**Parle Tilak Vidyalaya Associations**

**M. L. DAHANUKAR COLLEGE**

**Vile-Parle (East), Mumbai – 400 057.**

**Practical Journal**

**DEEP LEARNING**

**Submitted by**

**Trupti Hiru Bhostekar**

**Seat No.:**

**M.Sc. [I.T.]-Information Technology Part II**

**2023 – 2024**

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| **Sr.No.** | **Date** | **Aim** | **Sign** |
| 1 | 30/03/2024 | Performing matrix multiplication and finding eigen vectors and eigen values using TensorFlow |  |
| 2 | 30/03/2024 | Solving XOR problem using deep feed forward network |  |
| 3 | 04/04/2024 | Implementing deep neural network for performing binary classification task |  |
| 4.a | 25/04/2024 | Using deep feed forward network with two hidden layers for performing classification and predicting the class |  |
| 4.b | 25/04/2024 | Using deep feed forward network with two hidden layers for performing classification and predicting the probability of class |  |
| 5 | 15/05/2024 | Evaluating feed forward deep network for regression using KFold cross validation |  |
| 6 | 15/05/2024 | Implementing regularization to avoid overfitting in binary classification using TensorFlow |  |
| 7 | 22/05/2024 | Implementing Text classification with an RNN |  |
| 8 | 29/05/2024 | Implementation of Autoencoders |  |
| 9 | 30/05/2024 | Implementation of convolutional neural network to predict numbers from number images |  |
| 10 | 30/05/2024 | Implementing Denoising of images using Autoencoder |  |

**Practical No 1**

**AIM : Performing matrix multiplication and finding eigen vectors and eigen values using TensorFlow**

Matrix multiplication and finding eigenvalues and eigenvectors are fundamental operations in many deep learning algorithms. TensorFlow, a powerful library for deep learning, can handle these operations efficiently.

**Code :**

# importing numpy library

import numpy as np

# create numpy 2d-array

m = np.array([[1, 2],[2, 3]])

print("Printing the Original square array:\n",m)

print()

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

print()

# finding eigenvalues and eigenvectors

w, v = np.linalg.eig(m)

# printing eigen values

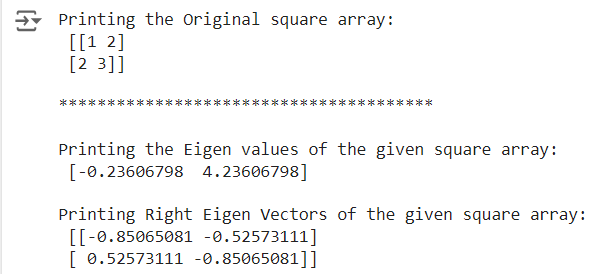
print("Printing the Eigen values of the given square array:\n",w)

print()

# printing eigen vectors

print("Printing Right Eigen Vectors of the given square array:\n",v)

**Output :**



# importing numpy library

import numpy as np

# create numpy 2d-array

m = np.array([[1, 2, 3],[2, 3, 4],[4, 5, 6]])

print("Printing the Original square array:\n",m)

print()

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

print()

# finding eigenvalues and eigenvectors

w, v = np.linalg.eig(m)

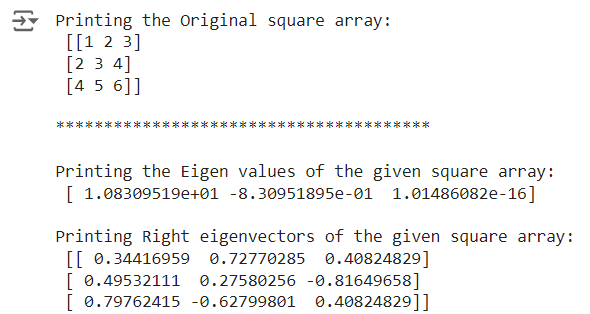
# printing eigen values

print("Printing the Eigen values of the given square array:\n",w)

print()

# printing eigen vectors

print("Printing Right eigenvectors of the given square array:\n",v)



!pip install tensorflow

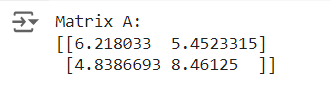
!pip install tensorflow[and-cuda]

import tensorflow as tf

# Let's see how we can compute the eigen vectors and values from a matrix

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

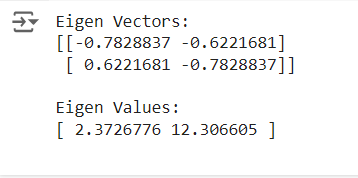
print("Matrix A: \n{}\n\n".format(e\_matrix\_A))



# Calculating the eigen values and vectors using tf.linalg.eigh, if you only want the values you can use eigvalsh

eigen\_values\_A, eigen\_vectors\_A = tf.linalg.eigh(e\_matrix\_A)

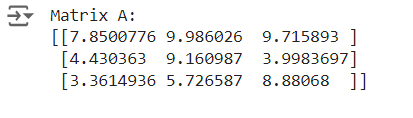
print("Eigen Vectors: \n{} \n\nEigen Values: \n{}\n".format(eigen\_vectors\_A, eigen\_values\_A))



# Let's see how we can compute the eigen vectors and values from a matrix

e\_matrix\_A = tf.random.uniform([3, 3], minval=3, maxval=10, dtype=tf.float32, name="matrixA")

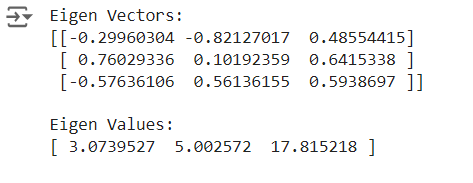
print("Matrix A: \n{}\n\n".format(e\_matrix\_A))



# Calculating the eigen values and vectors using tf.linalg.eigh, if you only want the values you can use eigvalsh

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))



**Practical No 2**

**AIM : Solving XOR problem using deep feed forward network**

Solving the XOR problem using a deep feed-forward neural network (also known as a multilayer perceptron, or MLP) is a classic example of using neural networks for a non-linearly separable problem. The XOR problem cannot be solved using a single layer of neurons but can be solved using a network with at least one hidden layer.

**Code :**

# importing Python library

import numpy as np

# define Unit Step Function

def unitStep(v):

if v >= 0:

return 1

else:

return 0

# design Perceptron Model

def perceptronModel(x, w, b):

v = np.dot(w, x) + b

y = unitStep(v)

return y

# NOT Logic Function

# wNOT = -1, bNOT = 0.5

def NOT\_logicFunction(x):

wNOT = -1

bNOT = 0.5

return perceptronModel(x, wNOT, bNOT)

# AND Logic Function

# here w1 = wAND1 = 1,

# w2 = wAND2 = 1, bAND = -1.5

def AND\_logicFunction(x):

w = np.array([1, 1])

bAND = -1.5

return perceptronModel(x, w, bAND)

# OR Logic Function

# w1 = 1, w2 = 1, bOR = -0.5

def OR\_logicFunction(x):

w = np.array([1, 1])

bOR = -0.5

return perceptronModel(x, w, bOR)

# XOR Logic Function

# with AND, OR and NOT

# function calls in sequence

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

# testing the Perceptron Model

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

test2 = np.array([1, 1])

test3 = np.array([0, 0])

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

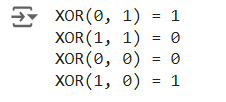
print("XOR({}, {}) = {}".format(0, 1, XOR\_logicFunction(test1)))

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 :**



**Practical No 3**

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

The dataset we will use in this is the Sonar dataset.

This is a dataset that describes sonar chirp returns bouncing off different services.

The 60 input variables are the strength of the returns at different angles.

It is a binary classification problem that requires a model to differentiate rocks from metal cylinders.

It is a well-understood dataset.

All of the variables are continuous and generally in the range of 0 to 1.

The output variable is a string “M” for mine and “R” for rock, which will need to be converted to integers 1 and 0.

A benefit of using this dataset is that it is a standard benchmark problem.

This means that we have some idea of the expected skill of a good model.

Using cross-validation, a neural network should be able to achieve performance around 84% with an upper bound on accuracy for custom models at around 88%.

