7135 - accuracy: 0.8667 - val_loss: 0.7125 - val_accuracy: 0.7429
Epoch 5/100
1/1 [==============================] - 0s 42ms/step - loss: 0.6985 - accuracy: 0.8667 - val_loss: 0.7029 - val_accuracy: 0.7286

Epoch 95/100
1/1 [==============================] - 0s 153ms/step - loss: 0.2591 - accuracy: 0.9000 - val_loss: 0.4713 - val_accuracy: 0.7571
Epoch 96/100
1/1 [==============================] - 0s 136ms/step - loss: 0.2582 - accuracy: 0.9000 - val_loss: 0.4705 - val_accuracy: 0.7571
Epoch 97/100
1/1 [==============================] - 0s 91ms/step - loss: 0.2574 - accuracy: 0.9000 - val_loss: 0.4698 - val_accuracy: 0.7571
Epoch 98/100
1/1 [==============================] - 0s 40ms/step - loss: 0.2566 - accuracy: 0.9000 - val_loss: 0.4690 - val_accuracy: 0.7571
Epoch 99/100
1/1 [==============================] - 0s 43ms/step - loss: 0.2558 - accuracy: 0.9000 - val_loss: 0.4682 - val_accuracy: 0.7571
Epoch 100/100
1/1 [==============================] - 0s 43ms/step - loss: 0.2550 - accuracy: 0.9000 - val_loss: 0.4675 - val_accuracy: 0.7571
```



Practical No: 7

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

Code:

```
import numpy as np
import tensorflow_datasets as tfds
import tensorflow as tf
tfds.disable_progress_bar()
import matplotlib.pyplot as plt
def plot_graphs(history, metric):
  plt.plot(history.history[metric])
  plt.plot(history.history['val_'+metric], '')
  plt.xlabel("Epochs")
  plt.ylabel(metric)
  plt.legend([metric, 'val_'+metric])
dataset, info = tfds.load('imdb_reviews', with_info=True,
                          as_supervised=True)
train_dataset, test_dataset = dataset['train'], dataset['test']
train_dataset.element_spec

for example, label in train_dataset.take(5):
  print('text: ', example.numpy())
  print('label: ', label.numpy())

BUFFER_SIZE = 10000
```

```
BATCH_SIZE = 64
train_dataset =
train_dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
test_dataset = test_dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)

for example, label in train_dataset.take(1):
  print('texts: ', example.numpy()[:3])
  print()
  print('labels: ', label.numpy()[:3])

VOCAB_SIZE = 1000
encoder = tf.keras.layers.TextVectorization(max_tokens=VOCAB_SIZE)
encoder.adapt(train_dataset.map(lambda text, label: text))

vocab = np.array(encoder.get_vocabulary())
vocab[:20]

encoded_example = encoder(example)[:3].numpy()
encoded_example

for n in range(3):
  print("Original: ", example[n].numpy())
  print("Round-trip: ", " ".join(vocab[encoded_example[n]]))
  print()

model = tf.keras.Sequential([
    encoder,
```

```
    tf.keras.layers.Embedding(
        input_dim=len(encoder.get_vocabulary()),
        output_dim=64,
        # Use masking to handle the variable sequence lengths
        mask_zero=True),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1)
])
print([layer.supports_masking for layer in model.layers])

# predict on a sample text without padding.
sample_text = ('The movie was cool. The animation and the graphics '
               'were out of this world. I would recommend this movie.')
predictions = model.predict(np.array([sample_text]))
print(predictions[0])

# predict on a sample text with padding
padding = "the " * 2000
predictions = model.predict(np.array([sample_text, padding]))
print(predictions[0])

model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              optimizer=tf.keras.optimizers.Adam(1e-4),
              metrics=['accuracy'])
history = model.fit(train_dataset, epochs=10,
                    validation_data=test_dataset,
```

```
                validation_steps=30)

test_loss, test_acc = model.evaluate(test_dataset)

print('Test Loss:', test_loss)
print('Test Accuracy:', test_acc)

plt.figure(figsize=(16, 8))
plt.subplot(1, 2, 1)
plot_graphs(history, 'accuracy')
plt.ylim(None, 1)
plt.subplot(1, 2, 2)
plot_graphs(history, 'loss')
plt.ylim(0, None)

sample_text = ('The movie was cool. The animation and the graphics '
            'were out of this world. I would recommend this movie.')
predictions = model.predict(np.array([sample_text]))
predictions

model = tf.keras.Sequential([
    encoder,
    tf.keras.layers.Embedding(len(encoder.get_vocabulary()), 64, mask_zero=True),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64,  return_sequences=True)),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dropout(0.5),
```

