Udacity MLND P4: Train a Smartcab

Tasks

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?

The agent is randomly picking actions, no matter what the condition of the environment. It does not care whether or not there is an oncoming vehicle, nor does it care about vehicles on its left nor right. It also seems to disregard the traffic light.

It seems to reward the following:

Going right when traffic light is red:

```
LearningAgent.update(): deadline = 20,
 inputs = {'light': 'red',
 'oncoming': None,
 'right': None,
 'left': None},
 action = right,
 reward = 2

LearningAgent.update(): deadline = 14,
 inputs = {'light': 'red',
 'oncoming': None,
 'right': None,
 'left': None},
 action = right,
 reward = 2

LearningAgent.update(): deadline = 13,
 inputs = {'light': 'red',
 'oncoming': None,
 'right': None,
 'left': None},
```

```
 action = right,
 reward = 2

LearningAgent.update(): deadline = 12,
 inputs = {'light': 'red',
 'oncoming': None,
 'right': None,
 'left': None},
 action = right,
 reward = 0.5

LearningAgent.update(): deadline = 11,
 inputs = {'light': 'red',
 'oncoming': None,
 'right': None,
 'left': None},
 action = right,
 reward = 2

LearningAgent.update(): deadline = 8,
 inputs = {'light': 'red',
 'oncoming': None,
 'right': None,
 'left': None},
 action = right,
 reward = 0.5
```

Going forward when traffic light is green:

```
LearningAgent.update(): deadline = 19,
 inputs = {'light': 'green',
 'oncoming': None,
 'right': None,
 'left': None},
 action = forward,
```

```
 reward = 2

LearningAgent.update(): deadline = 16,
 inputs = {'light': 'green',
 'oncoming': None,
 'right': None,
 'left': None},
 action = forward,
 reward = 0.5
```

Going forward when traffic light is green and car in the right is trying to move forward:

```
LearningAgent.update(): deadline = 10,
 inputs = {'light': 'green',
 'oncoming': None,
 'right': 'forward',
 'left': None},
 action = forward,
 reward = 0.5
```

Going right when traffic light is green:

```
LearningAgent.update(): deadline = 11,
 inputs = {'light': 'green',
 'oncoming': None,
 'right': None,
 'left': None},
 action = right,
 reward = 2
```

```
LearningAgent.update(): deadline = 6,
 inputs = {'light': 'green',
 'oncoming': None,
 'right': None,
 'left': None},
 action = right,
 reward = 0.5
```

Staying at the same location when deadline is red:

```
LearningAgent.update(): deadline = 17,
 inputs = {'light': 'red',
 'oncoming': None,
 'right': None,
 'left': None},
 action = None,
 reward = 1
```

Staying at the same location when stoplight is green:

```
LearningAgent.update(): deadline = 7,
 inputs = {'light': 'green',
 'oncoming': None,
 'right': None,
 'left': None},
 action = None,
 reward = 1

LearningAgent.update(): deadline = 5,
 inputs = {'light': 'green',
 'oncoming': None,
 'right': None,
```

```
 'left': None},
 action = None,
 reward = 1
```

It penalizes the following:

Going forward when the traffic light is red:

```
LearningAgent.update(): deadline = 18,
 inputs = {'light': 'red',
 'oncoming': None,
 'right': None,
 'left': None},
 action = forward,
 reward = -1

LearningAgent.update(): deadline = 15,
 inputs = {'light': 'red',
 'oncoming': None,
 'right': None,
 'left': None},
 action = forward,
 reward = -1
```

Going left when the traffic light is red:

```
LearningAgent.update(): deadline = 9,
 inputs = {'light': 'red',
 'oncoming': None,
 'right': None,
 'left': None},
```

```
 action = left,
 reward = -1
```

Interesting to note that environment seems to prevent the car from moving forward.

It does eventually make it to the target location, but only through many iterations, and it seems to take so long that it is way past the deadline.

