Part 1: Implement a basic driving agent

I've saved this basic implementation as agent1.py which made just one simple change to the original file:

Line 28 was originally: action = None

And has been changed to action = random.choice(Environment.valid_actions)

Which chooses an action randomly (None, left, right, forward.)

The agent will move in accordance to the random action chosen as long as the action is allowed by the current state of the traffic lights.

Result: The agent eventually reaches the destination through trial and error but not after considerable time as it moves in random directions.

Part 2: Identify and update state

Every state will be determined by a 2-tuple: (Waypoint, Light)

Due to the # of cars in the simulation, it's very rare that we run into another car on the left, right, or oncoming positions and even then, there are no rewards or penalties for running into other cars. So to simplify the learning, I've left those variables out of the state.

Right now, there are 3 possible choices for Waypoint (Left, Right, and Forward) plus 2 choices for Light (green or red), making it 6 possible states. Couple it with 4 different actions (None, Left, Right, and Forward) and that gives 24 different State + Action combos that need to be trained.

By adding unnecessary state variables, the agent will take longer to be fully trained because the state space will become sparser.

I also don't think including Deadline into the state variable is worthwhile since the agent doesn't get extra rewards for arriving early. Also, since destinations can be between 1-12 action steps away, the

domain for the Deadline variable will span between 5 and 60, a total of 56 values. If we cross that with our original state-action space of size 24, we'll be looking at 24*56 = 1344 state-action pairs to train for!

I've implemented this intermediate step as agent2.py.

Part 3: Implement Q-Learning

I implemented Q-Learning with the learning _rate = 0.5 and discount_factor = 0.5. The Q dictionary is initialized with all values set to 0 by default. At every step, the agent checks the Q-values for all possible actions (None, left, right, and forward) and picks the one that yields the largest Q-value. If all possible actions lead to a Q-value of 0 (which means those Q-values have never been calculated yet), then the agent will just pick one of the 4 actions randomly.

I also started to enforce deadlines and changed the # of trials to 100.

With the above implementation, I found that my agent would quickly learn not to disobey the traffic signals after several trials. However, it would get stuck in a local optima where it ends up staying in the same place (action = None) and collecting a reward of 1 every time. Since the traffic lights seem to cycle every 5 rounds, I've modified my agent to keep track of how many rounds in a row did it choose action = None and if it's greater than 6, then it chooses an action at random regardless of Q values. With this change, the agent doesn't get stuck in one place and can continue learning.

With these changes, the agent seems to make it to its destination a bit more frequently than moving completely randomly. These changes can be found in agent3.py. Line 79 is where the Q-value are updated.

After finishing each run (of 100 trials), a scatter plot will be generated. A dot at "0" indicates that the cab failed to reach the destination during that particular trial and a dot at "1" indicates that the cab reached the destination. Here's how it performed:

As you can see, there are far more failures (y=0) than successes (y=1) so I'll explore improving the parameters in the next section.

Part 4: Enhance the Driving Agent

The parameters that we can change (along with their default values) are as follows:

- Initial Q value = 0
- Discount Factor = 0.5
- Learning Rate = 0.5

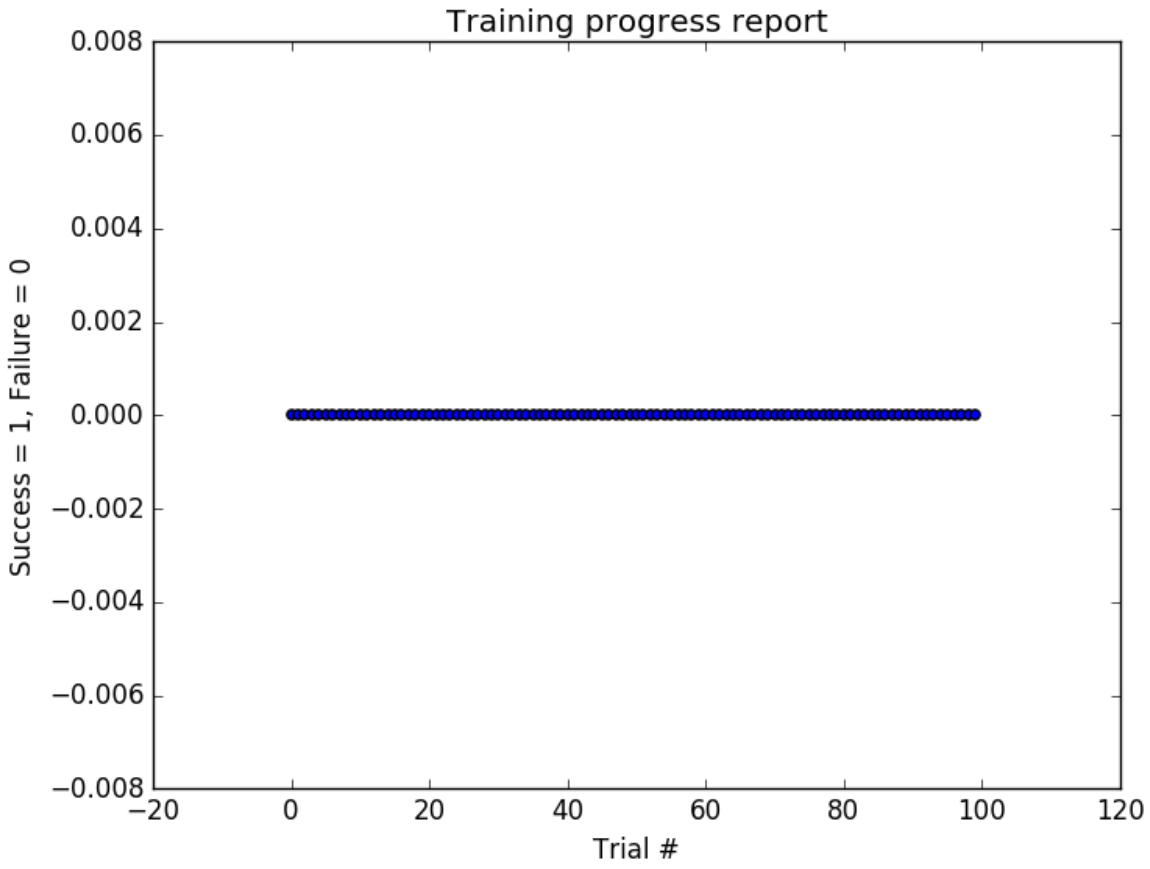
From this point on, I'll represent the parameters as a 3-tuple in our discussion: (Q0, Discount, Learning)

Let's try changing the Discount Factor to 0.75: (0, 0.75, 0.5). Here's the resulting graph:

It seems the results are still not up to par. Let's try using a discount factor of 0.25 instead (0,0.25,0.5):

Still not good enough. Let's try adjusting the learning rate to 0.75.

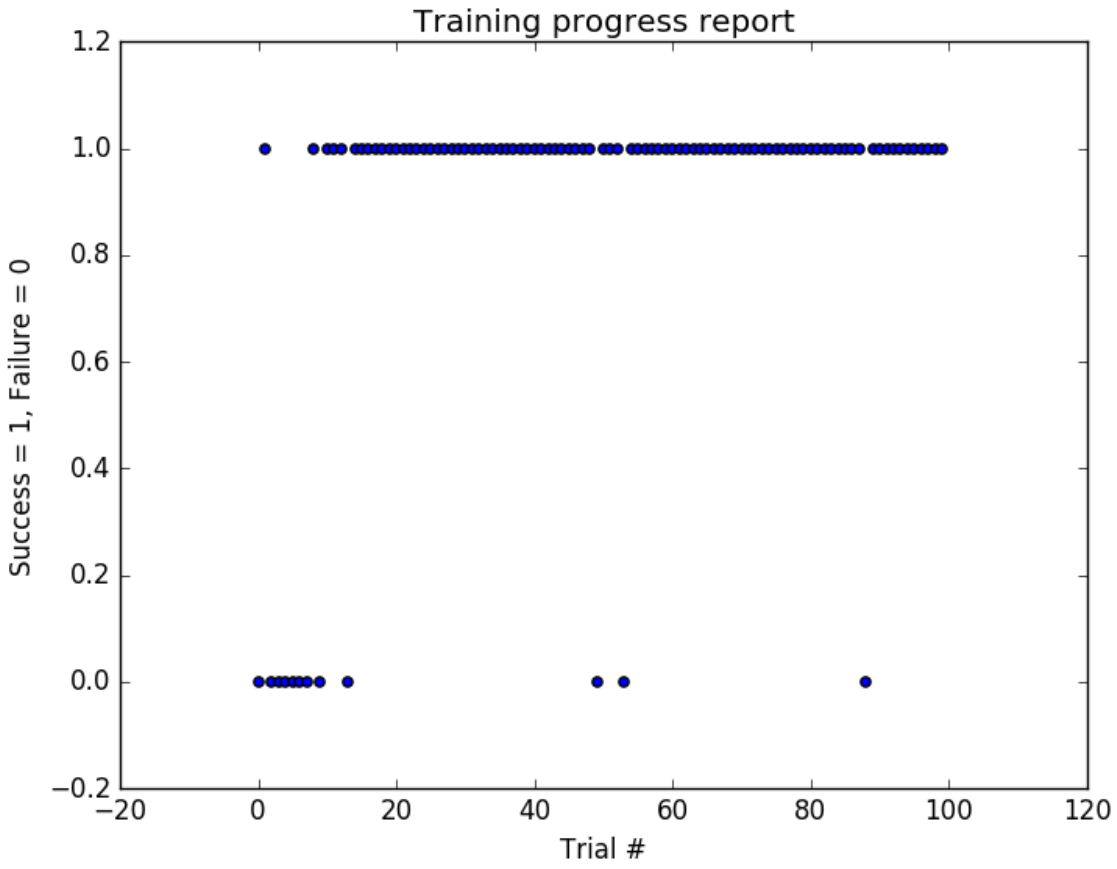
(0,0.25,0.75):