```
    tf.keras.layers.Dense(1)
])
model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              optimizer=tf.keras.optimizers.Adam(1e-4),
              metrics=['accuracy'])
history = model.fit(train_dataset, epochs=10,
                    validation_data=test_dataset,
                    validation_steps=30)

test_loss, test_acc = model.evaluate(test_dataset)
print('Test Loss:', test_loss)
print('Test Accuracy:', test_acc)

# predict on a sample text without padding.
sample_text = ('The movie was not good. The animation and the graphics '
               'were terrible. I would not recommend this movie.')
predictions = model.predict(np.array([sample_text]))
print(predictions)

plt.figure(figsize=(16, 6))
plt.subplot(1, 2, 1)
plot_graphs(history, 'accuracy')
plt.subplot(1, 2, 2)
plot_graphs(history, 'loss')
```

Output:

```
text:  b"This was an absolutely terrible movie. Don't be lured in by Christopher Walken or Michael Ironside. Both are great actors, but this must simply be their worst role in history. Even their great act
label:  0
text:  b'I have been known to fall asleep during films, but this is usually due to a combination of things including, really tired, being warm and comfortable on the sette and having just eaten a lot. Howe
label:  0
text:  b'Mann photographs the Alberta Rocky Mountains in a superb fashion, and Jimmy Stewart and Walter Brennan give enjoyable performances as they always seem to do. <br /><br />But come on Hollywood - a
label:  0
text:  b'This is the kind of film for a snowy Sunday afternoon when the rest of the world can go ahead with its own business as you descend into a big arm-chair and mellow for a couple of hours. Wonderful
label:  1
text:  b'As others have mentioned, all the women that go nude in this film are mostly absolutely gorgeous. The plot very ably shows the hypocrisy of the female libido. When men are around they want to be
label:  1
```

```
array([[ 10,   1, 442, ...,   0,   0,   0],
       [  1,  16,  49, ...,   0,   0,   0],
       [  4, 220,  12, ...,   0,   0,   0]])
```

```
array(['', '[UNK]', 'the', 'and', 'a', 'of', 'to', 'is', 'in', 'it', 'i',
       'this', 'that', 'br', 'was', 'as', 'for', 'with', 'movie', 'but'],
      dtype='<U14')
```

```
Original:  b'I voted 3 for this movie because it looks great as does all of Greenaways output. However it was his usual mix of "art" sex and pretentious crap.I know lots of people like this film but I gre
Round-trip:  i [UNK] 3 for this movie because it looks great as does all of [UNK] [UNK] however it was his usual [UNK] of art sex and [UNK] [UNK] know lots of people like this film but i [UNK] [UNK] of it

Original:  b'Justifications for what happened to his movie in terms of distributors and secondary directors, drunks and receptionists doing script rewrites aside, let\'s just take this movie as it\'s offer
Round-trip:  [UNK] for what happened to his movie in [UNK] of [UNK] and [UNK] directors [UNK] and [UNK] doing script [UNK] [UNK] lets just take this movie as its [UNK] without [UNK] [UNK] br this movie is