!pip uninstall tensorflow

!pip install tensorflow==2.12.0

import pandas as pd

from keras.models import Sequential

from keras.layers import Dense

from keras.wrappers.scikit\_learn 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("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

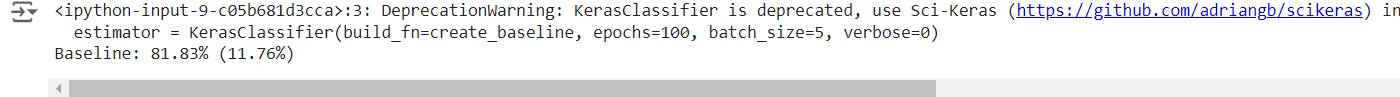
# evaluate model with standardized dataset

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()))

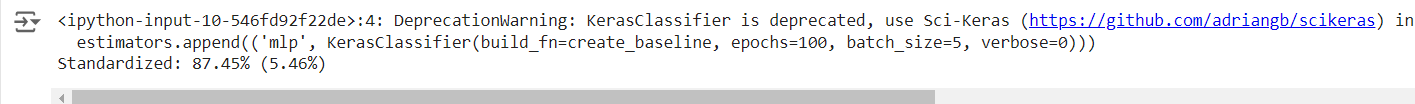
estimators.append(('mlp', KerasClassifier(build\_fn=create\_baseline, epochs=100, batch\_size=5, verbose=0)))

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))



# smaller model

def create\_smaller():

# create model

model = Sequential()

model.add(Dense(30, input\_dim=60, 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()))

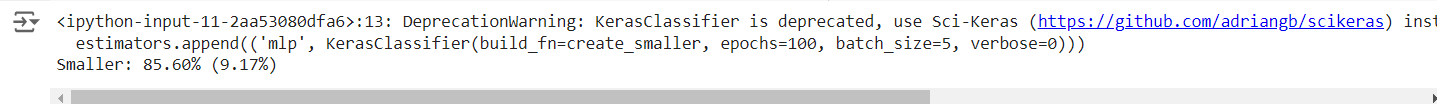
estimators.append(('mlp', KerasClassifier(build\_fn=create\_smaller, epochs=100, batch\_size=5, verbose=0)))

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()))

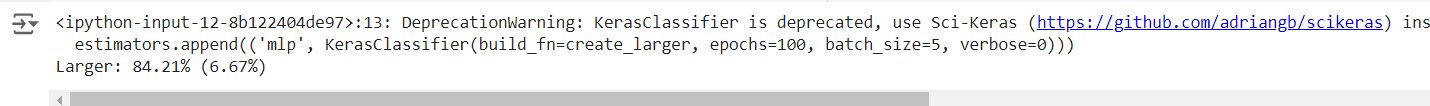
estimators.append(('mlp', KerasClassifier(build\_fn=create\_larger, epochs=100, batch\_size=5, verbose=0)))

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))



**Practical No 4**

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

To build and use a deep feed-forward neural network for performing classification and predicting the class in a general deep learning problem, we'll use TensorFlow and Keras. Let's take a typical classification dataset, such as the Iris dataset, and demonstrate how to build, train, evaluate, and make predictions with a deep feed-forward neural network.

Step-by-Step Implementation

1. Import Libraries
2. Load and Prepare the Data
3. Build the Model
4. Compile the Model
5. Train the Model
6. Evaluate the Model
7. Make Predictions

**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()

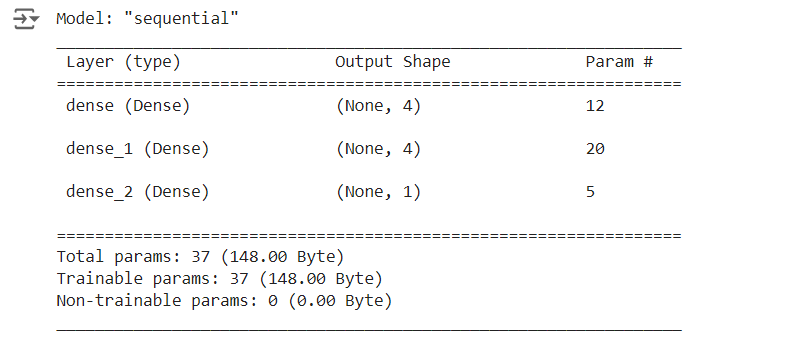
model.add(Dense(4,input\_dim=2,activation='relu'))

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

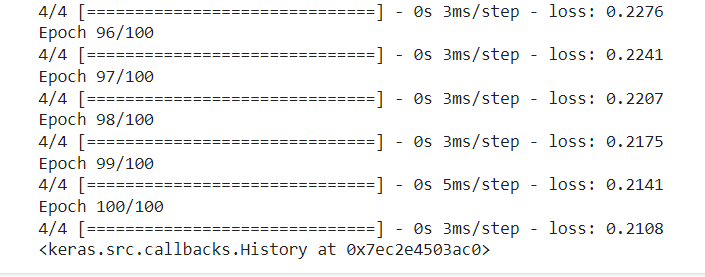
model.add(Dense(1,activation='sigmoid'))

model.compile(loss='binary\_crossentropy',optimizer='adam')

model.summary()



model.fit(X,Y,epochs=100) # u can use 150 epochs also…



Xnew,Yreal=make\_blobs(n\_samples=3,centers=2,n\_features=2,random\_state=1)

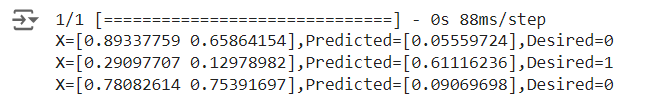
Xnew=scalar.transform(Xnew)

Ynew=model.predict(Xnew)

for i in range(len(Xnew)):

print("X=%s,Predicted=%s,Desired=%s"%(Xnew[i],Ynew[i],Yreal[i]))

**Output :**



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

To perform classification and predict the probability of each class using a deep feed-forward network with two hidden layers, you can follow a similar approach as outlined previously. We will use the Iris dataset as an example and modify the steps to include predicting the probability of each class.

Step-by-Step Implementation

1. Import Libraries
2. Load and Prepare the Data
3. Build the Model
4. Compile the Model
5. Train the Model
6. Evaluate the Model
7. Make Predictions and Predict Probability of Each 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()

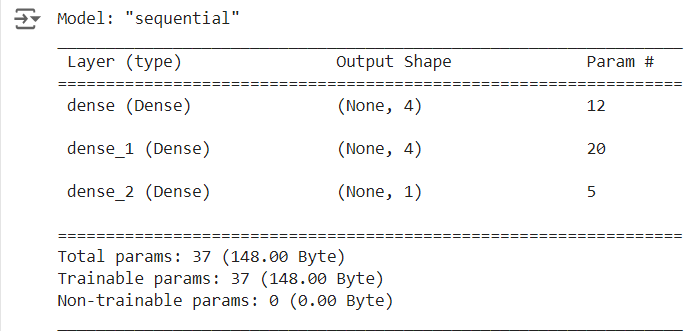
model.add(Dense(4,input\_dim=2,activation='relu'))

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

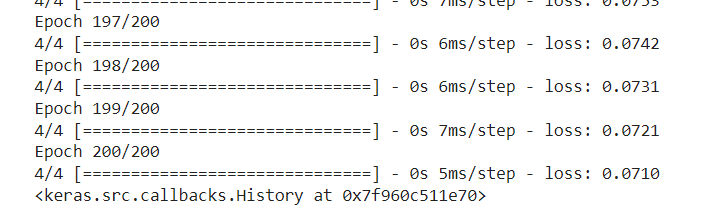
model.add(Dense(1,activation='sigmoid'))

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)

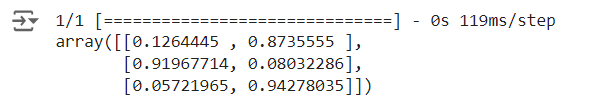
import numpy as np

def predict\_prob(number):

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

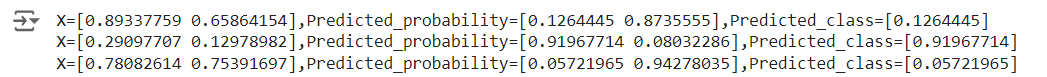
y\_prob = np.array(list(map(predict\_prob, model.predict(Xnew))))

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



**Practical No 5**

**AIM : Evaluating feed forward deep network for regression using KFold cross validation**

Evaluating a feed-forward deep network for regression using KFold cross-validation is a common approach to ensure that your model performs well across different subsets of the data. Here's how you can do this using TensorFlow and Keras.

Step-by-Step Implementation

1. Import Libraries
2. Load and Prepare the Data
3. Define the Model
4. Evaluate 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)))

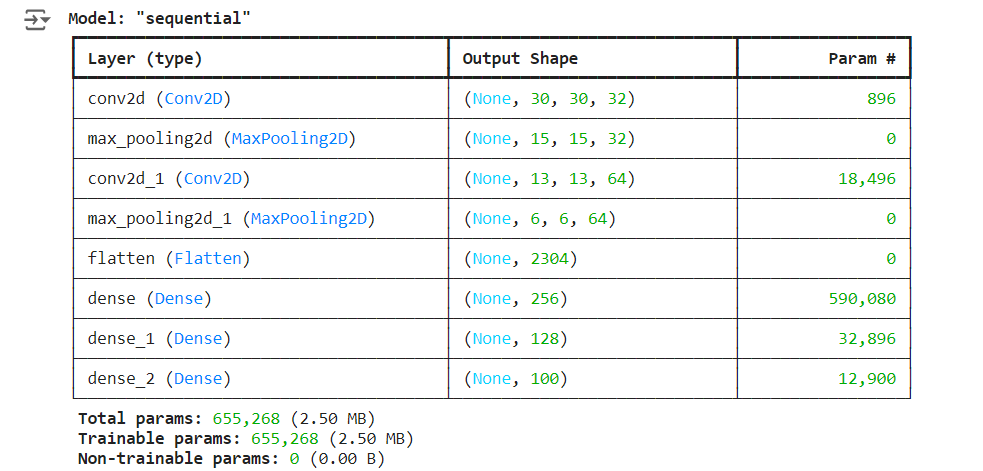
model.add(Flatten())

model.add(Dense(256, activation='relu'))

model.add(Dense(128, activation='relu'))

model.add(Dense(no\_classes, activation='softmax'))

model.summary()

