Challenge #10

1. Code and its bottlenecks

# -\*- coding: utf-8 -\*-

"""

Created on Sat Mar 28 12:59:23 2020

Assignment 2 - Agents and Reinforcement Learning

@author: Ronan Murphy - 15397831

"""

import numpy as np

import random

import matplotlib.pyplot as plt

#set the rows and columns length

BOARD\_ROWS = 5

BOARD\_COLS = 5

#initalise start, win and lose states

START = (0, 0)

WIN\_STATE = (4, 4)

HOLE\_STATE = [(1,0),(3,1),(4,2),(1,3)]

#class state defines the board and decides reward, end and next position

class State:

def \_\_init\_\_(self, state=START):

#initalise the state to start and end to false

self.state = state

self.isEnd = False

def getReward(self):

#give the rewards for each state -5 for loss, +1 for win, -1 for others

for i in HOLE\_STATE:

if self.state == i:

return -5

if self.state == WIN\_STATE:

return 1

else:

return -1

def isEndFunc(self):

#set state to end if win/loss

if (self.state == WIN\_STATE):

self.isEnd = True

for i in HOLE\_STATE:

if self.state == i:

self.isEnd = True

def nxtPosition(self, action):

#set the positions from current action - up, down, left, right

if action == 0:

nxtState = (self.state[0] - 1, self.state[1]) #up

elif action == 1:

nxtState = (self.state[0] + 1, self.state[1]) #down

elif action == 2:

nxtState = (self.state[0], self.state[1] - 1) #left

else:

nxtState = (self.state[0], self.state[1] + 1) #right

#check if next state is possible

if (nxtState[0] >= 0) and (nxtState[0] <= 4):

if (nxtState[1] >= 0) and (nxtState[1] <= 4):

#if possible change to next state

return nxtState

#Return current state if outside grid

return self.state

#class agent to implement reinforcement learning through grid

class Agent:

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

#inialise states and actions

self.states = []

self.actions = [0,1,2,3] # up, down, left, right

self.State = State()

#set the learning and greedy values

self.alpha = 0.5

self.gamma = 0.9

self.epsilon = 0.1

self.isEnd = self.State.isEnd

# array to retain reward values for plot

self.plot\_reward = []

#initalise Q values as a dictionary for current and new

self.Q = {}

self.new\_Q = {}

#initalise rewards to 0

self.rewards = 0

#initalise all Q values across the board to 0, print these values

for i in range(BOARD\_ROWS):

for j in range(BOARD\_COLS):

for k in range(len(self.actions)):

self.Q[(i, j, k)] =0

self.new\_Q[(i, j, k)] = 0

print(self.Q)

#method to choose action with Epsilon greedy policy, and move to next state

def Action(self):

#random value vs epsilon

rnd = random.random()

#set arbitraty low value to compare with Q values to find max

mx\_nxt\_reward =-10

action = None

#9/10 find max Q value over actions

if(rnd >self.epsilon) :

#iterate through actions, find Q value and choose best

for k in self.actions:

i,j = self.State.state

nxt\_reward = self.Q[(i,j, k)]

if nxt\_reward >= mx\_nxt\_reward:

action = k

mx\_nxt\_reward = nxt\_reward

#else choose random action

else:

action = np.random.choice(self.actions)

#select the next state based on action chosen

position = self.State.nxtPosition(action)

return position,action

#Q-learning Algorithm

def Q\_Learning(self,episodes):

x = 0

#iterate through best path for each episode

while(x < episodes):

#check if state is end

if self.isEnd:

#get current rewrard and add to array for plot

reward = self.State.getReward()

self.rewards += reward

self.plot\_reward.append(self.rewards)

#get state, assign reward to each Q\_value in state

i,j = self.State.state

for a in self.actions:

self.new\_Q[(i,j,a)] = round(reward,3)

#reset state

self.State = State()

self.isEnd = self.State.isEnd

#set rewards to zero and iterate to next episode

self.rewards = 0

x+=1

else:

