Smart Cab

Project Report

You will be required to submit a project report along with your modified agent code as part of your submission. As you complete the tasks below, include thorough, detailed answers to each question *provided in italics*.

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?*

ANSWER: To start, I only change one line in agent.py.

>> action = None
becomes
>> action = random.choice(self.env.valid_actions)

This sets the action to a random choice between: forward', 'left', 'right', and None.

Now running the agent.py file, we can observe how the agent (car) reacts.

- The target location and initial starting point change every trial.
- -1 is rewarded if agent violates a traffic rule
- +2 is rewarded if car moves towards target location while obeying traffic rules
- -.05 rewarded if obeys traffic rules but doesn't move towards target (way point)
- 0 is rewarded for null move
- 10 points is awarded if agent gets to target location before time limit
- Reaching target location early doesn’t get extra rewarded
- The car eventually reaches target location but rarely within time limit.

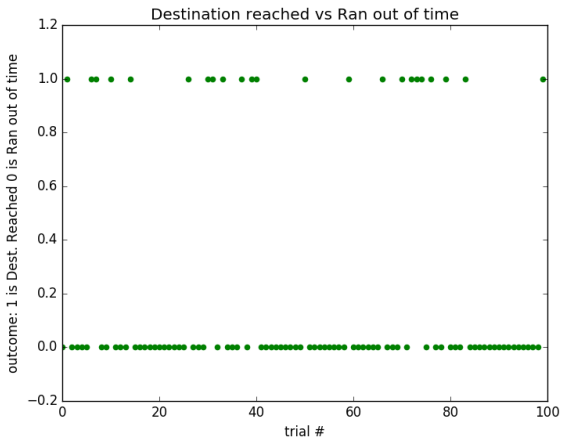
After my initial run, I changed the `enforce_deadline` to `True,` and ran the script three times with random action. I then copied the results from the terminal and saved them in the following three text files: ‘trial_one_random.txt’, ‘trial_two_random.txt’, and ‘trial_three_random.txt’.

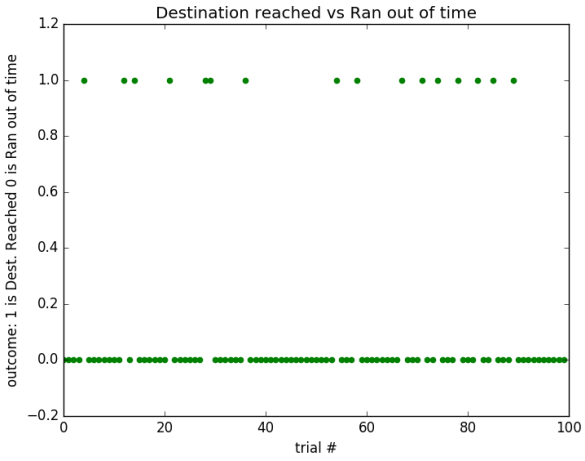
I then created a script called “get_stats.py’ and ran the script on each on of the text files. Below are the results of the script. It is the number of successes out of 100 trials and a graph.

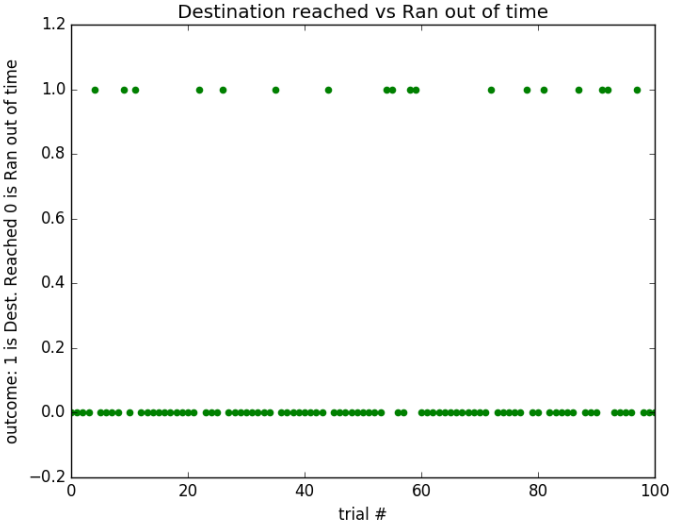
```
In [42]: %run get_stats.py
There were 23 successes out of 100 trials

In [43]: %run get_stats.py
There were 31 successes out of 100 trials

In [44]: %run get_stats.py
There were 16 successes out of 100 trials
```







As we can see the random action agent doesn’t do very well.