Original:  b'A comedy that worked surprisingly well was the little British effort "The Divorce Of Lady X (1938)" . It marks the first pairing of Laurence Olivier and Merle Oberon, before that little film
Round-trip:  a comedy that worked [UNK] well was the little british effort the [UNK] of lady [UNK] [UNK] it [UNK] the first [UNK] of [UNK] [UNK] and [UNK] [UNK] before that little film about [UNK] [UNK] or
```

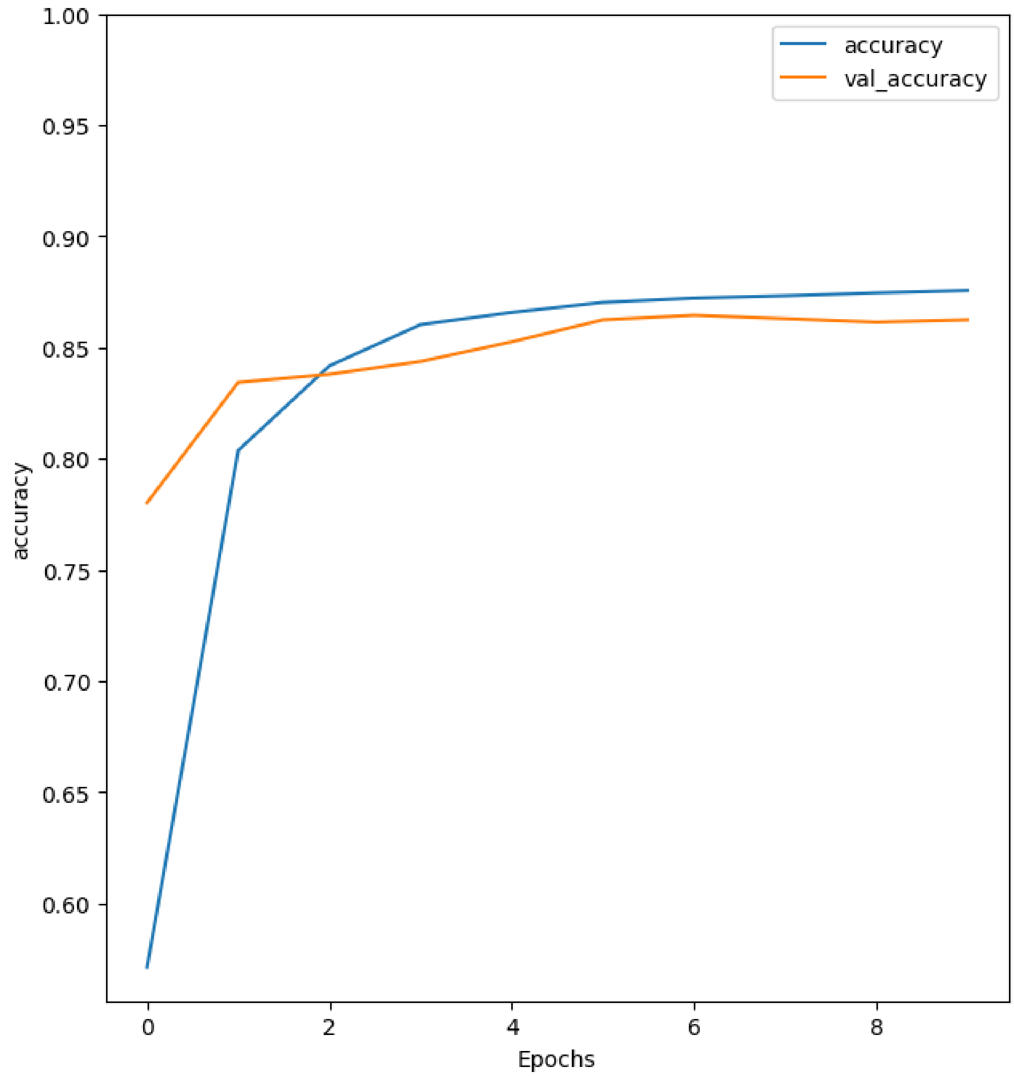
```
[False, True, True, True, True]
```

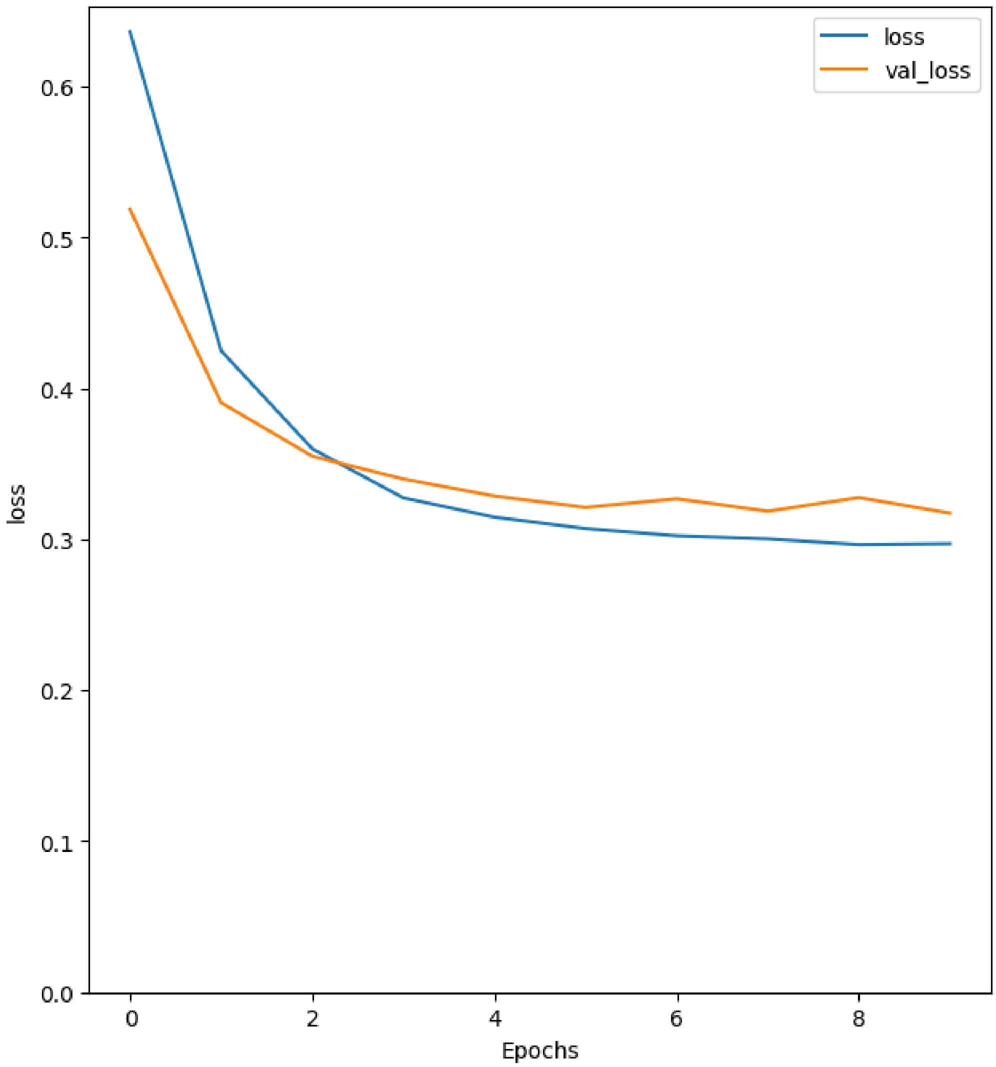
```
1/1 [==============================] - 4s 4s/step
[-0.01484437]
```

