Udacity Machine Learning Engineer Nanodegree
Reinforcement Learning
Project: Train a Smartcab How to Drive
Author: Yitaek Hwang

Implement a Basic Driving Agent

To begin, your only task is to get the **smartcab** to move around in the environment. At this point, you will not be concerned with any sort of optimal driving policy. Note that the driving agent is given the following information at each intersection:

- The next waypoint location relative to its current location and heading.
- The state of the traffic light at the intersection and the presence of oncoming vehicles from other directions.
- The current time left from the allotted deadline.

To complete this task, simply have your driving agent choose a random action from the set of possible actions (None, 'forward', 'left', 'right') at each intersection, disregarding the input information above. Set the simulation deadline enforcement, enforce_deadline to False and observe how it performs.

QUESTION: *Observe what you see with the agent's behavior as it takes random actions. Does the* ***smartcab*** *eventually make it to the destination? Are there any other interesting observations to note?*

When the agent takes random actions, the smartcab rarely reaches the goal when enforce-deadline is set to True (time limit enforced). This is an expected behavior, because the smartcab is not learning or maximizing on the rewards on every iteration. It is clearly not taking the optimal path since one of the possible action is to do nothing. Also, it does not learn from actions that result in negative rewards (breaking traffic laws). In essence, every iteration is like starting fresh and knowing nothing about the system and hoping that the smartcab will randomly reach the destination.

Inform the Driving Agent

Now that your driving agent is capable of moving around in the environment, your next task is to identify a set of states that are appropriate for modeling the **smartcab** and environment. The main source of state variables are the current inputs at the intersection, but not all may require representation. You may choose to explicitly define states, or use some combination of inputs as an implicit state. At each time step, process the inputs and update the agent's current state using the self.state variable. Continue with the simulation deadline enforcement enforce_deadline being set to False, and observe how your driving agent now reports the change in state as the simulation progresses.

QUESTION: *What states have you identified that are appropriate for modeling the* ***smartcab*** *and environment? Why do you believe each of these states to be appropriate for this problem?*

The driving agent is given the next waypoint location relative to its current location and heading, the state of the traffic light and the presence of oncoming vehicles, and the current time left from the allotted deadline. The next waypoint is an important input to help craft the optimal path to the destination. Following this waypoint will minimize the distance to the destination and is therefore a good input to model the smartcab's state. The state of the traffic light is important as there is a negative reward for disobeying traffic laws. The traffic data of the oncoming traffic is important to avoid collisions and determine when a car can turn left or right without violating the traffic laws. So in summary, status of traffic lights (green or red) and the direction of oncoming traffic from the left and oncoming side is important to model the state. The status of the traffic from the right side does not affect the smartcab assuming the car on the right behaves rationally. Lastly, time left to the allotted deadline is not important based on the way the simulation does not allow running red lights. So remaining time isn't useful for modeling the smartcab and environment in this particular situation. Moreover, adding deadline as an input to model the state will significantly increase the complexity with little information gain.

OPTIONAL: *How many states in total exist for the* ***smartcab*** *in this environment? Does this number seem reasonable given that the goal of Q-Learning is to learn and make informed decisions about each state? Why or why not?*

Using the above assumptions, there are a total of 2 (light status) x 3 (next way point) x 4 (oncoming traffic) x 4 (oncoming traffic from the left) = 96 total states. The number seems a bit high since Q-learning will have to calculate the argmax of Q_hat, but in most cases computation will not be completely expensive as some situations force other factors to be irrelevant. For example, if the light is red, it doesn't really matter what the oncoming traffic is doing, if our next way point is to go straight.

Implement a Q-Learning Driving Agent

With your driving agent being capable of interpreting the input information and having a mapping of environmental states, your next task is to implement the Q-Learning algorithm for your driving agent to choose the *best* action at each time step, based on the Q-values for the current state and action. Each action taken by the **smartcab** will produce a reward which depends on the state of the environment. The Q-Learning driving agent will need to consider these rewards when updating the Q-values. Once implemented, set the simulation deadline enforcement `enforce_deadline` to `True`. Run the simulation and observe how the **smartcab** moves about the environment in each trial.

The formulas for updating Q-values can be found in this video.

QUESTION: *What changes do you notice in the agent's behavior when compared to the basic driving agent when random actions were always taken? Why is this behavior occurring?*

My implementation for Q-Learning can be found in QLearnAgent.py. Once Q-Learning is applied to the agent, the car "learns" from the rewards given. For example, the agent learns the traffic light rules in route to the final destination. Now the agent will not stay in one place (action = None) when it is a green light. As the iteration goes on, the agent start to reach the destination more. This is happening because the policy is learned based on the state, action, and reward. Before, the action was randomized so policy isn't updated based on the reward. Now with Q-learning, the agent learns to take the best possible route that maximizes the reward in trying to get to the destination.

Improve the Q-Learning Driving Agent

Your final task for this project is to enhance your driving agent so that, after sufficient training, the **smartcab** is able to reach the destination within the allotted time safely and efficiently. Parameters in the Q-Learning algorithm, such as the learning rate (alpha), the discount factor (gamma) and the exploration rate (epsilon) all contribute to the driving agent's ability to learn the best action for each state. To improve on the success of your **smartcab**:

- Set the number of trials, n_trials, in the simulation to 100.
- Run the simulation with the deadline enforcement enforce_deadline set to True (you will need to reduce the update delay update_delay and set the display to False).
- Observe the driving agent's learning and **smartcab's** success rate, particularly during the later trials.
- Adjust one or several of the above parameters and iterate this process.