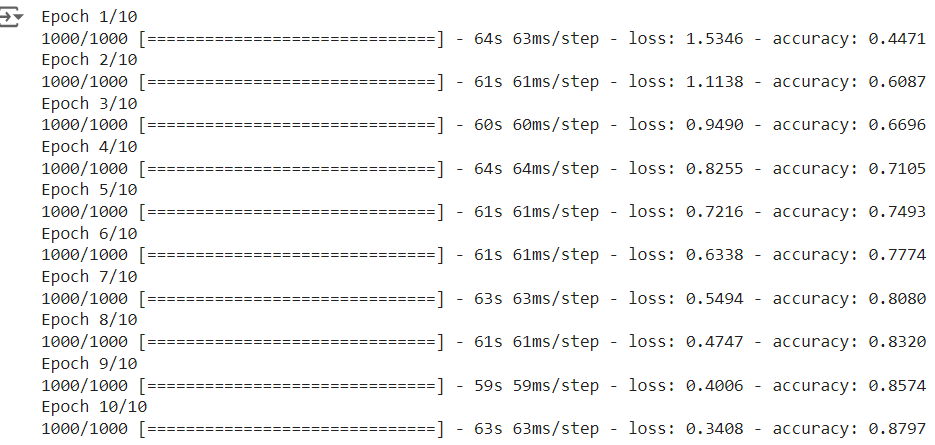


# Compile the model

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

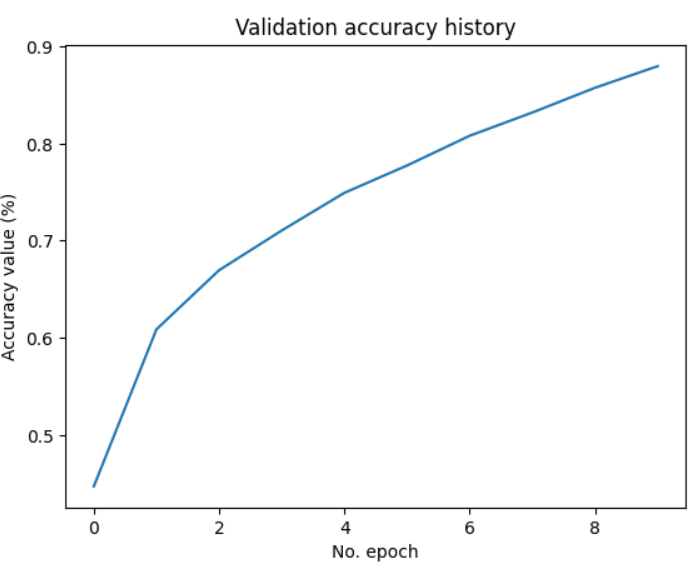
plt.plot(history.history['accuracy'])

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

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)

import tensorflow as tf

from tensorflow.keras.optimizers.legacy import SGD

tf.keras.optimizers.legacy.SGD(learning\_rate=0.1)

# 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'))

optimizer = tf.keras.optimizers.legacy.Adam()

# 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)

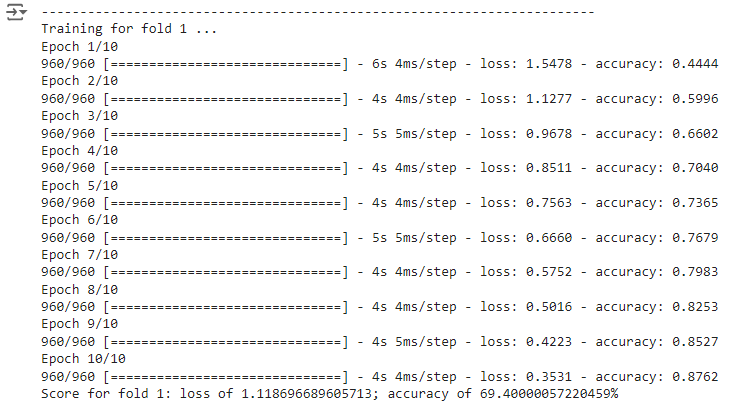
print(f'Score for fold {fold\_no}: {model.metrics\_names[0]} of {scores[0]}; {model.metrics\_names[1]} of {scores[1]\*100}%')

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]}%')

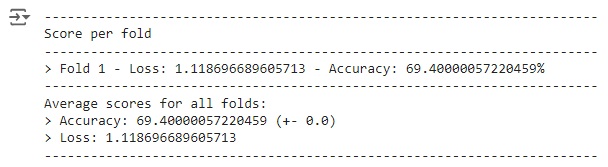
print('------------------------------------------------------------------------')

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('------------------------------------------------------------------------')



**Practical 6**

**AIM : Implementing regularization to avoid overfitting in binary classification using TensorFlow**

Implementing regularization is crucial for avoiding overfitting in deep learning models. For binary classification, common regularization techniques include L2 regularization (also known as weight decay) and dropout.

Step-by-Step Implementation

1. Import Libraries
2. Load and Prepare the Data
3. Build the Model with Regularization
4. Compile the Model
5. Train the Model
6. Evaluate the Model
7. Make Predictions

**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:]

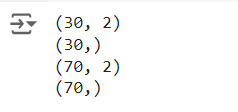
trainY,testY=Y[:n\_train],Y[n\_train:]

print(trainX.shape)

print(trainY.shape)

print(testX.shape)

print(testY.shape)



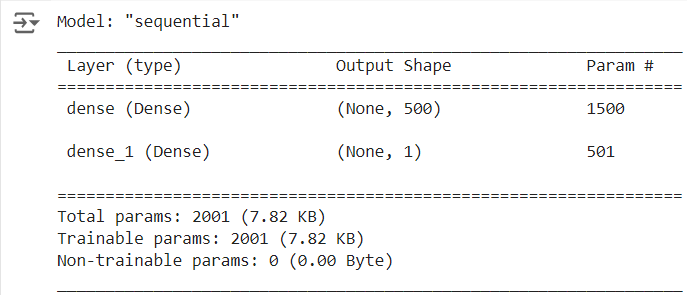
model=Sequential()

model.add(Dense(500,input\_dim=2,activation='relu'))

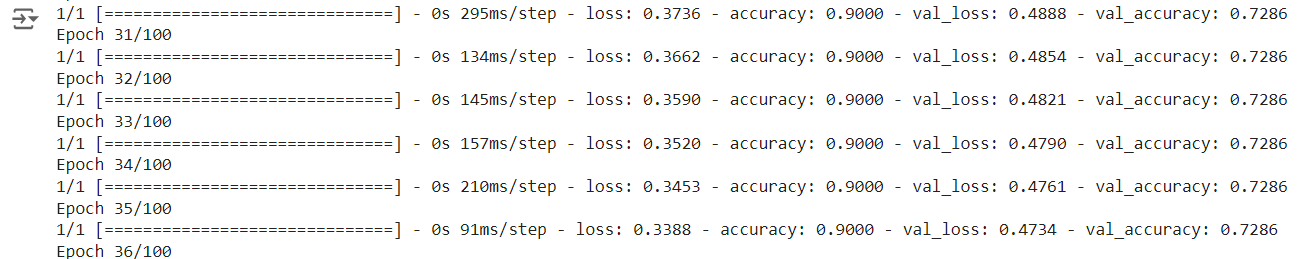
model.add(Dense(1,activation='sigmoid'))

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

model.summary()



history=model.fit(trainX,trainY,validation\_data=(testX,testY),epochs=100)

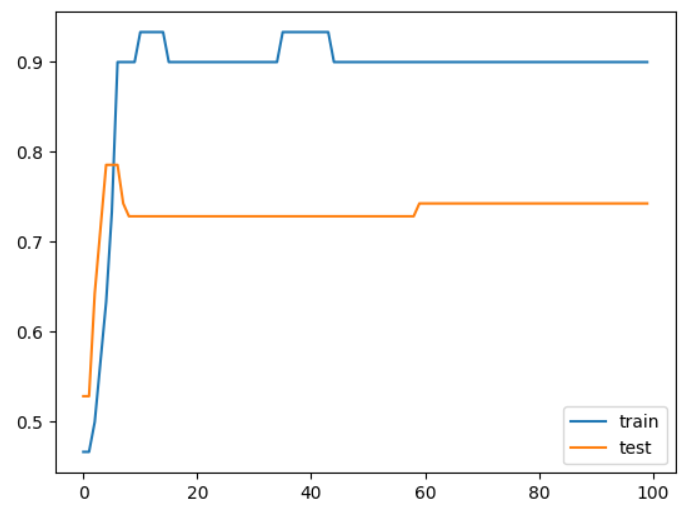


pyplot.plot(history.history['accuracy'],label='train')

pyplot.plot(history.history['val\_accuracy'],label='test')

pyplot.legend()

pyplot.show()



