

Udacity MLND P4: Train a Smart Cab

P.S: I noticed that the newest starting code for this project has been changed for the reward functions. Previously if action is null (i.e. stay as is) the reward is 1, but now it is 0. And before the agent gets 0.5 reward if the move is valid but not what the planner says, now this reward is changed to -0.5. I think these two changes make a lot sense, since the ultimate goal is to let the agent behave what the planner tells it to do, but not violating the traffic rules at the same moment.

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?

Answer: The agent will pick a random action from the 4 options, not caring about any other rules or instructions at all. Given unlimited time, the agent will eventually make it to the target location. The problem of this randomized policy is that first the agent ignores all traffic rules, so at times there will be traffic rule violation, even causing accident. Second, it also ignores the instructions from the planner, which would make it very difficult to get to the target location before 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.

Answer:

Initially I tried to put all variables from the inputs, i.e. [self.next_waypoint, self.env.sense(), deadline]. But this is a 6-dimensional model, and due to the “Curse of Dimensionality” it would take huge amount of space and time for the training. Then I realized that some of the inputs are not necessary.

The inputs[‘right’] is not necessary, since the agent doesn’t have to worry about who is on its right at all. If the light is red, and the agent’s action is right, it only needs to care about inputs[‘left’]. If the light is red, and the agent’s action is not right, then it is not allowed to move or will get a negative reward. If the light is green, the agent can move as long as its action is not left, otherwise it needs to check what is the situation from inputs[“oncoming”]. In any case, the agent doesn’t have to check inputs[‘right’] at all.

But 5-dimension is still high! Then I noticed that the ultimate goal of the learning agent is as follows: it needs to do whatever planner tells it to do, but not violating the traffic rules, except that when the deadline is coming close, it may violate the rules to follow the planner’s instruction to reach to the destination in time. Therefore, I constructed a new binary variable called “action_okay” that is defined in the following code:

```
def is_action_okay(self, inputs):  
    action_okay = True  
    if self.next_waypoint == 'right':
```

```

        if inputs['light'] == 'red' and inputs['left'] == 'forward':
            action_okay = False
    elif self.next_waypoint == 'forward':
        if inputs['light'] == 'red':
            action_okay = False
    elif self.next_waypoint == 'left':
        if inputs['light'] == 'red' or (inputs['oncoming'] == 'forward' or inputs['oncoming'] ==
'right'):
            action_okay = False
    return action_okay

```

With this “action_okay” and the way_point and deadline, we have reduced the dimensionality to 3. Further, the original deadline variable is an integer, we need to convert this state to a binary state, such that if the deadline is less than a threshold (say, 3) its value is True, otherwise it is False.

Finally, we have a 3-dimensional state model with the cardinality as follows: 3 for way_point, 2 for action_okay, and 2 for deadline.

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?

Answer: The basic Q Learning algorithm is fairly standard. The hyper-parameters are the learning rate, the discount factor for future rewards, and the probability of random actions

for each step in order to get rid of the local optima. After implementing this, and tuning these hyper-parameters, the agent will, as expected, try to follow the planner and the traffic rules at the same moment. When deadline is close, it may occasionally violate the traffic rule to get to the destination in time and get the 10 points of the reward.

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?

Answer: The basic Q learning works fine, but ideally we want a faster learning at the beginning, and slow it down as we approach the optimal point. In other words, we need a large learning rate initially, then slowly reduce it. Thus I use the idea of “learning rate decay” with this formula: $\text{learning_rate} = \text{learning_rate} * \text{decay_factor}$, for each update step. After reaching a lower bound (say, 0.01) we don’t reduce it anymore.

Similarly, we want to large probability for a random move at the beginning, but we need to reduce it to a small value after a while.

The new Q learning would have two new hyper-parameters, the learning rate decay factor, and the probability decay factor. After tuning these parameters (detailed values can be found in code), the agent can reach to the destination in time even before 50 trials. Due to the stochasticity, sometimes it need more than 50 trials to train, however it will have the desired behavior within 100 trials for most of the times.

