**Program 1**

**Aim: Write a program to implement water jug problem with two jugs the capacity of both the jugs should be entered by user. The quantity of the water to be stored should also be dynamic. The output will show all the steps to get the final state.**

**Code:**

n1=int(input("Enter the capacity of first jug: "))

n2=int(input("Enter the capacity of second jug: "))

n3=int(input("In which jug to be filled :"))

n4=int(input("How much to be filled: "))

class Waterjug:

def \_\_init\_\_(self,am,bm,a,b,g):

self.a\_max = am;

self.b\_max = bm;

self.a = a;

self.b = b;

self.goal = g;

def fillA(self):

self.a = self.a\_max;

print ('(', self.a, ',',self.b, ')')

def fillB(self):

self.b = self.b\_max;

print ('(', self.a, ',', self.b, ')')

def emptyA(self):

self.a = 0;

print ('(', self.a, ',', self.b, ')')

def emptyB(self):

self.b = 0;

print ('(', self.a, ',', self.b, ')')

def transferAtoB(self):

while (True):

self.a = self.a - 1

self.b = self.b + 1

if (self.a == 0 or self.b == self.b\_max):

break

print ('(', self.a, ',', self.b, ')')

def main(self):

while (True):

if (self.a == self.goal or self.b == self.goal):

break

if (self.a == 0):

self.fillA()

elif (self.a > 0 and self.b != self.b\_max):

self.transferAtoB()

elif (self.a > 0 and self.b == self.b\_max):

self.emptyB()

def pour(jug1, jug2):

max1, max2, fill = n1, n2, n4

print("%d\t%d" % (jug1, jug2))

if jug2 is fill:

return elif jug2 is max2:

pour(0, jug1)

elif jug1 != 0 and jug2 is 0:

pour(0, jug1)

elif jug1 is fill:

pour(jug1, 0)

elif jug1 < max1:

pour(max1, jug2)

elif jug1 < (max2-jug2):

pour(0, (jug1+jug2))

else:

pour(jug1-(max2-jug2), (max2-jug2)+jug2)

print("JUG1\tJUG2")

if(n3==2):

pour(0, 0)

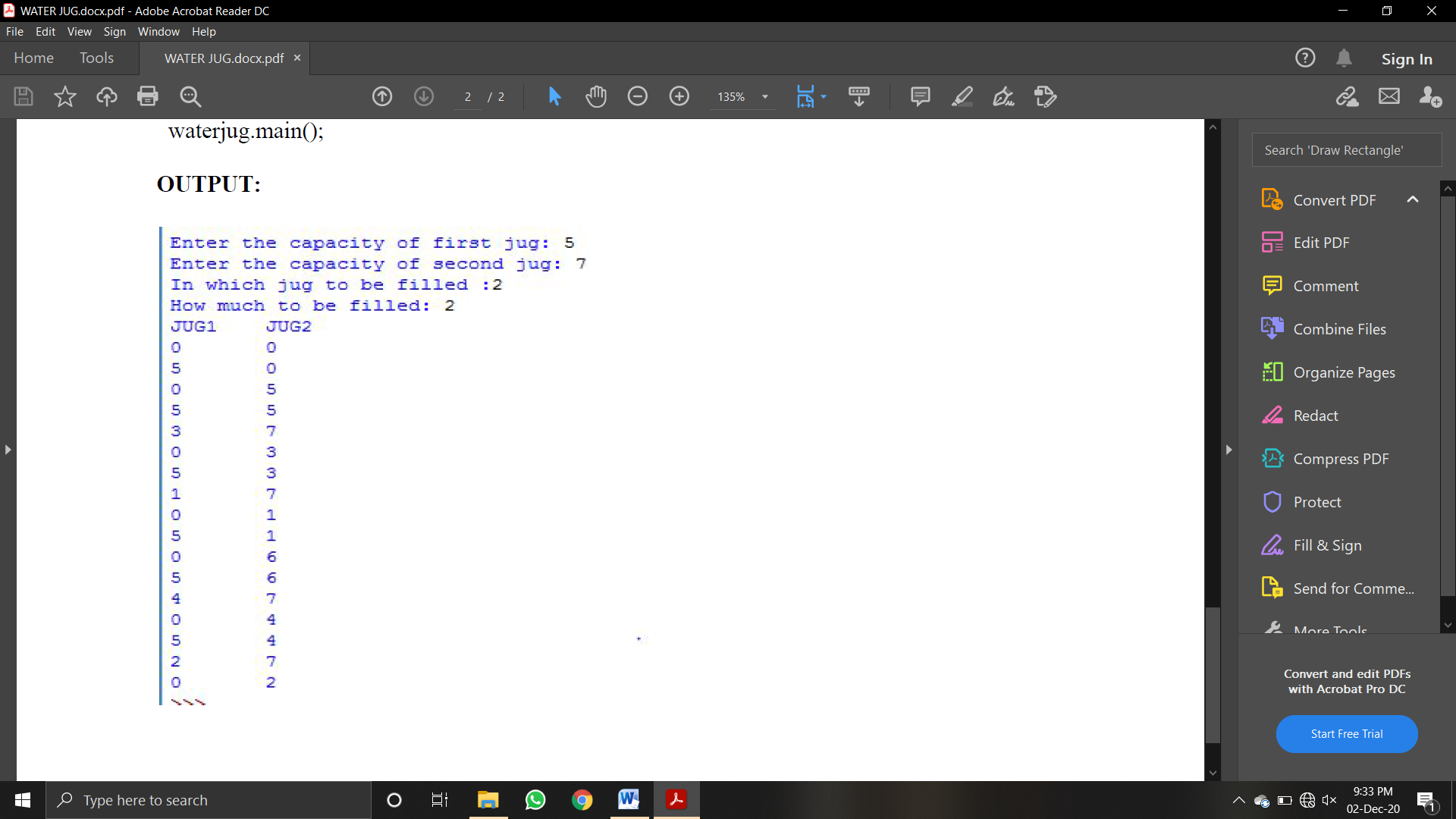
elif(n3==1):

print ('(', '0',',', '0', ')')

waterjug=Waterjug(n1,n2,0,0,n4);

waterjug.main();

**Output:**



**Program 2**

**Aim: Write a program in Python to implement single layer perceptron for ANDNOT function.**

**Code:**

import numpy as np

x=np.array([[1,1],[1,-1],[-1,1],[-1,-1]])

t=np.array([[-1],[1],[-1],[-1]])

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

b=0

theta=float(input("Enter new theta:"))

alpha=float(input("Enter new alpha:"))

yin=np.zeros(shape=(4,1))

y=np.zeros(shape=(4,1))

i=0

found=0

while(found==0):

yin=x[i][0]\*w[0]+x[i][1]\*w[1]

yin = yin+b

if(yin>theta):

y[i] = 1

elif(yin<=theta and yin>=-theta):

y[i]=0

else:

y[i]=-1

if (y[i]==t[i]):

print("NO UPDATION REQUIRED")

print(y[i])

if(i<3):

i=i+1

else:

i=0

else:

print("MODEL IS NOT TRAINED")

print("The value of output is")

print(y)

w[0]=w[0]+alpha\*x[i][0]\*t[i]

w[1]=w[1]+alpha\*x[i][1]\*t[i]

b = b+alpha\*t[i]

if(i<3):

i=i+1

else:

i=0

if(y==t).all():

found=1

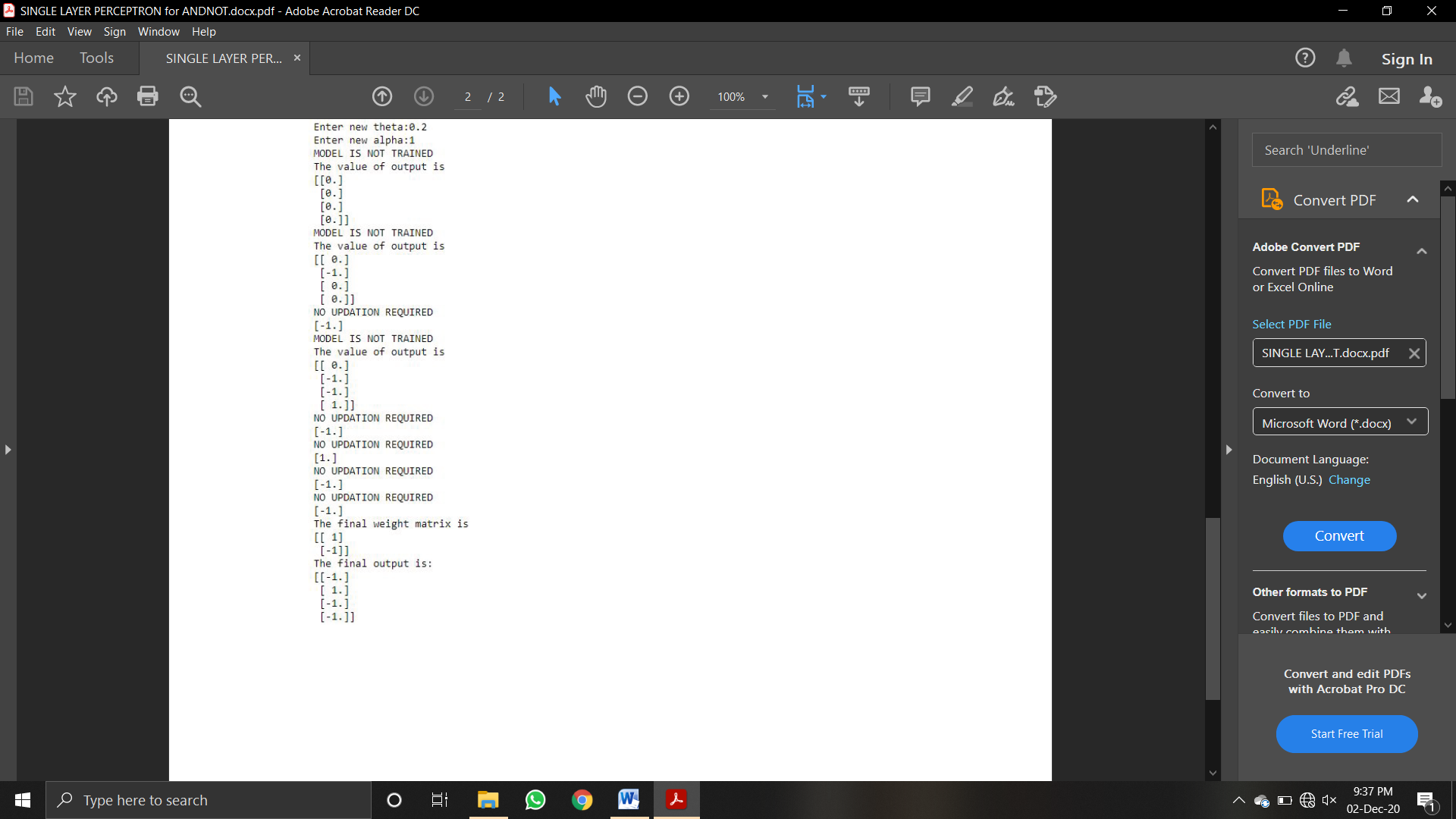
print("The final weight matrix is ")

print(w)

print("The final output is:")

print(y)

**Output:**



**Program 3**

**Aim: Write a program in Python to implement single layer perceptron for AND function.**

**Code:**

import numpy as np

x=np.array([[1,1],[1,-1],[-1,1],[-1,-1]])

t=np.array([[1],[1],[1],[-1]])