# After 75 epochs it started overfitting by giving same validation accuracy on the test data, so let us use regularization technique

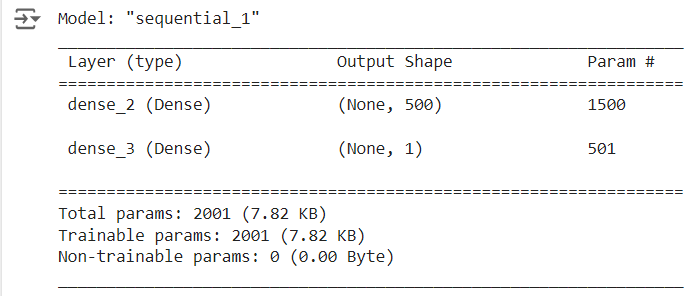
from keras.regularizers import l2

model=Sequential()

model.add(Dense(500,input\_dim=2,activation='relu',kernel\_regularizer=l2(0.001)))

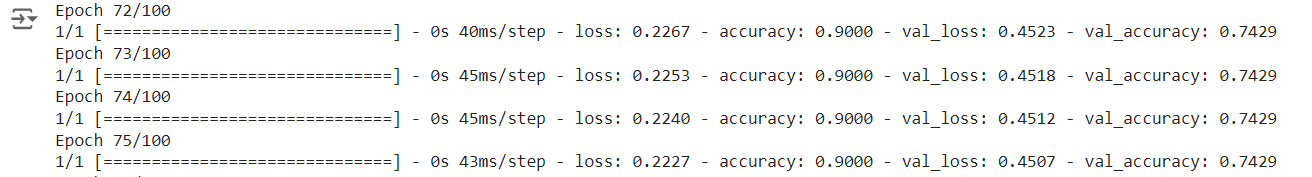
model.add(Dense(1,activation='sigmoid'))

model.summary()



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

history=model.fit(trainX,trainY,validation\_data=(testX,testY),epochs=100)

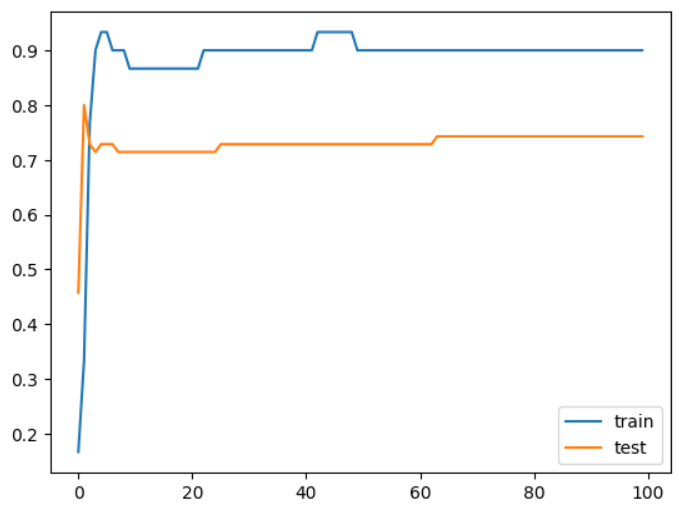


pyplot.plot(history.history['accuracy'],label='train')

pyplot.plot(history.history['val\_accuracy'],label='test')

pyplot.legend()

pyplot.show()



Lets apply l1 and l2 together to the model using below code

from keras.regularizers import l1\_l2

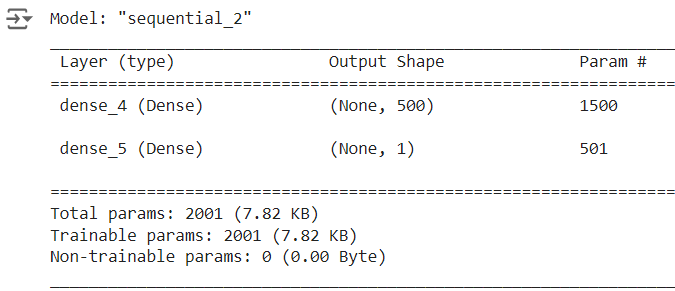
model=Sequential()

model.add(Dense(500,input\_dim=2,activation='relu',kernel\_regularizer=l1\_l2(l1=0.001,l2=0.001)))

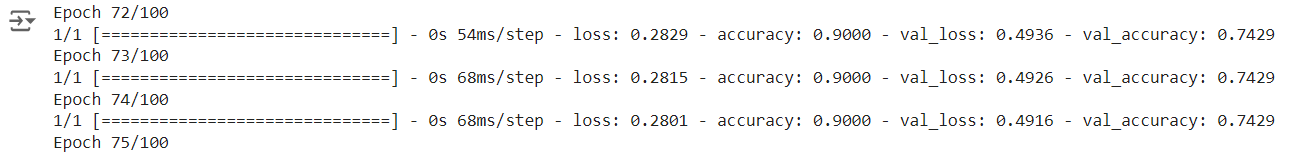
model.add(Dense(1,activation='sigmoid'))

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

model.summary()



history=model.fit(trainX,trainY,validation\_data=(testX,testY),epochs=100)

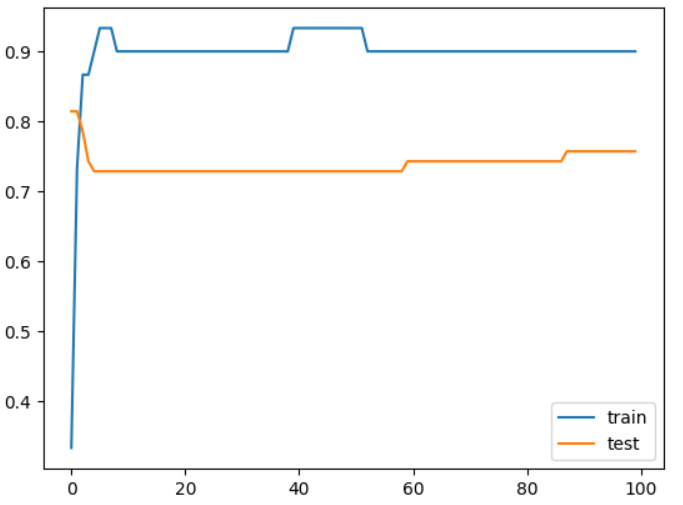


pyplot.plot(history.history['accuracy'],label='train')

pyplot.plot(history.history['val\_accuracy'],label='test')

pyplot.legend()

pyplot.show()



**Practical 7**

**AIM : Implementing Text classification with an RNN**

Implementing text classification with a Recurrent Neural Network (RNN) is a common task in natural language processing (NLP).

Step-by-Step Implementation

1. Import Libraries
2. Load and Prepare the Data
3. Build the RNN Model
4. Compile the Model
5. Train the Model
6. Evaluate the Model
7. Make Predictions

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)

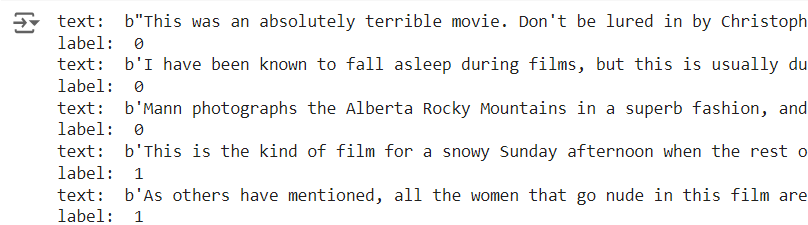
train\_dataset, test\_dataset = dataset['train'], dataset['test']

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)

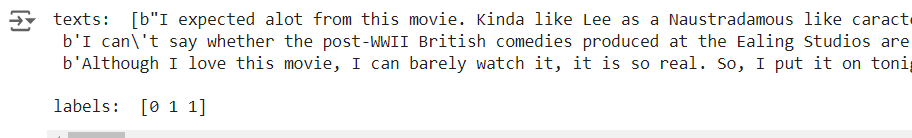
test\_dataset = test\_dataset.batch(BATCH\_SIZE).prefetch(tf.data.AUTOTUNE)

for example, label in train\_dataset.take(1):

print('texts: ', example.numpy()[:3])

print()

print('labels: ', label.numpy()[:3])



Create the text encoder

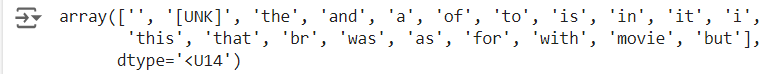
VOCAB\_SIZE = 1000

encoder = tf.keras.layers.TextVectorization(max\_tokens=VOCAB\_SIZE)

encoder.adapt(train\_dataset.map(lambda text, label: text))

