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.

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 decision. We still need traffic light, oncoming/left/right traffic & direction, and time steps remaining.

One of the issues with this is that by only following the route planner we may not get to our destination in time. 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 large 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)
- Right 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 three 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*4 = 384 states, which is a lot for the trainer.

Implement Q-Learning

First, I create a table of Q-values for each state. There are 384 states which are denoted by the Python tuple (waypoint, trafficlight, oncoming, left, right). Each unique state gets initialized as a key into the *states* dictionary. The value for each key is initialized as another dictionary: {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

The first time we encounter a state, we have to choose a move randomly. Based on these reward values, if we choose a value other than None, we will have learned either that this is always the best action or if this will never be the best action. If the randomly chosen move receives a reward of +2.0, all other possible moves will either receive a reward that is 0 or negative. When the Q-table is updated, the correct move will have a value of 2.0, so it will always be chosen in the future (largest Q-value), and since the rewards are consistent across time steps, the correct move will continue to receive 2.0 each time it is chosen in the future.

On the other hand, if the randomly chosen move is valid but incorrect or invalid, it will receive a negative reward. The Q-values are initialized at 0, so the next time the agent encounters the state it will randomly select between all moves except the move that now has a negative reward.

In summary, the learner will at most need to encounter a state three times to learn the proper action in the state. When I trained the agent this way, it reaches the destination within the deadline close to 100% of the time, even in the first few simulations.

Enhancing the Driving Agent

One quick way to enhance the driving agent is to understand that in scenarios (2) and (3) where random choice is required, we do not improve the agent by randomly choosing the None action. For example, if the first action on the first trial is None, the Q-values for that state will all remain at 0. If the agent returns to that state it will again randomly choose between all four possible actions.

To fix this we can eliminate None from the random choice. If None is in fact the optimal action in a given state, we will try all the other actions first. To evaluate our agent, I use the following metrics:

- **Success rate:** number of successful trips / number of trials
- **Trip efficiency:** L1 distance between start and destination / Number of actions taken
- **Average efficiency:** sum of trip efficiency / number of trials

Mainly I want to look at three things: success rate, average efficiency, and trip efficiency over time.

Success Rate

Trials = 100	Success Rate 1	Success Rate 2	Success Rate 3	Success Rate 4	Success Rate 5
Before Enhancement	99%	100%	99%	100%	100%
After Enhancement	99%	99%	99%	100%	99%

I ran the simulation 5 times for both the pre and post enhancement agents. There's almost no variation in success rate with the enhancement. This is not because the enhancement failed to work, but actually because the pre-enhancement agent was either 99% or 100% accurate. There just wasn't any room to improve. All this points to the conclusion that we should look at other metrics, like trip efficiency.

Average Efficiency

Trials = 100	Average Efficiency 1	Average Efficiency 2	Average Efficiency 3	Average Efficiency 4	Average Efficiency 5
Before Enhancement	0.527	0.551	0.499	0.530	0.537

After Enhancement	0.496	0.490	0.484	0.496	0.473

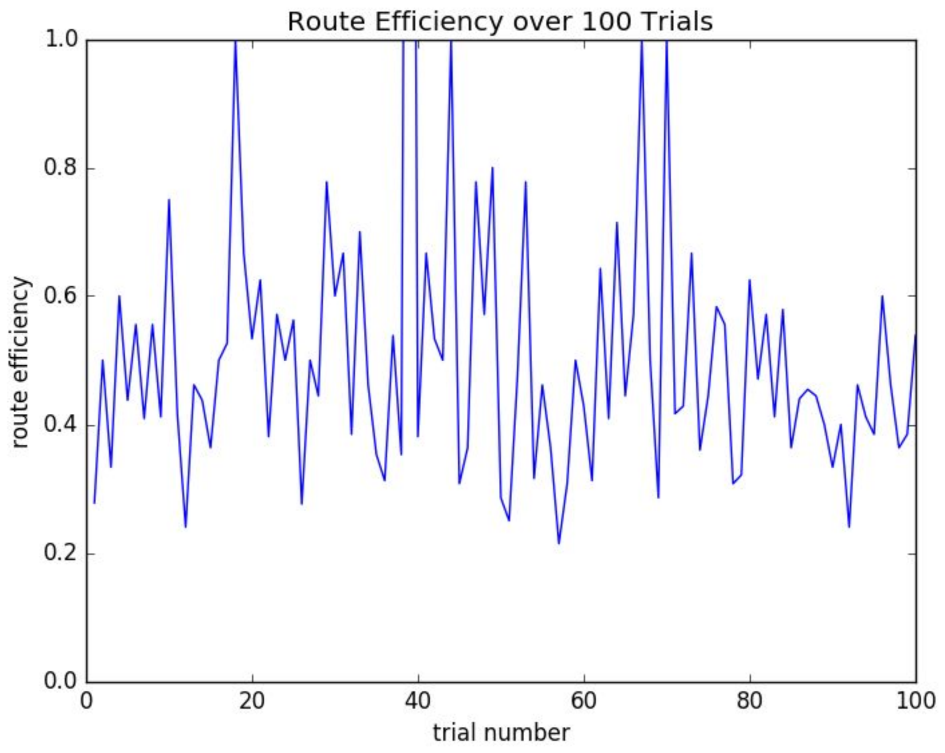
Surprisingly the non-enhanced agent has a slightly better efficiency over 100 trials.