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

b=0

theta=float(input("Enter new theta:"))

alpha=float(input("Enter new alpha:"))

yin=np.zeros(shape=(4,1))

y=np.zeros(shape=(4,1))

i=0

found=0

while(found==0):

yin=x[i][0]\*w[0]+x[i][1]\*w[1]

yin = yin+b

if(yin>theta):

y[i] = 1

elif(yin<=theta and yin>=-theta):

y[i]=0

else:

y[i]=-1

if (y[i]==t[i]):

print("NO UPDATION REQUIRED")

print(y[i])

if(i<3):

i=i+1

else:

i=0

else:

print("MODEL IS NOT TRAINED")

print("The value of output is")

print(y)

w[0]=w[0]+alpha\*x[i][0]\*t[i]

w[1]=w[1]+alpha\*x[i][1]\*t[i]

b = b+alpha\*t[i]

if(i<3):

i=i+1

else:

i=0

if(y==t).all():

found=1

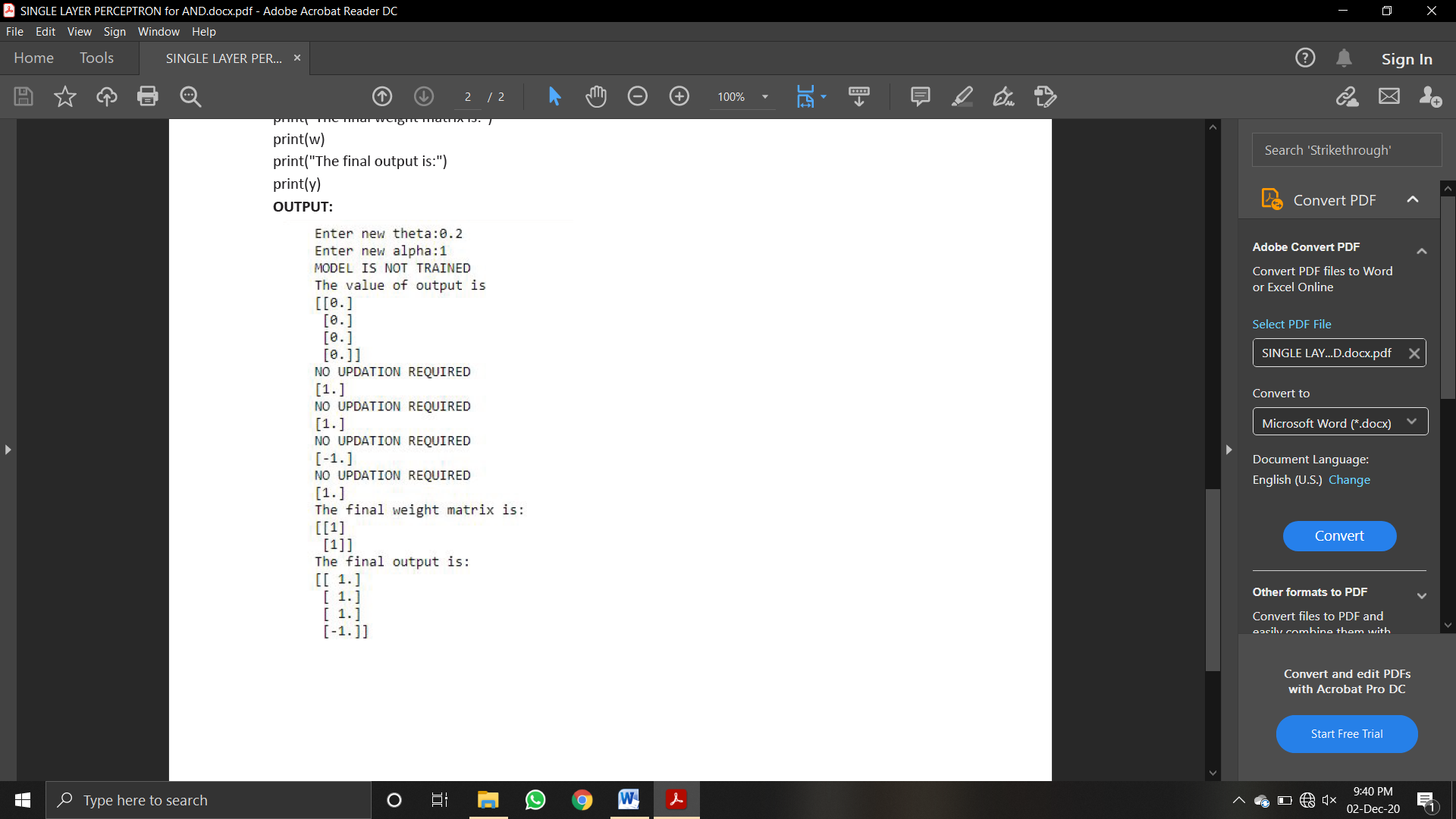
print("The final weight matrix is:")

print(w)

print("The final output is:")

print(y)

**Output:**



**Program 4**

**Aim:Write a program to implement OR logic functions using numpy neuron.**

**Code:**

Import numpy as np

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

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

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

theta=1

yin=np.zeros(shape=(4,1))

y=np.zeros(shape=(4,1))

yin=np.dot(x,w)

i=0

found=0

while(found==0):

i=0

yin=np.dot(x,w)

print("Y is initiallised",yin)

while(i<4):

if yin[i]>=theta:

y[i]=1

i=i+1

else:

y[i]=0

i=i+1

print("Calculated y",y)

print("Expected Target t",t)

if (y==t).all():

print("MODEL IS TRAINED ")

print("\nOutput : \n",y)

print("\nweights : ",w,"\n")

print("theta : ",theta)

found=1

else:

print("MODEL IS NOT TRAINED")

w=np.zeros(shape=(0,0))

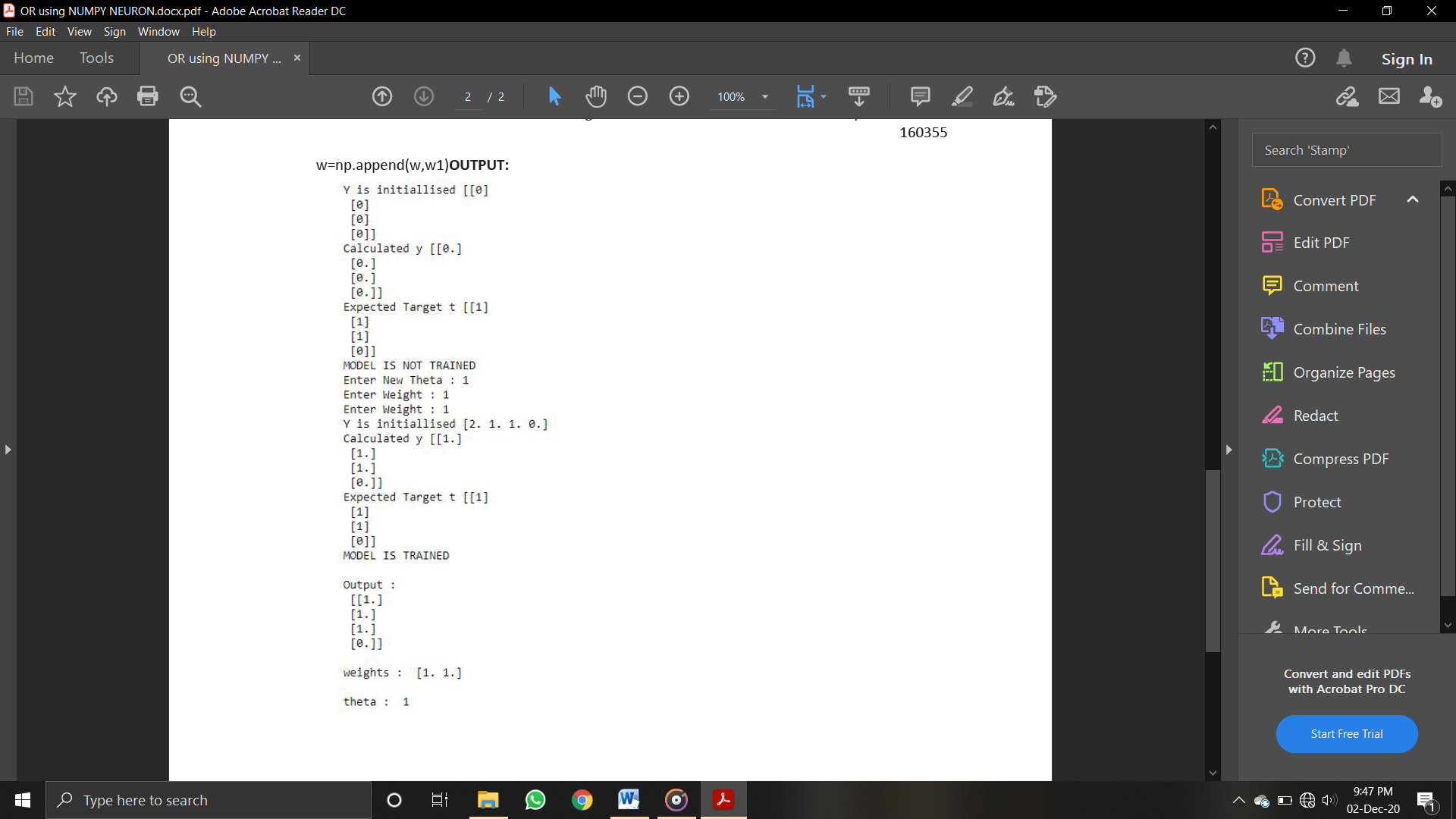
theta=int(input("Enter New Theta : "))

for k in range(int(2)):

w1=int(input("Enter Weight : "))

w=np.append(w,w1)

**Output:**



**Program 5**

**Aim:Write a program to implement NOT logic functions using numpy neuron.**

**Code:**

import numpy as np

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

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

w=np.array([0])

theta=1

yin=np.zeros(shape=(2,1))

y=np.zeros(shape=(2,1))

yin=np.dot(x,w)

i=0

found=0

while(found==0):

i=0

yin=np.dot(x,w)

print(yin)

while(i<2):

if yin[i]>=theta:

y[i]=1

i=i+1

#if(i==4):

#break

else:

y[i]=0

i=i+1

print("y",y)

print("t",t)

if (y==t).all():

print("MODEL IS TRAINED ")

print("\nOutput : \n",y)

print("\nweights : ",w,"\n")

print("theta : ",theta)

found=1

else:

print("MODEL IS NOT TRAINED")