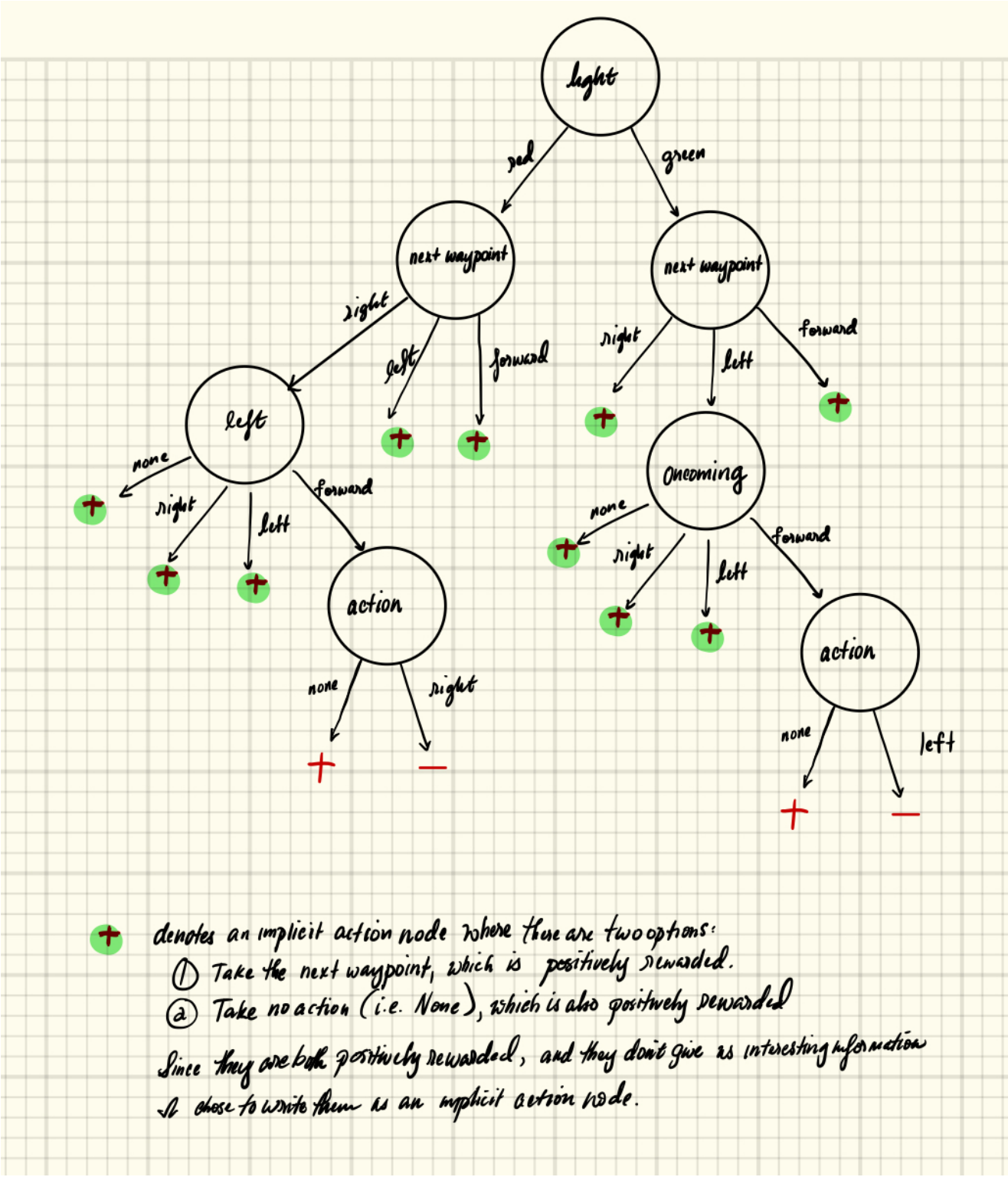
Identify and update state

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.