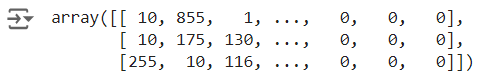
vocab = np.array(encoder.get\_vocabulary())

vocab[:20]



encoded\_example = encoder(example)[:3].numpy()

encoded\_example

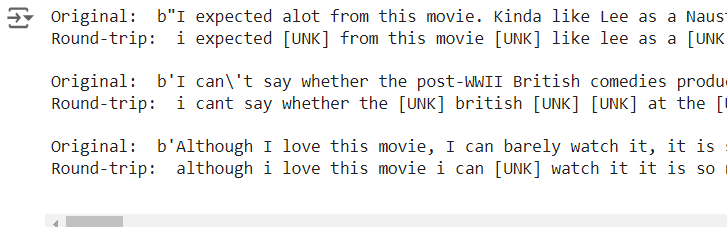


for n in range(3):

print("Original: ", example[n].numpy())

print("Round-trip: ", " ".join(vocab[encoded\_example[n]]))

print()



Create the model

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),

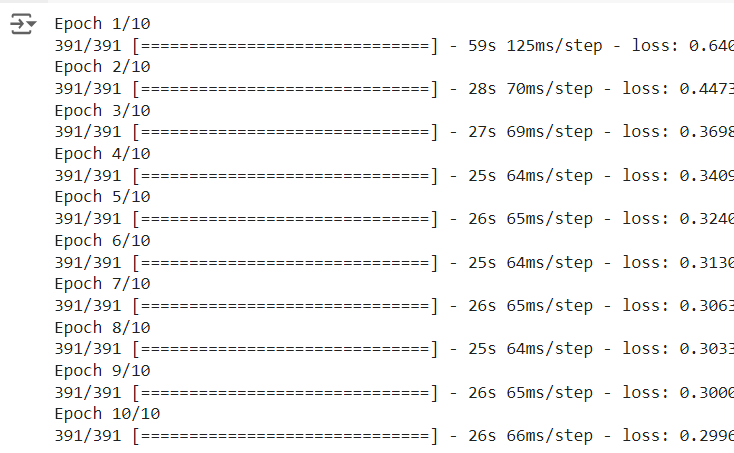
metrics=['accuracy'])

Train the model

history = model.fit(train\_dataset, epochs=10,

validation\_data=test\_dataset,

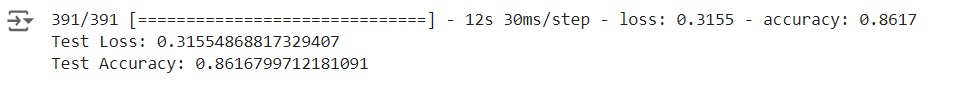
validation\_steps=30)



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)

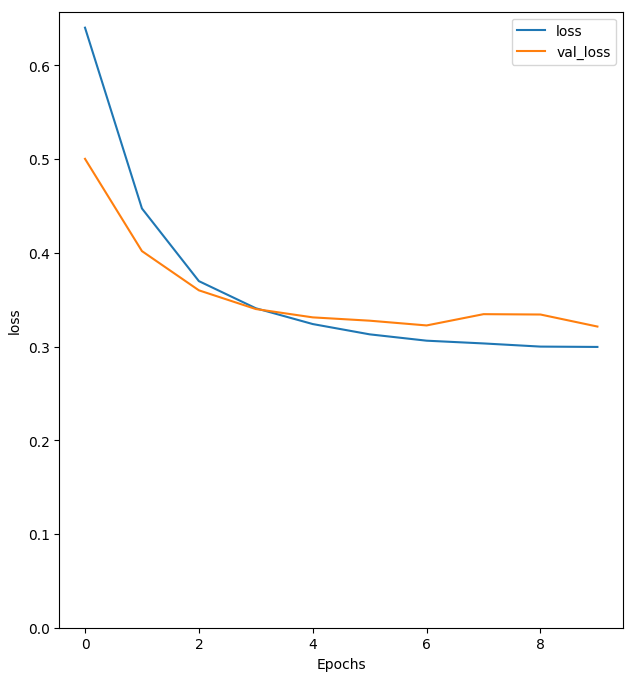
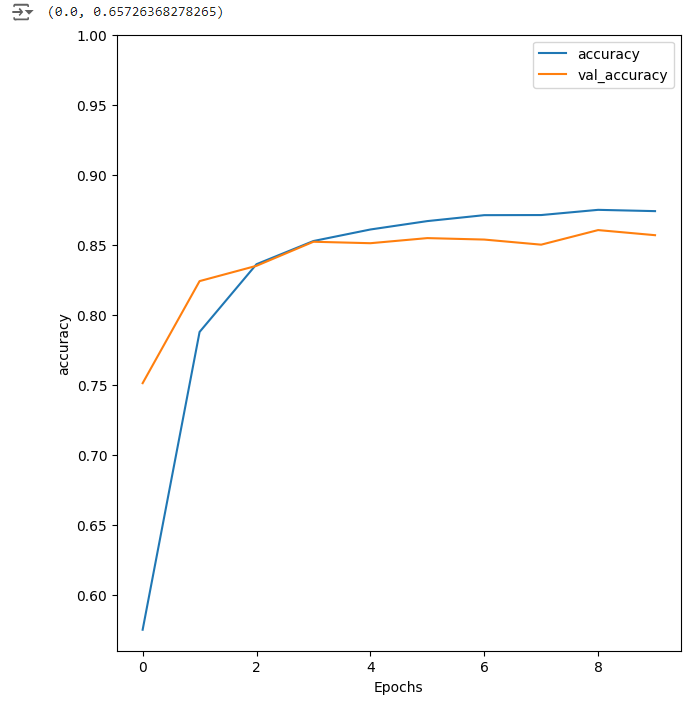
plot\_graphs(history, 'accuracy')

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),

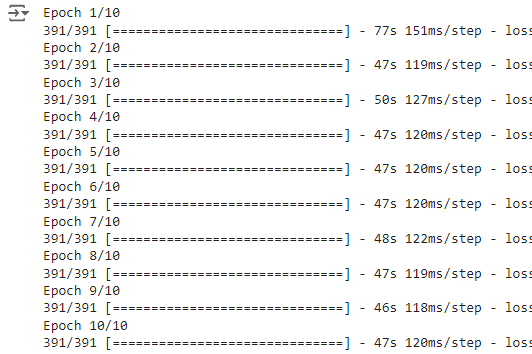
optimizer=tf.keras.optimizers.Adam(1e-4),

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)



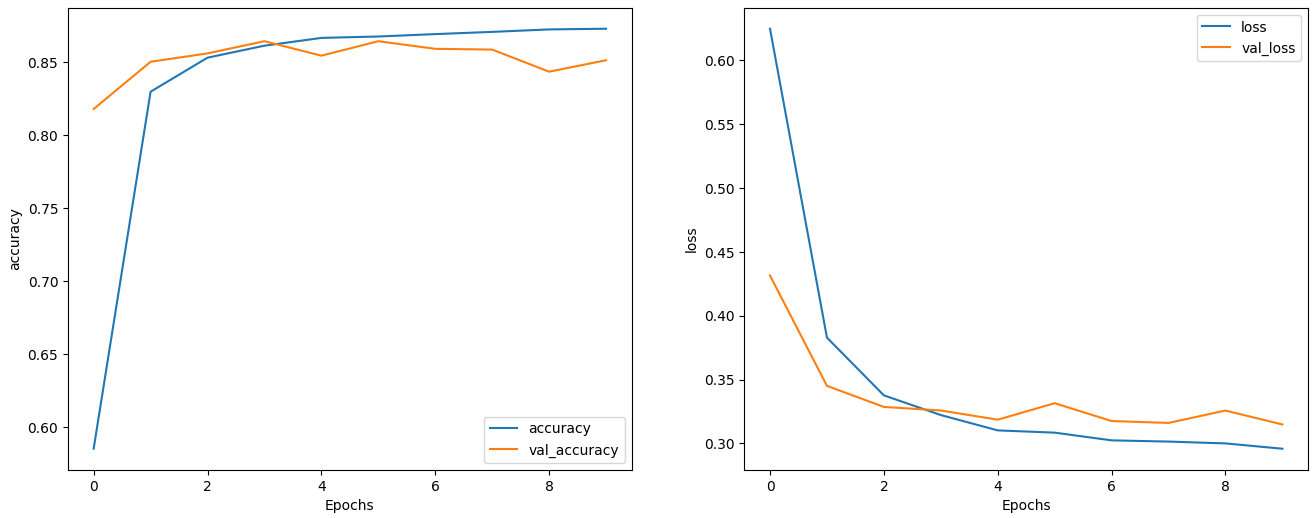
plt.figure(figsize=(16, 6))

plt.subplot(1, 2, 1)

plot\_graphs(history, 'accuracy')

plt.subplot(1, 2, 2)

plot\_graphs(history, 'loss')



**Practical No 8**

**AIM : Implementation of Autoencoders**

Autoencoders are a type of neural network used for unsupervised learning. They are designed to encode input data into a lower-dimensional representation and then reconstruct the data from this representation. This can be useful for tasks like dimensionality reduction, feature learning, and data denoising.

Here’s a step-by-step implementation of a simple autoencoder using TensorFlow and Keras.