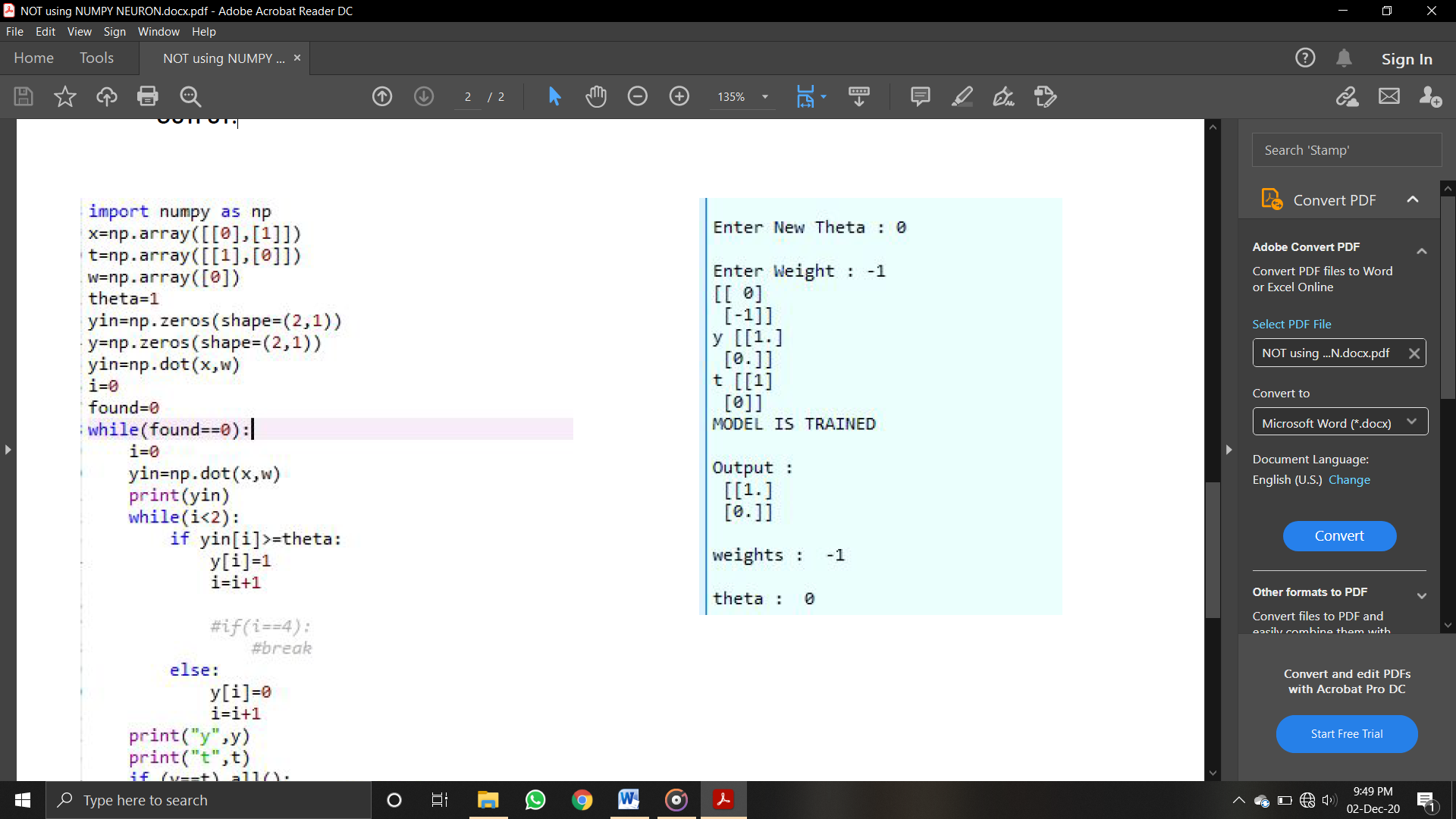
w=np.zeros(shape=(0,0))

theta=int(input("Enter New Theta : "))

for k in range(int(1)):

w=int(input("Enter Weight : "))

**Output:**



**Program 6**

**Aim:Write a program to implement NOR logic functions using numpy neuron.**

**Code:**

import numpy as np

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

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

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

theta=1

yin=np.zeros(shape=(4,1))

y=np.zeros(shape=(4,1))

yin=np.dot(x,w)

i=0

found=0

while(found==0):

i=0

yin=np.dot(x,w)

print("Y is initiallised",yin)

while(i<4):

if yin[i]>=theta:

y[i]=1

i=i+1

else:

y[i]=0

i=i+1

print("Calculated y",y)

print("Expected Target t",t)

if (y==t).all():

print("MODEL IS TRAINED ")

print("\nOutput : \n",y)

print("\nweights : ",w,"\n")

print("theta : ",theta)

found=1

else:

print("MODEL IS NOT TRAINED")

w=np.zeros(shape=(0,0))

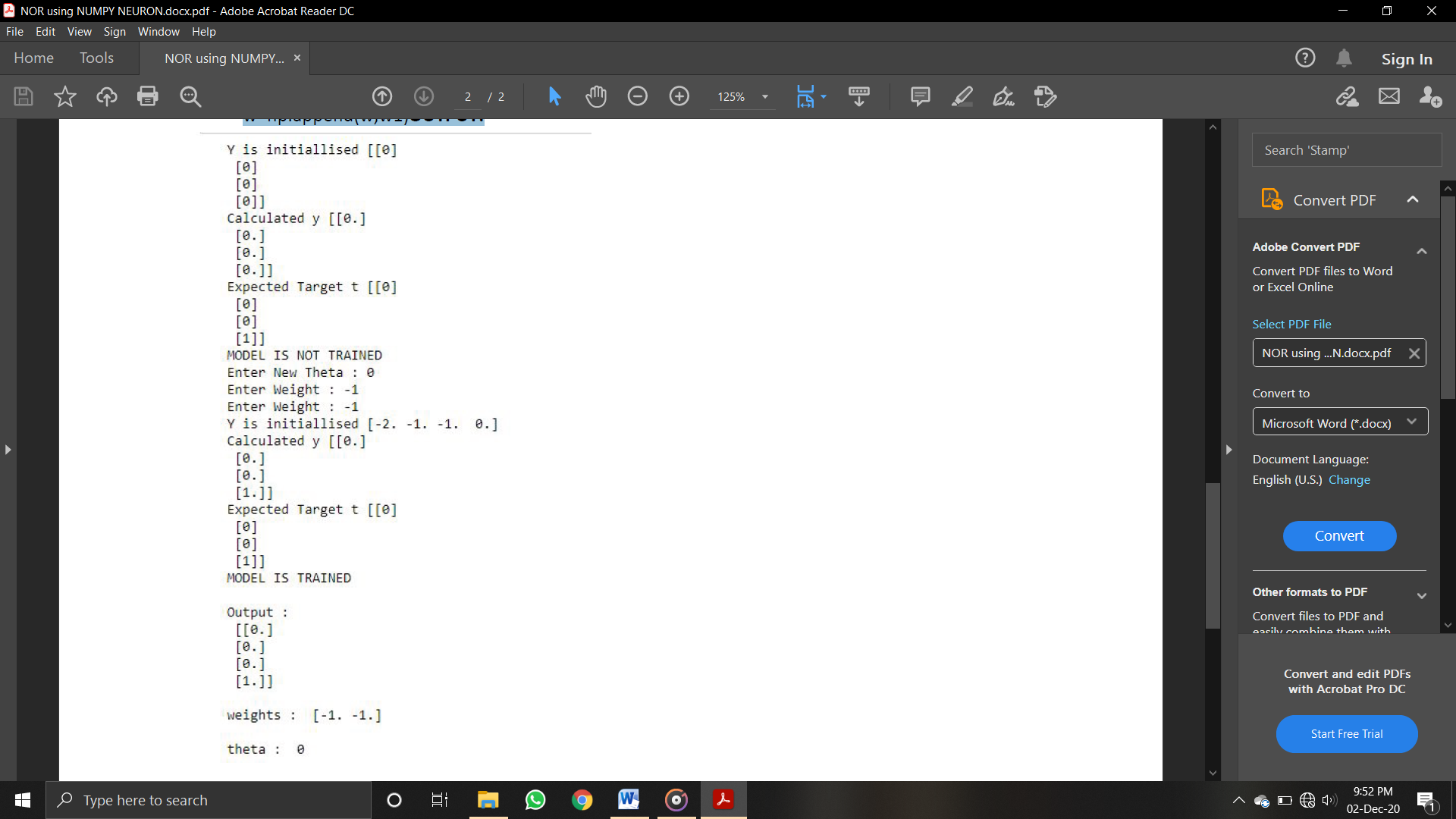
theta=int(input("Enter New Theta : "))

for k in range(int(2)):

w1=int(input("Enter Weight : "))

w=np.append(w,w1)

**Output:**



**Program 7**

**Aim:Write a program in Python to implement NAND function using MP neuron.**

**Code :**

w1 = int(input("enter the weight 1: "))

w2 = int(input("enter the weight 2: "))

theta = int(input("enter the theta:"))

inputarray = [[0,0],[0,1],[1,0],[1,1]]

expected = [1,1,1,0]

actual = []

for i in range(0,4):

temp = inputarray[i][0]\*w1+inputarray[i][1]\*w2

if(temp >= theta):

actual.append(1)

else:

actual.append(0)

for i in range(0,4):

if(expected[i] == actual[i]):

found = 1

else:

found = 0

break

print("Input array")

print(inputarray)

print("Actual output")

print(actual)

print("Expected output")

print(expected)

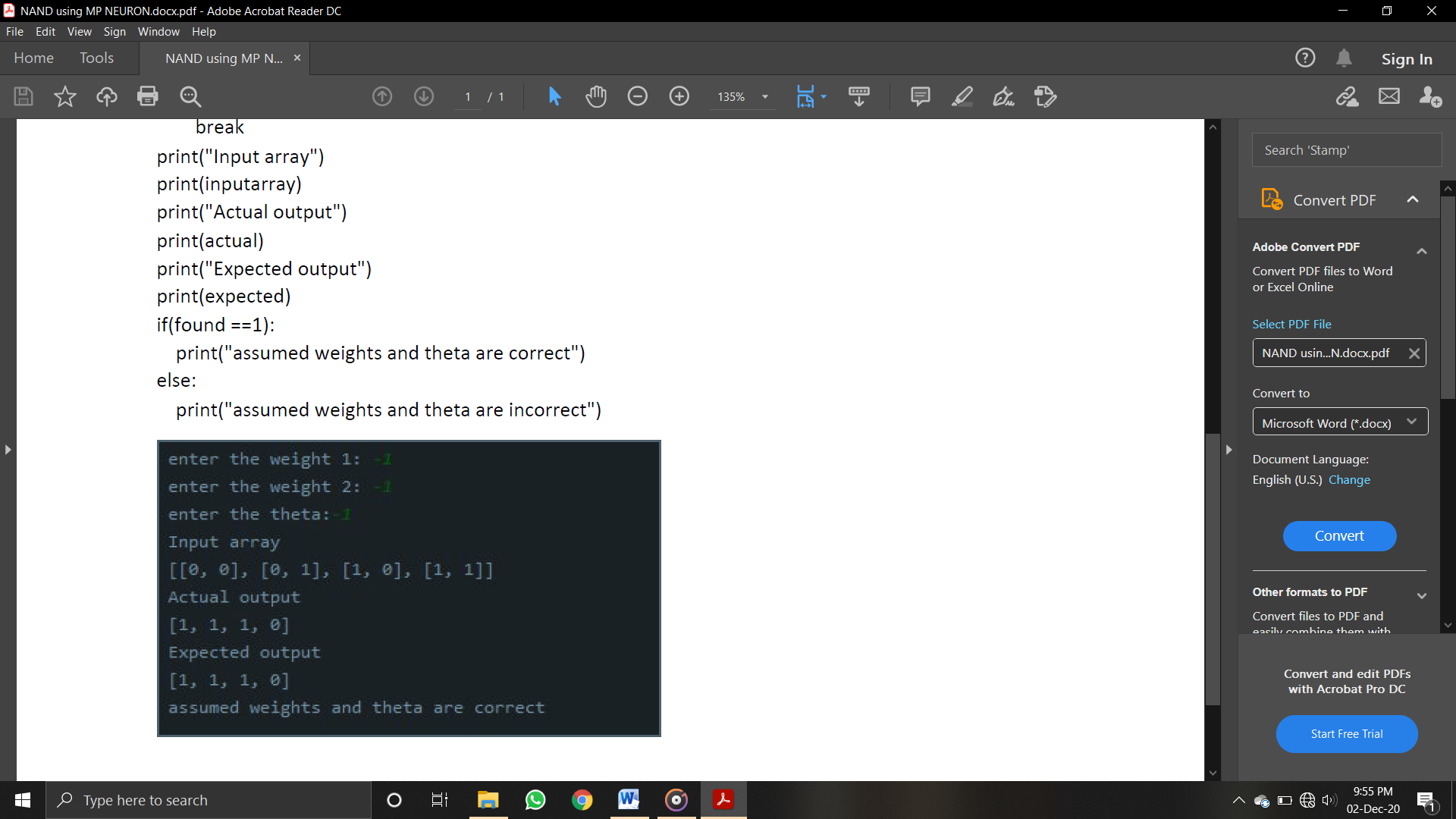
if(found ==1):

print("assumed weights and theta are correct")

else:

print("assumed weights and theta are incorrect")