I made a decision tree that seems to capture the reward system above:



Smartcab decision tree

One might notice that in the decision tree above, I do not consider the situation where the learning agent decided to turn right when the car is in an intersection where light is red and an oncoming car is turning left. This is

because the environment seems to reward such behavior, as seen below:

	action	left	light	next_waypoint	oncoming	reward	right
102	right	None	red	right	left	2.0	None
730	right	None	green	right	left	2.0	None
853	right	None	red	right	left	2.0	None

Getting positively rewarded for not following part of the U.S. right-of-way

Thus, the best states to include are the nodes in the decision tree, as follows:

1. Light: red or green.
2. Next waypoint: forward, left, or right.
3. Oncoming: forward, left, or right, or None.
4. Action: Take the next waypoint, or None.

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?

The Learning Agent gets to the destination much faster. In the earlier trials, the agent accumulates both positive and negative rewards (i.e. exploration). In the later trials, the learning agent tends to accumulate only positive rewards (i.e. exploitation).

It seems though, that it learns quickly. In the first 5 trials, it received some form of punishment (i.e. negative reward), but the next 5 trials, it did not receive punishment at all.

Trial #	received punishment
0	TRUE
1	TRUE
2	FALSE
3	TRUE
4	FALSE
5	FALSE

6	FALSE
7	FALSE
8	FALSE
9	FALSE

Whether or not learning agent has received punishment over first ten trials

Here was the mistake for trial 0. The Learning Agent learned to not go forward when light is red:

```
LearningAgent.update(): deadline = 39,
 inputs = {'light': 'red',
 'oncoming': None,
 'right': None,
 'left': None},
 action = forward,
 reward = -1,
 next_waypoint = forward
```

Here was the mistake for trial 1, which is similar to the mistake during trial 0, different only in that there was an oncoming car moving to the left. This makes sense because the two states are different. The Learning Agent learned to not go forward when light is red and when there's an oncoming car going to oncoming car's left:

```
LearningAgent.update(): deadline = 28,
 inputs = {'light': 'red',
 'oncoming': 'left',
 'right': None,
 'left': None},
 action = forward,
 reward = -1,
 next_waypoint = forward
```

Trial 3 had a mistake the same as trial 0:

```
LearningAgent.update(): deadline = 39,
 inputs = {'light': 'red',
 'oncoming': None,
 'right': None,
 'left': None},
 action = forward,
 reward = -1,
 next_waypoint = forward
```

Then for a while, it seems to be taking correct actions (trials 4-9). However, this does not mean that the learning agent is already perfect after only 5 trials. It still took incorrect actions intermittently, which are harder-to-catch because they involve cars being present in the learning agent's view.

Later errors, such as ones from trial 20, are exemplary of cases where the learning agent is learning to deal with driving rules when other cars are visible:

```
LearningAgent.update(): deadline = 32,
 inputs = {'light': 'red',
 'oncoming': 'forward',
 'right': None,
 'left': None},
 action = forward,
 reward = -1,
 next_waypoint = forward

LearningAgent.update(): deadline = 31,
 inputs = {'light': 'red',
 'oncoming': 'forward',
 'right': None,
 'left': 'right'},
 action = forward,
 reward = -1,
 next_waypoint = forward
```

Looking at the tail of the the 100 iterations, we could see that the driving is much improved. It did not get any sort of punishments in the last ten trials:

Trial #	received punishment
90	FALSE
91	FALSE
92	FALSE
93	FALSE

94	FALSE
95	FALSE
96	FALSE
97	FALSE
98	FALSE
99	FALSE

Learning agent's actions in the last ten trials were all correct.

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?

