Smartcab Reinforcement Learning Project

Implement a Basic Driving Agent

The agent randomly chooses one of four moves: none, forward, left, or right. When the deadline is not enforced, the agent eventually makes it to the destination, but it can take a long time. The agent only arrives at the destination by luck and random chance.

In a test of 100 trials, the agent arrives at the destination within the deadline only 19 times.

Identify and Update the State

From a global perspective, these are the variables that we would want to consider when choosing an action:

- Current location
- Current heading
- Destination
- Traffic light at current intersection
- Oncoming traffic & direction
- Left traffic & direction
- Right traffic & direction
- Time steps remaining

However, we can narrow this down because at any given intersection we are only considering the local environment. The RoutePlanner has already determined which direction we should go, so we can replace current location and current heading with next_waypoint. Essentially our AI should decide between making the decision from next_waypoint or remaining still. We don’t need destination because it is irrelevant to our local decision. We still need traffic light, oncoming/left traffic & direction, and time steps remaining. We actually don’t need to consider traffic from the right: because of right-of-way rules, it will never change the decision.

One of the issues with this is that by only following the route planner we may not get to our destination in time. However, there would have to be a lot of red lights for this to occur. For now, I will follow the route planner, but we may want to check if sometimes deviating from the route plan (and taking a small negative reward) could lead to a larger future reward by arriving at the destination on time.

For now, our state is defined as a Python list of the following variables:

- Next waypoint (‘forward’, ‘left’, ‘right’)
- Traffic light at intersection (‘red’, ‘green’)
- Oncoming traffic & direction (None, ‘left’, ‘right’, ‘forward’)
- Left traffic & direction (same as oncoming)

Next waypoint tells us the direction the route planner wants us to go. Traffic light provides information into what actions are legal: if green, all actions are legal; if red, only right turn is legal. All traffic directions tell us which actions will lead to an accident. We might also want to consider time steps remaining - L_1 distance from destination. If this number is less than 0, there is no way for the agent to reach the destination within the time allotted. The agent would therefore want to avoid any moves with a negative reward. I will run the learner for now without any timestep input. We already have 3*2*4*4 = 96 states, which is a lot for the trainer.

Implement Q-Learning

First, I create a table of Q-values for each state. There are 96 states which are denoted by the Python tuple (waypoint, trafficlight, oncoming, left). Each unique state gets initialized as a key into the *states* dictionary. The value for each state key is initialized as a dictionary of actions: {None:0, 'forward':0, 'right':0, 'left':0}. This is taking a naive approach where at the beginning we all actions in all states have an equal value of 0.

At each time step, the learner looks up the current state in the *states* dictionary. There are 3 possible scenarios: (1) one action has the largest Q-value, (2) two or three actions have the largest and equal Q-values, or (3) all four actions have equal Q-values. In scenario (1) the learner simply chooses the action with the largest Q-value. In scenario (2) the learner randomly chooses between the two or three actions with equal proportion. In scenario (3) the learner randomly chooses between all four actions.

We know (although the learner doesn't) that the reward will be consistent across all time steps at a specific state given the same action.

- A valid and correct action leads to a reward of +2.0
- A valid but incorrect action leads to a reward of -0.5
- A null action leads to a reward of 0.0
- An invalid action leads to a reward of -1.0

This game is non-deterministic, so we work backward to update Q values. The following version of Bellman's equation is how we update the Q values:

$$Q(s_{t-1}, a_{t-1}) \coloneqq Q(s_{t-1}, a_{t-1}) + \alpha(r_{t-1} + \gamma * max_{a_t}Q(s_t, a_t) - Q(s_{t-1}, a_{t-1}))$$

When the agent makes an action, it goes back and updates the Q table for the state it was in when the action occurred.

