Hunter Leise

CSCI 3202

Problem Set 4

**Problem 4.1**

Code:

import random  
import math  
  
class Perceptron:  
 *"""  
 Represents a perceptron and all of the methods necessary to train it.  
  
 Attributes:  
 num\_inputs (int): number of inputs to the perceptron  
 correct (function([list of inputs]) --> bool): function that returns  
 the correct output given a list of inputs. Used for training.  
 bias (float): initial perceptron bias  
 weights (list of float): list of input weights  
 """* def \_\_init\_\_(self, num\_inputs, correct\_func, bias):  
 self.num\_inputs = num\_inputs  
 self.correct = correct\_func  
 self.bias = bias  
 self.weights = self.initialize\_weights(self, num\_inputs, -1, 1)  
  
 @staticmethod  
 def initialize\_weights(*self*, num\_weights, range\_min, range\_max):  
 *"""  
 Randomize initial weights before training.  
  
 Args:  
 num\_weights (int): number of weights to create  
 range\_min (int): minimum weight for randomization  
 range\_max (int): maximum weight for randomization  
  
 Returns:  
 list of floats: list of weights for each input  
 """* weights = []  
 for \_ in range(num\_weights):  
 weights.append(random.uniform(range\_min, range\_max))  
 return weights  
  
 @staticmethod  
 def prediction(*self*, output):  
 *"""  
 Takes a sigmoid output value and returns its corresponding  
 boolean value.  
  
 Args:  
 output (float): sigmoid activation value  
 Returns:  
 bool: True if output >= 0.5  
 False otherwise  
 """* return True if output >= 0.5 else False  
  
 def train(self, iterations, learning\_rate):  
 *"""  
 Train the perceptron for the given number of iterations.  
  
 Args:  
 iterations (int): number of training iterations  
 learning\_rate (float): learning rate  
 Returns:  
 None  
 """* for i in range(iterations):  
 if i % 250 == 0: *# print weights every 250 iterations* print(self.weights)  
 inputs = [random.randint(0, 1) for \_ in range(self.num\_inputs)]  
 output = self.sig\_output(inputs)  
 if self.prediction(self, output) != get\_correct(inputs):  
 derivative = output \* (1.0 - output)  
 error = self.correct(inputs) - output  
 self.update(learning\_rate, error, derivative, inputs)  
  
 def sig\_output(self, inputs):  
 *"""  
 Calculate sigmoid activation function output  
  
 Args:  
 inputs (list of int): list of binary inputs  
 Returns:  
 float: sigmoid activation value  
 """* weighted\_sum = self.bias  
 for i in range(len(inputs)):  
 weighted\_sum += inputs[i] \* self.weights[i]  
 return 1.0 / (1.0 + math.exp(-weighted\_sum))  
  
 def update(self, learning\_rate, error, derivative, inputs):  
 *"""  
 Update the weights using the perceptron update function  
  
 Args:  
 learning\_rate (float): learning rate  
 error (float): calculated error  
 derivative (float): sigmoid derivative  
 inputs (list of int): list of perceptron binary inputs  
 Returns:  
 None  
 """* for i in range(len(self.weights)):  
 self.weights[i] = self.weights[i] + \  
 (learning\_rate \* error \* derivative \* inputs[i])  
  
  
  
  
  
  
  
  
  
  
  
def get\_correct(inputs):  
 *"""  
 Return the 'and' function of inputs 1 and 3.  
  
 Args:  
 inputs (list of int): list of binary inputs  
 Returns:  
 int: 1 for True (inputs 1 and 3 are both 1)  
 0 for False (either inputs 1 or 3 are 0)  
 """* return int(inputs[0] and inputs[2])  
  
  
def all\_correct(perceptron):  
 *"""  
 Check if all possible 3 input combinations return the expected values.  
  