This task is complete once you have arrived at what you determine is the best combination of parameters required for your driving agent to learn successfully.

QUESTION: *Report the different values for the parameters tuned in your basic implementation of Q-Learning. For which set of parameters does the agent perform best? How well does the final driving agent perform?*

The results for various parameters tuned for my implementation of Q-Learning is stored in sim-results. Epsilon controls for the frequency of exploration (random action) to avoid being stuck in a local minima. I varied the values of epsilon from 0.05 to 0.9. It turned out that smaller values of epsilon, particularly 0.05, helped the agent reach the destination more often. Alpha is the learning rate. Smaller values of learning rate means finer update policy for Q. So using that logic, smaller value of alpha will help the algorithm. I used alpha of 0.01 and saw that the agent only failed to reach the destination 5 times out of 100 and consistently reached the destination after the 3rd iteration. Finally, gamma is the discount value for the Q-learning equation. Discount factor determines the importance of future rewards. In this problem, future rewards may not have a big effect since the smartcab will behave best if it follows traffic laws at its current state. Still, for the purpose of the experiment, I implemented my version of gridsearch, varying different values of the parameters and used success rate of smartcab reaching the destination per 100 trial run using those parameters. Looking at the two simulation results, I see that epsilon=0.1 and alpha =0.1 gives the best success rate of 97% and 96% respectively. Gamma values actually did not really make a difference as you can see that in q_learn_2.txt, the algorithm performs best with epsilon = 0.1 and alpha = 0.1 for varying values of gamma.

The results are saved under SuccessRateAnalysis.ipynb. The top results for q_learn.txt and q_learn2.txt are shown below:

	Parameters	Success Rate
76	epsilon = 0.1, alpha = 0.1, gamma = 0.2	Success Rate:97%
60	epsilon = 0.05, alpha = 0.05, gamma = 0	Success Rate:96%
57	epsilon = 0.05, alpha = 0.01, gamma = 0.4	Success Rate:95%
105	epsilon = 0.2, alpha = 0.01, gamma = 0	Success Rate:95%
61	epsilon = 0.05, alpha = 0.05, gamma = 0.2	Success Rate:94%
82	epsilon = 0.1, alpha = 0.01, gamma = 0.4	Success Rate:94%

	Parameters	Success Rate
76	epsilon = 0.1, alpha = 0.1, gamma = 0.2	Success Rate:96%
75	epsilon = 0.1, alpha = 0.1, gamma = 0	Success Rate:96%
45	epsilon = 0.01, alpha = 0.2, gamma = 0	Success Rate:96%
96	epsilon = 0.1, alpha = 0.2, gamma = 0.2	Success Rate:96%
117	epsilon = 0.2, alpha = 0.1, gamma = 0.4	Success Rate:94%

In both iterations, epsilon = 0.1, alpha = 0.1, and gamma = 0.2 provide the best success rate.

QUESTION: *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? How would you describe an optimal policy for this problem?*

In the way the program is structured, intuitively, the optimal policy would reward the agent for obeying the traffic laws and moving in the shortest distance using waypoint. Q learning implementation gets close to finding an optimal policy as it consistently reached the destination in time after about 10-15 iterations based on the parameters. Particularly, if you look under q_table.txt under sim-results folder, in the last 10 trials, it follows traffic laws and takes the correct action. For example, in this state = ('forward', 'green', None, None), it goes forward with reward of 12. In another case, state = ('forward', 'red', None, None), action = None, reward = 0.0, it knows not to go forward on a red light and takes no action. The last couple of iterations for the smartcab is listed below and it shows pretty good adherence to the optimal policy:

state = ('right', 'green', None, None), action = right, reward = 2.0
state = ('right', 'green', None, None), action = right, reward = 2.0
state = ('forward', 'red', None, None), action = None, reward = 0.0
state = ('forward', 'green', None, None), action = forward, reward = 2.0
state = ('left', 'green', None, None), action = left, reward = 12.0
state = ('forward', 'green', None, None), action = forward, reward = 2.0
state = ('forward', 'green', None, None), action = forward, reward = 2.0
state = ('right', 'red', None, None), action = right, reward = 2.0
state = ('forward', 'green', None, None), action = forward, reward = 2.0
state = ('forward', 'green', None, None), action = forward, reward = 12.0
state = ('right', 'red', None, None), action = right, reward = 2.0
state = ('forward', 'red', None, None), action = None, reward = 0.0
state = ('forward', 'red', None, None), action = None, reward = 0.0
state = ('forward', 'green', None, None), action = forward, reward = 2.0
state = ('left', 'green', None, None), action = left, reward = 2.0
state = ('forward', 'green', None, None), action = forward, reward = 2.0
state = ('forward', 'green', None, None), action = forward, reward = 2.0
state = ('forward', 'green', None, None), action = forward, reward = 2.0
state = ('forward', 'green', None, None), action = forward, reward = 12.0