Machine Learning Engineer Nanodegree

Project 4: Reinforcement Learning

This time I submit two pieces of code, with different strategies of updating Q-learning. 'agent.py' uses 'PreviousState' and 'CurrentState'. 'agent_v2.py' uses 'CurrentState' and 'NextState'. Both of them can give correct results. I should mention here that I have add a few lines in environment.py so I can write the output in a txt file.

Part 1 Implement a basic agent

Q: what you see in the agent's behavior. Does it eventually make it to the target location?

A: The agent does random walk on the grid, just like a Brownian particle. Since now it can not response to environment, the agent often crashes other dummy agents or goes through red light. It can hardly reach the destination within time limit. If given long enough time, it will finally reach the destination, but this is not what we want.
Results for 1000 independent trials of a random walker is stored in 'logfile\randomWalk.txt'. Success rate is only 0.254.

Part 2 Identify and update state

Q: *Justify why you picked these set of states, and how they model the agent and its environment.*

A: Each state of the agent should contain two parts, one is which direction the agent prefer to go, following the guidance of the planner (self.planner.next_waypoint), and the other is environment information from environment sensor (self.env.sense). So possible states of the agent is every conceivable combination in the following dictionary.
{'guide': [None, 'forward', 'left', 'right'], 'light':['green', 'red'], 'oncoming': [None, 'forward', 'left', 'right'], 'left': [None, 'forward', 'left', 'right'], 'right': [None, 'forward', 'left', 'right']}

There are 4*2*4*4*4 = 512 possbile state. It is quite a large number that I can never write them down. Thus in practice, I just create a null dictionary named Qtable at the beginning. Once a new state appears, I add it to this dict and set the key like

(('guide', 'forward'), ('light', 'green'), ('oncoming', None), ('left', None), ('right', 'forward')). It is a tuple. This method makes the learning process of Q-value much more convenient.

However, there is another input I don't take into account, the deadline. One may argue this feature could affect the decision, but I prefer to remove it because it has too many possible values and will make state space quite complicated and slow down learning process.

reply to the review last time###
Since the agent don't know any traffic rules, I think it is necessary to record all possible states. During 1000 training trials, I write the states in Qtable in a txt file 'StateAppear.txt'. Only 36 states appear. It is already a small sub set. Maybe the reason is there are only 3 dummy agents. The probability for two or three of them locating in the same intersection is really small.

Part 3 Implement Q-Learning

Q: *What changes do you notice in the agent's behavior?*

A: In this part, I apply the Q-Learning algorithm to my agent:
Q(s,a) = (1-alpha)*Q(s,a)+alpha*(reward + gamma*max Q(s',a')).
In the algorithm, there are 3 basic parameters, the learning rate alpha, the discount gamma, and a probability to reject the optimal decision epsilon.

I have debug my last version of 'agent.py' carefully. If I change self.previousReward into reward as you said, success rate of the agent will be extremely low. See 'logfile_wrong.txt'. Acturally the expression in my code is like

Q(previsous S, previous A) = (1-γ)*Q(previous S, previous A) + α*(previous Reward + γ*max(Q(current S,~)

Comparing the two expressions above, you can see that previous state is corresponding to s, and current state is corresponding to s'. So it is correct to use self.previousReward. What I did is to update the Q – matrix for last step in the current run.
I have recorded the output of agent using 'reviousReward' in 'logfile_best.txt'. The success rate this time is high. ###

This time I follow your advice and write two function 'Alpha_decay' and 'Epsilon_decay' in class Learning Agent. One can drop the dacay function by setting

self.decayValid = False.

This time, I tried alpha = 0.5, gamma = 0.5 and epsilon = 0.5.
I find the frequency agent breaking traffic rules gradually decreases. It indicates Q-learning algorithm helps the agent choose the optimal action. As a result, the cab can get to the destination within the deadline more frequently. Results of a basic agent can be found in 'logfile\basicQlearningAgent.txt'.
However, since epsilon doesn't decay, the agent still have a large chance to randomly choose a direction Success rate has increased to approximately 0.56 (a random walker is 0.25). It is still not good enough.

