|  |  |
| --- | --- |
| **Name: Nitin Magdum** | **PRN NO: 19UEL325** |
| **Class: Final Year (Electronics Engineering)** | **Batch: FB4** |

**Experiment No-01**

**Design of McCulloch-Pitts Neuron Model in python**

**Code-**

from numpy import exp, dot, random, array

def initialize\_weights():

random.seed(1)

synaptic\_weights = random.uniform(low=-1, high=1, size=(3, 1))

return synaptic\_weights

def sigmoid(x):

return 1 / (1 + exp(-x))

def sigmoid\_derivative(x):

return x \* (1 - x)

def train(inputs, expected\_output, synaptic\_weights, bias, learning\_rate, training\_iterations):

for epoch in range(training\_iterations):

predicted\_output = learn(inputs, synaptic\_weights, bias)

error = sigmoid\_derivative(predicted\_output) \* (expected\_output - predicted\_output)

weight\_factor = dot(inputs.T, error) \* learning\_rate

bias\_factor = error \* learning\_rate

synaptic\_weights += weight\_factor

bias += bias\_factor

if ((epoch % 10) == 0):

print("Epoch", epoch)

print("Predicted Output = ", predicted\_output.T)

print("Expected Output = ", expected\_output.T)

print()

return synaptic\_weights

def learn(inputs, synaptic\_weights, bias):

return sigmoid(dot(inputs, synaptic\_weights) + bias)

if \_\_name\_\_ == "\_\_main\_\_":

synaptic\_weights = initialize\_weights()

inputs = array([[0, 1, 1],

[1, 0, 0],

[1, 0, 1]])

expected\_output = array([[1, 0, 1]]).T

test = array([1, 0, 1])

trained\_weights = train(inputs, expected\_output, synaptic\_weights, bias=0.01, learning\_rate=0.98,

training\_iterations=100)

accuracy = (learn(test, trained\_weights, bias=0.01)) \* 100

print("accuracy =", accuracy[0], "%")

Output-

Epoch 0

Predicted Output = [[0.36606807 0.46108984 0.23944453]]

Expected Output = [[1 0 1]]

Epoch 10

Predicted Output = [[0.88156946 0.27505523 0.77746895]]

Expected Output = [[1 0 1]]

Epoch 20

Predicted Output = [[0.9253287 0.18449832 0.85137421]]

Expected Output = [[1 0 1]]

Epoch 30

Predicted Output = [[0.94205725 0.14442224 0.88248086]]

Expected Output = [[1 0 1]]

Epoch 40

Predicted Output = [[0.95127929 0.12156497 0.90043602]]

Expected Output = [[1 0 1]]

Epoch 50

Predicted Output = [[0.9572632 0.10653564 0.91237607]]

Expected Output = [[1 0 1]]

Epoch 60

Predicted Output = [[0.96152527 0.0957633 0.92100566]]

Expected Output = [[1 0 1]]

Epoch 70

Predicted Output = [[0.96474996 0.08758666 0.92759626]]

Expected Output = [[1 0 1]]

Epoch 80

Predicted Output = [[0.96729517 0.08112231 0.93283113]]

Expected Output = [[1 0 1]]

Epoch 90

Predicted Output = [[0.96936789 0.07585408 0.937113 ]]

Expected Output = [[1 0 1]]

accuracy = 73.57932132581824 %

|  |  |
| --- | --- |
| **Name: Nitin Magdum** | **PRN NO: 19UEL325** |
| **Class: Final Year (Electronics Engineering)** | **Batch: FB4** |

**Experiment No-02**

**Implementation of XOR gates using McCulloch-Pitts Artificial Neuron Model in python**

**Code-**

import numpy as np

from matplotlib import pyplot as plt

def sigmoid(z):

return 1 / (1 + np.exp(-z))

def initializeParameters(inputFeatures, neuronsInHiddenLayers, outputFeatures):

W1 = np.random.randn(neuronsInHiddenLayers, inputFeatures)

W2 = np.random.randn(outputFeatures, neuronsInHiddenLayers)

b1 = np.zeros((neuronsInHiddenLayers, 1))

b2 = np.zeros((outputFeatures, 1))

parameters = {"W1" : W1, "b1": b1,

"W2" : W2, "b2": b2}

return parameters

def forwardPropagation(X, Y, parameters):

m = X.shape[1]

W1 = parameters["W1"]

W2 = parameters["W2"]

b1 = parameters["b1"]

b2 = parameters["b2"]

Z1 = np.dot(W1, X) + b1

A1 = sigmoid(Z1)

Z2 = np.dot(W2, A1) + b2

A2 = sigmoid(Z2)

cache = (Z1, A1, W1, b1, Z2, A2, W2, b2)

logprobs = np.multiply(np.log(A2), Y) + np.multiply(np.log(1 - A2), (1 - Y))

cost = -np.sum(logprobs) / m

return cost, cache, A2

def backwardPropagation(X, Y, cache):

m = X.shape[1]