Q-Learning Results

(i) Testing various values for α and γ

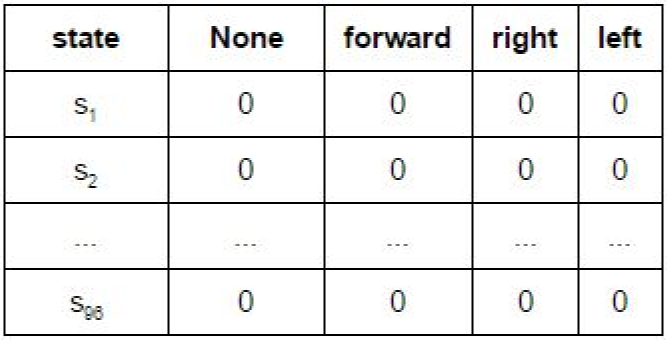
For the following α and γ values, I ran the simulation three times with 100 trials. The success rate is reported for the last 10 trials of each simulation.

α	γ	Success Rate 1	Success Rate 2	Success Rate 3	Comments
0.0	0.0	40%	40%	20%	No learning, all random actions.
0.25	0.25	100%	100%	90%	
0.5	0.5	100%	100%	100%	
0.75	0.75	100%	100%	100%	
1.0	1.0	100%	100%	100%	

What it looks like here is that there is no difference in the agent's learning ability when α > 0 and γ > 0.

(ii) Why α and γ don't really tune the agent (spoiler: initial conditions)

The reason that various values of α and γ such that both are greater than 0 doesn't tune the model is because of our initialization of each Q value as 0. Recall that our Q table is initialized with the values to the right.



state	None	forward	right	left
s_1	0	0	0	0
s_2	0	0	0	0
...	...	...	...	...
s_{96}	0	0	0	0

The first time we encounter a state, all Q values are 0, so we choose a move randomly. Suppose we choose the correct move. We receive a reward of 2.0, and if α > 0 then the Q table for that state and action will be updated to a positive value. Every time we arrive at the state in the future, we will again choose the proper action (this time not randomly) because it has a Q value that is positive whereas the other Q values for that state are all 0.

Now suppose the random action is either invalid or incorrect. The reward will either be -0.5 or -1.0, so the Q value for that state/action will be updated to a negative value. If the state is encountered again in the future, that action will not be taken because at least one other action will have a Q value >= 0.

The action None provides no benefit on two levels: the reward is 0 and no exploration occurs. We can improve our agent by removing None from the possible unexplored random actions. If None happens to be the right action, our agent won't know until it has explored all three other actions.

(iii) Rewards, Penalties, and Learning Time with varying α and γ

Simulation Values	Success Rate on Final 10 Trials	Success Rate on all 100 Trials
Random action	20%	29%
$\alpha = 1.0$, $\gamma = 1.0$	10%	32%
$\alpha = 0.75$, $\gamma = 0.75$	100%	100%
$\alpha = 0.5$, $\gamma = 0.5$	100%	98%
$\alpha = 0.25$, $\gamma = 0.25$	100%	99%

In the next section I will analyze each of these in further detail.

Enhancing the Driving Agent

(i) Initial Q values and exploration

With the initial Q values all set to 0 we could describe our agent's exploration as "explore when you have to". At most our agent will have to encounter a state 3 times before it always knows the best action to take.

There is a sneaky way we can program the agent to never make a move with a negative reward by updating a few parameters. First, we need to initialize the Q-table differently. It will now look like:

State	None	Forward	Right	Left
('forward', a, b, c)	0.5	0.5	0	0
...	...	...	...	...
('right', d, e, f)	0.5	0	0.5	0
...	...	...	...	...
('left', g, h, i)	0.5	0	0	0.5
...	...	...	...	...

Note: a,b,c,d,e,f,h,i are placeholders for traffic light, oncoming, and left traffic. Their specific values don't matter

We initialize the Q-table so that None starts with a value of 0.5 and the next waypoint action also starts with a value of 0.5. When an agent encounters a state for the first time it will now test the next waypoint action. If that action is invalid its Q value will decrease, and the agent will know that the proper action to take in that state (to avoid both improper and invalid actions) is None. With these parameters, the agent will only need to arrive at a state once to always make the proper action at that state in the future. In this scenario, we don't actually have to implement Q learning, we could just add the reward to the Q value. In all three simulations of 100 trials the agent successfully reached its destination within the allotted time on the last 10 trials 100% of the time.

