Smartcab Q-learning Report

This report was part of reinforcement learning project on Udacity, the goal was to train an AI driving agent for the smartcab which should receive inputs at each time step t, and generate an output move. Based on the rewards and penalties it gets, the agent should learn an optimal policy for driving on city roads, obeying traffic rules correctly, and trying to reach the destination within a goal time. The original data and codes were provided by the instructor.
My work concentrated on:

- Test the influences of different policies on the performance of the agent
- Define elements of a state based on the given information and programs
- Apply Reinforcement Learning method, exactly Q-learning method on the agent, train the agent to find the destination.
- tune parameters to make the smartcab reach the destination within deadlines in optimal ways.

Summary

First I set that the agent picked actions randomly, it could not find next way properly and failed to reach the destination within deadline many times. Then, I used Q-learning method to update the state and picked optimal action, the agent did not perform well initially, and gradually it could find right actions to reach the destination. Finally I tuned the learning parameters such as learning rate *alpha*, discount factor *gamma*, and greedy exploration parameter *epsilon*, to make the agent find an optimal way without penalties within 100 trials.

Background

Smartcab operates in an idealized grid-like city, with roads going North-South and East-West. Other vehicles may be present on the roads, but no pedestrians. There is a traffic light at each intersection that can be in one of two states: North-South open or East-West open. There may be other agents running in the city, which can lead to apply traffic regulations if meeting in the same crossroad.

Questions

1.Implement a basic driving agent

Implement the basic driving agent, which processes the following inputs at each time step:

- Next waypoint location, relative to its current location and heading,
- Intersection state (traffic light and presence of cars), and,
- Current deadline value (time steps remaining),

And produces some random move/action (None, 'forward', 'left', 'right'). Don’t try to

implement the correct strategy! That's exactly what your agent is supposed to learn.

Run this agent within the simulation environment with enforce_deadline set to False (see run function in agent.py), and observe how it performs. In this mode, the agent is given unlimited time to reach the destination. The current state, action taken by your agent and reward/penalty earned are shown in the simulator.

In your report, mention what you see in the agent's behavior. Does it eventually make it to the target location?

I set the agent to choose actions randomly, and the agent(red car) went in random directions, not correspondent to the next way direction, and it didn't consider traffic regulations, sometimes it violated the traffic rules. Eventually it reached the destination several times if the deadline was set false. I have run 9 trials, the results are below:

Steps	Count of Correct Action	Total Reward
113	18	61
250	36	94.5
120	24	63
135	21	60
69	10	30
44	7	37
261	49	139
21	3	19
44	8	17

We can see that, it was not stable for the primary agent to find destination, sometimes it took as long as 250 steps, sometimes it only took 21 steps. And the corrent actions counted less than 20%. But the total rewards were all positive, partly because violation of red lights was not frequent and the penalty was not heavy. It would be punished only if the action was 'forward' or 'left and the light was red, the probability was 0.5 * 0.5 = 0.25, which means the primary only had 25% posibility to be punished, and the penaly was just -1, whereas correct actions had rewards of 2, and incorrect actions had rewards of 0.5, None actions had rewards of 1, the expectation of rewards should be above 0.

2.Identify and update state

Identify a set of states that you think are appropriate for modeling the driving agent. The main source of state variables are current inputs, but not all of them may be worth representing. Also, you can choose to explicitly define states, or use some combination (vector) of inputs as an implicit state. At each time step, process the inputs and update the current state. Run it again (and as often as you need) to observe how the reported state changes through the run.

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

My initial definition of a state in this problem was set of traffic light status, states of other agents, locations, destination and headings. All of these factors could show us details about the primary agent. The traffic light status and oncoming, left and right cars were related to traffic rules, penalties would come if violated; the location and destination of primary agent determined whether it was in absorbing state, and whether the action was effective. Also, heading was used to predict next location, and judge violations of traffic rules.

It would be huge computing tasks if we considered all the factors directly. I have read the provided codes about reward and next waypoint functions. Actually, **the reward function did not take nearby agents' state into consideration, it only considered traffic lights, actions and locations**. Besides, **the next waypoint made decisions based on the distance between current location and destination and heading status**. Therefore, we need only pay attention on traffic light, differences between location and destination, heading for the state. Yet the Q matrix dimension should be 2*48*4*4=1536, still a very large matrix for computing, it would be not easy to converge without a huge number of samples. We should make it easier and more efficient.