Step-by-Step Implementation

1. Import Libraries
2. Load and Prepare the Data
3. Build the Autoencoder Model
4. Compile the Model
5. Train the Model
6. Evaluate the Model
7. Visualize the Results

**Code :**

import keras

from keras import layers

# This is the size of our encoded representations

encoding\_dim = 32 # 32 floats -> compression of factor 24.5, assuming the input is 784 floats

# 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)

# This model maps an input to its reconstruction

autoencoder = keras.Model(input\_img, decoded)

#Let's also create a separate encoder model:

# This model maps an input to its encoded representation

encoder = keras.Model(input\_img, encoded)

# This is our encoded (32-dimensional) input

encoded\_input = keras.Input(shape=(encoding\_dim,))

# Retrieve 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))

#Now let's train our autoencoder to reconstruct MNIST digits.

#First, we'll configure our model to use a per-pixel binary crossentropy loss, and the Adam optimizer:

autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

#Let's prepare our input data. We're using MNIST digits, and we're discarding the labels (since we're only interested in encoding/decoding the input images).

from keras.datasets import mnist

import numpy as np

(x\_train, \_), (x\_test, \_) = mnist.load\_data()



# We will normalize all values between 0 and 1 and we will flatten the 28x28 images into vectors of size 784.

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)



# Now let's train our autoencoder for 50 epochs:

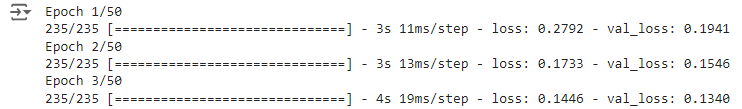
autoencoder.fit(x\_train, x\_train,

epochs=50,

batch\_size=256,

shuffle=True,

validation\_data=(x\_test, x\_test))



# Encode and decode some digits

# Note that we take them from the \*test\* set

encoded\_imgs = encoder.predict(x\_test)

decoded\_imgs = decoder.predict(encoded\_imgs)



# Use Matplotlib

import matplotlib.pyplot as plt

n = 10 # How many digits we will display

plt.figure(figsize=(20, 4))

for i in range(n):

# Display original

ax = plt.subplot(2, n, 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 reconstruction

ax = plt.subplot(2, n, i + 1 + n)

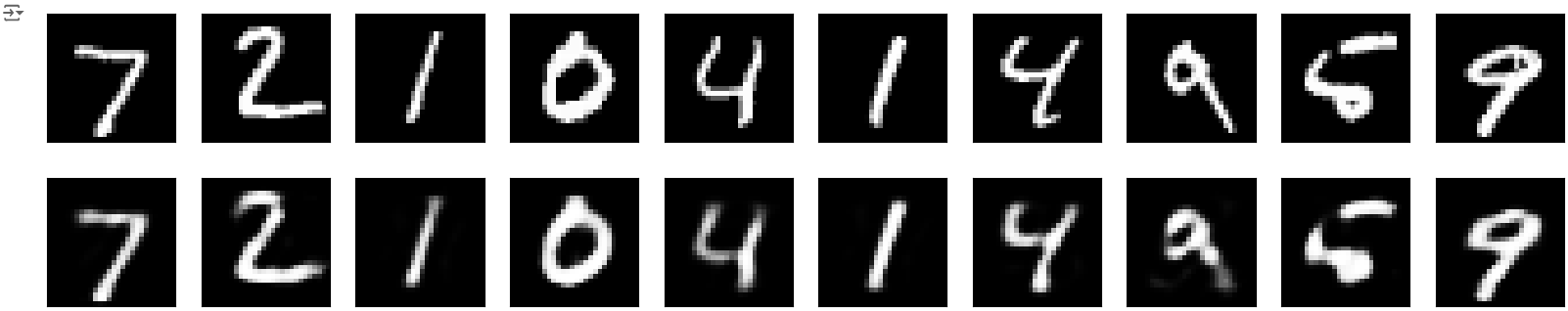
plt.imshow(decoded\_imgs[i].reshape(28, 28))

plt.gray()

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

plt.show()



**Adding a sparsity constraint on the encoded representations**

In the previous example, the representations were only constrained by the size of the hidden layer (32). In such a situation, what typically happens is that the hidden layer is learning an approximation of PCA (principal component analysis). But another way to constrain the representations to be compact is to add a sparsity contraint on the activity of the hidden representations, so fewer units would "fire" at a given time. In Keras, this can be done by adding an activity\_regularizer to our Dense layer:

from keras import regularizers

encoding\_dim = 32

input\_img = keras.Input(shape=(784,))

# Add a Dense layer with a L1 activity regularizer

encoded = layers.Dense(encoding\_dim, activation='relu',

activity\_regularizer=regularizers.l1(10e-5))(input\_img)

decoded = layers.Dense(784, activation='sigmoid')(encoded)

autoencoder = keras.Model(input\_img, decoded)

#Let's also create a separate encoder model:

# This model maps an input to its encoded representation

encoder = keras.Model(input\_img, encoded)

# This is our encoded (32-dimensional) input

encoded\_input = keras.Input(shape=(encoding\_dim,))

# Retrieve 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))

#Now let's train our autoencoder to reconstruct MNIST digits.

#First, we'll configure our model to use a per-pixel binary crossentropy loss, and the Adam optimizer:

autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

#Let's prepare our input data. We're using MNIST digits, and we're discarding the labels (since we're only interested in encoding/decoding the input images).

from keras.datasets import mnist

import numpy as np

(x\_train, \_), (x\_test, \_) = mnist.load\_data()

# We will normalize all values between 0 and 1 and we will flatten the 28x28 images into vectors of size 784.

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)



# Now let's train our autoencoder for 50 epochs:

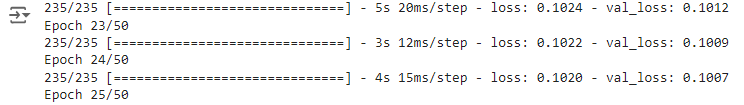
autoencoder.fit(x\_train, x\_train,

epochs=50,

batch\_size=256,

shuffle=True,

validation\_data=(x\_test, x\_test))



# Encode and decode some digits

# Note that we take them from the \*test\* set

encoded\_imgs = encoder.predict(x\_test)

decoded\_imgs = decoder.predict(encoded\_imgs)



# Use Matplotlib

import matplotlib.pyplot as plt

n = 10 # How many digits we will display

plt.figure(figsize=(20, 4))

for i in range(n):

# Display original

ax = plt.subplot(2, n, 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 reconstruction

ax = plt.subplot(2, n, i + 1 + n)

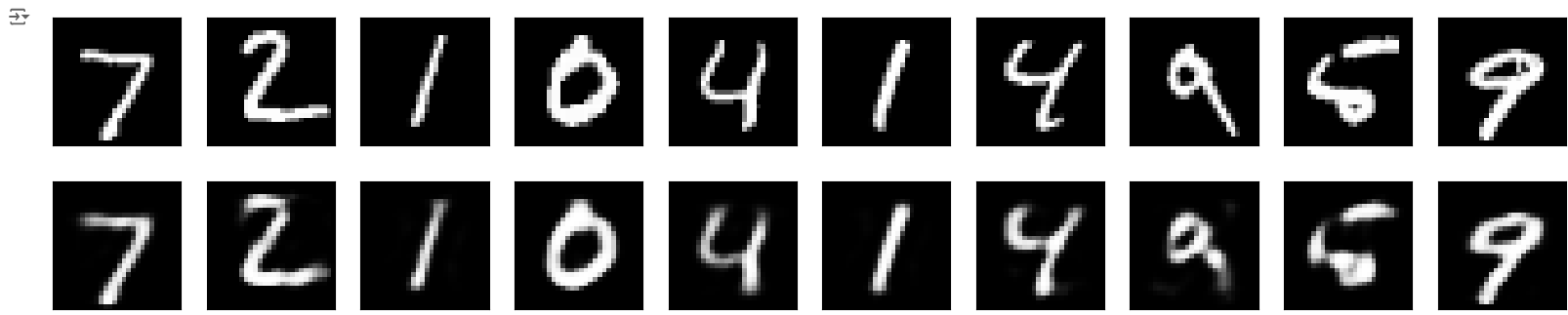
plt.imshow(decoded\_imgs[i].reshape(28, 28))

plt.gray()

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

plt.show()



**Deep autoencoder**

We do not have to limit ourselves to a single layer as encoder or decoder, we could instead use a stack of layers, such as:

input\_img = keras.Input(shape=(784,))

encoded = layers.Dense(128, activation='relu')(input\_img)

encoded = layers.Dense(64, activation='relu')(encoded)

encoded = layers.Dense(32, activation='relu')(encoded)

decoded = layers.Dense(64, activation='relu')(encoded)

decoded = layers.Dense(128, activation='relu')(decoded)

decoded = layers.Dense(784, activation='sigmoid')(decoded)

autoencoder = keras.Model(input\_img, decoded)

autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

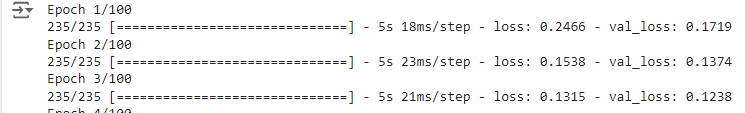
autoencoder.fit(x\_train, x\_train,

epochs=100,

batch\_size=256,

shuffle=True,

validation\_data=(x\_test, x\_test))