```
Epoch 1/10
391/391 [==============================] - 50s 103ms/step - loss: 0.6363 - accuracy: 0.5712 - val_loss: 0.5186 - val_accuracy: 0.7802
Epoch 2/10
391/391 [==============================] - 26s 67ms/step - loss: 0.4250 - accuracy: 0.8037 - val_loss: 0.3905 - val_accuracy: 0.8344
Epoch 3/10
391/391 [==============================] - 27s 69ms/step - loss: 0.3600 - accuracy: 0.8418 - val_loss: 0.3551 - val_accuracy: 0.8380
Epoch 4/10
391/391 [==============================] - 26s 67ms/step - loss: 0.3275 - accuracy: 0.8604 - val_loss: 0.3400 - val_accuracy: 0.8438
Epoch 5/10
391/391 [==============================] - 25s 65ms/step - loss: 0.3147 - accuracy: 0.8658 - val_loss: 0.3287 - val_accuracy: 0.8526
Epoch 6/10
391/391 [==============================] - 26s 66ms/step - loss: 0.3071 - accuracy: 0.8703 - val_loss: 0.3212 - val_accuracy: 0.8625
Epoch 7/10
391/391 [==============================] - 25s 64ms/step - loss: 0.3024 - accuracy: 0.8722 - val_loss: 0.3269 - val_accuracy: 0.8646
Epoch 8/10
391/391 [==============================] - 25s 64ms/step - loss: 0.3003 - accuracy: 0.8733 - val_loss: 0.3187 - val_accuracy: 0.8630
Epoch 9/10
391/391 [==============================] - 25s 64ms/step - loss: 0.2965 - accuracy: 0.8746 - val_loss: 0.3277 - val_accuracy: 0.8615
Epoch 10/10
391/391 [==============================] - 26s 65ms/step - loss: 0.2971 - accuracy: 0.8757 - val_loss: 0.3174 - val_accuracy: 0.8625
```