(Z1, A1, W1, b1, Z2, A2, W2, b2) = cache

dZ2 = A2 - Y

dW2 = np.dot(dZ2, A1.T) / m

db2 = np.sum(dZ2, axis = 1, keepdims = True)

dA1 = np.dot(W2.T, dZ2)

dZ1 = np.multiply(dA1, A1 \* (1- A1))

dW1 = np.dot(dZ1, X.T) / m

db1 = np.sum(dZ1, axis = 1, keepdims = True) / m

gradients = {"dZ2": dZ2, "dW2": dW2, "db2": db2,

"dZ1": dZ1, "dW1": dW1, "db1": db1}

return gradients

def updateParameters(parameters, gradients, learningRate):

parameters["W1"] = parameters["W1"] - learningRate \* gradients["dW1"]

parameters["W2"] = parameters["W2"] - learningRate \* gradients["dW2"]

parameters["b1"] = parameters["b1"] - learningRate \* gradients["db1"]

parameters["b2"] = parameters["b2"] - learningRate \* gradients["db2"]

return parameters

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

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

neuronsInHiddenLayers = 2

inputFeatures = X.shape[0]

outputFeatures = Y.shape[0]

parameters = initializeParameters(inputFeatures, neuronsInHiddenLayers, outputFeatures)

epoch = 100000

learningRate = 0.01

losses = np.zeros((epoch, 1))

for i in range(epoch):

losses[i, 0], cache, A2 = forwardPropagation(X, Y, parameters)

gradients = backwardPropagation(X, Y, cache)

parameters = updateParameters(parameters, gradients, learningRate)

# Evaluating the performance

plt.figure()

plt.plot(losses)

plt.xlabel("EPOCHS")

plt.ylabel("Loss value")

plt.show()

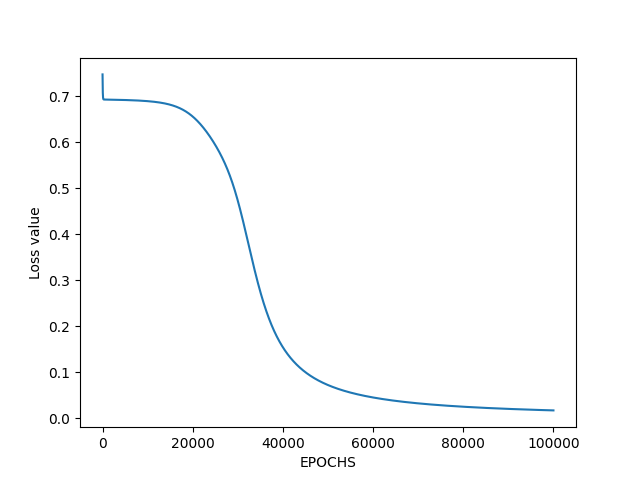
X1= np.array([[1, 0, 0, 0], [0, 1, 0, 1]])

cost, \_, A2 = forwardPropagation(X1, Y, parameters)

prediction = (A2 > 0.5) \* 1.0

print(prediction)

**Output-**

****

[[1. 1. 0. 1.]]

|  |  |
| --- | --- |
| **Name: Nitin Magdum** | **PRN NO: 19UEL325** |
| **Class: Final Year (Electronics Engineering)** | **Batch: FB4** |

**Experiment No-03**

**Write a python program to implement gradient descent algorithm**

**Code-**

import numpy as np

import matplotlib.pyplot as plt

def mean\_squared\_error(y\_true, y\_predicted):

cost = np.sum((y\_true-y\_predicted)\*\*2) / len(y\_true)

return cost

def gradient\_descent(x, y, iterations = 10, learning\_rate = 0.1,

stopping\_threshold = 1e-3):

current\_weight = 4

current\_bias = 0.01

iterations = iterations

learning\_rate = learning\_rate

n = float(len(x))

costs = []

weights = []

previous\_cost = None

for i in range(iterations):

y\_predicted = (current\_weight \* x) + current\_bias

current\_cost = mean\_squared\_error(y, y\_predicted)

if previous\_cost and abs(previous\_cost-current\_cost)<=stopping\_threshold:

break

previous\_cost = current\_cost

costs.append(current\_cost)

weights.append(current\_weight)

weight\_derivative = -(2/n) \* sum(x \* (y-y\_predicted))

bias\_derivative = -(2/n) \* sum(y-y\_predicted)

current\_weight = current\_weight - (learning\_rate \* weight\_derivative)

current\_bias = current\_bias - (learning\_rate \* bias\_derivative)

print(f"Iteration {i+1}: Cost {current\_cost}, Weight \

{current\_weight}, Bias {current\_bias}")

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

plt.plot(weights, costs)

plt.scatter(weights, costs, marker='o', color='red')

plt.title("Cost vs Weights")

plt.ylabel("Cost")

plt.xlabel("Weight")

plt.show()

return current\_weight, current\_bias

def main():

