CAM

import numpy as np

mem\_vectors=np.array([

    [1,1,0,0,0,0],

    [0,0,0,0,1,1],

    [0,0,1,1,0,0]

])

q=mem\_vectors.shape[0] #number of vectors

n=mem\_vectors.shape[1] #dim of the vectors

bip\_mem\_vecs=2\*mem\_vectors-1 

#Initialize and compute the weight matrix

zd\_wt\_mat=np.zeros((n,n))

for i in range (q):

  zd\_wt\_mat +=np.outer(bip\_mem\_vecs[i], bip\_mem\_vecs[i])

zd\_wt\_mat -=q\*np.eye(n) #zero diag

probe=input("Enter the Probe vector")

probe=np.array([int(x) for x in probe.split()])

signal\_vector=2\*probe-1

print(signal\_vector)

#output

Enter the Probe vector0 0 1 0 1 1 

[-1 -1  1 -1  1  1]

flag = 0 #Intialize flag

while flag !=n:

  permindex=np.random.permutation(n) #Randomize order

  old\_signal\_vector= signal\_vector.copy()

  for j in range(n):

    act\_vec=np.dot(signal\_vector,zd\_wt\_mat)

    if act\_vec[permindex[j]]>0:

      signal\_vector[permindex[j]]=1

    elif act\_vec[permindex[j]]<0:

      signal\_vector[permindex[j]]=-1

  flag = np.dot(signal\_vector,old\_signal\_vector)

print("The recalled vector is :")

print(0.5\*(signal\_vector + 1))

#output

The recalled vector is :   
[0. 0. 1. 1. 1. 1.]

BAM

import numpy as np

n = 5  # Dimension of Fx

p = 4  # Dimension of Fy

q = 2  # Number of associations

mem\_vectors\_x = np.array([[0, 1, 0, 1, 0], [1, 1, 0, 0, 0]])  # Specify Fx vectors

mem\_vectors\_y = np.array([[1, 0, 0, 1], [0, 1, 0, 1]])  # Specify Fy vectors

bip\_mem\_vecs\_x = 2 \* mem\_vectors\_x - 1  # Convert to bipolar

bip\_mem\_vecs\_y = 2 \* mem\_vectors\_y - 1

wt\_matrix = np.zeros((n, p))  # Initialize weight matrix

for i in range(q):

    wt\_matrix += np.outer(bip\_mem\_vecs\_x[i], bip\_mem\_vecs\_y[i])

k = 1  # Set up time index

probe = np.array([0, 1, 0, 1, 1])  # Set up probe

signal\_x = 2 \* probe - 1  # Set Fx signals to probe

signal\_y = np.random.choice([-1, 1], p)  # Randomize Fy signals

pattern\_x = np.zeros((2, n))  # Store patterns on Fx

pattern\_y = np.zeros((2, p))  # Store patterns on Fy  

pattern\_x[0] = signal\_x  # Store the initial pattern on Fx

pattern\_y[0] = signal\_y  # Store the initial pattern on Fy

flag = False  # Indicates bidirectional equilibrium

while not flag:

    act\_y = np.dot(signal\_x, wt\_matrix)  # Compute activations for Fy

    signal\_y = np.where(act\_y > 0, 1, -1)  # Update Fy signals based on activations

    if k >= pattern\_y.shape[0]:

        pattern\_y = np.vstack((pattern\_y, np.zeros((1, p))))  

    if k > 1:

        compare\_y = np.array\_equal(signal\_y, pattern\_y[k - 1])

    else:

        compare\_y = False  

    pattern\_y[k] = signal\_y  # Store the signal on Fy

    act\_x = np.dot(signal\_y, wt\_matrix.T)  # Compute activations for Fx

    signal\_x = np.where(act\_x > 0, 1, -1)  # Update Fx signals based on activations

    if k >= pattern\_x.shape[0]:

        pattern\_x = np.vstack((pattern\_x, np.zeros((1, n))))  

    compare\_x = np.array\_equal(signal\_x, pattern\_x[k - 1]) if k > 1 else False

    pattern\_x[k] = signal\_x  # Store the signal on Fx

    flag = compare\_x and compare\_y  # Check for bidirectional equilibrium

    k += 1  # Increment time

print("Pattern on Fx:")

print(pattern\_x)

print("\nPattern on Fy:")

print(pattern\_y)

Pattern on Fx: [[-1. 1. -1. 1. 1.] [-1. 1. -1. 1. -1.] [-1. 1. -1. 1. -1.]]

Pattern on Fy: [[-1. -1. 1. -1.] [ 1. -1. -1. 1.] [ 1. -1. -1. 1.]]

...