Note in the next waypoint function, it only used the signs of the difference between current location and destination, and the sign tuples of differences should be (1,0), (0,1), (0,0), (1,1), (-1,0), (0,-1), (1,-1), (-1,1), (-1,-1), only 9 cases, which could tell the agent was the right direction along with the heading. So the Q matrix should be 2*9*4*4=288, much easier to converge. But we still kept difference between current location and destination in order to computing next location, next state.

Above all, my state consist of traffic light status, signs of difference between current location and destination, and heading status. If the primary agent violated traffic lights, for example, moving forward when the light was red, it would get penalties(-1 in reward), which would be updated in the Q-learning, and next time it would try to avoid this case. The heading and sign of difference between current location and destination could determine a proper action, an action made the car close to the destination. Given specific heading and sign, if the agent chose correct action, it would get reward of 2, otherwise, it would only get 0.5, which would also be updated in the Q-learning, next time, the agent would try to choose correct actions with more rewards. In addition, if the sign of difference between current location and destination was (0,0), which meaned the agent reached the destination, and won an extra lsrge reward, and after learning, the agent would try to reach its destination.

3. Implement Q-Learning

Implement the Q-Learning algorithm by initializing and updating a table/mapping of

Q-values at each time step. Now, instead of randomly selecting an action, pick the best action available from the current state based on Q-values, and return that.

Each action generates a corresponding numeric reward or penalty (which may be zero). Your agent should take this into account when updating Q-values. Run it again, and observe the behavior.

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

I set *gamma* = 0.2, *alpha* = 0.8. First, the agent chose action according to the largest value in Q matrix. The agent went in random directions initially, but after many steps, if it returned to the same crossroad again, it seemed to have a memory, tried to avoid the wrong action which led to previous penalty, or chose the action which led to a big reward. Sometimes I watched a phenomenon, that the agent could fell into a trap, traveled through the same path, even stayed still again and again, seemed like a temporary deadlock. Because the Q values were all zeros in the beginning, if the agent chose a wrong action or stayed still, it would still get a reward above zero, then next time, it would choose this action again.

In order to avoid this phenomenon, I generated a random value and compared it with a parameter *epsilon* in each trial. If the random value was smaller than *epsilon*, chose the action randomly, otherwise chose the action with largest Q value. I set *epsilon = epsilon* * (t+0.5)/(t+0.8), so *epsilon* could decrease after 10 steps in each trial.

I have run 50 trials, it took many 22 trials to find the destination within given deadline, then it was more likely to find the destination with given deadlines. But it was not quite stable, obviously the Q matrix did not get converged within 50 trials, and there were still penalties in the last few trials.

Trials	Reach Destination	Penalty Count	Rewards	Step	Deadline
1	FALSE	2	20.5	25	25
2	FALSE	2	49.0	45	45
3	FALSE	4	27.0	30	30
4	FALSE	2	25.5	30	30
5	FALSE	4	28.5	30	30
6	FALSE	3	15.5	20	20
7	FALSE	3	21.0	30	30
8	FALSE	4	24.0	30	30
9	FALSE	2	35.5	35	35
10	FALSE	6	22.5	35	35
11	FALSE	2	27.0	25	25
12	FALSE	3	44.0	40	40
13	FALSE	3	33.0	35	35
14	FALSE	3	23.0	25	25

15	FALSE	5	50.5	55	55
16	FALSE	7	14.0	25	25
17	FALSE	4	27.5	35	35
18	FALSE	5	21.5	25	25
19	FALSE	3	19.0	25	25
20	FALSE	1	25.5	25	25
21	TRUE	1	17.0	7	13
22	FALSE	1	39.0	30	30
23	TRUE	3	33.0	20	10
24	FALSE	1	29.5	25	25
25	TRUE	1	24	10	15
26	TRUE	2	30.5	19	16
27	TRUE	1	35.5	20	20
28	FALSE	2	24.5	25	25
29	TRUE	1	27.5	12	28
30	TRUE	4	16.5	11	39
31	TRUE	3	23	12	8
32	TRUE	2	41.5	25	5
33	TRUE	2	36.0	26	24
34	TRUE	6	58.5	45	45
35	TRUE	5	36.0	27	8
36	FALSE	2	51.5	45	45
37	FALSE	6	11.5	20	20
38	TRUE	5	29.5	21	14
39	TRUE	1	43.5	31	9
40	TRUE	1	21	6	14
41	TRUE	0	26.5	11	9
42	TRUE	1	27	12	8
43	TRUE	2	40.5	24	16
44	TRUE	3	33.5	21	14
45	FALSE	2	20.0	20	20
46	TRUE	6	30.0	23	2
47	TRUE	3	45.0	34	6
48	TRUE	1	33.0	19	21
49	TRUE	1	32.5	16	4
50	TRUE	2	44.5	29	16