#set to arbitrary low value to compare net state actions

mx\_nxt\_value = -10

#get current state, next state, action and current reward

next\_state, action = self.Action()

i,j = self.State.state

reward = self.State.getReward()

#add reward to rewards for plot

self.rewards +=reward

#iterate through actions to find max Q value for action based on next state action

for a in self.actions:

nxtStateAction = (next\_state[0], next\_state[1], a)

q\_value = (1-self.alpha)\*self.Q[(i,j,action)] + self.alpha\*(reward + self.gamma\*self.Q[nxtStateAction])

#find largest Q value

if q\_value >= mx\_nxt\_value:

mx\_nxt\_value = q\_value

#next state is now current state, check if end state

self.State = State(state=next\_state)

self.State.isEndFunc()

self.isEnd = self.State.isEnd

#update Q values with max Q value for next state

self.new\_Q[(i,j,action)] = round(mx\_nxt\_value,3)

#copy new Q values to Q table

self.Q = self.new\_Q.copy()

#print final Q table output

print(self.Q)

#plot the reward vs episodes

def plot(self,episodes):

plt.plot(self.plot\_reward)

plt.show()

#iterate through the board and find largest Q value in each, print output

def showValues(self):

for i in range(0, BOARD\_ROWS):

print('-----------------------------------------------')

out = '| '

for j in range(0, BOARD\_COLS):

mx\_nxt\_value = -10

for a in self.actions:

nxt\_value = self.Q[(i,j,a)]

if nxt\_value >= mx\_nxt\_value:

mx\_nxt\_value = nxt\_value

out += str(mx\_nxt\_value).ljust(6) + ' | '

print(out)

print('-----------------------------------------------')

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

#create agent for 10,000 episdoes implementing a Q-learning algorithm plot and show values.

ag = Agent()

episodes = 10000

ag.Q\_Learning(episodes)

ag.plot(episodes)

ag.showValues()  
  
Key bottlenecks in the code: -

| **Bottleneck Area** | **Why It's a Bottleneck** | **Optimization Tip** |
| --- | --- | --- |
| 1. **Q-table Updates** | Q-table is updated in every step by scanning all possible actions again—even if only one action was taken. Uses nested loops across all states and actions. | Use vectorized data structures (e.g., NumPy arrays). |
| 2. **State Lookup via Dictionary** | Q-values stored in a Python dictionary self.Q[(i, j, a)], which is slow compared to direct array access. | Switch to a NumPy array Q[i][j][a]. |
| 3. **Action Selection Loop** | For every step, max Q-value over actions is computed using a loop, which slows things down during training. | Replace with np.argmax(Q[i][j]) in a NumPy-based version. |
| 4. **New Q-value Dictionary** | Every time step uses new\_Q.copy() to duplicate the dictionary, which is expensive for large episodes. | Update Q-table in-place or use NumPy arrays. |
| 5. **Redundant Object Recreation** | self.State = State() is called repeatedly, creating a new object even for trivial state transitions. | Use a mutable object or simply update the state tuple. |
| 6. **Inefficient Reward Accumulation** | The reward is fetched every step with method calls and control logic instead of being computed once during transition. | Refactor reward logic into transition function. |
| 7. **Matplotlib Plotting (after 10k episodes)** | Plotting 10,000 points at once without smoothing or downsampling can slow rendering. | Smooth rewards or sample fewer points. |

1. Do the suggestions make sense? How well is it able to identity bottlenecks?

Yes, the suggestions make sense. The GPT was able to identify key computational bottlenecks that align with. For example, it pointed out how using a dictionary for the Q-table and looping over all actions in every step adds overhead, especially when scaling to thousands of episodes. I had not thought much about how the repeated object creation could slow things down, so that was insightful.

It also highlighted how switching to NumPy arrays or vectorized operations could improve speed, which is something I know works well in Python for optimization. Overall, I think the assistant did a good job identifying both algorithmic and implementation-level inefficiencies.