No Successes!!

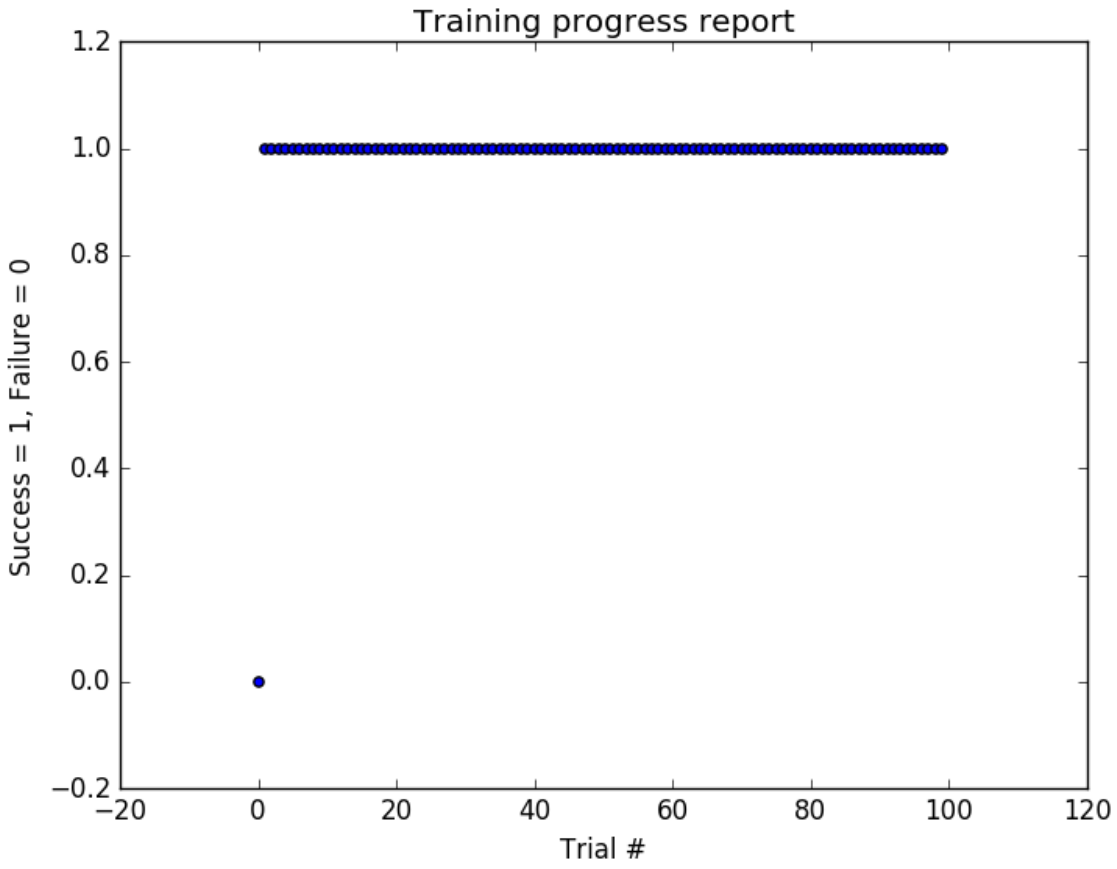
Let’s try resetting our parameters but increasing our initial Q-value to 1.