Part 4 Enhance the driving agent

Q: *Report what changes you made to your basic implementation of Q-Learning to achieve the final version of the agent. How well does it perform?*
Does your agent get close to finding an optimal policy, i.e. reach the destination in the minimum possible time, and not incur any penalties?

A: Based on what I learn, Alpha should decay as 1/t as iteration goes on. And epsilon also need to gradually decrease. I set the day function of epsilon like epsilon = epsilon_0/(1+0.05*t). Here t represents how many times state s has been visited. To activate the decay function, set self.decayValid = True.

reply to review last time###
Last time I initialize the Q-matrix for new state to be [0,0,0,0]. So argmax(Q) returns the first value, where represents 'None'. I fix this by using np.random.rand(4) instead. What's more, I find that action = None is somehow a local stationary point. I ran 100 independent training process. Sometimes the code works well, but sometimes the same code provides really low success rate, just as you find last time. In these bad conditions, the agent select 'None' many times.
Possible explanation is, when the agent choose to reject optimal action based on Q-matrix, the agent has 1/4 chance to choose a 'None' action. At this moment, the agent will get no reward or penalty. So the agent gains no information. If it is unlucky enough T_T, the Q-values don't change much before epsilon goes to zero. It often happens when beginning epsilon is small.
To fix this problem, I enforce the agent to go 'forward', 'left' or 'right' when it doesn't know where to go.
####
Then I try to find the optimal parameters from the following parameter grid:
{'alpha': [0.3,0.6,0.9], 'gamma': [0.3,0.6,0.9], 'epsilon': [0.3,0.6,0.9]}.
For each combination I run 10 independent training process. The average Success rate are presented in the following 3 tables.

Alpha = 0.3

	Gamma = 0.3	Gamma = 0.6	Gamma = 0.9
Epsilon = 0.3	0.833	0.604	0.396
Epsilon = 0.6	0.888	0.856	0.877
Epsilon = 0.9	0.885	0.812	0.766

Alpha = 0.6

	Gamma = 0.3	Gamma = 0.6	Gamma = 0.9
Epsilon = 0.3	0.948	0.874	0.626
Epsilon = 0.6	0.919	0.886	0.833
Epsilon = 0.9	0.901	0.888	0.872

Alpha = 0.9

	Gamma = 0.3	Gamma = 0.6	Gamma = 0.9
Epsilon = 0.3	0.943	0.844	0.794
Epsilon = 0.6	0.940	0.910	0.882
Epsilon = 0.9	0.9	0.888	0.851

According to the table above,{alpha = 0.6, gamma = 0.3,epsilon = 0.3}, {alpha = 0.9, gamma = 0.3,epsilon = 0.3},{alpha = 0.9, gamma = 0.3,epsilon = 0.6} are the top3.

I keep these three, and compare their performance on different epsilon decay rate

	Decay = 0.05	Decay = 0.1	Decay = 0.2
{alpha = 0.6, gamma = 0.3,epsilon = 0.3}	0.948	0.916	0.938
{alpha = 0.9, gamma = 0.3,epsilon = 0.3}	0.943	0.895	0.877
{alpha = 0.9, gamma = 0.3,epsilon = 0.6}	0.940	0.938	0.952

Combine all tables above, the optimal parameters are
{alpha = 0.9, gamma = 0.3,epsilon = 0.6}, and epsilon decrease according to the function epsilon = epsilon_0/(1+0.2*t)

In my mind, the optimal policy should be:
1 keep moving following the guide of planner.next_waypoint(), as long as it is allowed by the traffic rules.
2 If such action is forbidden, choose to wait at the present intersection.
This policy will enable the agent to reach destination within deadline and get none penalty.
As to the agent with best parameters, its final behavior is quite similar to this policy. The only difference I find is, when light is red, it sometimes turn right to find another way round instead of just wait there. Most of the trials it behaves perfect.