1. Hardware implementation of the biggest bottleneck.

GPT identified the biggest bottleneck as the Q-table update and access. Below is the hardware implementation of the same.

Q(s, a) = (1 - α) \* Q(s, a) + α \* (reward + γ \* max(Q(s’, a’)))

This can be offloaded as a hardware module that takes in:

* current state (s)
* next state (s’)
* action (a)
* reward
* current Q-values
* parameters α and γ

And outputs the updated Q-value.

// Q-learning update module

module q\_learning\_unit #(

parameter DATA\_WIDTH = 16,

parameter STATE\_WIDTH = 5,

parameter ACTION\_WIDTH = 2

)(

input logic clk,

input logic rst,

input logic [STATE\_WIDTH-1:0] s\_i,

input logic [STATE\_WIDTH-1:0] s\_j,

input logic [ACTION\_WIDTH-1:0] a,

input logic [STATE\_WIDTH-1:0] s\_prime\_i,

input logic [STATE\_WIDTH-1:0] s\_prime\_j,

input logic signed [DATA\_WIDTH-1:0] reward,

input logic signed [DATA\_WIDTH-1:0] Q\_sa,

input logic signed [DATA\_WIDTH-1:0] Q\_max\_sp,

input logic [DATA\_WIDTH-1:0] alpha,

input logic [DATA\_WIDTH-1:0] gamma,

output logic signed [DATA\_WIDTH-1:0] Q\_sa\_updated,

output logic done

);

logic signed [DATA\_WIDTH-1:0] q1, q2, q\_target;

typedef enum logic [1:0] {IDLE, CALC\_TARGET, CALC\_Q, DONE} fsm\_t;

fsm\_t current\_state, next\_state;

always\_ff @(posedge clk or posedge rst) begin

if (rst)

current\_state <= IDLE;

else

current\_state <= next\_state;

end

always\_comb begin

next\_state = current\_state;

case (current\_state)

IDLE: next\_state = CALC\_TARGET;

CALC\_TARGET: next\_state = CALC\_Q;

CALC\_Q: next\_state = DONE;

DONE: next\_state = IDLE;

endcase

end

always\_ff @(posedge clk) begin

if (current\_state == CALC\_TARGET)

q\_target <= reward + ((gamma \* Q\_max\_sp) >>> 4); // fixed-point scale

if (current\_state == CALC\_Q) begin

q1 <= ((16 - alpha) \* Q\_sa) >>> 4;

q2 <= (alpha \* q\_target) >>> 4;

end

if (current\_state == DONE)

Q\_sa\_updated <= q1 + q2;

end

assign done = (current\_state == DONE);

endmodule

// Top module

module top\_q\_learning #(

parameter DATA\_WIDTH = 16,

parameter STATE\_WIDTH = 5,

parameter ACTION\_WIDTH = 2,

parameter ACTIONS = 4

)(

input logic clk,

input logic rst

);

logic [DATA\_WIDTH-1:0] alpha = 16'd8; // 0.5 in Q4.12

logic [DATA\_WIDTH-1:0] gamma = 16'd14; // 0.875 in Q4.12

logic [STATE\_WIDTH-1:0] s\_i = 5'd2, s\_j = 5'd2;

logic [STATE\_WIDTH-1:0] s\_prime\_i = 5'd3, s\_prime\_j = 5'd2;

logic [ACTION\_WIDTH-1:0] a = 2'd1;

logic signed [DATA\_WIDTH-1:0] reward = -16'sd1;

logic signed [DATA\_WIDTH-1:0] Q\_table [0:4][0:4][0:ACTIONS-1];

logic signed [DATA\_WIDTH-1:0] Q\_sa;

logic signed [DATA\_WIDTH-1:0] Q\_max\_sp;

logic signed [DATA\_WIDTH-1:0] Q\_sa\_updated;

logic done, done\_d, done\_2d;