(1,0.5,0.5):



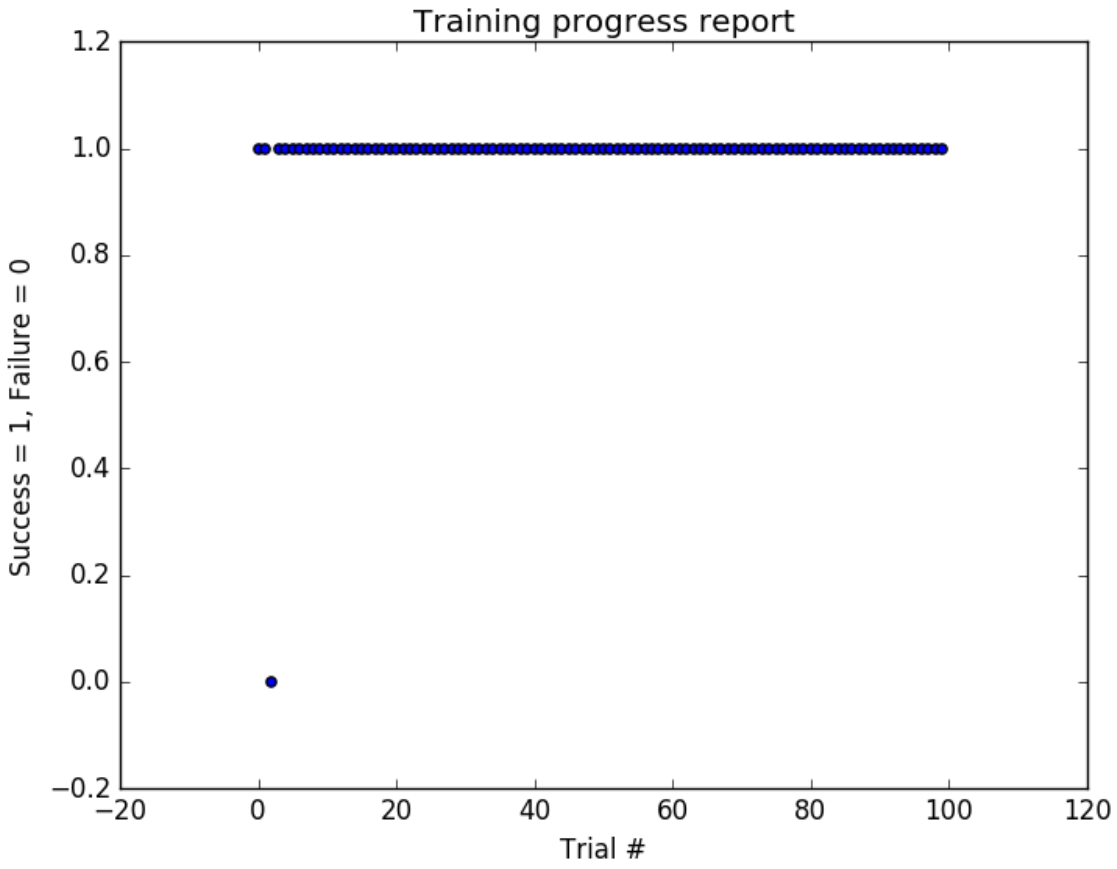
Seems like a vast improvement. Let's try an initial Q of 2:

(2,0.5,0.5):



I've noticed that the agent receives a reward of 2 when it follows a waypoint, the highest reward possible besides reaching the destination (which awards 12.) So it seems that by choosing an initial Q of 2, most state changes will reduce the Q value. This has the effect of forcing the agent to visit all possible states in the beginning which improves the agent's judgement.

Let's try running (2, 0.5, 0.5) again:



It seems the agent is able to reach its destination well over 90% of the time with these parameters:

- Initial Q: 2
- Discount Factor: 0.5
- Learning Rate: 0.5

And the agent learns very quickly in the first trial!

The final code, agent.py, uses the above parameters. I've disabled the plotting feature to make it more in line with the original code.