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?*

ANSWER:

The possible states that we could look at for modeling the smart cab belong to one of two flavors: Handling traffic rules and Reaching way point (target location).

For handling traffic rules the states are:

Traffic light: {red, green}
Car oncoming with their action: {forward, left, right, None}
Car to left with their action: {forward, left, right, None}
Car to right with their action: {forward, left, right, None}

For reaching way point:

Direction to way point:{forward, left, right}

Therefore by combining these we can get a 5-tuple (traffic light, oncoming, left, right, way point) to indicate the state.

Note: this state information is justifiable because it gives us the direction to the way point which helps us choose the correct path, and it gives us information that can help us avoid being negatively rewarded for not obeying traffic rules.

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?*

ANSWER:

For the 5-tuple there exist (2*4*4*4*3) 384 possible states (combined with a possibility of 4 actions for each space, that is a lot of q updating). This state space makes sense to me in general (meaning if I had a lot of trials and larger time limits, I could learn all states with a q value associate to one of the four options); however, I will not be using this state space. Because we only perform a hundred trials, have a relatively low time limit to reach our destination, and there only exists a few other cars on the road, I

am going to choose a reduced state space that is a 2-tuple. It is just (traffic light, way point), which has a state space of (2*3) 6 .

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?*

ANSWER:

To understand an agent's behavior, we need to understand how Q learning is working. In python we initiate a default Q table, which has a key that is the state and a value which is an action associated with a Q value. For us,we initiate all possible action values as 1. (We can easily change it to what ever we want by changing 'init_value' under 'Initialize any additional variables here' in 'agent.py.

Then after actions are taken, the Q's are updated with the Q learning formula:

Q[state,action] ← (1-alpha)Q[state,action] + alpha (r + gamma(max_action(Q[state',action'])))

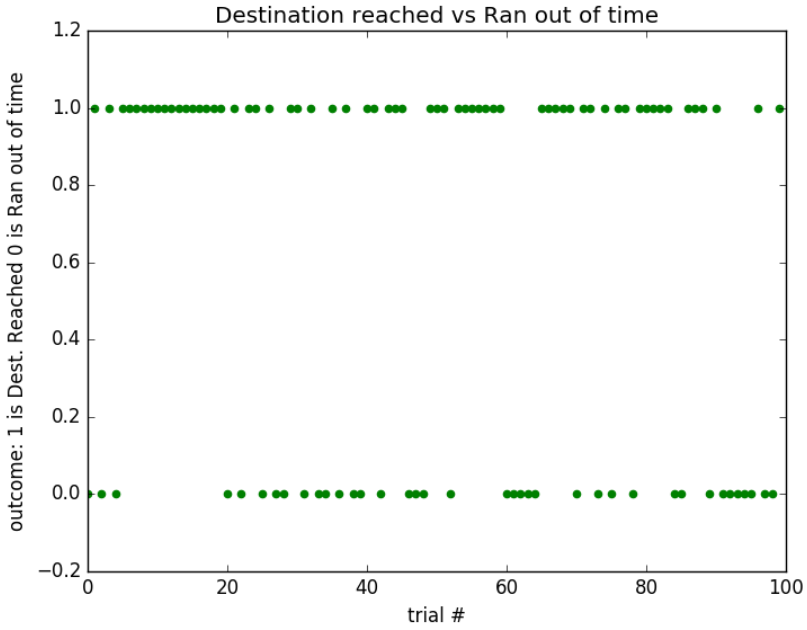
where alpha is the learning rate and gamma is the discount factor.

Also note, in the code there is an epsilon. The epsilon takes a value between [0,1] and is lets an agent take a random action with a probability epsilon. The point of epsilon is to help exploration (help us get out of local min if we get in one). By not taking random action we just rely on what we know- the Q values.

'ql_e5g5a5.txt' holds results in which we run 'get_stats.py'
Running the agent under Q learning with parameters: Q table all 0's, discount factor=0.5, learning rate=0.5, and epsilon=0.5

>>%run get_stats.py

There were 62 successes out of 100 trials



This scenario has a high epsilon. Therefore, it fails to use it knowledge a lot and explores too much. In the next section we will lower the exploration and try other parameters.