// Instantiate the Q-learning unit

q\_learning\_unit #(

.DATA\_WIDTH(DATA\_WIDTH),

.STATE\_WIDTH(STATE\_WIDTH),

.ACTION\_WIDTH(ACTION\_WIDTH)

) q\_unit (

.clk(clk),

.rst(rst),

.s\_i(s\_i),

.s\_j(s\_j),

.a(a),

.s\_prime\_i(s\_prime\_i),

.s\_prime\_j(s\_prime\_j),

.reward(reward),

.Q\_sa(Q\_sa),

.Q\_max\_sp(Q\_max\_sp),

.alpha(alpha),

.gamma(gamma),

.Q\_sa\_updated(Q\_sa\_updated),

.done(done)

);

// Done delay chain

always\_ff @(posedge clk or posedge rst) begin

if (rst) begin

done\_d <= 0;

done\_2d <= 0;

end else begin

done\_d <= done;

done\_2d <= done\_d;

end

end

// Single always\_ff block for ALL Q\_table operations

always\_ff @(posedge clk) begin

if (rst) begin

Q\_sa <= 0;

Q\_max\_sp <= 0;

for (int i = 0; i < 5; i++) begin

for (int j = 0; j < 5; j++) begin

for (int k = 0; k < ACTIONS; k++) begin

Q\_table[i][j][k] <= 0;

end

end

end

end else begin

// Read Q values

Q\_sa <= Q\_table[s\_i][s\_j][a];

Q\_max\_sp <= Q\_table[s\_prime\_i][s\_prime\_j][0];

for (int i = 1; i < ACTIONS; i++) begin

if (Q\_table[s\_prime\_i][s\_prime\_j][i] > Q\_max\_sp)

Q\_max\_sp <= Q\_table[s\_prime\_i][s\_prime\_j][i];

end

// ✅ Write Q\_sa\_updated after 2-clock delay

if (done\_2d)

Q\_table[s\_i][s\_j][a] <= Q\_sa\_updated;

end

end

endmodule

Below is the testbench code.

`timescale 1ns / 1ps

module testbench\_q\_learning;

parameter DATA\_WIDTH = 16;

parameter STATE\_WIDTH = 5;

parameter ACTION\_WIDTH = 2;

logic clk;

logic rst;

top\_q\_learning #(

.DATA\_WIDTH(DATA\_WIDTH),

.STATE\_WIDTH(STATE\_WIDTH),

.ACTION\_WIDTH(ACTION\_WIDTH)

) dut (

.clk(clk),

.rst(rst)

);

// Clock generation

always #5 clk = ~clk;

// Main test sequence

initial begin

$display("Running Q-learning update in hardware...");

clk = 0;

rst = 1;

$dumpfile("dump.vcd");

$dumpvars(0, testbench\_q\_learning);

#20;

rst = 0;

// Wait for FSM to raise done signal

wait (dut.done == 1);

// Wait 2 extra clock cycles to allow Q\_table write to complete

repeat (2) @(posedge clk);

// Now print all values safely

$display("Q\_sa = %0d", dut.Q\_sa);

$display("Q\_max\_sp = %0d", dut.Q\_max\_sp);

$display("Updated Q(s,a) = %0d", dut.Q\_sa\_updated);

$display("Q\_table[%0d][%0d][%0d] = %0d",

dut.s\_i, dut.s\_j, dut.a,

dut.Q\_table[dut.s\_i][dut.s\_j][dut.a]);

#20;

$finish;

end

endmodule

After executing the above code in EDA Playground, I saw that the Q-table is not getting updated properly. After giving GPT with extensive prompts, it was not able modify it. Below is the output expected:

Q\_sa = 0

Q\_max\_sp = 0

Updated Q(s,a) = -2048

Q\_table[2][2][1] = -2048  
  
Obtained output:

Q\_sa = 0  
Q\_max\_sp = 0  
Updated Q(s,a) = 4095  
Q\_table[2][2][1] = 0