# Encode and decode some digits

# Note that we take them from the \*test\* set

encoded\_imgs = encoder.predict(x\_test)

decoded\_imgs = decoder.predict(encoded\_imgs)



# Use Matplotlib

import matplotlib.pyplot as plt

n = 10 # How many digits we will display

plt.figure(figsize=(20, 4))

for i in range(n):

# Display original

ax = plt.subplot(2, n, 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 reconstruction

ax = plt.subplot(2, n, i + 1 + n)

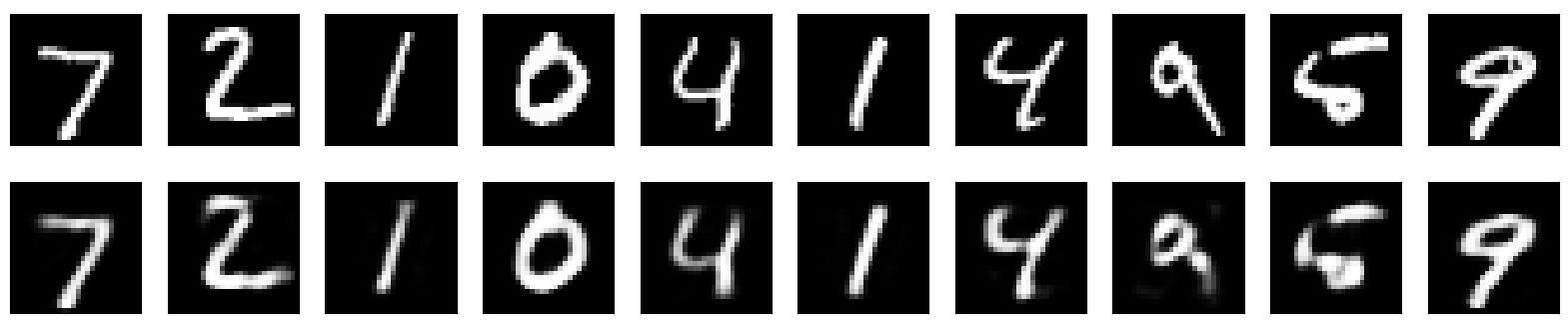
plt.imshow(decoded\_imgs[i].reshape(28, 28))

plt.gray()

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

plt.show()



**Practical No 9**

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

Implementing a Convolutional Neural Network (CNN) to predict numbers from images is a classic task in deep learning, often referred to as digit recognition. We'll use the MNIST dataset, which consists of 28x28 grayscale images of handwritten digits (0 to 9).

Step-by-Step Implementation

1. Import Libraries
2. Load and Prepare the Data
3. Build the CNN Model
4. Compile the Model
5. Train the Model
6. Evaluate the Model
7. Make Predictions

**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

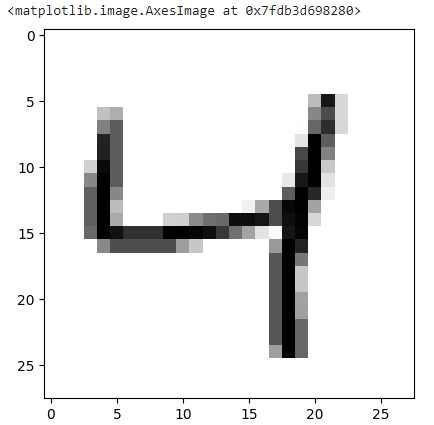
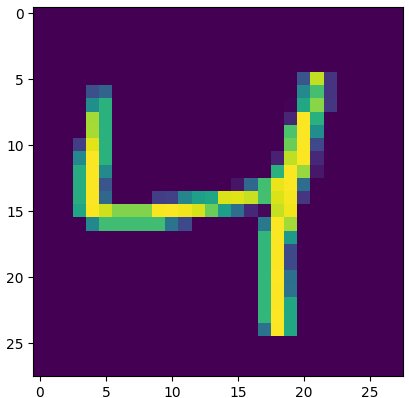


import matplotlib.pyplot as plt

plt.imshow(X\_train[2])

plt.show()

plt.imshow(X\_train[2], cmap=plt.cm.binary)



X\_train[2]

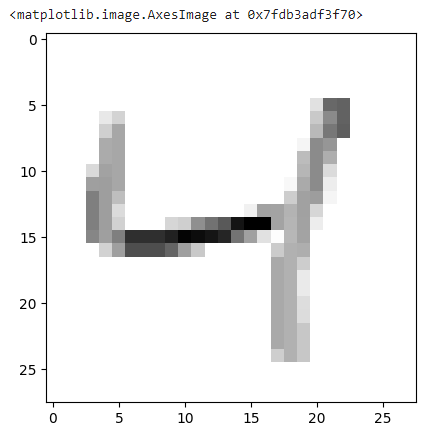


**Normalizing the data**

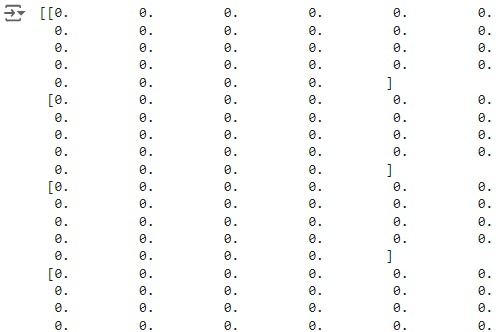
X\_train = tf.keras.utils.normalize(X\_train, axis=1)

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

## Model

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)

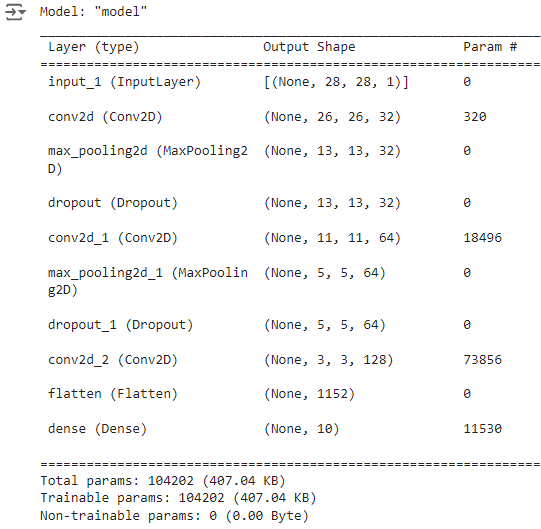
f = KL.Flatten()(c)

outputs = KL.Dense(10, activation=tf.nn.softmax)(f)

model = KM.Model(inputs, outputs)

model.summary()

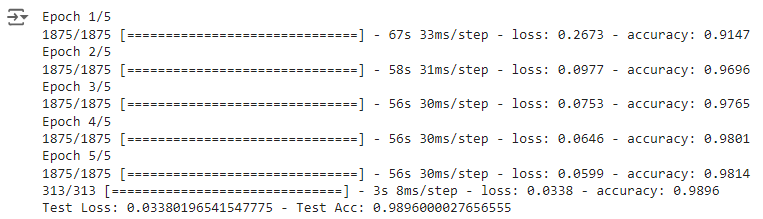
model.compile(optimizer="adam", loss="sparse\_categorical\_crossentropy", metrics=["accuracy"])



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))



**Practical No 10**

**AIM : Implementing Denoising of images using Autoencoder**

Implementing denoising of images using an autoencoder involves training an autoencoder to remove noise from input images. Here's how you can do it step by step using TensorFlow and Keras:

Step-by-Step Implementation

1. Import Libraries
2. Load and Prepare the Data
3. Add Noise to the Images
4. Build the Autoencoder Model
5. Compile the Model
6. Train the Model
7. Evaluate the Model
8. Denoise Images

**Code :**

%matplotlib inline

%config InlineBackend.figure\_format = 'retina'

import matplotlib.pyplot as plt

import pandas as pd

import numpy as np

import seaborn as sns

import warnings

warnings.filterwarnings('ignore')

from \_\_future\_\_ import print\_function

from keras.models import Model

from keras.layers import Dense, Input

from keras.datasets import mnist

from keras.regularizers import l1

from keras.optimizers import Adam

**Utility Functions**

def plot\_autoencoder\_outputs(autoencoder, n, dims):

decoded\_imgs = autoencoder.predict(x\_test)

# number of example digits to show

n = 5

plt.figure(figsize=(10, 4.5))

for i in range(n):

# plot original image

ax = plt.subplot(2, n, i + 1)

plt.imshow(x\_test[i].reshape(\*dims))

plt.gray()

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

if i == n/2:

ax.set\_title('Original Images')

# plot reconstruction

ax = plt.subplot(2, n, i + 1 + n)

plt.imshow(decoded\_imgs[i].reshape(\*dims))

plt.gray()