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?*

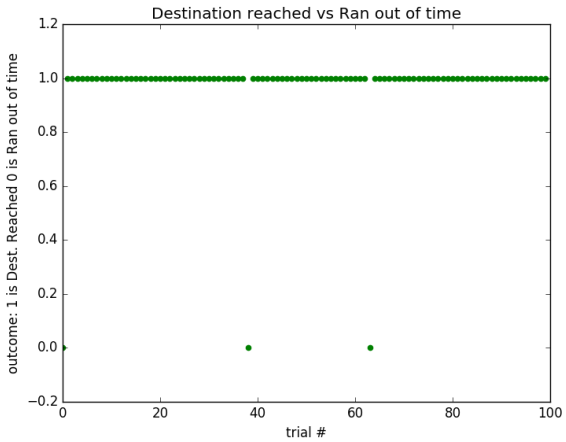
ANSWER:

As I stated previously, epsilon is way too high. By changing that alone, we will get a better number of successes. Below are the following setups. Above the setups will be the file that holds the results.

‘ql_qt0e1g5a5.txt’
Running the agent under Q learning with parameters: Q table all 0’s, discount factor=0.5, learning rate=0.5, and epsilon=0.1

>>%run get_stats.py

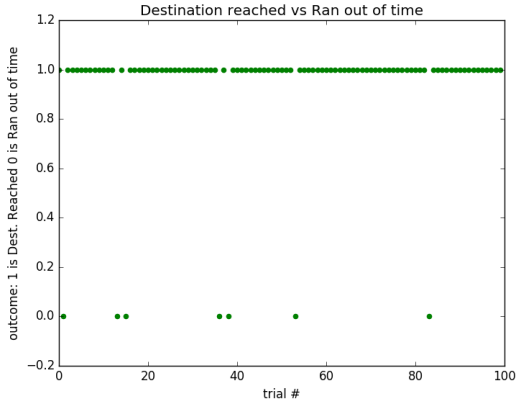
There were 97 successes out of 100 trials



ql_qt1e1g5a5.txt
Running the agent under Q learning with parameters: Q table all 1’s, discount factor=0.5, learning rate=0.5, and epsilon=0.1

>>%run get_stats.py

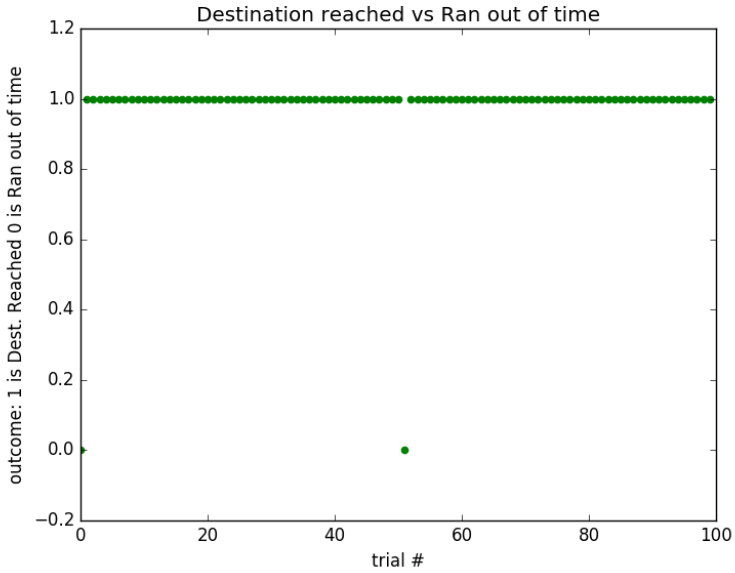
There were 93 successes out of 100 trials



ql_qt15e1g3a5.txt
Running the agent under Q learning with parameters: Q table all 1.5's, discount factor=0.3, learning rate=0.5, and epsilon=0.1

>>%run get_stats.py

There were 98 successes out of 100 trials



This last agent performs very well, where well is defined as reaching the way point before the deadline expires. In the next section, I explore whether this last agent actually gives an optimal policy that would avoid breaking laws or accidents.