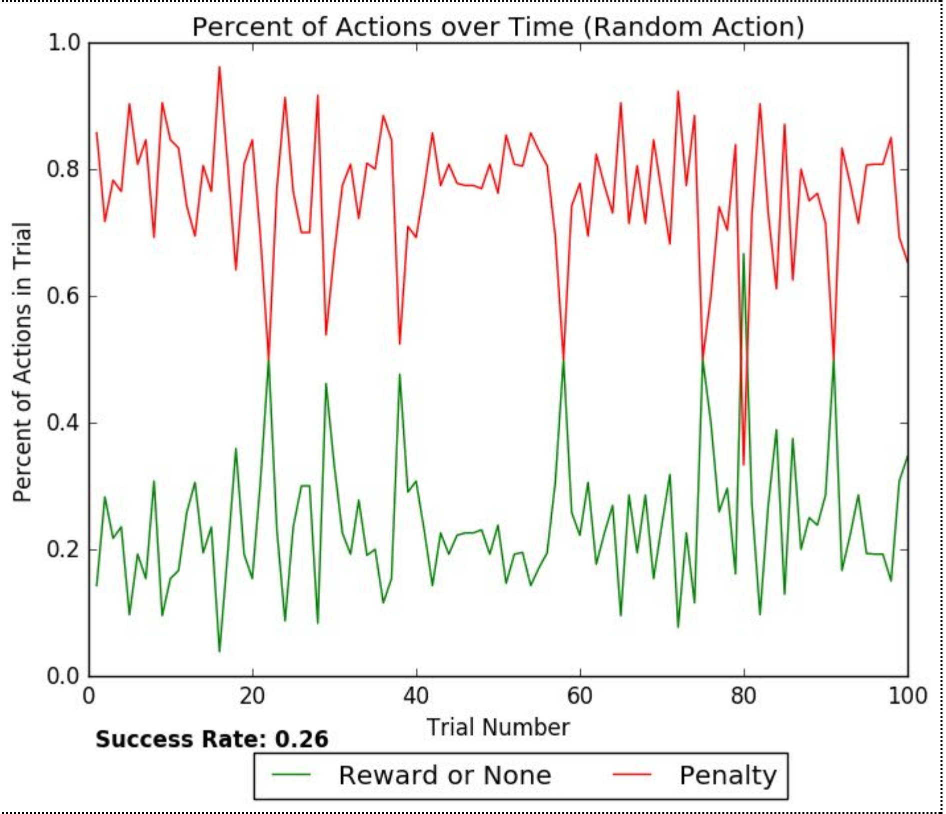
(ii) Thinking about ideal policy

In the case of this game, the agent can continue to accrue rewards until time runs out. Because the reward for a proper move (2.0) is greater than the negative reward for an improper or invalid move (-0.5, -1.0), the agent may have an incentive to make a few improper moves if it means they will be able to accrue more proper moves before the time runs out.

