import numpy as np

def decimal\_to\_bipolar(decimal\_value, length):

  binary\_str = format (decimal\_value, f'0{length}b') # Convert to binary string of fixed length

  print(binary\_str)

  return np.array([1 if bit == '1' else -1 for bit in binary\_str]) # Convert to bipolar (-1, 1)

def bipolar\_to\_decimal (bipolar\_vector):

  binary\_str = ''.join(['1' if bit == 1 else '0' for bit in bipolar\_vector])

  return int(binary\_str, 2)

data = [

  (12,345), # ID: 12, Phone: 345

  (23, 678) # ID: 23, Phone: 678

]

id\_length = 8 # 8-bit representation for IDs

phone\_length = 12 # 12-bit representation for phone numbers

# Convert decimal IDs and phone numbers to bipolar vectors

ids\_bipolar = [decimal\_to\_bipolar(id\_val, id\_length) for id\_val, \_ in data]

phones\_bipolar = [decimal\_to\_bipolar(phone\_val, phone\_length) for \_, phone\_val in data]

weight = np.zeros((id\_length, phone\_length))

# Compute the weight matrix (sum of outer products)

for id\_vec, phone\_vec in zip(ids\_bipolar, phones\_bipolar):

  weight += np.outer(id\_vec, phone\_vec)

print(weight)

import numpy as np

step=lambda x: 1 if x>0 else 0

def unitstep(v):

  if v>=0:

    return 1

  else:

    return 0

training\_data=[

    {'input':[1,1,0,0,0,0],'label':1},

    {'input':[1,1,0,0,0,1],'label':0},

    {'input':[1,1,0,0,1,0],'label':1},

    {'input':[1,1,0,1,1,1],'label':0},

    {'input':[1,1,0,1,0,0],'label':1},

    {'input':[1,1,0,1,0,1],'label':0},

    {'input':[1,1,0,1,1,0],'label':1},

    {'input':[1,1,0,0,1,1],'label':0},

    {'input':[1,1,0,0,0,0],'label':1},

    {'input':[1,1,0,0,0,1],'label':0},

]

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

for data in training\_data:

  input=np.array(data['input'])

  label=data['label']

  output=step(np.dot(input.T,weights))

  error=label-output

  weights=weights+input\*error

  print("output",output)

  print("error",error)

  input=np.array([0,0,0,1,1,0])

  def ans(x):

    if x==1:

      return "even"

    else:

      return "odd"

output=ans(np.dot(input,weights))

print(input,"is",output)

import numpy as np

def unitstep(v):

  if v>=0:

    return 1

  else:

    return 0

def perceptron(x,w,b):

  v=np.dot(x,w)+b

  y=unitstep(v)

  return y

def and\_logicFunction(x):

  w=np.array([1,1])

  b=-1.5

  return perceptron(x,w,b)

def or\_logicfunction(x):

  w=np.array([1.5,1.5])

  b=-1.5

  return perceptron(x,w,b)

def not\_logicfunction(x):

  w=-1

  b=0.5

  return perceptron(x,w,b)

def xor\_logicfunction(x):

  y1=and\_logicFunction(x)

  y2=or\_logicfunction(x)

  y3=not\_logicfunction(y1)

  result=np.array([y2,y3])

  return and\_logicFunction(result)

t1=np.array([0,0])

t2=np.array([1,0])

t3=np.array([0,1])

t4=np.array([1,1])

print("XOR({},{})={}".format(0,0,xor\_logicfunction(t1)))

print("XOR({},{})={}".format(1,0,xor\_logicfunction(t2)))

print("XOR({},{})={}".format(0,1,xor\_logicfunction(t3)))

print("XOR({},{})={}".format(1,1,xor\_logicfunction(t4)))

//bidirectional

import numpy as np

#input pattern

x1=np.array([1,1,1,1,1,1]).reshape(6,1)

x2=np.array([-1,-1,-1,-1,-1,-1]).reshape(6,1)

x3=np.array([1,1,-1,-1,1,1]).reshape(6,1)

x4=np.array([-1,-1,1,1,-1,-1]).reshape(6,1)