**Output:**



**Program 8**

**Aim:** Write a program in Python to implement Madaline Neural Network.

**Code:**

import numpy as np

x=np.array([[1,1],[1,-1],[-1,1],[-1,-1]])

t=np.array([[1],[1],[1],[-1]])

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

b=0

theta=float(input("enter new theta"))

alpha=float(input("enter new alpha"))

yin=np.zeros(shape=(4,1))

y=np.zeros(shape=(4,1))

i=0

found=0

while(found==0):

yin=x[i][0]\*w[0]+x[i][1]\*w[1]

yin = yin+b

if(yin>theta):

y[i] = 1

elif(yin<=theta and yin>=-theta):

y[i]=0

else:

y[i]=-1

if (y[i]==t[i]):

print("NO UPDATION REQUIRED")

print(y[i])

if(i<3):

i=i+1

else:

i=0

else:

print("MODEL IS NOT TRAINED")

print("The value of output is")

print(y)

w[0]=w[0]+alpha\*x[i][0]\*t[i]

w[1]=w[1]+alpha\*x[i][1]\*t[i]

b = b+alpha\*t[i]

if(i<3):

i=i+1

else:

i=0

if(y==t).all():

found=1

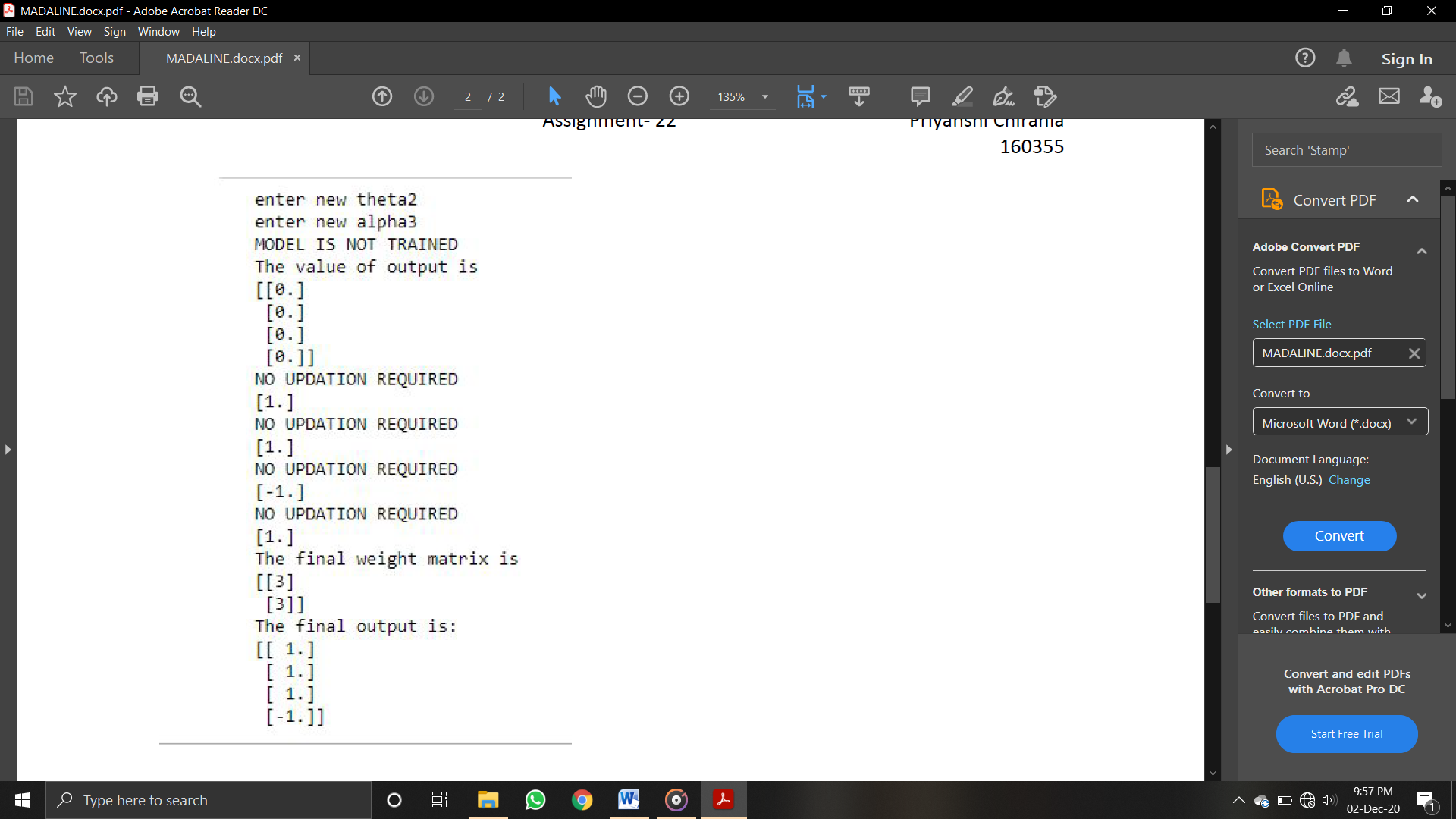
print("The final weight matrix is ")

print(w)

print("The final output is:")

print(y)

**OUTPUT:**



**Program 9**

**Aim: Write a program to implement linear separability for AND function** .

**Code:**

import numpy as np

import matplotlib.pyplot as plt

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

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

plt.scatter(x,y,c='red')

plt.scatter(1,1,c="blue")

plt.xlabel('Input 1')

plt.ylabel('Input 2')

w=-1

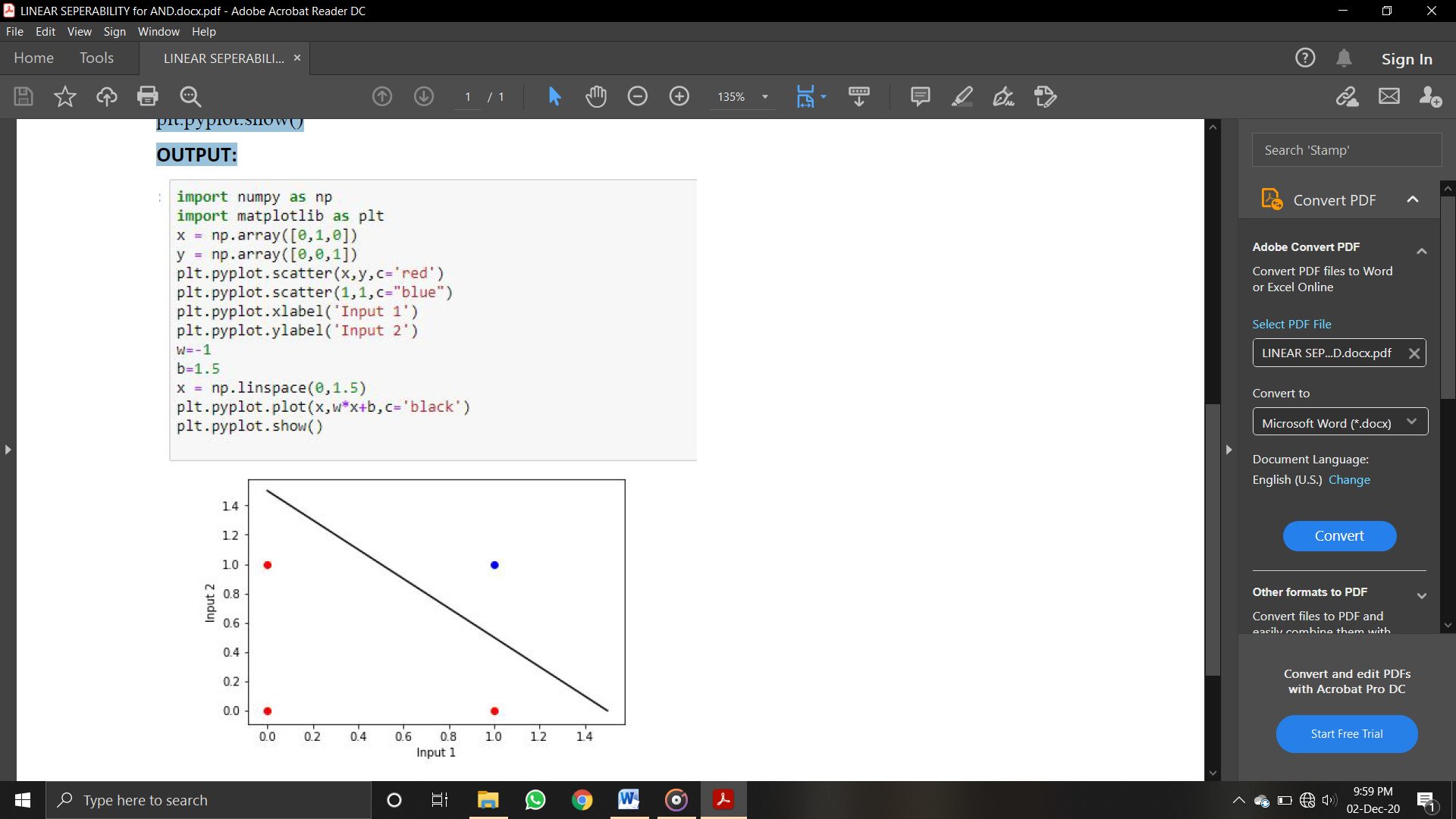
b=1.5

x = np.linspace(0,1.5)

plt.plot(x,w\*x+b,c='black')

plt.show()

**Output:**



**Program 10**

**Aim:** Write a program to implement linear separability for OR function.

**Code :**

import numpy as np

import matplotlib.pyplot as plt

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

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

plt.scatter(x,y,c='red')