```
391/391 [==============================] - 18s 45ms/step - loss: 0.3137 - accuracy: 0.8618
Test Loss: 0.3137068450450897
Test Accuracy: 0.8617600202560425
```

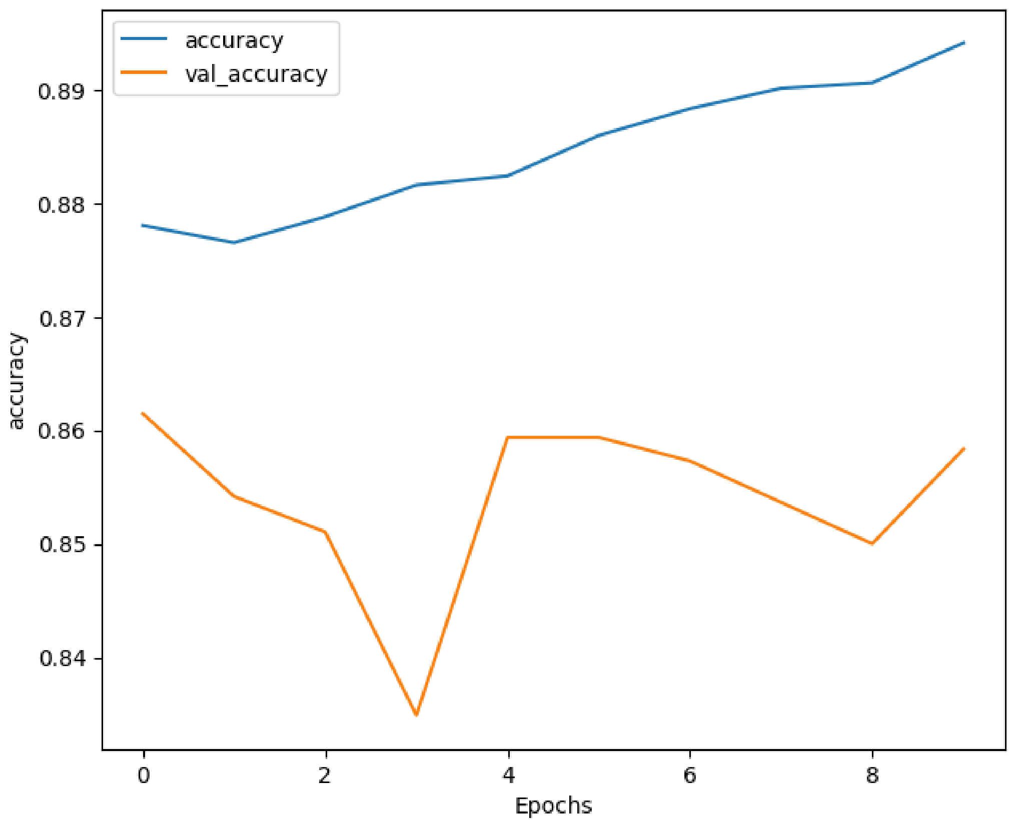
```
(0.0, 0.6532905504107476)
```

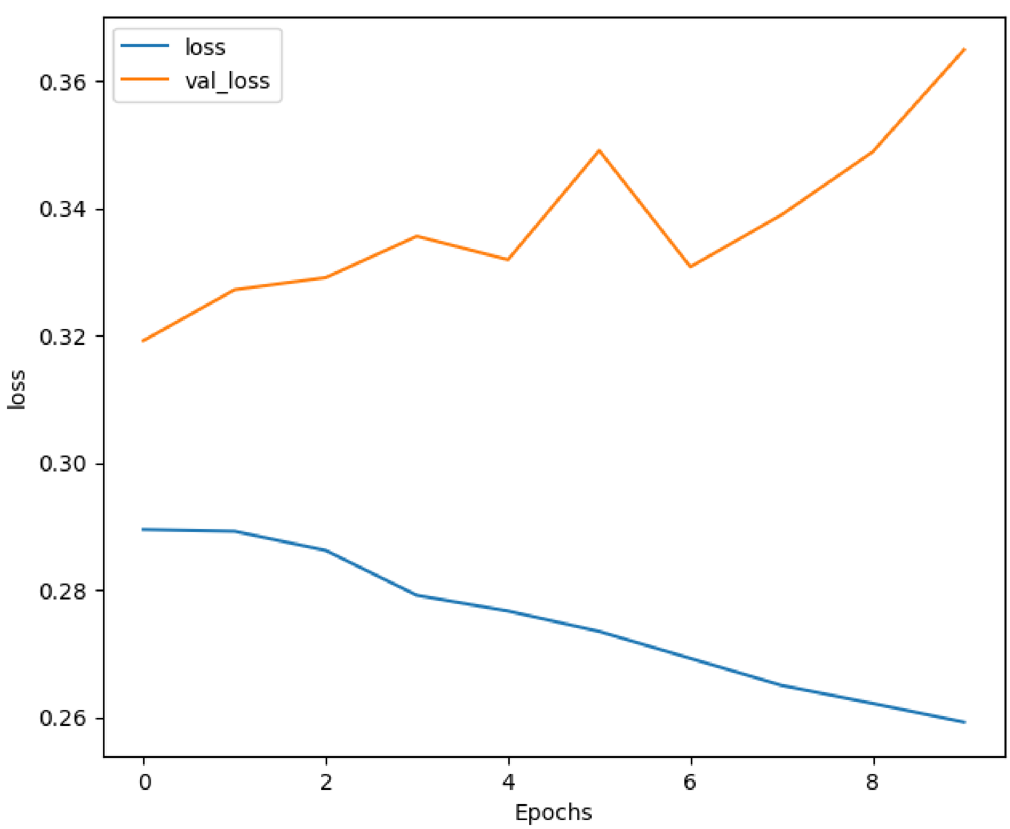