 Args:  
 perceptron (Perceptron): perceptron class instance  
 Returns:  
 bool: True if all input possibilities return the expected values,  
 False otherwise.  
 """* for i in range(2):  
 for j in range(2):  
 for k in range(2):  
 output = perceptron.sig\_output([i, j, k])  
 prediction = perceptron.prediction(perceptron, output)  
 if prediction != get\_correct([i, j, k]):  
 return False  
 return True  
  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 perceptron = Perceptron(3, get\_correct, -1)  
 perceptron.train(8000, 1.0)  
 print(all\_correct(perceptron))

Example Output:

[0.04605156097206797, 0.4743032771212059, -0.42877117354423055]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
[0.5998231184667563, 0.3382992044547937, 0.5424929969569187]  
True

**Problem 4.2**

Code:

import random  
  
  
class City:  
 *"""  
 Represents a ring city and the necessary methods for how it operates.  
  
 Attributes:  
 neighbors (list of int): list of neighbor types  
 vacant (list of int): indexes of vacant homes in the neighbors list.  
 """* def \_\_init\_\_(self):  
 self.neighbors = [0] \* 6 + [1] \* 27 + [2] \* 27  
 random.shuffle(self.neighbors)  
 self.vacant = [i for i, x in enumerate(self.neighbors) if x == 0]  
  
 def iterate(self, iterations):  
 *"""  
 Iterate until all neighbors are satisfied or until the maximum  
 number of iterations is reached.  
  
 Args:  
 iterations (int): Maximum number of times the cycle will be run  
  
 Returns:  
 None  
 """* for iteration in range(iterations):  
 if iteration % 20 == 0: *# print every 20 iterations* print(self.neighbors)  
  
 dissatisfied = []  
 for i in range(len(self.neighbors)):  
 if self.neighbors[i] != 0 and not self.is\_satisfied(i):  
 dissatisfied.append(i)  
 if len(dissatisfied) == 0: *# Yay! Everyone is satisfied!* print(**"all satisfied"**)  
 break  
 else: *# Move unsatisfied people to vacant house* rand\_dis = random.choice(dissatisfied)  
 self.neighbors[self.vacant[0]] = self.neighbors[rand\_dis]  
 self.neighbors[rand\_dis] = 0  
 self.vacant.pop(0)  
 self.vacant.append(rand\_dis)  
  
 def is\_satisfied(self, house):

*"""  
 Returns whether a given house is satisfied. In other words,  
 whether it has at least two neighbors of its own type within  
 two houses on either side.  
  
 Args:  
 house (int): index for the queried house  
  
 Returns:  
 bool: True if satisfied, False if not satisfied  
 """* same\_neighbors = 0  
 for i in range(1, 3):  
 if self.neighbors[(house + i) % len(self.neighbors)] \  
 == self.neighbors[house]:  
 same\_neighbors += 1  
 if self.neighbors[(house - i) % len(self.neighbors)] \  
 == self.neighbors[house]:  
 same\_neighbors += 1  
 return same\_neighbors >= 2  
  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 city = City()  
 city.iterate(400)  
 print(city.neighbors) *# Print final neighborhood*

Example Output:

[1, 1, 1, 2, 0, 0, 1, 2, 2, 2, 1, 1, 1, 2, 1, 2, 0, 1, 1, 0, 2, 2, 2, 2, 2, 2, 1, 1, 1, 2, 1, 2, 1, 1, 1, 2, 1, 0, 1, 1, 2, 1, 2, 2, 2, 1, 2, 1, 2, 2, 0, 2, 1, 2, 1, 1, 2, 2, 1, 2]  
  
[1, 1, 1, 2, 2, 1, 1, 2, 2, 2, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 1, 1, 1, 2, 1, 0, 1, 1, 1, 0, 1, 2, 1, 1, 2, 2, 2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 1, 0, 1, 0, 0, 2, 1, 2]  
  
[1, 1, 1, 0, 1, 0, 0, 2, 2, 2, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 1, 1, 1, 0, 1, 1, 1, 1, 1, 2, 1, 1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 0, 2, 2, 2, 2, 1, 1]  
  
all satisfied   
  
[1, 1, 1, 0, 0, 0, 0, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 2, 2, 2, 2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1]

The ring city does reach a totally satisfied state, and in surprisingly few iterations! This generally looks like a few very large groupings, such as the large grouping of two near the end of the example above, combined with a few smaller groupings.