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

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

plt.scatter(x,y,c="blue")

plt.xlabel('Input 1')

plt.ylabel('Input 2')

w=-1

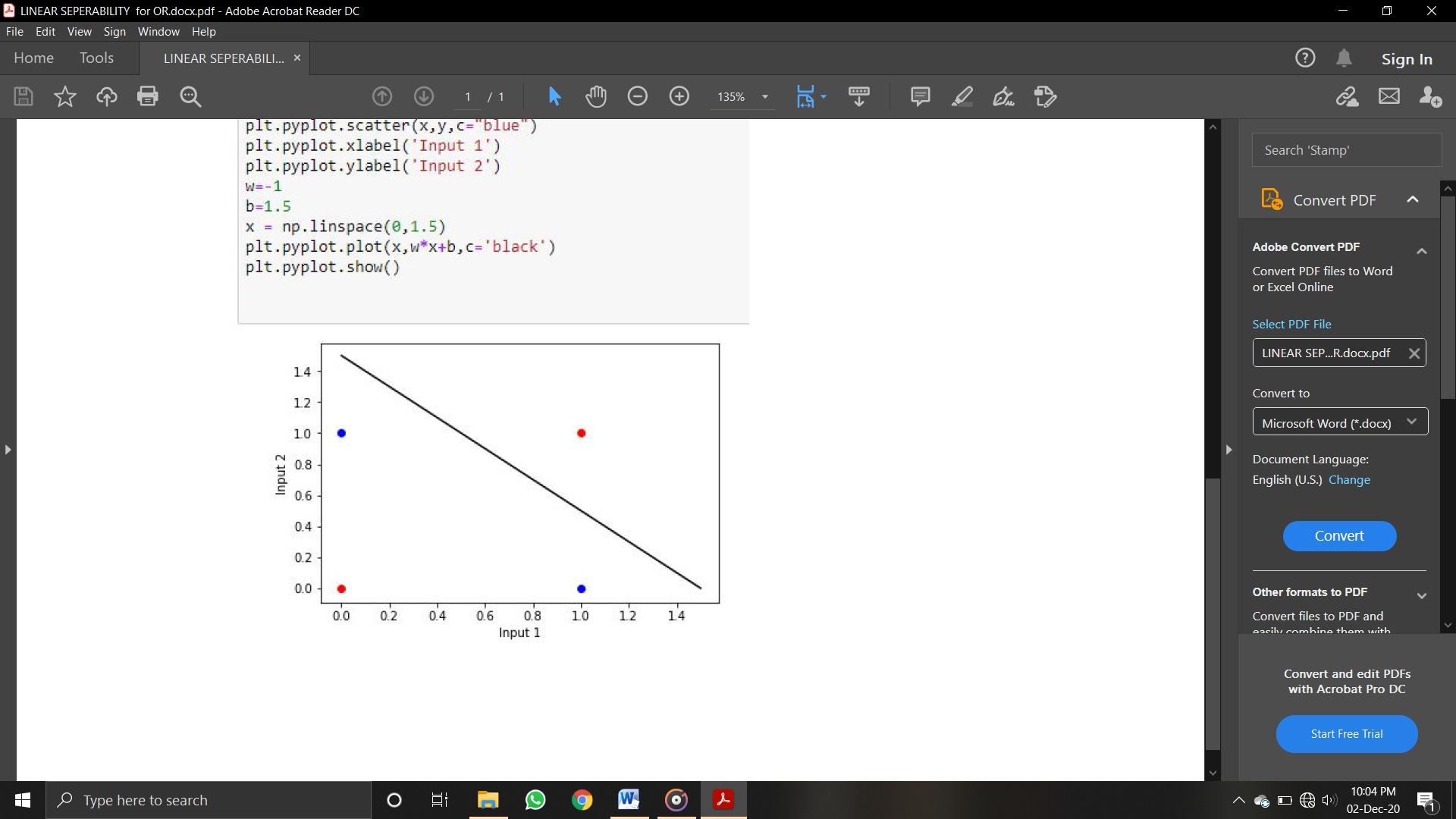
b=1.5

x = np.linspace(0,1.5)

plt.plot(x,w\*x+b,c='black')

plt.show()

**Output:**



**Program 11**

**Aim:Write a program to implement iterative deepening dfs with using variable limit and return the path to traverse till input node.**

**Code:**

from collections import defaultdict

class Graph:

def \_\_init\_\_(self,vertices):

self.V = vertices

self.graph = defaultdict(list)

def addEdge(self,u,v):

self.graph[u].append(v)

def DLS(self,src,target,maxDepth):

if src == target : return True

if maxDepth <= 0 : return False

for i in self.graph[src]:

if(self.DLS(i,target,maxDepth-1)):

return True

return False

def IDDFS(self,src, target, maxDepth):

for i in range(maxDepth):

if (self.DLS(src, target, i)):

return True

return False

g = Graph (7);

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 3)

g.addEdge(1, 4)

g.addEdge(2, 5)

g.addEdge(2, 6)

target = int(input("enter the node to be searched"));

maxDepth = int(input("enter the depth"));

src = 0

found = 1

while(found):

if g.IDDFS(src, target, maxDepth) == True:

print ("Target is reachable from source " +

Assignment- 9 Priyanshi Chirania

160355

"within max depth : ")

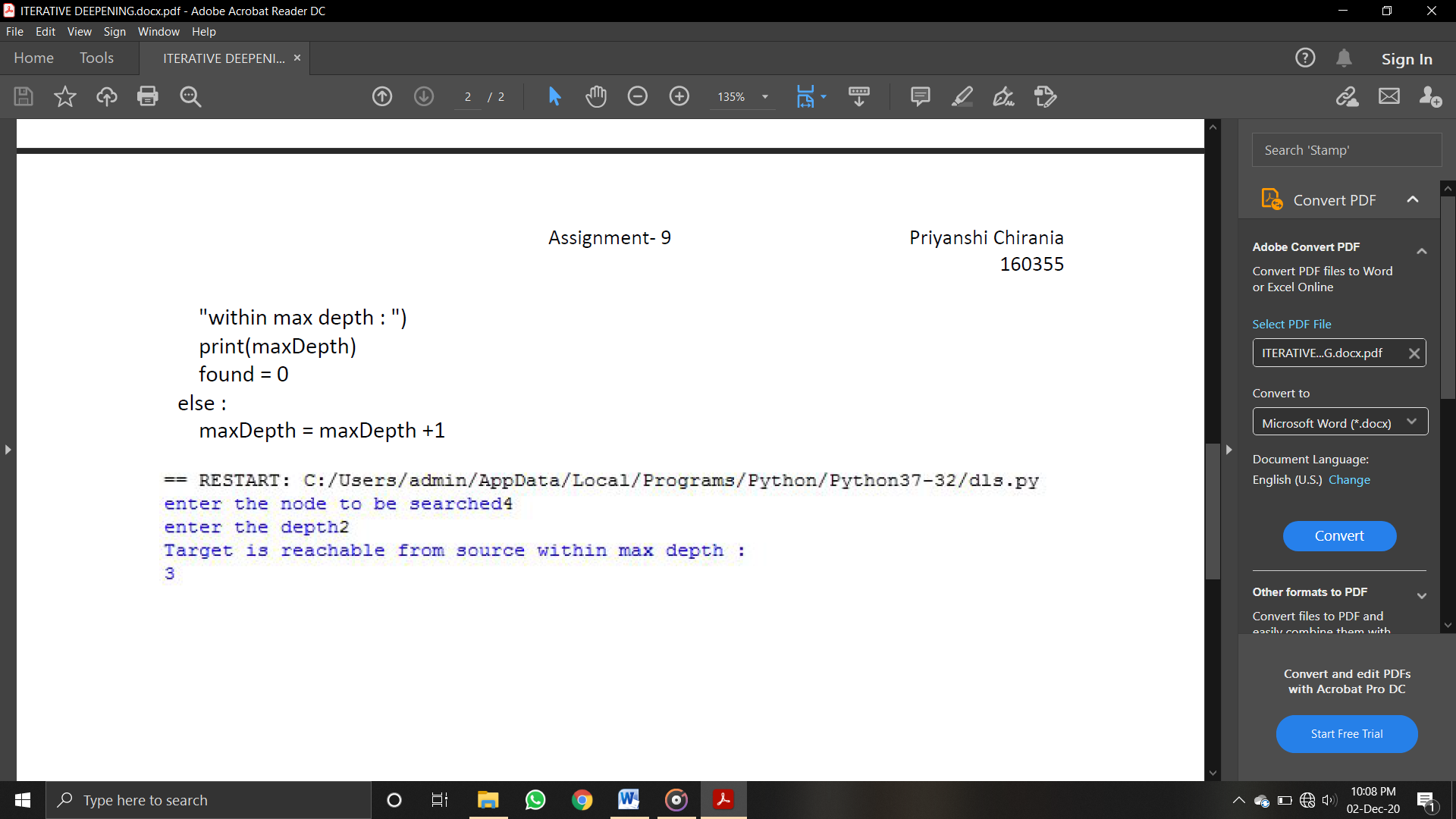
print(maxDepth)

found = 0

else :

maxDepth = maxDepth +1

**Output:**



**Program 12**

**Aim: Write a program in Python to implement Hopfield neural network**

**Code:**

Import numpy as np

x=np.array([[1,1,1,1,1],[1,-1,-1,1,-1],[-1,1,-1,-1,-1]])

x1=np.transpose(x)

t1=np.array([[1,1,1,-1,1]])

t2=np.array([[1,-1,-1,-1,-1]])

t3=np.array([[1,1,-1,-1,-1]])

w=np.zeros((5,5))

i=0

j=0

k=0

for i in range(len(x1)):

for j in range(len(x[0])):

for k in range(len(x)):

w[i][j] += x1[i][k] \* x[k][j]

print('Weight Matrix:\n')

for r in w:

print(r)

print('\n\nWeight Matrix with no self connection:\n')

i=0

j=0

for i in range(int(5)):

for j in range(int(5)):

if(i==j):

w[i][j]=0

for r in w:

print(r)

E1=0

E2=0

E3=0

x11= x[0].reshape(5,1)

x12=x[1].reshape(5,1)

x13=x[2].reshape(5,1)

E1= -0.5 \* np.matmul(x[0],np.matmul(w,x11))

print('\n\nEnergy Calculations for pattern [1,1,1,1,1]:',E1)

E2= -0.5 \* np.matmul(x[1],np.matmul(w,x12))

print('\n\nEnergy Calculations for pattern [1,-1,-1,1,-1]:',E2)

E3= -0.5 \* np.matmul(x[2],np.matmul(w,x13))

print('\n\nEnergy Calculations for pattern [-1,1,-1,1,-1]:',E3)

print('\n\nTESTING PHASE')

w\_dash=np.transpose(w)

Yin1=t1[0][3]+ np.matmul(x[0],w\_dash[3])

if(Yin1>0):

t1[0][3]=1

else:

t1[0][3]=-1

if((t1==x).any()):

print('\nPattern [1,1,1,-1,1] Recognized ')

else:

print('\nPattern [1,1,1,-1,1] not Recognized ')

Yin2=t2[0][3]+ np.matmul(x[1],w\_dash[3])

if(Yin2>0):

t2[0][3]=1

else:

t2[0][3]=-1

if((t2==x).any()):

print('\nPattern [1,-1,-1,-1,-1] Recognized ')

else:

print('\nPattern [1,-1,-1,-1,-1] not Recognized ')