4. Enhance the driving agent

Apply the reinforcement learning techniques you have learnt, and tweak the parameters (e.g. learning rate, discount factor, action selection method, etc.), to improve the performance of your agent. Your goal is to get it to a point so that within 100 trials, the agent is able to learn a feasible policy - i.e. reach the destination within

the allotted time, with net reward remaining positive.

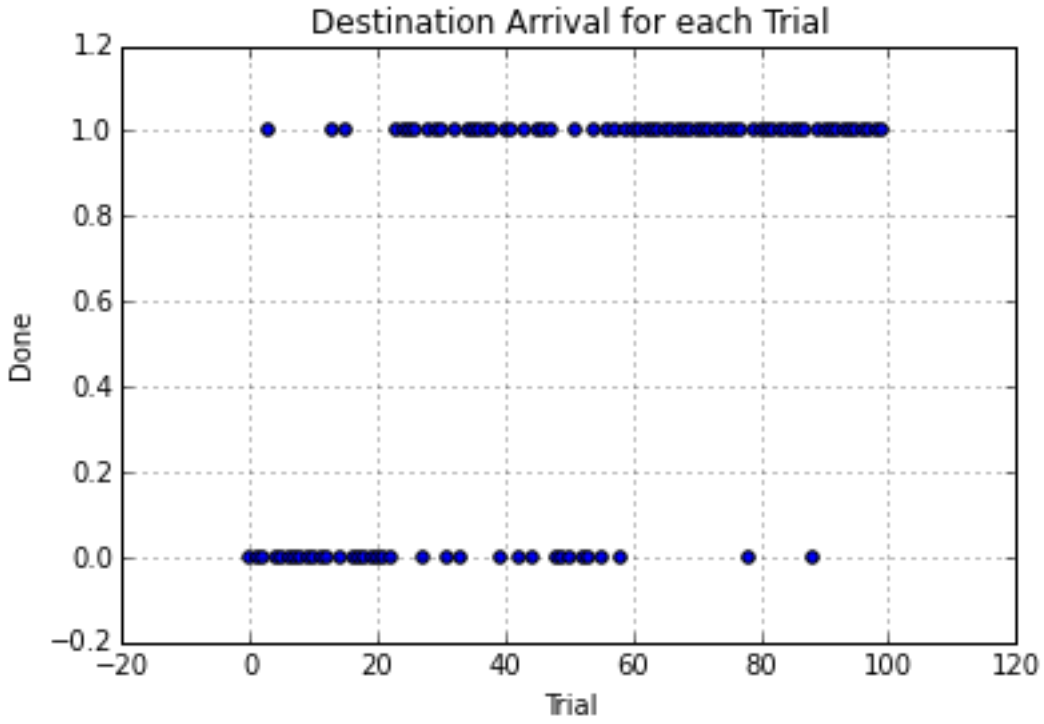
Report what changes you made to your basic implementation of Q-Learning to achieve the final version of the agent. How well does it perform?