X = np.array([1, 2, 3, 4])

Y = np.array([2, 4, 6, 8])

estimated\_weight, eatimated\_bias = gradient\_descent(X, Y, iterations=10)

print(f"Estimated Weight: {estimated\_weight}\nEstimated Bias: {eatimated\_bias}")

Y\_pred = estimated\_weight\*X + eatimated\_bias

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

plt.scatter(X, Y, marker='o', color='red')

plt.plot([min(X), max(X)], [min(Y\_pred), max(Y\_pred)], color='blue',

markerfacecolor='red',

markersize=10,linestyle='dashed')

plt.xlabel("X")

plt.ylabel("Y")

plt.show()

Y\_pred = estimated\_weight\*7 + eatimated\_bias

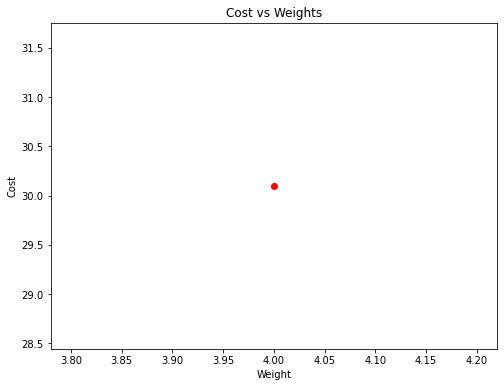
print(Y\_pred)

if \_\_name\_\_=="\_\_main\_\_":

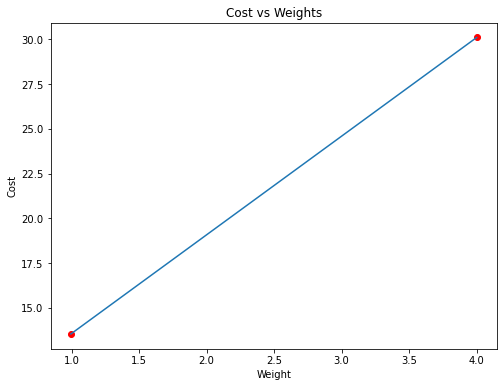
main()

**Output:**

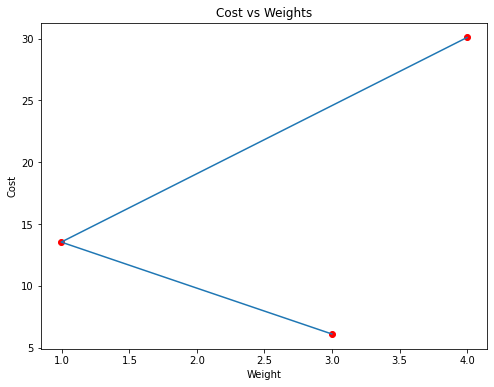
Iteration 1: Cost 30.100100000000005, Weight 0.9949999999999992, Bias -0.992



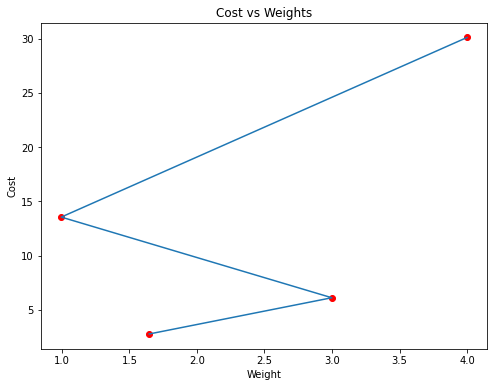
Iteration 2: Cost 13.544051500000016, Weight 2.9985000000000004, Bias -0.2910999999999996



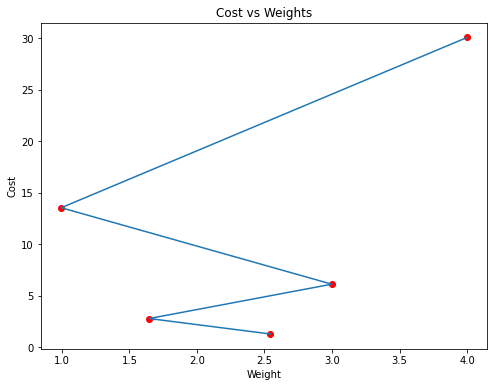
Iteration 3: Cost 6.108939335000006, Weight 1.6462999999999997, Bias -0.73213



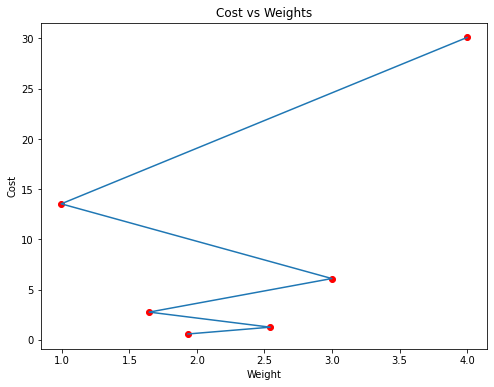
Iteration 4: Cost 2.769063916900002, Weight 2.5429150000000003, Bias -0.4088539999999998