```
Epoch 1/10
391/391 [==============================] - 56s 142ms/step - loss: 0.2895 - accuracy: 0.8780 - val_loss: 0.3192 - val_accuracy: 0.8615
Epoch 2/10
391/391 [==============================] - 48s 123ms/step - loss: 0.2892 - accuracy: 0.8765 - val_loss: 0.3272 - val_accuracy: 0.8542
Epoch 3/10
391/391 [==============================] - 47s 120ms/step - loss: 0.2862 - accuracy: 0.8788 - val_loss: 0.3291 - val_accuracy: 0.8510
Epoch 4/10
391/391 [==============================] - 47s 119ms/step - loss: 0.2792 - accuracy: 0.8816 - val_loss: 0.3356 - val_accuracy: 0.8349
Epoch 5/10
391/391 [==============================] - 46s 117ms/step - loss: 0.2767 - accuracy: 0.8824 - val_loss: 0.3319 - val_accuracy: 0.8594
Epoch 6/10
391/391 [==============================] - 48s 121ms/step - loss: 0.2735 - accuracy: 0.8860 - val_loss: 0.3491 - val_accuracy: 0.8594
Epoch 7/10
391/391 [==============================] - 46s 117ms/step - loss: 0.2692 - accuracy: 0.8883 - val_loss: 0.3308 - val_accuracy: 0.8573
Epoch 8/10
391/391 [==============================] - 45s 115ms/step - loss: 0.2650 - accuracy: 0.8901 - val_loss: 0.3390 - val_accuracy: 0.8536
Epoch 9/10
391/391 [==============================] - 47s 120ms/step - loss: 0.2622 - accuracy: 0.8906 - val_loss: 0.3489 - val_accuracy: 0.8500
Epoch 10/10
391/391 [==============================] - 47s 119ms/step - loss: 0.2592 - accuracy: 0.8941 - val_loss: 0.3649 - val_accuracy: 0.8583
```

```
391/391 [==============================] - 20s 51ms/step - loss: 0.3604 - accuracy: 0.8529
Test Loss: 0.3603912889957428
Test Accuracy: 0.8529199957847595
```


accuracy
val_accuracy
0.89
0.88
0.87
0.86
0.85
0.84
accuracy
0
2
4
6
8
Epochs


loss
val_loss
0.36
0.34
0.32
0.30
0.28
0.26
loss
0
2
4
6
8
Epochs

Practical No: 8

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

Code:

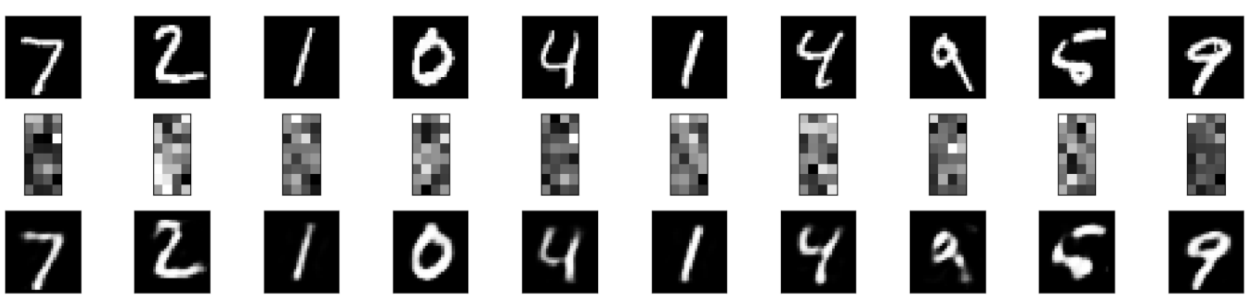
```
import keras
from keras import layers
from keras.datasets import mnist
import numpy as np
encoding_dim=32
#this is our input image
input_img=keras.Input(shape=(784,))
#"encoded" is the encoded representation of the input
encoded=layers.Dense(encoding_dim, activation='relu')(input_img)
#"decoded"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

```
Epoch 1/50
235/235 [==============================] - 4s 12ms/step - loss: 0.2743 - val_loss: 0.1868
Epoch 2/50
235/235 [==============================] - 4s 16ms/step - loss: 0.1695 - val_loss: 0.1527
Epoch 3/50
235/235 [==============================] - 3s 11ms/step - loss: 0.1440 - val_loss: 0.1334
Epoch 4/50
235/235 [==============================] - 3s 12ms/step - loss: 0.1283 - val_loss: 0.1212
Epoch 5/50
235/235 [==============================] - 3s 11ms/step - loss: 0.1180 - val_loss: 0.1127

Epoch 45/50
235/235 [==============================] - 3s 12ms/step - loss: 0.0927 - val_loss: 0.0917
Epoch 46/50
235/235 [==============================] - 3s 12ms/step - loss: 0.0927 - val_loss: 0.0915
Epoch 47/50
235/235 [==============================] - 4s 16ms/step - loss: 0.0927 - val_loss: 0.0915
Epoch 48/50
235/235 [==============================] - 3s 12ms/step - loss: 0.0926 - val_loss: 0.0915
Epoch 49/50
235/235 [==============================] - 3s 12ms/step - loss: 0.0926 - val_loss: 0.0915
Epoch 50/50
235/235 [==============================] - 3s 12ms/step - loss: 0.0926 - val_loss: 0.0915
```