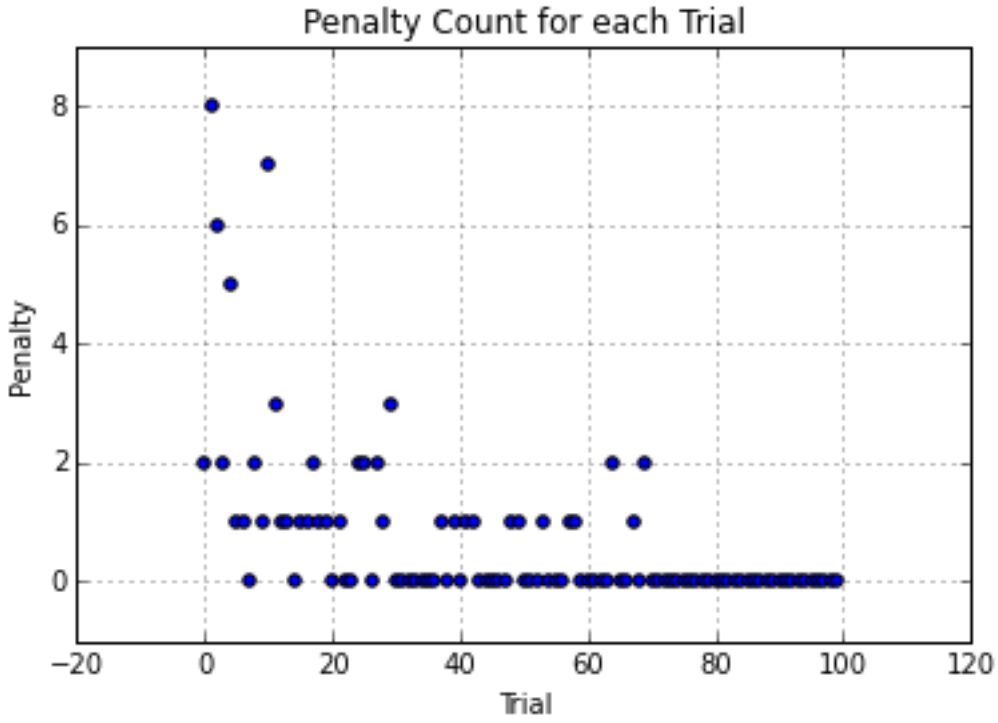
Does your agent get close to finding an optimal policy, i.e. reach the destination in the minimum possible time, and not incur any penalties?

I used exploration/exploitation trade-off strategy because our initial Q estimates might be incomplete and noisy, and the corresponding policy was weak. I selected *epsilon*-greedy algorithm, specifically, with probability epsilon, the agent chose a random action, and with probability (1-*epsilon*) the agent adopted Q-values policy. Epsilon decreased over time *t*, so finally the agent would follow actions induced by Q-values.

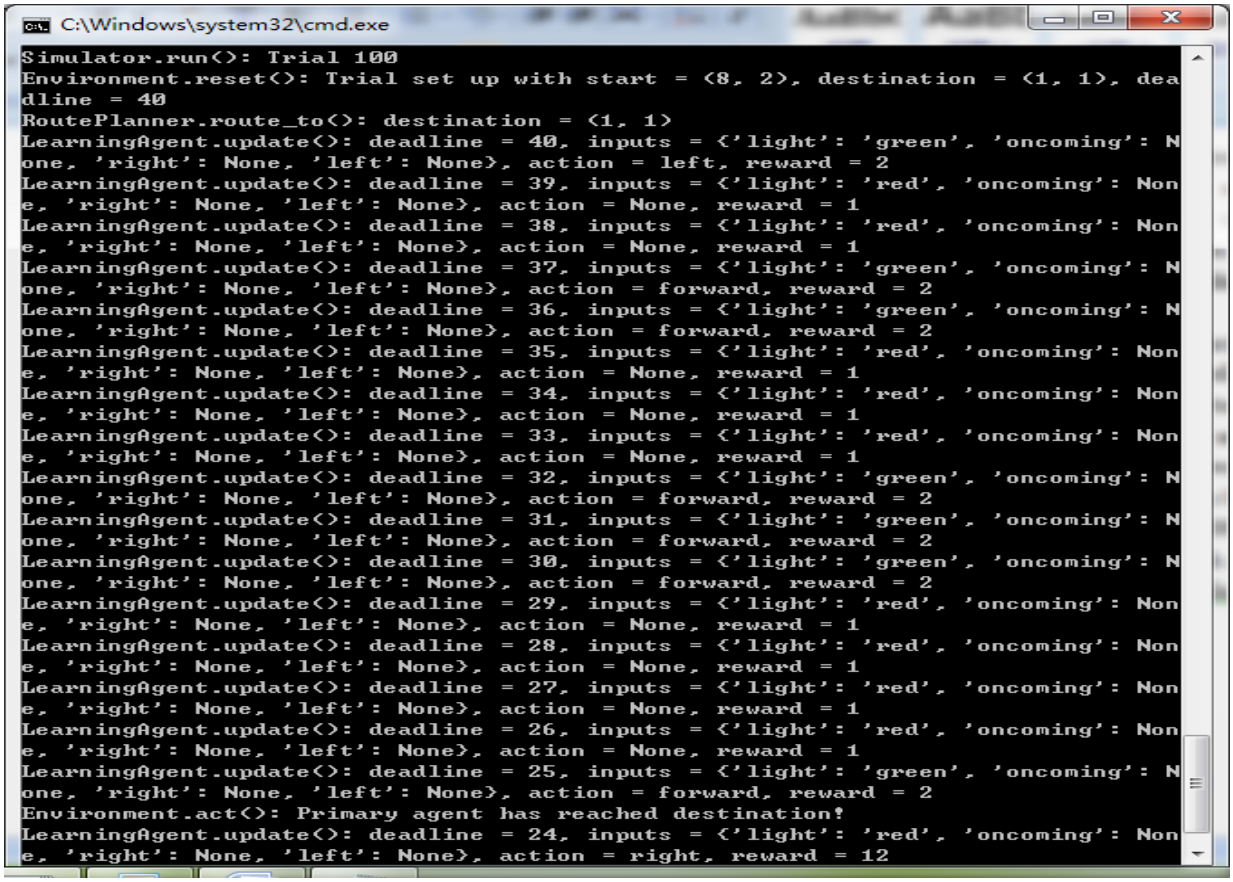
On the basis of Part 3, I tuned three parameters: *learning rate*, *discount factor* and *epsilon*, used a global time in epsilon instead reset it each trial. I found as *discount factor gamma* increased, it was less likely to reach destinations within deadlines, which meaned future states didn't contribute so much as current reward. The algorithm performed quite well, after 60 trials, it was very likely to reach destinations within deadlines if *gamma* was quite small such as 0.2.