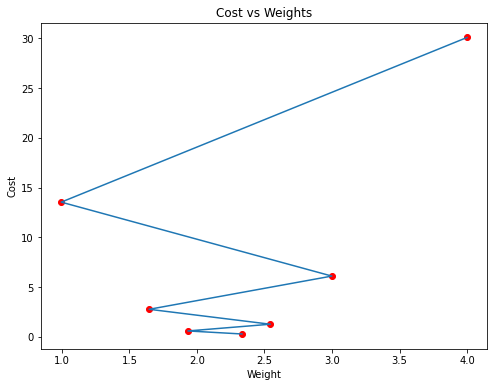
Iteration 5: Cost 1.2679719754535015, Weight 1.9329694999999998, Bias -0.5985406999999999



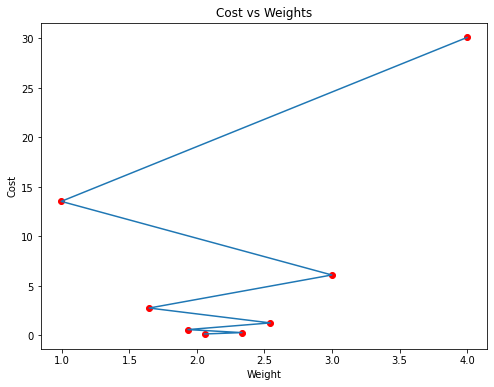
Iteration 6: Cost 0.5925515409901158, Weight 2.3327856000000002, Bias -0.44531730999999986



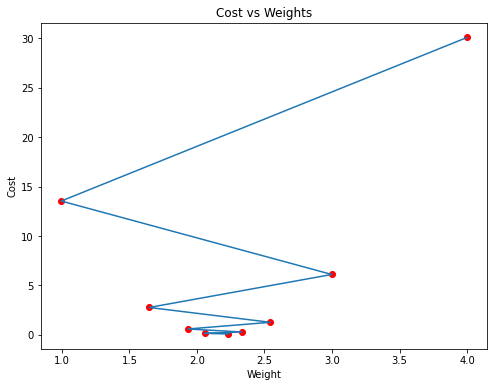
Iteration 7: Cost 0.2879284823471567, Weight 2.056265855, Bias -0.5226466479999999



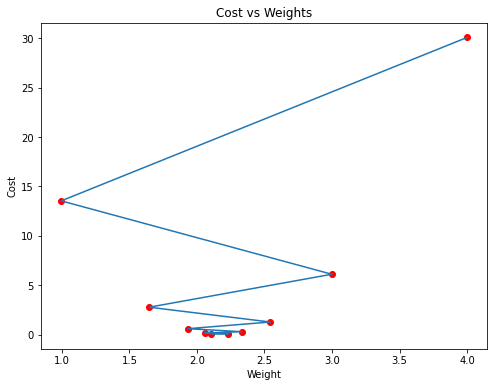
Iteration 8: Cost 0.1498675643942235, Weight 2.2331903965, Bias -0.4462502458999999



Iteration 9: Cost 0.08666613071629707, Weight 2.1065299247, Bias -0.4735953949699999

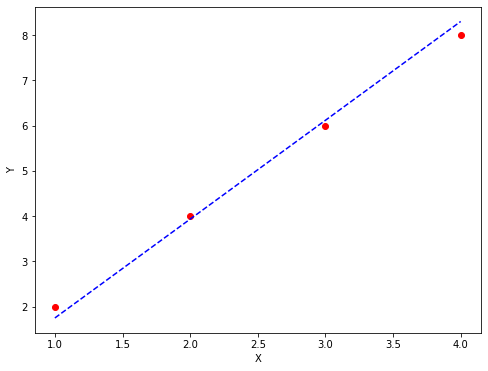


Iteration 10: Cost 0.057146875739093696, Weight 2.1835327351350005, Bias -0.4321412783259998



Estimated Weight: 2.1835327351350005

Estimated Bias: -0.4321412783259998



|  |  |
| --- | --- |
| **Name: Nitin Magdum** | **PRN NO: 19UEL325** |
| **Class: Final Year (Electronics Engineering)** | **Batch: FB4** |