I think the action selection method is quite optimal already in my “basic” implementation of Q-Learning. At any point in time, I made the learning agent only choose between two actions: either choose the next waypoint as the

action, or stay in the same intersection. The benefit of this is that there is less space for the learning agent to "explore." My *a priori* knowledge of the reward system helped me limit the actions to choices that are generally good choices. Taking the action of going to the next waypoint brings us closer the destination, but risking violating traffic laws. On the other hand, doing nothing does not get us a step closer to reaching the destination, but may prevent violation of traffic laws (and be less of a safety risk). Thus in optimizing my implementation, I chose to fiddle with the learning rate and the discount factor instead.

First, to measure success when comparing different parameters, I thought that looking into $average\ reward\ /\ total\ moves$ might be a good metric to use. For example, let's assume that there exists two sets of 100 trials A & B. If both sets have the same number of moves, and if the ratio is higher for A, than B, then A must be incurring more rewards than B. Likewise, if A & B got the same reward sum but if B took more moves, then A is more efficient than B. This metric is also great because when the learning agent makes mistakes, the environment prevents it from taking a step, which means that the number of actions it takes increases without actually getting closer to the goal. Thus, when it makes a lot of mistakes, the $average\ reward\ /\ total\ moves$ ratio will be lower than when it does not make a lot of mistakes, assuming everything else equal.

So I actually made my own “grid-search” by testing different learning rate and discount factor values, and calculating the *average reward / total moves* metric for each combination. I ran each combination 10 times because there is inherent randomness in the placement of dummy cars relative to the learning agent, and also there is randomness in the placement of the learning agent relative to the destination. I wanted to find the average ratio for each combination, and I got the following results:

	alpha	avg_total_actions	avg_total_rewards	gamma	std_total_actions	std_total_rewards	reward_per_action
0	0.1	1275.8	2870.2	0.1	49.746960	60.439722	2.249726
1	0.1	1259.4	2853.7	0.3	62.121172	68.095595	2.265920
2	0.1	1272.2	2868.5	0.7	72.755481	90.109101	2.254756
3	0.1	1276.2	2867.9	0.9	53.966286	63.315796	2.247218
4	0.3	1276.6	2866.9	0.1	30.604575	38.239901	2.245731
5	0.3	1290.6	2883.6	0.3	67.486591	80.508633	2.234310
6	0.3	1252.1	2831.8	0.7	54.784031	53.942191	2.261640
7	0.3	1285.8	2881.0	0.9	70.455376	80.142373	2.240628
8	0.7	1340.5	2942.5	0.1	51.761472	77.223377	2.195076
9	0.7	1248.8	2836.9	0.3	28.049242	33.634655	2.271701
10	0.7	1256.5	2846.1	0.7	48.276806	58.390838	2.265101
11	0.7	1270.1	2863.3	0.9	43.882685	56.091087	2.254389
12	0.9	1277.8	2868.0	0.1	50.429753	64.992307	2.244483
13	0.9	1247.0	2837.4	0.3	57.856720	74.135282	2.275381
14	0.9	1283.7	2894.0	0.7	59.888313	70.155541	2.254421
15	0.9	1273.2	2863.5	0.9	68.146607	83.332167	2.249057

Results of Running Experiments with Different Learning Rate and Discount Factor Values

I found out that the “best” combination seems to be when learning rate

alpha is 0.9 and *gamma* is 0.3, which gives us 2.275381 as the average reward per action. However this metric seems to be pretty close throughout each combination, and it does not look like there really is a clear winner.

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?

My agent is definitely close to having the optimal policy. In 100 iterations, it consistently reaches the destination, and minimally incurs penalties (especially in the beginning). After 100 iterations, it seems to consistently take correct actions. One thing I would add is that the "learning" could probably be accelerated by creating more dummy agents (i.e. it would speed up convergence for states that were happening less often by making those events happen more frequently). By increasing the number of dummy agents, the car would probably interact more with them, and would accumulate penalties to avoid earlier than later. This would then translate to less "training time", and would be a nice enhancement.