Finally, I set the epsilon as 100/(*t*+100), learning rate alpha as 2000/(*t* + 2000), with the increase of the global time *t* (started when the agent was activated), both *epsilon* and *alpha* decreased gradually. The discount factor was 0.2. After 60 trials, the Q values seemed converged, the primary agent could reach destinations within deadline without penalties most of trials, as the screenshot showed below. But there were two failures, perhaps because the agent had never been that state before.





There was an example in the last trial, the primary agent could wait until the red light turned green, then it moved forward and waited again when the light turned red, there were no penalties.


C:\Windows\system32\cmd.exe
Simulator.run(): Trial 100
Environment.reset(): Trial set up with start = (8, 2), destination = (1, 1), dea
dline = 40
RoutePlanner.route_to(): destination = (1, 1)
LearningAgent.update(): deadline = 40, inputs = {'light': 'green', 'oncoming': N
one, 'right': None, 'left': None}, action = left, reward = 2
LearningAgent.update(): deadline = 39, inputs = {'light': 'red', 'oncoming': Non
e, 'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 38, inputs = {'light': 'red', 'oncoming': Non
e, 'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 37, inputs = {'light': 'green', 'oncoming': N
one, 'right': None, 'left': None}, action = forward, reward = 2
LearningAgent.update(): deadline = 36, inputs = {'light': 'green', 'oncoming': N
one, 'right': None, 'left': None}, action = forward, reward = 2
LearningAgent.update(): deadline = 35, inputs = {'light': 'red', 'oncoming': Non
e, 'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 34, inputs = {'light': 'red', 'oncoming': Non
e, 'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 33, inputs = {'light': 'red', 'oncoming': Non
e, 'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 32, inputs = {'light': 'green', 'oncoming': N
one, 'right': None, 'left': None}, action = forward, reward = 2
LearningAgent.update(): deadline = 31, inputs = {'light': 'green', 'oncoming': N
one, 'right': None, 'left': None}, action = forward, reward = 2
LearningAgent.update(): deadline = 30, inputs = {'light': 'green', 'oncoming': N
one, 'right': None, 'left': None}, action = forward, reward = 2
LearningAgent.update(): deadline = 29, inputs = {'light': 'red', 'oncoming': Non
e, 'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 28, inputs = {'light': 'red', 'oncoming': Non
e, 'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 27, inputs = {'light': 'red', 'oncoming': Non
e, 'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 26, inputs = {'light': 'red', 'oncoming': Non
e, 'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 25, inputs = {'light': 'green', 'oncoming': N
one, 'right': None, 'left': None}, action = forward, reward = 2
Environment.act(): Primary agent has reached destination!
LearningAgent.update(): deadline = 24, inputs = {'light': 'red', 'oncoming': Non
e, 'right': None, 'left': None}, action = right, reward = 12