**Experiment No-04**

**Build a single neuron to predict the output value for given input data**

**Code-**

class NeuralNetwork:

def \_\_init\_\_(self, learning\_rate):

self.weights = np.array([np.random.randn(), np.random.randn()])

self.bias = np.random.randn()

self.learning\_rate = learning\_rate

def \_sigmoid(self, x):

return 1 / (1 + np.exp(-x))

def \_sigmoid\_deriv(self, x):

return self.\_sigmoid(x) \* (1 - self.\_sigmoid(x))

def predict(self, input\_vector):

layer\_1 = np.dot(input\_vector, self.weights) + self.bias

layer\_2 = self.\_sigmoid(layer\_1)

prediction = layer\_2

return prediction

def \_compute\_gradients(self, input\_vector, target):

layer\_1 = np.dot(input\_vector, self.weights) + self.bias

layer\_2 = self.\_sigmoid(layer\_1)

prediction = layer\_2

derror\_dprediction = 2 \* (prediction - target)

dprediction\_dlayer1 = self.\_sigmoid\_deriv(layer\_1)

dlayer1\_dbias = 1

dlayer1\_dweights = (0 \* self.weights) + (1 \* input\_vector)

derror\_dbias = (derror\_dprediction \* dprediction\_dlayer1 \* dlayer1\_dbias )

derror\_dweights = (derror\_dprediction \* dprediction\_dlayer1 \* dlayer1\_dweights )

return derror\_dbias, derror\_dweights

def \_update\_parameters(self, derror\_dbias, derror\_dweights):

self.bias = self.bias - (derror\_dbias \* self.learning\_rate)

self.weights = self.weights - (derror\_dweights \* self.learning\_rate )

def train(self, input\_vectors, targets, iterations):

cumulative\_errors = []

for current\_iteration in range(iterations):

# Pick a data instance at random

random\_data\_index = np.random.randint(len(input\_vectors))

input\_vector = input\_vectors[random\_data\_index]

target = targets[random\_data\_index]

# Compute the gradients and update the weights

derror\_dbias, derror\_dweights = self.\_compute\_gradients(input\_vector, target )

self.\_update\_parameters(derror\_dbias, derror\_dweights)

# Measure the cumulative error for all the instances

if current\_iteration % 1000 == 0:

cumulative\_error = 0

# Loop through all the instances to measure the error

for data\_instance\_index in range(len(input\_vectors)):

data\_point = input\_vectors[data\_instance\_index]

target = targets[data\_instance\_index]

prediction = self.predict(data\_point)

error = np.square(prediction - target)

cumulative\_error = cumulative\_error + error

cumulative\_errors.append(cumulative\_error)

return cumulative\_errors

import numpy as np

import matplotlib.pyplot as plt

input\_vectors = np.array([[3, 1.5],[2, 1],[4, 1.5], [3, 4],[3.5, 0.5],[2, 0.5],[5.5, 1],[1, 1]])

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

learning\_rate = 0.1

neural\_network = NeuralNetwork(learning\_rate)

training\_error = neural\_network.train(input\_vectors, targets, 10000)

plt.plot(training\_error)

plt.xlabel("Iterations")

plt.ylabel("Error for all training instances")

plt.savefig("cumulative\_error.png")

pr=neural\_network.predict(input\_vectors)

print(pr)

derror\_dprediction = 2 \* (pr - targets)

print(abs(derror\_dprediction))

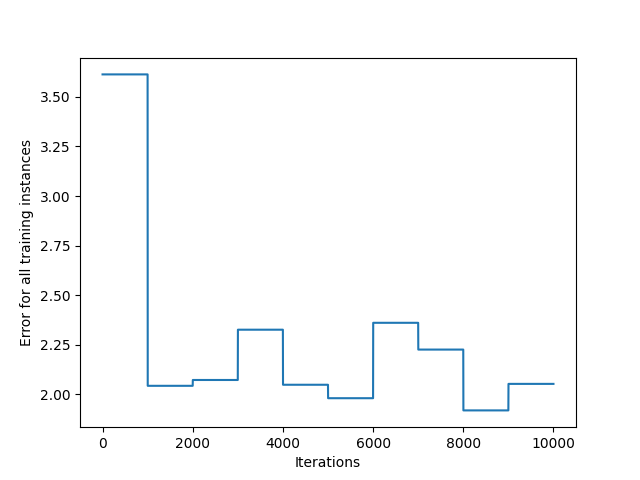
**Output:**

[0.40573302 0.35573326 0.38219723 0.76366049 0.25870163 0.28805913

0.28109658 0.37863818]

[0.81146603 1.28853347 0.76439445 0.47267901 0.51740325 1.42388174

1.43780683 0.75727636]



|  |  |
| --- | --- |
| **Name: Nitin Magdum** | **PRN NO: 19UEL325** |
| **Class: Final Year (Electronics Engineering)** | **Batch: FB4** |

**Experiment No-05**

**Design and develop a Neural Network for classifying Breast Cancer Dataset**

**Code-**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.preprocessing import StandardScaler

from sklearn.preprocessing import LabelEncoder

from sklearn.model\_selection import train\_test\_split

import keras

from keras.models import Sequential

from keras.layers import Dense, Dropout

from sklearn.metrics import confusion\_matrix

import sklearn.metrics as metrics

data = pd.read\_csv('data.csv')

del data['Unnamed: 32']