Practical No: 9

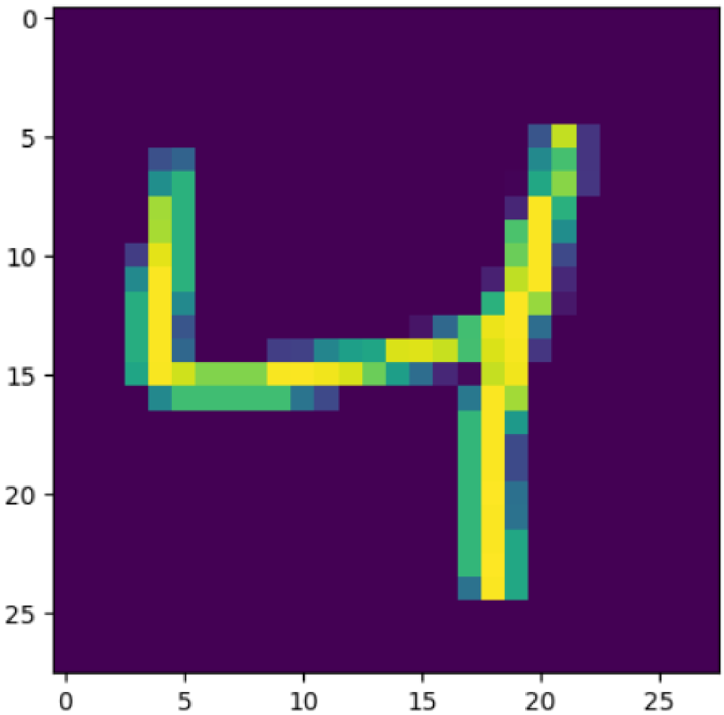
Aim: Implementation of convolutional neural network to predict numbers from number images.

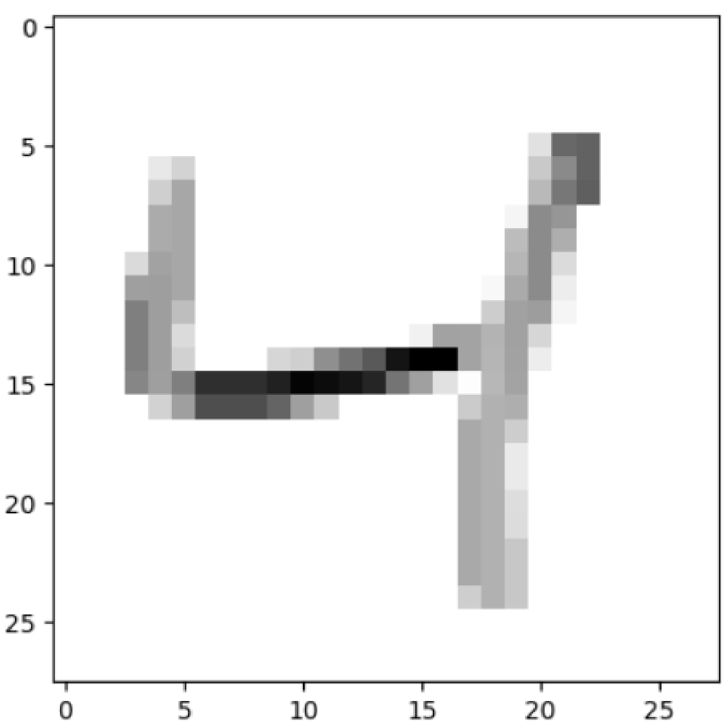
Code:

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train.shape
y_train.shape
X_test.shape
y_test.shape

import matplotlib.pyplot as plt
plt.imshow(X_train[2])
plt.show()
plt.imshow(X_train[2], cmap=plt.cm.binary)

X_train[2]
X_train = tf.keras.utils.normalize(X_train, axis=1)
X_test = tf.keras.utils.normalize(X_test, axis=1)
plt.imshow(X_train[2], cmap=plt.cm.binary)
print(X_train[2])

import tensorflow as tf
```

```
import tensorflow.keras.layers  as KL
import tensorflow.keras.models  as KM

inputs = KL.Input(shape=(28, 28, 1))
c = KL.Conv2D(32, (3, 3), padding="valid", activation=tf.nn.relu)(inputs)
m = KL.MaxPool2D((2, 2), (2, 2))(c)
d = KL.Dropout(0.5)(m)
c = KL.Conv2D(64, (3, 3), padding="valid", activation=tf.nn.relu)(d)
m = KL.MaxPool2D((2, 2), (2, 2))(c)
d = KL.Dropout(0.5)(m)
c = KL.Conv2D(128, (3, 3), padding="valid", activation=tf.nn.relu)(d)
f = KL.Flatten()(c)
outputs = KL.Dense(10, activation=tf.nn.softmax)(f)
model = KM.Model(inputs, outputs)
model.summary()
model.compile(optimizer="adam", loss="sparse_categorical_crossentropy",
metrics=["accuracy"])

model.fit(X_train, y_train, epochs=5)
test_loss, test_acc = model.evaluate(X_test, y_test)
print("Test Loss: {0} - Test Acc: {1}".format(test_loss, test_acc))
```

Output:


0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0

 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2  (None, 13, 13, 32)        0
 D)

 dropout (Dropout)           (None, 13, 13, 32)        0

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 5, 5, 64)          0
 g2D)

 dropout_1 (Dropout)         (None, 5, 5, 64)          0

 conv2d_2 (Conv2D)           (None, 3, 3, 128)         73856

 flatten (Flatten)           (None, 1152)              0

 dense (Dense)               (None, 10)                11530

=================================================================
Total params: 104202 (407.04 KB)
Trainable params: 104202 (407.04 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________

Epoch 1/5
1875/1875 [==============================] - 65s 34ms/step - loss: 0.2675 - accuracy: 0.9175
Epoch 2/5
1875/1875 [==============================] - 62s 33ms/step - loss: 0.0983 - accuracy: 0.9699
Epoch 3/5
1875/1875 [==============================] - 64s 34ms/step - loss: 0.0755 - accuracy: 0.9767
Epoch 4/5
1875/1875 [==============================] - 62s 33ms/step - loss: 0.0654 - accuracy: 0.9796
Epoch 5/5
1875/1875 [==============================] - 64s 34ms/step - loss: 0.0568 - accuracy: 0.9822
313/313 [==============================] - 3s 9ms/step - loss: 0.0309 - accuracy: 0.9897
Test Loss: 0.03090468980371952 - Test Acc: 0.9897000193595886
```

Practical No: 10

Aim: Denoising of images using autoencoder.

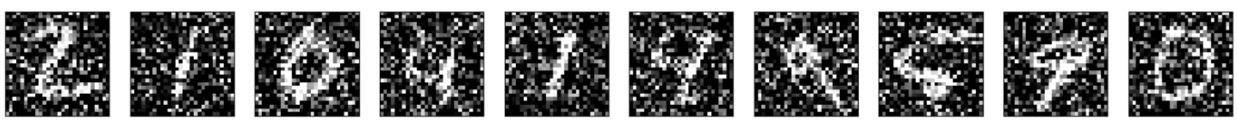
Code:

```
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 [==============================] - 133s 281ms/step - loss: 0.1604 - val_loss: 0.1171
Epoch 2/3
469/469 [==============================] - 115s 245ms/step - loss: 0.1126 - val_loss: 0.1077
Epoch 3/3
469/469 [==============================] - 114s 242ms/step - loss: 0.1073 - val_loss: 0.1063
313/313 [==============================] - 5s 15ms/step
```