Yin3=t3[0][0]+ np.matmul(x[2],w\_dash[0])

if(Yin3>0):

t3[0][0]=1

else:

t3[0][0]=-1

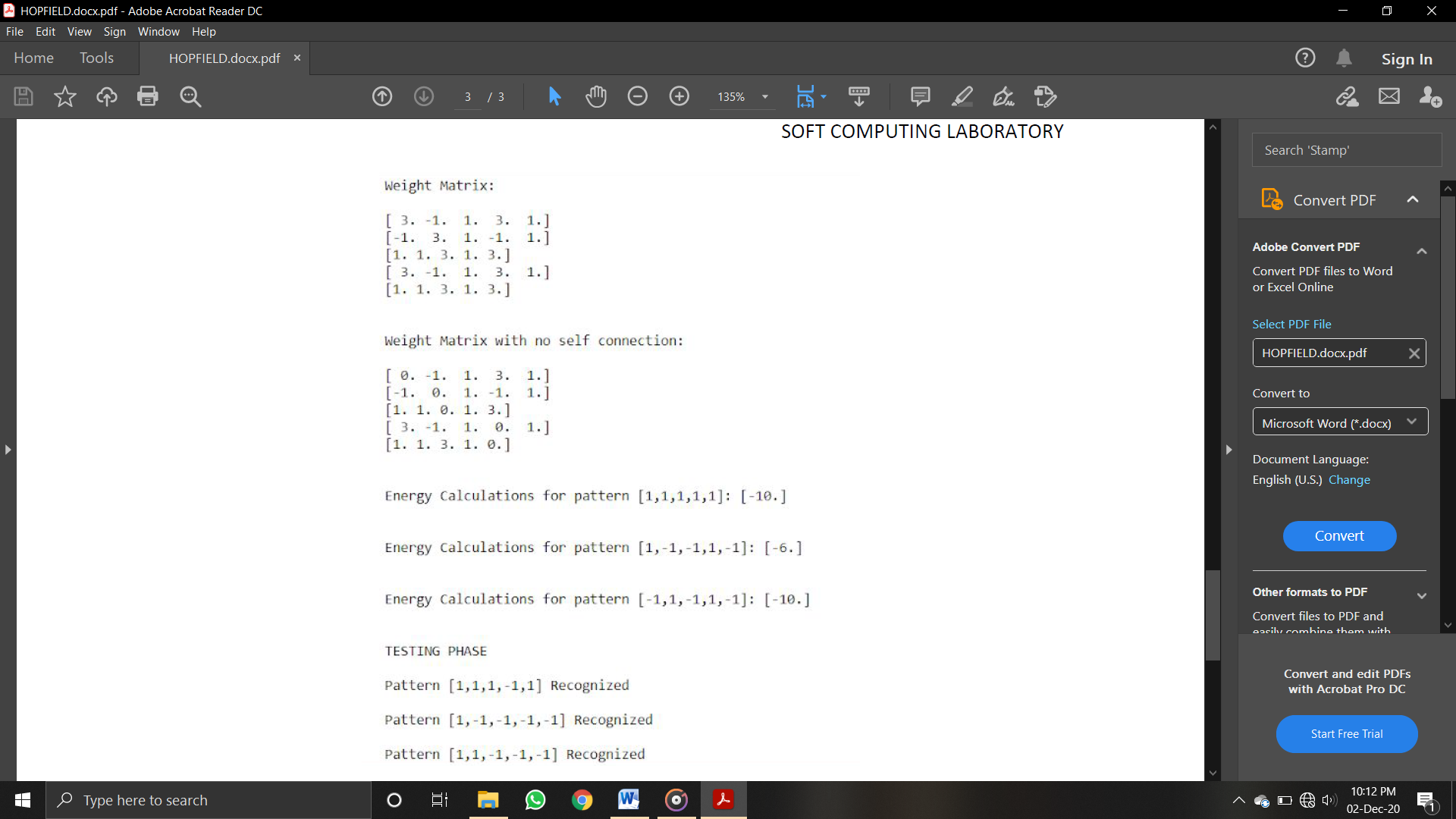
if((t3==x).any()):

print('\nPattern [1,1,-1,-1,-1] Recognized ')

else:

print('\nPattern [1,1,-1,-1,-1] not Recognized ')

**Output:**



**Program 13**

**Aim:Write a program to implement dfs with using a fix limit and return the path to**

**traverse till input node.**

**Code:**

from collections import defaultdict

class Graph:

def \_\_init\_\_(self,vertices):

self.V = vertices

self.graph = defaultdict(list)

def addEdge(self,u,v):

self.graph[u].append(v)

def DLS(self,src,target,maxDepth):

if src == target : return True

if maxDepth <= 0 : return False

for i in self.graph[src]:

if(self.DLS(i,target,maxDepth-1)):

return True

return False

def IDDFS(self,src, target, maxDepth):

for i in range(maxDepth):

if (self.DLS(src, target, i)):

return True

return False

g = Graph (7);

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 3)

g.addEdge(1, 4)

g.addEdge(2, 5)

g.addEdge(2, 6)

target = int(input("enter the node to be searched"));

maxDepth = int(input("enter the depth"));

src = 0

if g.IDDFS(src, target, maxDepth) == True:

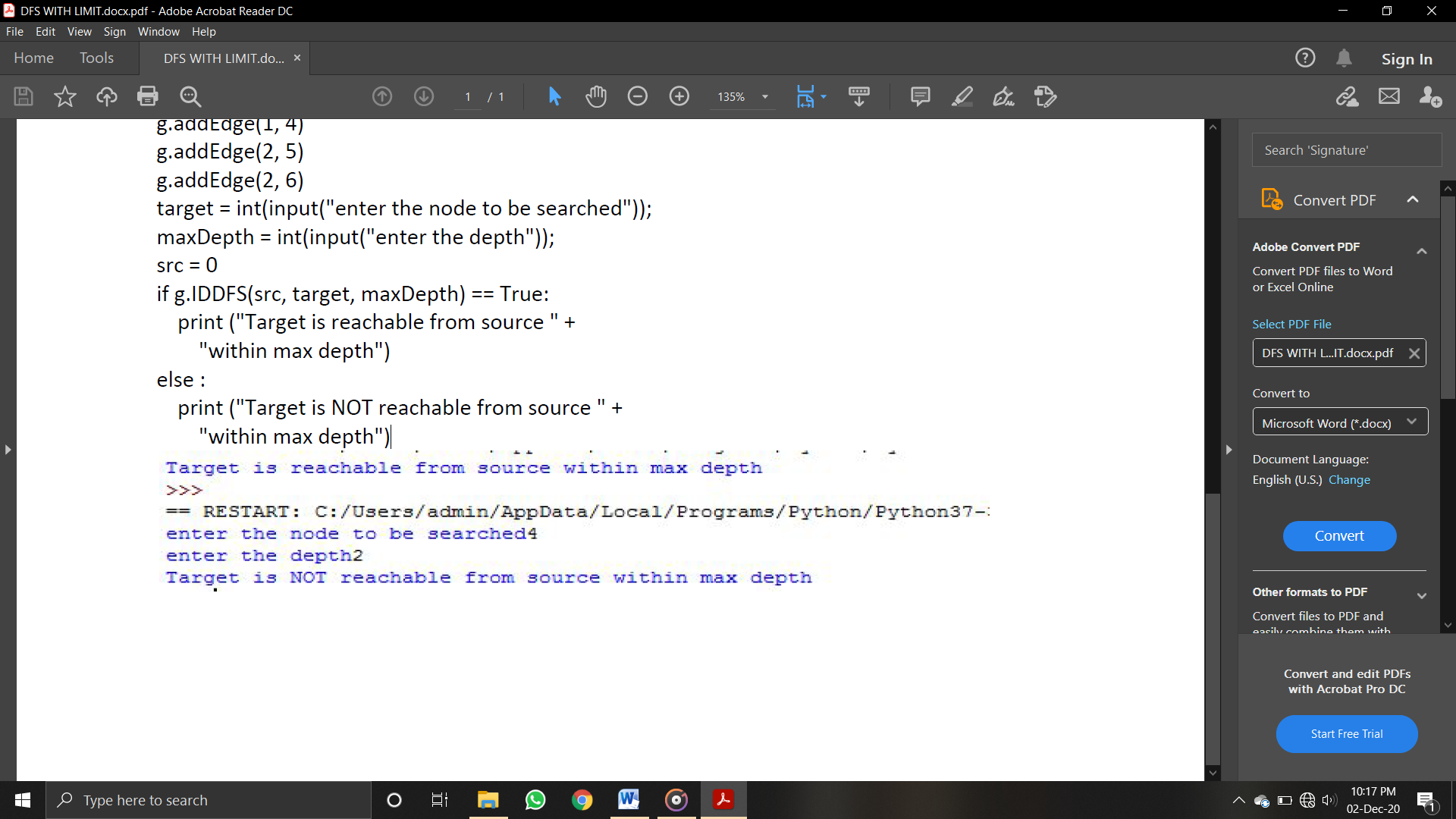
print ("Target is reachable from source " +

"within max depth")

else :

print ("Target is NOT reachable from source " +"within max depth")

**Output:**



**Program 14**

**AIM:Write a program to implement bfs and dfs with the input of graph and the goal node to be searched, your output will show the path from the root node to goal node only**

**Code:**

def bfs(graph,start,search):

explored = []

queue = [start]

found = 1

while found:

node = queue.pop(0)

if(node == search):

found = 0

if node not in explored:

explored.append(node)

neighbours = graph[node]

for neighbour in neighbours:

queue.append(neighbour)

print(explored)

search = int(input("enter the number you want to search-"))

graph = {1: [2, 3, 5],

2: [1,4, 5],

3: [1, 6, 7],

4: [2],

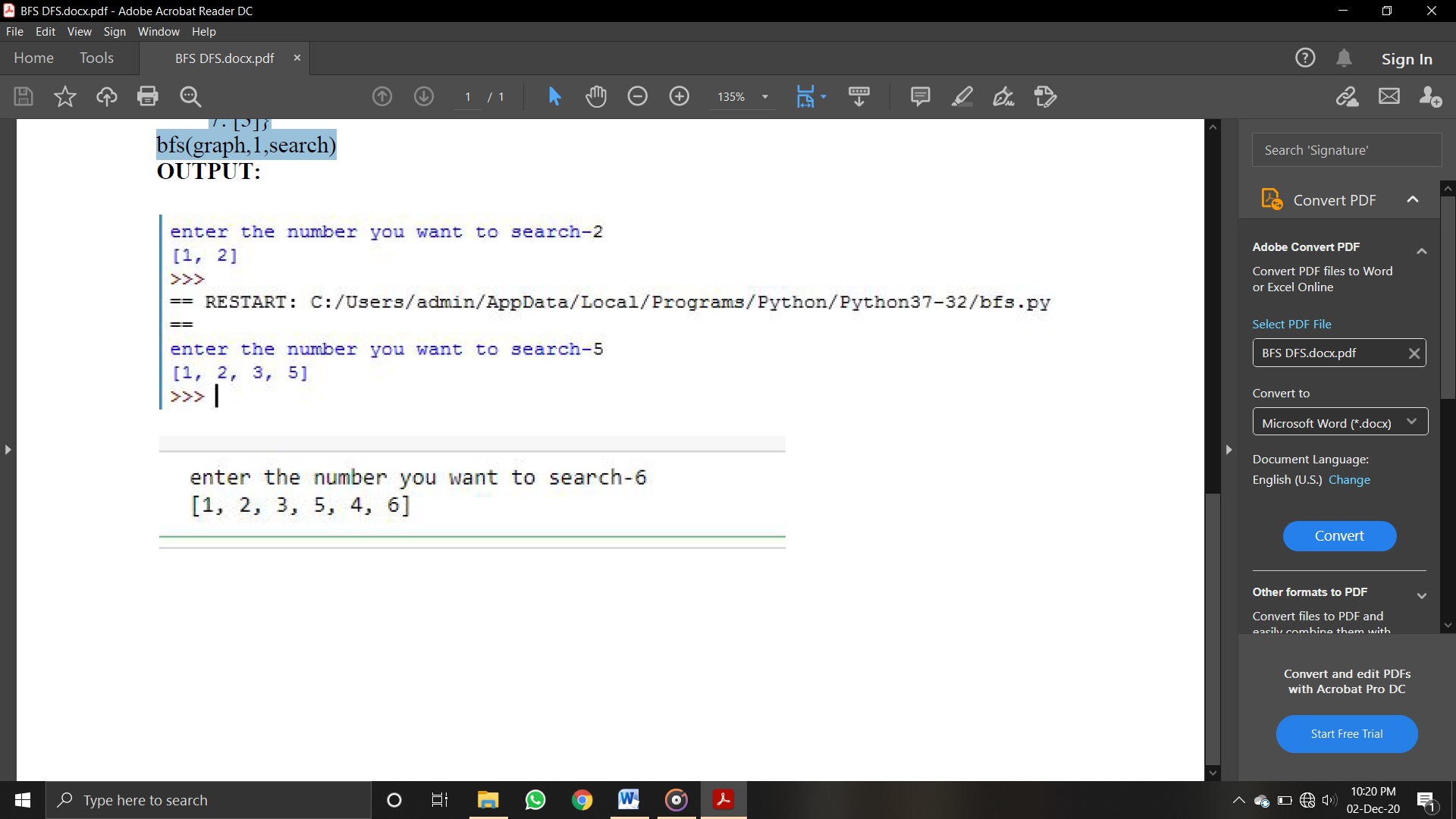
5: [1, 2,4],

6: [3],

7: [3]}

bfs(graph,1,search)

**Output:**



**Program 15**

**Aim:** Write a program in Python to implement Bidirectional Associative Memory (BAM) network to store and test the given patterns.