X = data.iloc[:, 2:].values

y = data.iloc[:, 1].values

labelencoder\_X\_1 = LabelEncoder()

y = labelencoder\_X\_1.fit\_transform(y)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.3, random\_state = 0)

sc = StandardScaler()

X\_train = sc.fit\_transform(X\_train)

X\_test = sc.transform(X\_test)

classifier = Sequential()

classifier.add(Dense(output\_dim=16, init='uniform', activation='relu', input\_dim=30))

classifier.add(Dropout(p=0.1))

classifier.add(Dense(output\_dim=16, init='uniform', activation='relu'))

classifier.add(Dropout(p=0.1))

classifier.add(Dense(output\_dim=1, init='uniform', activation='sigmoid'))

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

classifier.fit(X\_train, y\_train, batch\_size=2, nb\_epoch=150)

acc = classifier.evaluate(X\_test,y\_test)

print("Accuracy is:",acc)

y\_pred = classifier.predict(X\_test)

y\_pred = (y\_pred > 0.5)

cm = confusion\_matrix(y\_test, y\_pred)

print("[Epoch:150] Our accuracy is {}%".format(((cm[0][0] + cm[1][1])/175)\*100))

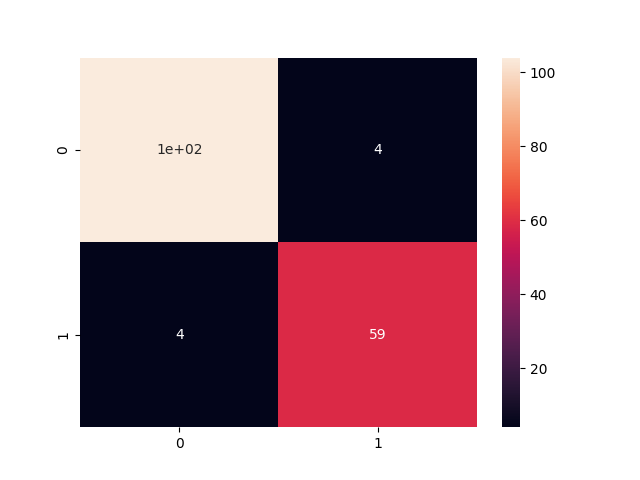
sns.heatmap(cm,annot=True)

plt.savefig('epoch150.png')

**Output:**

Accuracy is: [0.47546069604465646, 0.9532163742690059]

[Epoch:150] Our accuracy is 93.14285714285714%

****

|  |  |
| --- | --- |
| **Name: Nitin Magdum** | **PRN NO: 19UEL325** |
| **Class: Final Year (Electronics Engineering)** | **Batch: FB4** |

**Experiment No-06**

**Design and develop a Deep Convolutional Neural Network for identifying hand written digits from MNIST dataset**

**Code-**

from tensorflow.keras.datasets import mnist

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D

from tensorflow.keras.layers import MaxPool2D

from tensorflow.keras.layers import Flatten

from tensorflow.keras.layers import Dropout

from tensorflow.keras.layers import Dense

from matplotlib import pyplot as plt

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

X\_train = X\_train.reshape((X\_train.shape[0], X\_train.shape[1], X\_train.shape[2], 1))

X\_test = X\_test.reshape((X\_test.shape[0],X\_test.shape[1],X\_test.shape[2],1))

print(X\_train.shape)

print(X\_test.shape)

X\_train=X\_train/255

X\_test=X\_test/255

model=Sequential()

#adding convolution layer

model.add(Conv2D(32,(3,3),activation='relu',input\_shape=(28,28,1)))

#adding pooling layer

model.add(MaxPool2D(2,2))

#adding fully connected layer

model.add(Flatten())

model.add(Dense(100,activation='relu'))

#adding output layer

model.add(Dense(10,activation='softmax'))

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

history=model.fit(X\_train,y\_train,epochs=10)

print("Test Accuracy is:",model.evaluate(X\_test,y\_test) plt.plot(history.history['accuracy'])

plt.title('model accuracy')

plt.ylabel('accuracy')

plt.xlabel('epoch')

plt.legend(['train', 'test'], loc='upper left')

plt.show()

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

plt.title('model loss')

plt.ylabel('loss')

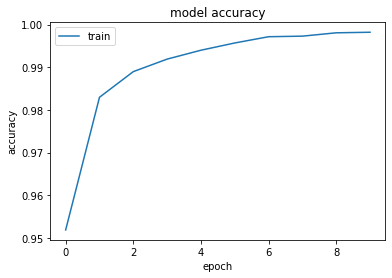
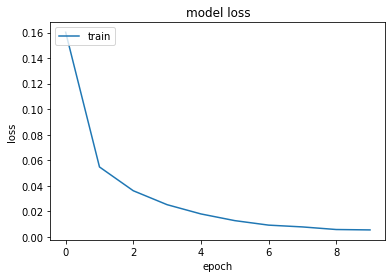
plt.xlabel('epoch')

plt.legend(['train', 'test'], loc='upper left')

plt.show()