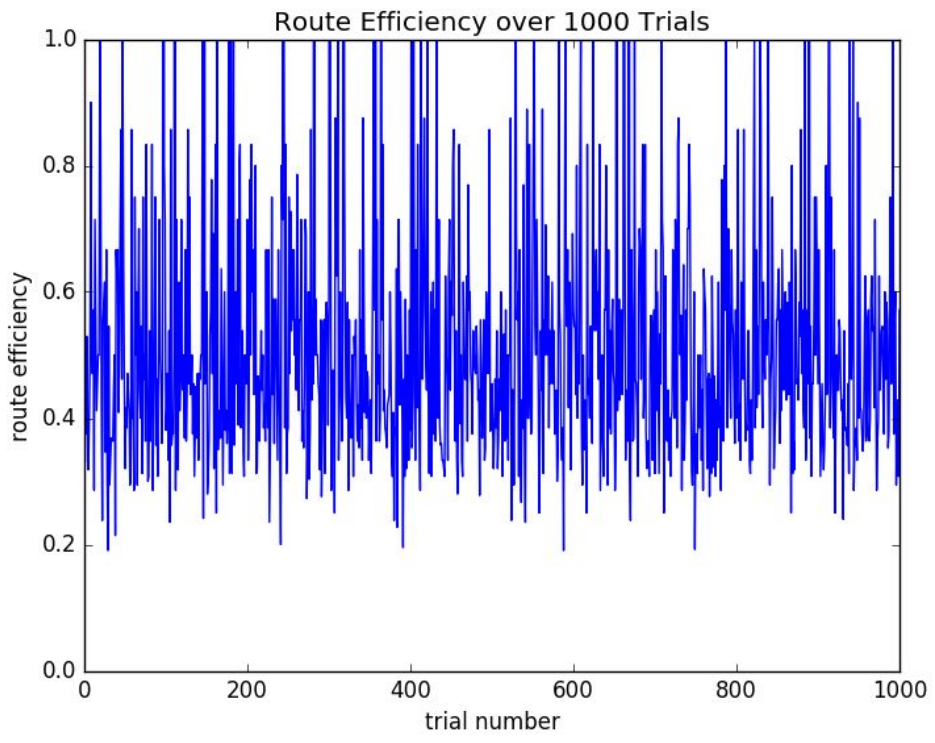
Trip Efficiency as Number of Trials Increases

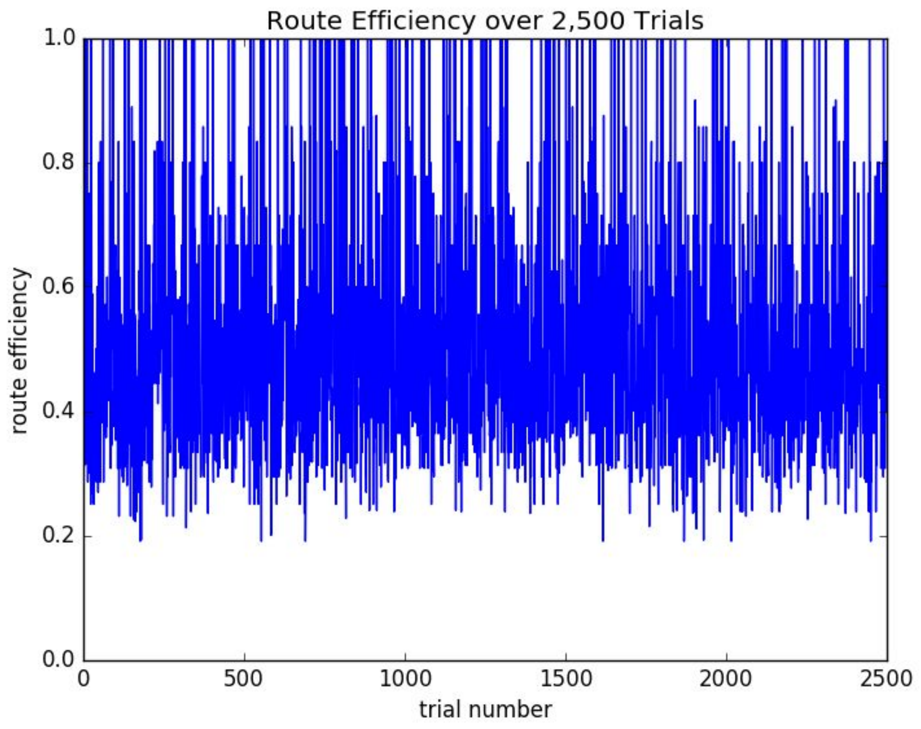
In the charts below I've plotted the individual trip efficiencies for 100 trials, 1,000 trials, and 10,000 trials. We should expect the trip efficiencies to improve as the number of trials improves.