**Code:**

Import numpy as np

x1=np.array([[1,1,1,-1,1,-1,-1,1,-1,-1,1,-1]])

x2=np.array([[1,1,1,1,-1,1,1,-1,1,1,1,1]])

x3=np.array([[1,1,1,-1,1,-1,-1,1,-1,1,1,1]])

t1 = np.array([[-1],[1]])

t2 = np.array([[1],[1]])

w1=np.zeros((12,2),dtype=int)

w2=np.zeros((12,2),dtype=int)

w=np.zeros((12,2),dtype=int)

i=0

while(i!=12):

w1[i][0]=x1[0][i]\*t1[0][0]

w1[i][1]=x1[0][i]\*t1[1][0]

w2[i][0]=x2[0][i]\*t2[0][0]

w2[i][1]=x2[0][i]\*t2[1][0]

i=i+1

w=w1+w2

print('The Weight Matrix is:\n')

print(w)

Yin11=Yin12=Yin21=Yin22=Yin31=Yin32=0

y1=0

y2=0

i=0

while(i!=12):

Yin11=Yin11+(x1[0][i]\*w[i][0])

Yin12=Yin12+(x1[0][i]\*w[i][1])

Yin21=Yin21+(x2[0][i]\*w[i][0])

Yin22=Yin22+(x2[0][i]\*w[i][1])

Yin31=Yin31+(x3[0][i]\*w[i][0])

Yin32=Yin32+(x3[0][i]\*w[i][1])

i=i+1

if(Yin11>0):

Yin11=1

else:

Yin11=-1

if(Yin12>0):

Yin12=1

else:

Yin12=-1

if(Yin21>0):

Yin21=1

else:

Yin21=-1

if(Yin22>0):

Yin22=1

else:

Yin22=-1

if(Yin31>0):

Yin31=1

else:

Yin31=-1

if(Yin32>0):

Yin32=1

else:

Yin32=-1

if((Yin11==-1) and (Yin12==1)):

print('Pattern T is recognized for Y-Layer')

else:

print('Pattern T is not recognized for Y-Layer')

if((Yin21==1) and (Yin22==1)):

print('Pattern O is recognized for Y-Layer')

else:

print('Pattern O is not recognized for Y-Layer')

i=0

Xin1=np.zeros((12,1),dtype=int)

Xin2=np.zeros((12,1),dtype=int)

while(i!=12):

Xin1[i][0]=Xin1[i][0]+((Yin11\*w[i][0])+(Yin12\*w[i][1]))

if(Xin1[i][0]>0):

Xin1[i][0]=1

else:

Xin1[i][0]=-1

Xin2[i][0]=Xin2[i][0]+((Yin21\*w[i][0])+(Yin22\*w[i][1]))

if(Xin2[i][0]>0):

Xin2[i][0]=1

else:

Xin2[i][0]=-1

i=i+1

Xin1=Xin1.T

Xin2=Xin2.T

print('\n')

if((Xin1==x1).all()):

print('Pattern T is recognized for X-Layer')

else:

print('Pattern T is not recognized for X-Layer')

if((Xin2==x2).all()):

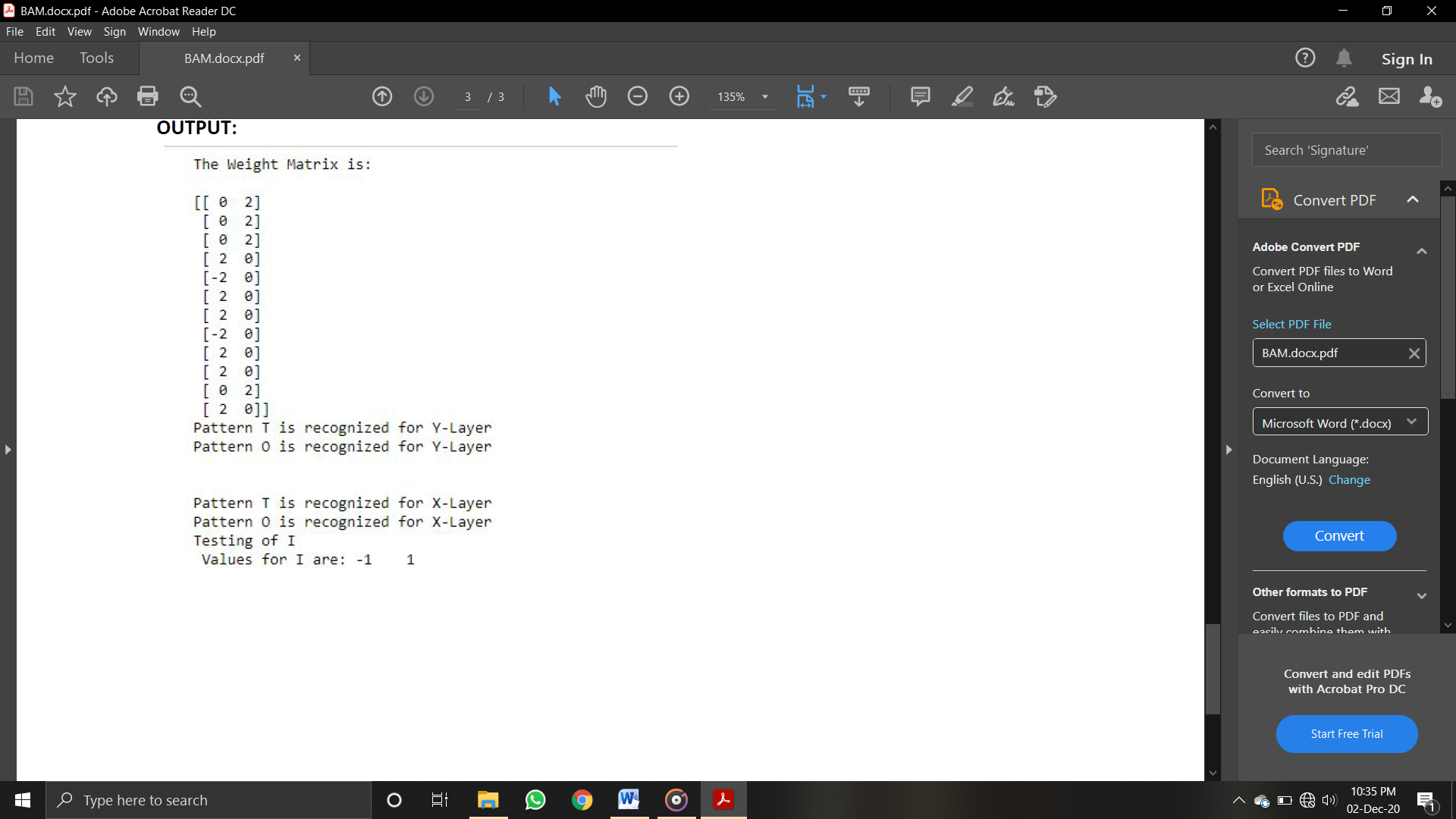
print('Pattern O is recognized for X-Layer')

else:

print('Pattern O is not recognized for X-Layer')

print('Testing of I \n Values for I are:', Yin31 ,'\t',Yin32)

**Output:**



**Program 16**

**Aim:** Write a program in Python to implement Back-Propagation Neural Network.

**Code:**

import math

import random

import string

class NN:

def \_\_init\_\_(self, NI, NH, NO):

# number of nodes in layers

self.ni = NI + 1 # +1 for bias

self.nh = NH

self.no = NO

self.ai, self.ah, self.ao = [],[], []

self.ai = [1.0]\*self.ni

self.ah = [1.0]\*self.nh

self.ao = [1.0]\*self.no

self.wi = makeMatrix (self.ni, self.nh)

self.wo = makeMatrix (self.nh, self.no)

# initialize node weights to random vals

randomizeMatrix ( self.wi, -0.2, 0.2 )

randomizeMatrix ( self.wo, -2.0, 2.0 )

self.ci = makeMatrix (self.ni, self.nh)

self.co = makeMatrix (self.nh, self.no)

defrunNN (self, inputs):

iflen(inputs) != self.ni-1:

print('incorrect number of inputs')

for i in range(self.ni-1):

self.ai[i] = inputs[i]

for j in range(self.nh):

sum = 0.0

for i in range(self.ni):

sum +=( self.ai[i] \* self.wi[i][j] )

self.ah[j] = sigmoid (sum)

for k in range(self.no):

sum = 0.0

for j in range(self.nh):

sum +=( self.ah[j] \* self.wo[j][k] )

self.ao[k] = sigmoid (sum)

return self.ao

defbackPropagate (self, targets, N, M):

output\_deltas = [0.0] \* self.no

for k in range(self.no):

error = targets[k] - self.ao[k]

output\_deltas[k] = error \* dsigmoid(self.ao[k])

for j in range(self.nh):

for k in range(self.no):

change = output\_deltas[k] \* self.ah[j]

self.wo[j][k] += N\*change + M\*self.co[j][k]

self.co[j][k] = change

hidden\_deltas = [0.0] \* self.nh

for j in range(self.nh):

error = 0.0

for k in range(self.no):

error += output\_deltas[k] \* self.wo[j][k]

hidden\_deltas[j] = error \* dsigmoid(self.ah[j])

for i in range (self.ni):

for j in range (self.nh):

change= hidden\_deltas[j] \* self.ai[i]

self.wi[i][j] += N\*change + M\*self.ci[i][j]

self.ci[i][j] = change

error = 0.0

for k in range(len(targets)):

error = 0.5 \* (targets[k]-self.ao[k])\*\*2

return error

def weights(self):

print('Input weights:')

for i in range(self.ni):

print (self.wi[i])

print()

print('Output weights:')

for j in range(self.nh):

print (self.wo[j])

print ('')

def test(self, patterns):

for p in patterns:

inputs = p[0]

print('Inputs:', p[0], '-->', self.runNN(inputs), '\tTarget', p[1])

def train (self, patterns, max\_iterations = 1000, N=0.5, M=0.1):

for i in range(max\_iterations):

for p in patterns:

inputs = p[0]

targets = p[1]

self.runNN(inputs)

error = self.backPropagate(targets, N, M)

if i % 50 == 0:

print('Combined error', error)

self.test(patterns)

def sigmoid (x):

returnmath.tanh(x)

defdsigmoid (y):

return 1 - y\*\*2

defmakeMatrix ( I, J, fill=0.0):

m = []

for i in range(I):

m.append([fill]\*J)

return m

defrandomizeMatrix ( matrix, a, b):

for i in range ( len (matrix) ):

for j in range ( len (matrix[0]) ):

matrix[i][j] = random.uniform(a,b)

def main ():

pat = [

[[0,0], [1]],

[[0,1], [1]],

[[1,0], [1]],

[[1,1], [0]]