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

if i == n/2:

ax.set\_title('Reconstructed Images')

plt.show()

def plot\_loss(history):

historydf = pd.DataFrame(history.history, index=history.epoch)

plt.figure(figsize=(8, 6))

historydf.plot(ylim=(0, historydf.values.max()))

plt.title('Loss: %.3f' % history.history['loss'][-1])

def plot\_compare\_histories(history\_list, name\_list, plot\_accuracy=True):

dflist = []

min\_epoch = len(history\_list[0].epoch)

losses = []

for history in history\_list:

h = {key: val for key, val in history.history.items() if not key.startswith('val\_')}

dflist.append(pd.DataFrame(h, index=history.epoch))

min\_epoch = min(min\_epoch, len(history.epoch))

losses.append(h['loss'][-1])

historydf = pd.concat(dflist, axis=1)

metrics = dflist[0].columns

idx = pd.MultiIndex.from\_product([name\_list, metrics], names=['model', 'metric'])

historydf.columns = idx

plt.figure(figsize=(6, 8))

ax = plt.subplot(211)

historydf.xs('loss', axis=1, level='metric').plot(ylim=(0,1), ax=ax)

plt.title("Training Loss: " + ' vs '.join([str(round(x, 3)) for x in losses]))

if plot\_accuracy:

ax = plt.subplot(212)

historydf.xs('acc', axis=1, level='metric').plot(ylim=(0,1), ax=ax)

plt.title("Accuracy")

plt.xlabel("Epochs")

plt.xlim(0, min\_epoch-1)

plt.tight\_layout()

**Deep Autoencoder**

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

x\_train = x\_train.astype('float32') / 255.0

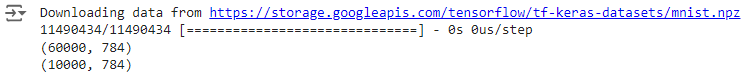
x\_test = x\_test.astype('float32') / 255.0

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)



input\_size = 784

hidden\_size = 128

code\_size = 32

input\_img = Input(shape=(input\_size,))

hidden\_1 = Dense(hidden\_size, activation='relu')(input\_img)

code = Dense(code\_size, activation='relu')(hidden\_1)

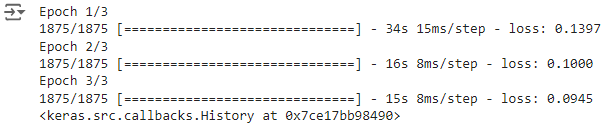
hidden\_2 = Dense(hidden\_size, activation='relu')(code)

output\_img = Dense(input\_size, activation='sigmoid')(hidden\_2)

autoencoder = Model(input\_img, output\_img)

autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

autoencoder.fit(x\_train, x\_train, epochs=3)



plot\_autoencoder\_outputs(autoencoder, 5, (28, 28))



weights = autoencoder.get\_weights()[0].T

n = 10

plt.figure(figsize=(20, 5))

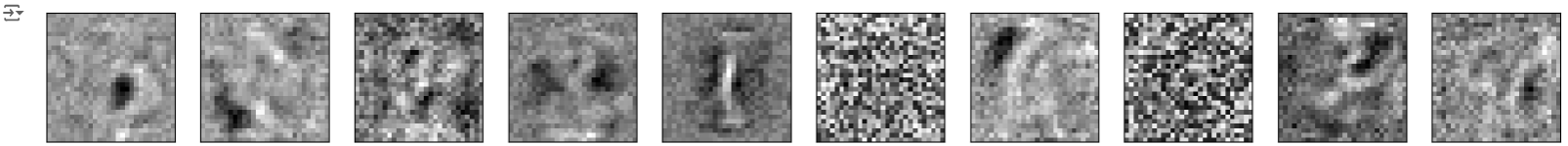
for i in range(n):

ax = plt.subplot(1, n, i + 1)

plt.imshow(weights[i+0].reshape(28, 28))

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)



**Shallow Autoencoder**

input\_size = 784

code\_size = 32

input\_img = Input(shape=(input\_size,))

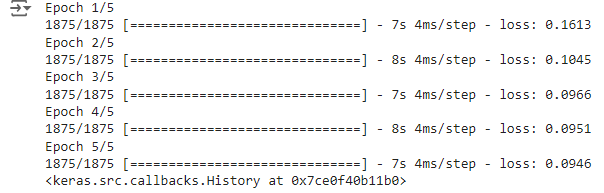
code = Dense(code\_size, activation='relu')(input\_img)

output\_img = Dense(input\_size, activation='sigmoid')(code)

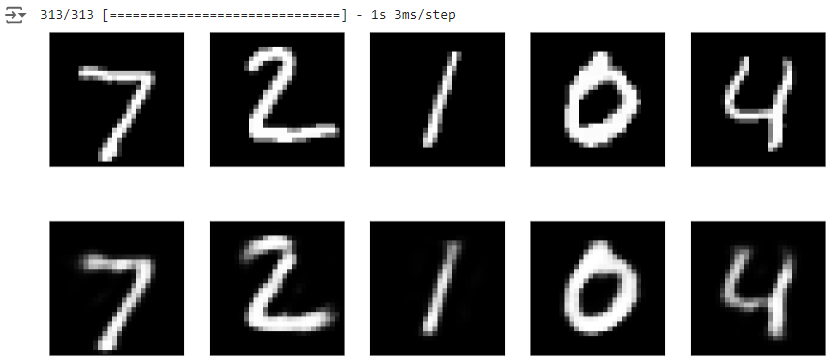
autoencoder = Model(input\_img, output\_img)

autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

autoencoder.fit(x\_train, x\_train, epochs=5)



plot\_autoencoder\_outputs(autoencoder, 5, (28, 28))



weights = autoencoder.get\_weights()[0].T

n = 10

plt.figure(figsize=(20, 5))

for i in range(n):

ax = plt.subplot(1, n, i + 1)

plt.imshow(weights[i+20].reshape(28, 28))

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

Denoising Autoencoder

noise\_factor = 0.4

x\_train\_noisy = x\_train + noise\_factor \* np.random.normal(size=x\_train.shape)

x\_test\_noisy = x\_test + noise\_factor \* np.random.normal(size=x\_test.shape)

x\_train\_noisy = np.clip(x\_train\_noisy, 0.0, 1.0)

x\_test\_noisy = np.clip(x\_test\_noisy, 0.0, 1.0)

n = 5

plt.figure(figsize=(10, 4.5))

for i in range(n):

# plot original image

ax = plt.subplot(2, n, i + 1)

plt.imshow(x\_test[i].reshape(28, 28))

plt.gray()

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

if i == n/2:

ax.set\_title('Original Images')

# plot noisy image

ax = plt.subplot(2, n, i + 1 + n)

plt.imshow(x\_test\_noisy[i].reshape(28, 28))

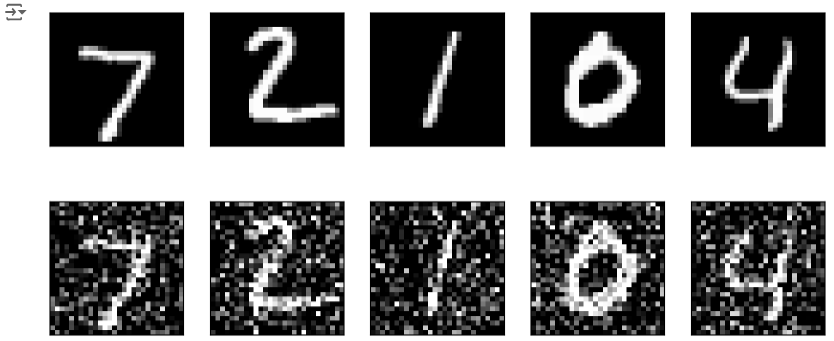
plt.gray()

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

if i == n/2:

ax.set\_title('Noisy Input')



input\_size = 784

hidden\_size = 128

code\_size = 32

input\_img = Input(shape=(input\_size,))

hidden\_1 = Dense(hidden\_size, activation='relu')(input\_img)

code = Dense(code\_size, activation='relu')(hidden\_1)

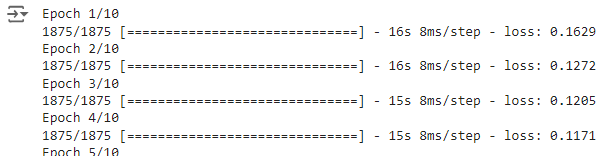
hidden\_2 = Dense(hidden\_size, activation='relu')(code)

output\_img = Dense(input\_size, activation='sigmoid')(hidden\_2)

autoencoder = Model(input\_img, output\_img)

autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

autoencoder.fit(x\_train\_noisy, x\_train, epochs=10)



n = 5

plt.figure(figsize=(10, 7))

images = autoencoder.predict(x\_test\_noisy)

for i in range(n):

# plot original image

ax = plt.subplot(3, n, i + 1)

plt.imshow(x\_test[i].reshape(28, 28))

plt.gray()

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

if i == n/2:

ax.set\_title('Original Images')

# plot noisy image

ax = plt.subplot(3, n, i + 1 + n)

plt.imshow(x\_test\_noisy[i].reshape(28, 28))

plt.gray()

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

if i == n/2:

ax.set\_title('Noisy Input')

# plot noisy image

ax = plt.subplot(3, n, i + 1 + 2\*n)

plt.imshow(images[i].reshape(28, 28))

plt.gray()

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

if i == n/2:

ax.set\_title('Autoencoder Output')

