Report

April 21, 2016

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

Answers: By applying random actions to the cab, it will generate random actions on the map, but it will still obey the traffic rules since the act function in enviroment.py will not perform that random action if the agent breaks the rule. The angent will finally ready the target location by a small chance since the map in finite when set enforce_deadline to false. The rewards are negative.

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

Answers: The states feature I choose include traffic light, incoming traffic, left traffic, right traffic and the action the agent intend to perform. I choose all of them to fully capture what's going on there.

3. Implement Q-Learning. What changes do you notice in the agent's behavior?

Answers: It begins to reach the destination on time and get positive final reward. But with a high chance the agent will stuck in a rut to just make right turns in a loop all the time and miss the deadline. Maybe it's because I set the learning rate(0.5) and gamma(0.7) too high so that the agent by a change make a lot of right turn at the begining and quickly learns it can make more right turns to collect a reward of 2. Even in the situation it should make a left turn, it sacrifise to get a negative reward -0.5 (since action != agent.get_next_waypoint()) in order to make three more right turns because the q table shows more action values.

Some time it works with One time when I run the 1000 trials the agents get really stuck in loop with 977 times failures to get destination, which means 97.7% times the agent fails.

So I reduce gamma to 0.5 and alpha to 0.2, then the failure rate reduces to around 4%.

4. Enhance the driving agent

4. 1 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?

Answers: To prevent the agent stucking, one way is to do a grid search of alpha and gamma, in this way we can find a good parameter combination but only for this model, which means these parameters can not be generilized, so I just skip this.

The other way is to us a exploration function. I implement it in function *next_action_index_with_exploration.* The basic idea is cache the time of state->action the agent have tried. If it doesn't meet a certain threshold(*self.NE*) then we use a certain positive q_value for that state->action (*self.reward_plus*). After we meet enought numbers of that state->action status, we begin to use the values we've learned in q_table. I set *self.NE* to be 2

The third way is to reduce the size of all the states. I implement a state filter in function *inputs_filter* following the rules below:

- Ignore right traffic
- If light is green, ignore left traffic
- if light is red, ignore oncoming traffic

I define the failure rate as the percentage that the agent fails among all the trails, in order to better capture the rate, I define *run_multi* function to get 100 fresh start, and keep the 100 trails. Since there're lots of randomness involved, by just increasing the number of trails won't prove anything since the agent rarely fails after it get a acceptable q table. Here's the result:

	No Filter(Baseline)	With Filter	With Exploration	With Filter & Exploration
AVE Fail rate	4.4%	3.64%	4.48%	3.45%
AVE Q table size	34	13	35	14

We can see that with filter and exploration we can get the lowest failure rate.

4. 2 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?

For the baseline without filtering and exploration, I got one policy like this #light:green#oncoming:*#right:forward#left:*#waypoint:left[0 , 0 , 0 , 0. 4 7 7]

- When the light is green, no oncoming traffic and the agent tends to go left, but according to the Q table, it will choose to turn right(0.477).

With filtering and exploration, the agent can get much better policies:

1 light:green#oncoming:*#right:*#left:*#waypoint:left[0 . 0 , - 0 . 2 5 2 , 7 . 1 8 0 3 9 2 3 5 5 3 7 5 5 0 8, - 0 . 0 0 6 9 0 6 5 6 0 0 0 0 0 0 0 0 0 1]

- When the light is green, no oncoming traffic and the agent tends to go left, the just go(7.18)

2 light:green#oncoming:forward#right:*#left:*#waypoint:left[0 . 4 5 2 8 8 2 9 1 4 8 3 6 0 2 1 8 , 0 , 0 , 0]

- If there's oncoming traffic going forward, then stay(0.4) instead of going left(0)

```
In [ ]:
```