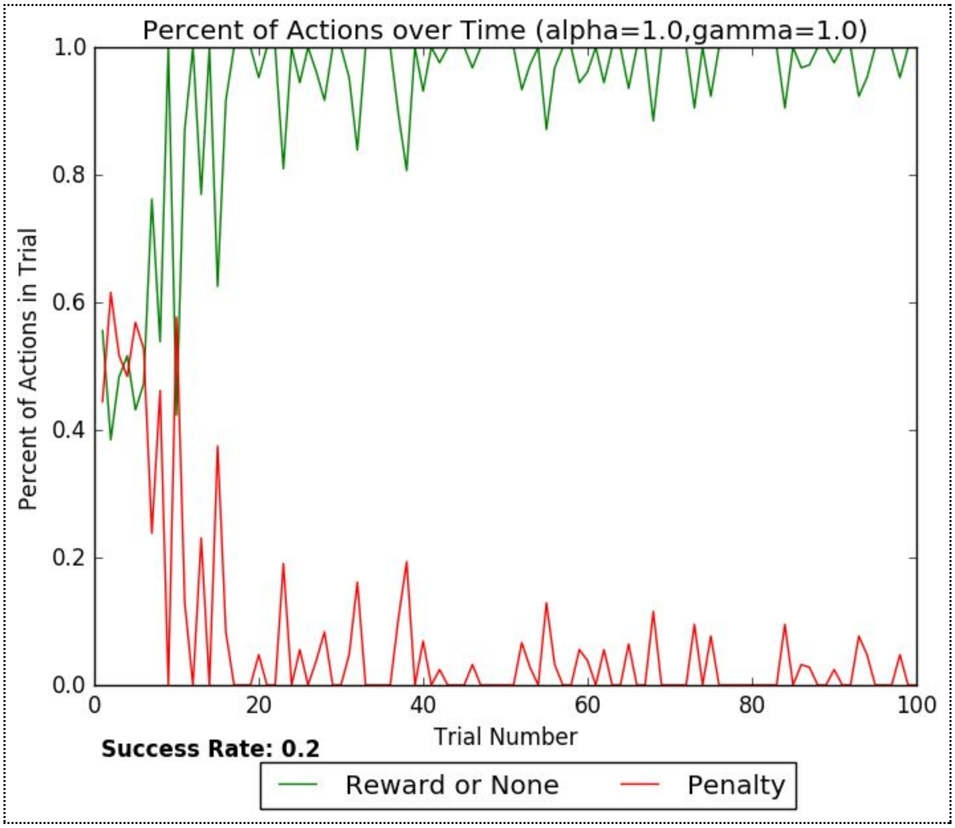
If this were the real world, we'd most likely want our agent to take the shortest possible route while following all the rules of the road. This is precisely what the agent learns to do in the scenario stated in (i) where the Q-values are not all initialized to 0.

(iii) Looking at agent performance over time

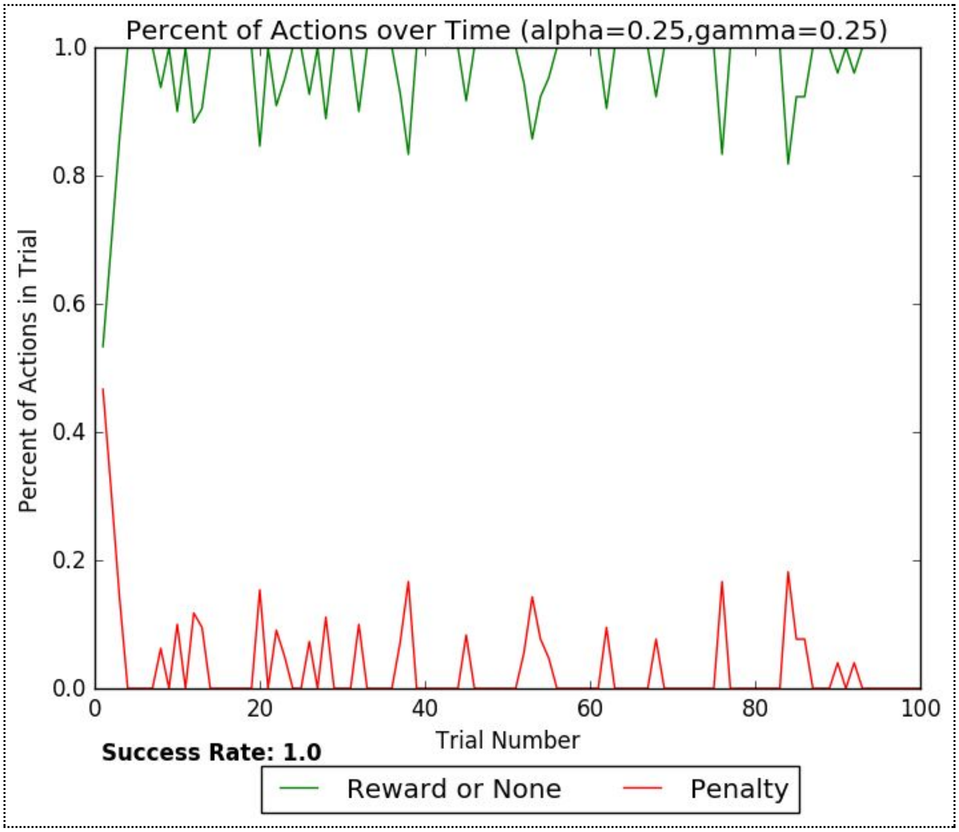
The charts below in this section show the percentage of actions an agent makes on each of the 100 trials. The actions are put into two categories: 'Reward or None' and 'Penalty'. In the chart to the right, the agent always makes random actions. We can see that the 'Reward or None' rate really doesn't improve over time (nor would we expect that).