#output pattern

y1=np.array([1,1,1]).reshape(3,1)

y2=np.array([-1,-1,-1]).reshape(3,1)

y3=np.array([1,-1,1]).reshape(3,1)

y4=np.array([-1,1,-1]).reshape(3,1)

c1=np.concatenate((x1,x2,x3,x4),axis=1)

c2=np.concatenate((y1.T,y2.T,y3.T,y4.T))

print(c1)

print(c2)

w=np.dot(c1,c2)

print(w)

def prediction(x,w):

  net\_sum=np.dot(w.T,x)

  net\_sum[net\_sum<0]=-1

  net\_sum[net\_sum>0]=1

  return net\_sum

print(prediction(x1,w))

print(prediction(x2,w))

print(prediction(x3,w))

print(prediction(x4,w))

print(np.dot(x1.T,x2))

print(np.dot(x1.T,x2))

print(np.dot(x1.T,x4))

import numpy as np

def unitStep(v):

    return np.where(v >= 0, 1, 0)

def sigmoid(v):

  return 1/(1+np.exp(-v))

def linear(x):

    return x

def perceptronModel(x, w, b):

    v = np.dot(x, w.T) + b

    unitStep\_result = unitStep(v)

    sigmoid\_result = sigmoid(v)

    linear\_result = linear(v)

    return unitStep\_result, sigmoid\_result, linear\_result

def Fun(x):

    w = np.array(2.3)

    b = -3

    return perceptronModel(x, w, b)

input\_data = np.array(2.0)

unitStep\_result, sigmoid\_result, linear\_result = Fun(input\_data)

print("Unit Step Function:", unitStep\_result)

print("Sigmoid Function:", sigmoid\_result)

print("Linear Function:", linear\_result)

import numpy as np

# Step function

def step(x):

  return 1 if x >= 0 else 0

# Training data for NOT gate

training\_data\_not = [

    {'input': [0], 'label': 1},

    {'input': [1], 'label': 0},

]

# Initial weights and bias for NOT gate

weights\_not = np.array([0.5])

bias\_not = 0.1

# Perceptron learning for NOT gate

for data in training\_data\_not:

  input\_data = np.array(data['input'])

  label = data['label']

  output = step(np.dot(input\_data.T, weights\_not) + bias\_not)

  error = label - output

  weights\_not += error \* input\_data

  bias\_not += error

  print("NOT Gate - Output:", output)

  print("NOT Gate - Error:", error)

  print("NOT Gate - Weights:", weights\_not)

  print("NOT Gate - Bias:", bias\_not)

  # Training data for XOR gate

training\_data\_xor = [

    {'input': [0, 0], 'label': 0},

    {'input': [0, 1], 'label': 1},

    {'input': [1, 0], 'label': 1},

    {'input': [1, 1], 'label': 0},

]

# Initial weights and bias for XOR gate

weights\_xor = np.array([0.5, 0.5])

bias\_xor = 0.1

# Perceptron learning for XOR gate (This is a simplified XOR implementation, it might not converge perfectly)

for data in training\_data\_xor:

  input\_data = np.array(data['input'])

  label = data['label']

  output = step(np.dot(input\_data.T, weights\_xor) + bias\_xor)

  error = label - output

  weights\_xor += error \* input\_data

  bias\_xor += error

  print("XOR Gate - Output:", output)

  print("XOR Gate - Error:", error)

  print("XOR Gate - Weights:", weights\_xor)

  print("XOR Gate - Bias:", bias\_xor)

import numpy as np

def step(x):

return 1 if x >= 0 else 0

# Training data

training\_data = [

{'input': [2.0], 'label': 1},

{'input': [1.7], 'label': 1},

{'input': [1.3], 'label': 0},

{'input': [1.8], 'label': 1},

{'input': [1.2], 'label': 0},

{'input': [1.0], 'label': 0},

]

# Initial weights

weights = np.array([2.3])

b = -3

# Perceptron learning

for data in training\_data:

input\_data = np.array(data['input'])

label = data['label']

output = step(np.dot(input\_data.T, weights))

error = label - output

weights += error \* input\_data

print("output", output)

print("Error:", error)

print("weights:", weights)

print("b:", b)

import numpy as np

def unitstep(v):

  print("UNITSTEP")

  if v>=0:

    return 1

  else:

    return 0

def signfunc(v):

   print("SIGNFUNCTION")

   if v>=0:

     return 1

   else:

     return -1

def linear(v):

   print("LINEAR FUNCTION")

   return v

def perceptron(x):

   w=np.array([3,2])

   b=1.2

   v=np.dot(x,w)+b

   return unitstep(v), signfunc(v), linear(v)

   #return

   #return

#input

test1=np.array([-5,6])

perceptron(test1)