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?*

ANSWER:

The agent's policy given the q learning is:

given state: ('green', 'right') → turn right
given state: ('green', 'forward') → go straight
given state: ('red', 'right') → turn right
given state: ('green', 'left') → turn left
given state: ('red', 'left') → stay put
given state: ('red', 'forward') → stay put

we get this from the final Q table after running 'agent.py'

++
final Q table
++
('green', 'right') ['0.30', '-0.14', '2.65', '1.50']

('green', 'forward') ['2.14', '0.20', '0.02', '0.70']
('red', 'right') ['-0.25', '-0.22', '2.55', '0.87']
('green', 'left') ['0.32', '2.16', '0.57', '1.50']
('red', 'left') ['-0.80', '-0.89', '-0.15', '0.00']
('red', 'forward') ['-1.00', '-1.00', '-0.07', '0.00']

However, though it does do well at reaching the way point under the deadline. It does not avoid breaking traffic rules. Below we see the penalties occurred during each trial. Earlier trials have more penalties, which makes sense because we are still learning. Yet, we never get rid of all penalties even in later trials, and the reason for this is our 2 term state space. We never see opposing cars, therefore we will always act as if they are not there (this can be seen from -1 in later trials). Therefore we occur penalties in later trials (in all) because we are performing illegal activities with other cars around. We could increase the state space to account for them, but because of our small amount of trials and limited deadlines we would never learn all the state space and would reach the way point in the alloted time less often than we do now.

==========
An Array of Arrays where each subarray shows neg rewards for a trial
==========
[['', '-0.5', '-0.5', '-0.5', '-1.0', '-1.0', '-1.0', '-1.0', '-0.5', '-0.5', '-0.5', ''], ['', '-1.0', '-1.0', '-0.5', '-1.0', '-1.0', '-0.5', '-0.5', ''], ['', '-0.5', '-0.5', ''], ['', '-0.5', '-1.0', '-1.0', ''], ['', '-1.0', ''], ['', ''], ['', '-0.5', '-0.5', '-0.5', '-1.0', ''], ['', '-1.0', ''], ['', ''], ['', '-1.0', ''], ['', '-1.0', '-0.5', ''], ['', '-0.5', '-1.0', ''], ['', '-1.0', '-1.0', ''], ['', '-0.5', '-1.0', '-0.5', ''], ['', ''], ['', '-0.5', '-1.0', '-0.5', ''], ['', '-0.5', '-0.5', '-0.5', ''], ['', ''], ['', '-1.0', ''], ['', ''], ['', '-1.0', '-0.5', ''], ['', '-0.5', '-0.5', '-1.0', '-1.0', ''], ['', '-0.5', ''], ['', ''], ['', ''], ['', '-1.0', ''], ['', '-0.5', '-1.0', ''], ['', '-1.0', '-0.5', ''], ['', ''], ['', '-0.5', '-1.0', ''], ['', '-0.5', '-1.0', '-0.5', '-1.0', '-0.5', ''], ['', ''], ['', '-0.5', ''], ['', ''], ['', '-0.5', ''], ['', '-1.0', ''], ['', ''], ['', ''], ['', '-1.0', ''], ['', '-1.0', ''], ['', '-0.5', '-0.5', '-0.5', '-0.5', '-0.5', ''], ['', ''], ['', ''], ['', ''], ['', ''], ['', '-1.0', ''], ['', '-0.5', ''], ['', '-1.0', '-1.0', '-1.0', ''], ['', '-0.5', ''], ['', '-0.5', '-0.5', '-1.0', '-0.5', ''], ['', '-0.5', ''], ['', ''], ['', '-1.0', '-1.0', ''], ['', '-0.5', ''], ['', '-1.0', ''], ['', '-0.5', '-0.5', '-1.0', '-1.0', ''], ['', ''], ['', ''], ['', '-0.5', ''], ['', '-0.5', '-1.0', '-0.5', ''], ['', '-0.5', '-0.5', ''], ['', '-0.5', '-0.5', ''], ['', '-0.5', ''], ['', '-1.0', ''], ['', ''], ['', ''], ['', '-0.5', ''], ['', '-1.0', ''], ['', '-0.5', '-0.5', ''], ['', '-0.5', ''], ['', '-1.0', '-0.5', ''], ['', '-1.0', ''], ['', '-0.5', ''], ['', ''], ['', ''], ['', '-0.5', '-1.0', ''], ['', '-0.5', ''], ['', '-1.0', '-0.5', '-0.5', '-0.5', ''], ['', '-1.0', '-0.5', ''], ['', '-1.0', '-1.0', ''], ['', '-1.0', ''], ['', '-1.0', ''], ['', '-1.0', ''], ['', '-0.5', ''], ['', '']]

Because of this, we have to decide on what we are trying to accomplish and how we could do it. In an ideal world, we would use the 5-tuple for the state and just increase the deadline and amount of trials so we could update (learn) q values for all states (therefore getting an optimal policy).