**Output:**

Test Accuracy is: [0.056114569306373596, 0.9861000180244446]

|  |  |
| --- | --- |
| **Name: Nitin Magdum** | **PRN NO: 19UEL325** |
| **Class: Final Year (Electronics Engineering)** | **Batch: FB4** |

**Experiment No-07**

**Design and develop a model for object detection using VGG16 model**

**Code-**

import tensorflow as tf

import numpy as np

import tensorflow\_datasets as tfds

from tensorflow.keras.utils import to\_categorical

(train\_ds, train\_labels), (test\_ds, test\_labels) = tfds.load(

"tf\_flowers",

split=["train[:70%]", "train[:30%]"],

batch\_size=-1,as\_supervised=True, # Include labels)

train\_ds.shape

size = (150, 150)

train\_ds = tf.image.resize(train\_ds, (150, 150))

test\_ds = tf.image.resize(test\_ds, (150, 150))

train\_labels = to\_categorical(train\_labels, num\_classes=5)

test\_labels = to\_categorical(test\_labels, num\_classes=5)

train\_ds.shape

from tensorflow.keras.applications.vgg16 import VGG16

from tensorflow.keras.applications.vgg16 import preprocess\_input

train\_ds = preprocess\_input(train\_ds)

test\_ds = preprocess\_input(test\_ds)

base\_model = VGG16(weights="imagenet", include\_top=False, input\_shape=train\_ds[0].shape)

base\_model.trainable = False

base\_model.summary()

from tensorflow.keras import layers, models

flatten\_layer = layers.Flatten()

dense\_layer\_1 = layers.Dense(50, activation='relu')

dense\_layer\_2 = layers.Dense(20, activation='relu')

prediction\_layer = layers.Dense(5, activation='softmax')

model = models.Sequential([

base\_model,

flatten\_layer,

dense\_layer\_1,

dense\_layer\_2,

prediction\_layer

])

model.summary()

from tensorflow.keras.callbacks import EarlyStopping

model.compile(

optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'],

)

es = EarlyStopping(monitor='val\_accuracy', mode='max', patience=5, restore\_best\_weights=True)

model.fit(train\_ds, train\_labels, epochs=50, validation\_split=0.2, batch\_size=32, callbacks=[es])

model.evaluate(test\_ds, test\_labels)

**Output:**

Model: "vgg16"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_1 (InputLayer) [(None, 150, 150, 3)] 0

block1\_conv1 (Conv2D) (None, 150, 150, 64) 1792

block1\_conv2 (Conv2D) (None, 150, 150, 64) 36928

block1\_pool (MaxPooling2D) (None, 75, 75, 64) 0

block2\_conv1 (Conv2D) (None, 75, 75, 128) 73856

block2\_conv2 (Conv2D) (None, 75, 75, 128) 147584

block2\_pool (MaxPooling2D) (None, 37, 37, 128) 0

block3\_conv1 (Conv2D) (None, 37, 37, 256) 295168

block3\_conv2 (Conv2D) (None, 37, 37, 256) 590080

block3\_conv3 (Conv2D) (None, 37, 37, 256) 590080

block3\_pool (MaxPooling2D) (None, 18, 18, 256) 0

block4\_conv1 (Conv2D) (None, 18, 18, 512) 1180160

block4\_conv2 (Conv2D) (None, 18, 18, 512) 2359808

block4\_conv3 (Conv2D) (None, 18, 18, 512) 2359808

block4\_pool (MaxPooling2D) (None, 9, 9, 512) 0

block5\_conv1 (Conv2D) (None, 9, 9, 512) 2359808

block5\_conv2 (Conv2D) (None, 9, 9, 512) 2359808

block5\_conv3 (Conv2D) (None, 9, 9, 512) 2359808

block5\_pool (MaxPooling2D) (None, 4, 4, 512) 0

=================================================================

Total params: 14,714,688

Trainable params: 0

Non-trainable params: 14,714,688

Model: "sequential"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

vgg16 (Functional) (None, 4, 4, 512) 14714688

flatten (Flatten) (None, 8192) 0

dense (Dense) (None, 50) 409650

dense\_1 (Dense) (None, 20) 1020

dense\_2 (Dense) (None, 5) 105

=================================================================

Total params: 15,125,463

Trainable params: 410,775

Non-trainable params: 14,714,688

[0.07233612984418869, 0.9845594763755798]

|  |  |
| --- | --- |
| **Name: Nitin Magdum** | **PRN NO: 19UEL325** |
| **Class: Final Year (Electronics Engineering)** | **Batch: FB4** |