]

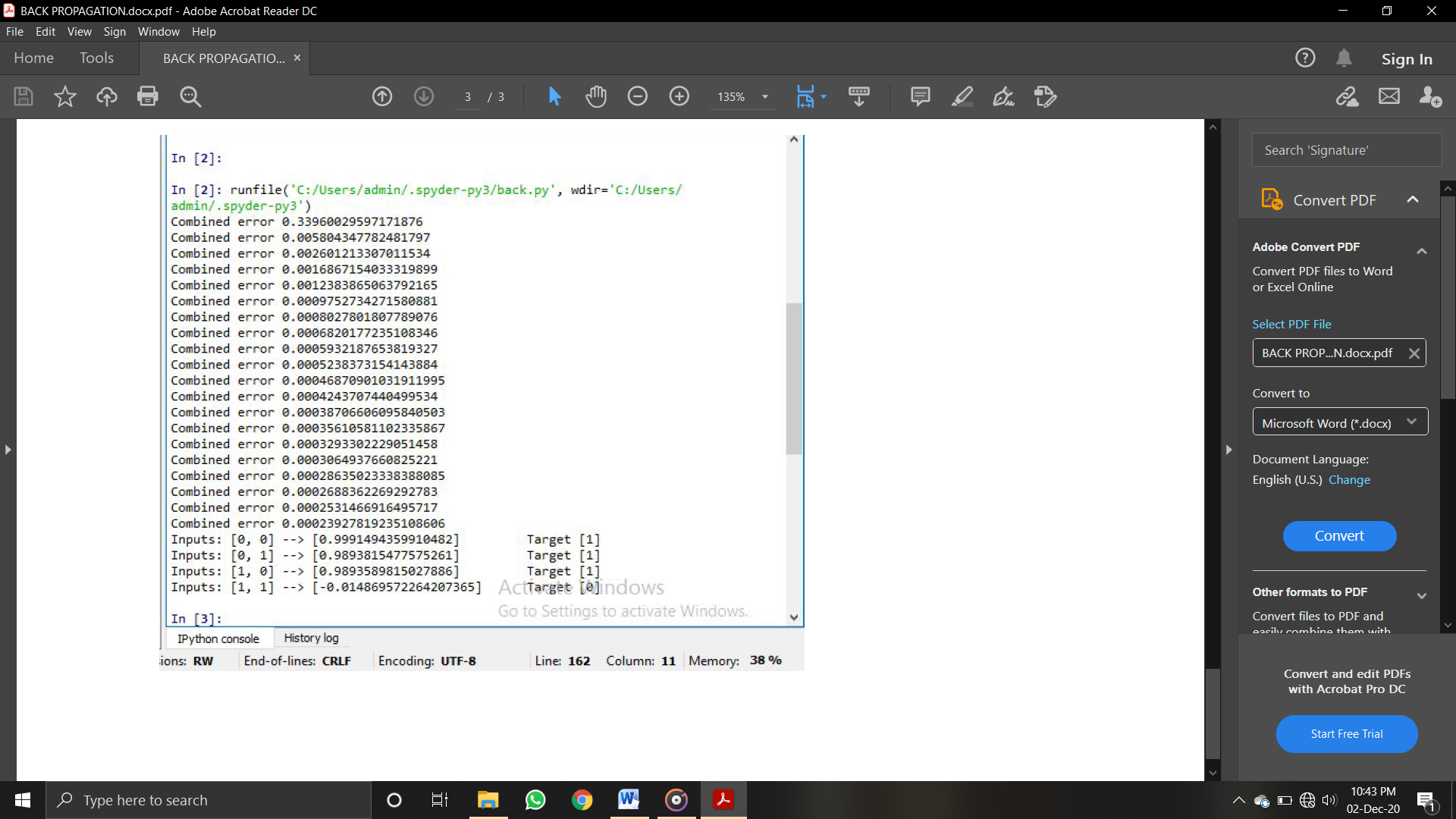
myNN = NN ( 2, 2, 1)

myNN.train(pat)

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

main()

**Output:**



**Program 17**

**Aim:** Write a program to implement AND logic functions.

**Code:**

import numpy as np

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

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

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

theta=1

yin=np.zeros(shape=(4,1))

y=np.zeros(shape=(4,1))

yin=np.dot(x,w)

i=0

found=0

while(found==0):

i=0

yin=np.dot(x,w)

#print(yin)

while(i<4):

if yin[i]>=theta:

y[i]=1

i=i+1

else:

y[i]=0

i=i+1

#print("y",y)

#print("t",t)

if (y==t).all():

print("MODEL IS TRAINED ")

print("\nOutput : \n",y)

print("\nweights : ",w,"\n")

print("theta : ",theta)

found=1

else:

print("MODEL IS NOT TRAINED")

w=np.zeros(shape=(0,0))

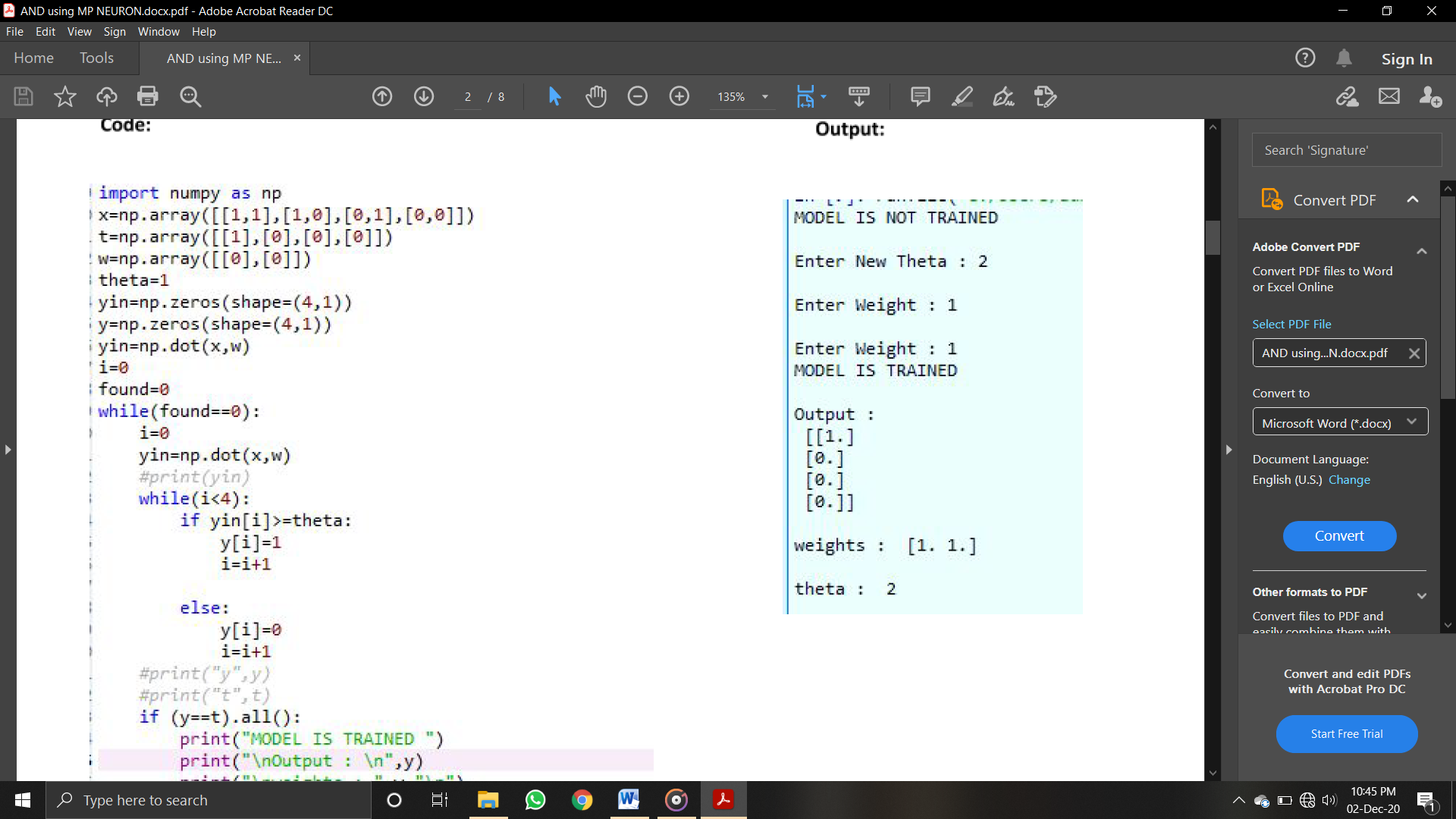
theta=int(input("Enter New Theta : "))

for k in range(int(2)):

w1=int(input("Enter Weight : "))

w=np.append(w,w1)

**Output:**



**Program 18**

**Aim:** Write a program in Python to implement Adaline Neural Network.

**Code:**

Import numpy as np

x1=np.array([[1,1,-1,-1]])

x2=np.array([[1,-1,1,-1]])

t=np.array([[1],[1],[1],[-1]])

w11=0.1

w21=0.1

w01=0.1

alpha=0.1

i=0

bias=1

w1=np.zeros((4,1))

w2=np.zeros((4,1))

w0=np.zeros((4,1))

Yin=np.zeros((4,1))

y=np.zeros((4,1))

error=np.zeros((4,1))

count=0

while(count!=3):

i=0

if(count!=0):

w11=w1[3]

w21=w2[3]

w01=w0[3]

while(i!=4):

if(i==0):

Yin[i]= (x1[0][i]\*w11)+(x2[0][i]\*w21)+(bias\*w01)

y[i]=t[i][0]-Yin[i]

w1[i]=w11+(alpha\*y[i]\*x1[0][i])

w2[i]=w21+(alpha\*y[i]\*x2[0][i])

w0[i]=w01+(alpha\*y[i]\*bias)

else:

if(i>0 & i<=4):

Yin[i]= (x1[0][i]\*w1[i-1])+(x2[0][i]\*w2[i-1])+(bias\*w0[i-1])

y[i]=t[i][0]-Yin[i]

w1[i]=w1[i-1]+(alpha\*y[i]\*x1[0][i])

w2[i]=w2[i-1]+(alpha\*y[i]\*x2[0][i])

w0[i]=w0[i-1]+(alpha\*y[i]\*bias)

error[i]=(y[i])\*\*2

i=i+1

print('EPOCH',(count+1),':')

print('\n')

print('w1:',w1)

print('\n')

print('w2:',w2)

print('\n')

print('w0:',w0)

print('\n')

print('error',error)

print('\n\n')

count=count+1

**Output:**