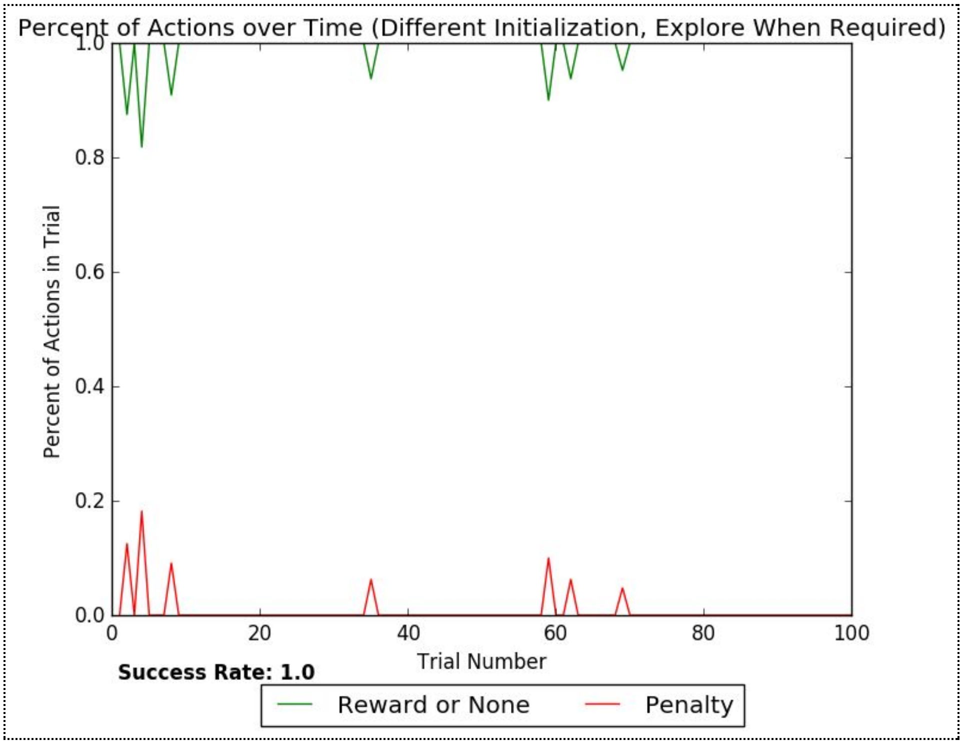
On the other hand, when α = 1.0 and γ = 1.0 as in the chart to the right, the agent frequently makes penalty actions for the first 15 trials. Around about trial 20 the agent consistently makes fewer penalties, somewhere around 5% of the time. However, this agent still only succeeds on 20% of the trials.



The chart to the right shows performance when α = 0.25 and γ = 0.25. Here we see quicker learning with the Reward or None rate dropping close to 0 within the first 10 trials. We see various small spikes throughout all 100 trials when the agent has to explore a new state. These parameters work well with a 100% success rate over 100 trials.



In this final chart we see the results when we use the different Q value initialization (None=0.5, next waypoint=0.5, other actions=0). We see that the agent does the majority of its learning in the first 10 trials. The four red spikes to the right of the 20th trial denote instances when the agent encountered a novel state and happened to make a decision resulting in a penalty. This agent performs very well with a 100% success rate.


Percent of Actions over Time (Different Initialization, Explore When Required)
Percent of Actions in Trial
Trial Number
Success Rate: 1.0
Reward or None
Penalty