**Experiment No-08**

**Develop a linear regression model using deep neural network for predicting price of houses in Boston city**

**Code-**

import pandas as pd

import numpy as np

import tensorflow as tf

train\_df = pd.read\_csv('https://firebasestorage.googleapis.com/v0/b/bible-project-2365c.appspot.com/o/train.csv?alt=media&token=9c5d17c2-0589-43ea-b992-e7c2ad02d714', index\_col='ID')

train\_df.head()

test\_df = pd.read\_csv('https://firebasestorage.googleapis.com/v0/b/bible-project-2365c.appspot.com/o/test.csv?alt=media&token=99688b27-9fdb-4ac3-93b8-fa0e0f4d7540', index\_col='ID')

test\_df.head()

predictors = ['crim', 'zn', 'indus', 'chas', 'nox', 'rm', 'age', 'dis', 'rad', 'tax', 'ptratio', 'black', 'lstat']

target = 'medv'

from sklearn.preprocessing import MinMaxScaler

print("Note: median values were scaled by multiplying by {:.10f} and adding {:.6f}".format(scaler.scale\_[13], scaler.min\_[13]))

multiplied\_by = scaler.scale\_[13]

added = scaler.min\_[13]

print(type(scaled\_train))

scaled\_train\_df = pd.DataFrame(scaled\_train, columns=train\_df.columns.values)

model = tf.keras.Sequential()

model.add(tf.keras.layers.Dense(50, activation='relu'))

model.add(tf.keras.layers.Dense(100, activation='relu'))

model.add(tf.keras.layers.Dense(50, activation='relu'))

model.add(tf.keras.layers.Dense(1))

model.compile(loss='mean\_squared\_error', optimizer='adam')

X = scaled\_train\_df.drop(target, axis=1).values

Y = scaled\_train\_df[[target]].values# Train the model

model.fit(

X[10:],

Y[10:],

epochs=50,

shuffle=True,

verbose=2

)

test\_error\_rate = model.evaluate(X[:10], Y[:10], verbose=0)

print("The mean squared error (MSE) for the test data set is: {}".format(test\_error\_rate))

prediction = model.predict(X[:1])

y\_0 = prediction[0][0]

print('Prediction with scaling - {}',format(y\_0))

y\_0 -= added

y\_0 /= multiplied\_by

print("Housing Price Prediction - ${}".format(y\_0))

Y\_0 = Y[0]

print('Ground truth with scaling - {}'.format(Y\_0))

Y\_0 -= added

Y\_0 /= multiplied\_by

print('Ground Truth Price - ${}'.format(Y\_0))

**Output:**

|  | **crim** | **zn** | **indus** | **chas** | **nox** | **rm** | **age** | **dis** | **rad** | **tax** | **ptratio** | **black** | **lstat** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ID** |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **3** | 0.02729 | 0.0 | 7.07 | 0 | 0.469 | 7.185 | 61.1 | 4.9671 | 2 | 242 | 17.8 | 392.83 | 4.03 |
| **6** | 0.02985 | 0.0 | 2.18 | 0 | 0.458 | 6.430 | 58.7 | 6.0622 | 3 | 222 | 18.7 | 394.12 | 5.21 |
| **8** | 0.14455 | 12.5 | 7.87 | 0 | 0.524 | 6.172 | 96.1 | 5.9505 | 5 | 311 | 15.2 | 396.90 | 19.15 |
| **9** | 0.21124 | 12.5 | 7.87 | 0 | 0.524 | 5.631 | 100.0 | 6.0821 | 5 | 311 | 15.2 | 386.63 | 29.93 |
| **10** | 0.17004 | 12.5 | 7.87 | 0 | 0.524 | 6.004 | 85.9 | 6.5921 | 5 | 311 | 15.2 | 386.71 | 17.10 |

|  | **crim** |  | **zn** | **indus** | **chas** | **nox** | **rm** | **age** | **dis** | **rad** | **tax** | **ptratio** | **black** | **lstat** | **medv** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ID** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **1** | 0.00632 |  | 18.0 | 2.31 | 0 | 0.538 | 6.575 | 65.2 | 4.0900 | 1 | 296 | 15.3 | 396.90 | 4.98 | 24.0 |
| **2** | 0.02731 |  | 0.0 | 7.07 | 0 | 0.469 | 6.421 | 78.9 | 4.9671 | 2 | 242 | 17.8 | 396.90 | 9.14 | 21.6 |
| **4** | 0.03237 |  | 0.0 | 2.18 | 0 | 0.458 | 6.998 | 45.8 | 6.0622 | 3 | 222 | 18.7 | 394.63 | 2.94 | 33.4 |
| **5** | 0.06905 |  | 0.0 | 2.18 | 0 | 0.458 | 7.147 | 54.2 | 6.0622 | 3 | 222 | 18.7 | 396.90 | 5.33 | 36.2 |
| **7** | 0.08829 |  | 12.5 | 7.87 | 0 | 0.524 | 6.012 | 66.6 | 5.5605 | 5 | 311 | 15.2 | 395.60 | 12.43 | 22.9 |

The mean squared error (MSE) for the test data set is: 0.00301399640739

('Prediction with scaling - {}', '0.473798304796')

Housing Price Prediction - $26.3209237158

Ground truth with scaling - [0.42222222]

Ground Truth Price - $[24.]