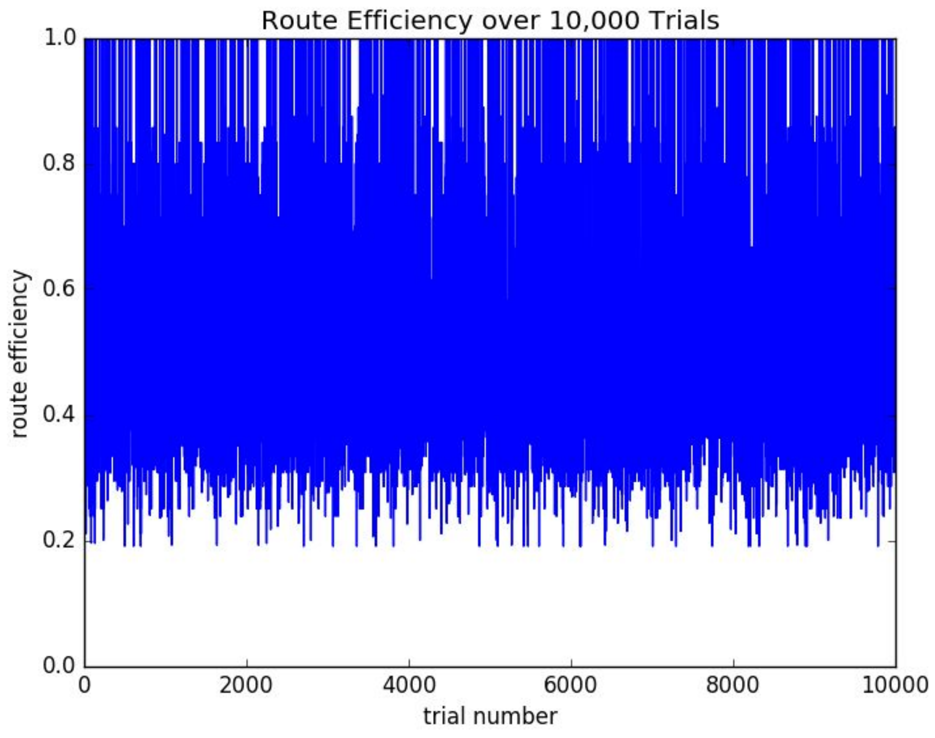
Trial Start Number	Trial End Number	Average Route Efficiency
1	1000	0.509879
1001	2000	0.520208
2001	3000	0.536719
3001	4000	0.519104
4001	5000	0.524663
5001	6000	0.521894
6001	7000	0.509658
7001	8000	0.525568
8001	9000	0.530189
9001	10000	0.518038

Surprisingly we don't see that average route efficiency improves as the number of trials increases. Ultimately our agent arrives at the destination within the deadline 99%+ of the time.


Route Efficiency over 100 Trials
route efficiency
trial number
0.0
0.2
0.4
0.6
0.8
1.0
0
20
40
60
80
100


Route Efficiency over 1000 Trials
route efficiency
trial number
0.0
0.2
0.4
0.6
0.8
1.0
0
200
400
600
800
1000


Route Efficiency over 2,500 Trials
route efficiency
trial number
0.0
0.2
0.4
0.6
0.8
1.0
0
500
1000
1500
2000
2500


Route Efficiency over 10,000 Trials
route efficiency
trial number
0.0
0.2
0.4
0.6
0.8
1.0
0
2000
4000
6000
8000
10000