Project 4 - Smart Cab Driving Agent

Implement a basic driving agent

The code was run with `enforce_deadline=False` and it was observed that one of either two things happened on a trial:

1. The agent made it to the destination, by just randomly fumbling its way into the target location.
2. I ran out of patience and cancelled the simulation.

Identify and update state

The learning agent has available the following data about the intersection it is currently located at:

- **Inputs** :
  - The traffic light state
  - Whether a car is approaching this intersection from the front, left and right
- **Next waypoint** : The agent has access to the next waypoint which is all it knows about where the destination is
- **Deadline** : The agent knows how much time it has left to get to the destination

I chose to give the agent all of this information to let it learn for itself what to do and what not to do at each intesection, except the deadline. I did not include the deadline because that would explode the state space for the Q-table.

With that in mind, the input vector is defined as:

` def get_state(self):

```
    inputs = self.env.sense(self)

    return ( inputs['light'], inputs['oncoming'], inputs['left'],
 inputs['right'], self.next_waypoint )
```

`

Implement Q-learning

I implementd a basic Q-learner to allow the agent to learn from its surroundings. The parameters for the Q-learner were set to explore the performance of the algorithm and see how each parameter might affect it:

- ϵ: This is our representation of the exploration-exploitation dilemma. ϵ represents the probability of exploration, i.e. the probability that the learner will pick a random action instead of *exploiting* the knowledge that it has through the Q-learning process. To start with, I set $\epsilon = 0.5$ to see how the algorithm would perform if I gave the learner an equal opportunity to explore or exploit.
- ϵ decay: This is a boolean value that represents whether we want to decay our ϵ value as we go through the trials. One reason we might want to do this is that when our learner starts, we want to explore, but as it gets more knowledge of the state space, we might want it start using that knowledge. For the first run, I set it to *False* to see how the algorithm would perform without a decay.
- γ: This is the discount factor. This represents the weight we are attaching to future rewards. In our case, we don't really know much about the future, so a low value of γ might work better. For our first run, I set $\gamma = 0.4$
- α: This is the learning rate, i.e the rate at which we weight the information we already have and the weight we give to the information we are sensing about the future. Again, we might want to decay this, because as our learner learns more, it can presume that its representation of the world as it sees it in the present is getting more and more accurate. To start with, I set $\alpha = 0.6$. Also, an increased learning rate corresponds to a higher chance of local minima.

I implemented a class for Q-learning and ran the code with the values as mentioned above: $(\epsilon = 0.5, \gamma = 0.4, \alpha = 0.6, \epsilon_{Decay} = False)$. I noticed from the simulation that the agent starts with random choices of actions, but very quickly seems to converge on the strategy of using the `next_waypoint` input to move towards the goal.

With these values, I got a **success rate of 0.62**. This is good, but I believe we can do better.

Improving Q

To improve the performance of the algorithm, I decided to perform a grid-search over the various parameters available to the Q-learning algorithm. The values for the parameters were chosen to represent a range of possibilities for the learner:

- $\epsilon = [0.1, 0.3, 0.5]$
- $\gamma = [0.2, 0.4, 0.5]$
- $\alpha = [0.4, 0.6, 0.8]$
- $\epsilon_{Decay} = [True, False]$

The results are tabulated below.

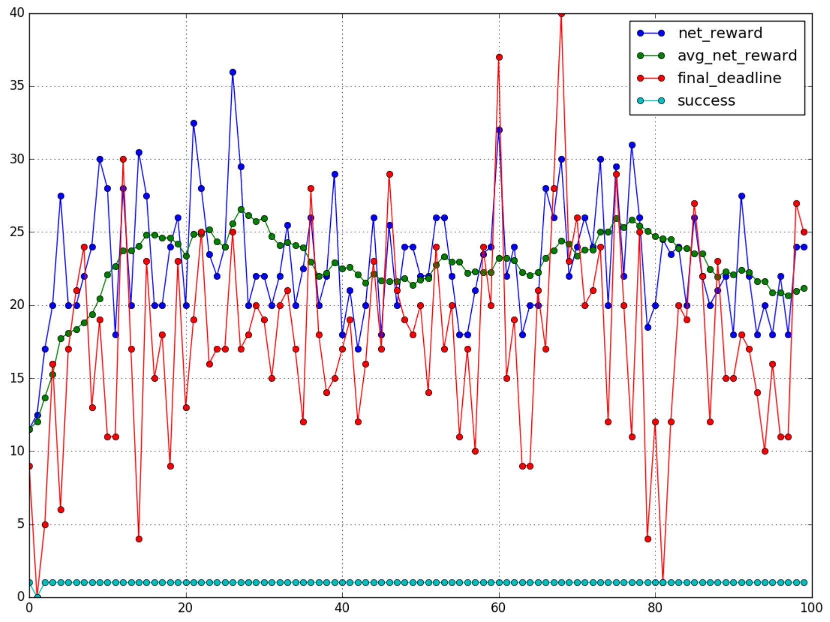
γ	α	ϵ	ϵ **decay**	**Performance**
.2	0.6	0.1	True	1.00
0.2	0.8	0.1	True	1.00
0.5	0.8	0.1	True	1.00
0.2	0.4	0.3	True	1.00
0.2	0.4	0.5	True	1.00
0.4	0.6	0.5	True	1.00
0.5	0.8	0.5	True	1.00
0.4	0.8	0.1	False	0.99
0.5	0.4	0.1	True	0.99
0.5	0.6	0.1	True	0.99
0.2	0.6	0.3	True	0.99
0.2	0.8	0.3	True	0.99
0.4	0.4	0.3	True	0.99
0.4	0.6	0.3	True	0.99
0.4	0.8	0.3	True	0.99
0.5	0.4	0.3	True	0.99
0.2	0.6	0.5	True	0.99
0.2	0.8	0.5	True	0.99
0.4	0.4	0.5	True	0.99
0.4	0.8	0.5	True	0.99

0.2	0.4	0.1	True	0.98
0.4	0.4	0.1	True	0.98
0.5	0.8	0.3	True	0.98
0.5	0.6	0.5	True	0.98
0.4	0.6	0.1	True	0.97
0.4	0.8	0.1	True	0.97
0.5	0.6	0.3	True	0.97
0.5	0.4	0.5	True	0.97
0.2	0.6	0.1	False	0.96
0.2	0.8	0.1	False	0.96
0.2	0.4	0.1	False	0.94
0.4	0.4	0.1	False	0.93
0.5	0.4	0.1	False	0.93
0.4	0.6	0.1	False	0.91
0.2	0.8	0.3	False	0.87
0.5	0.6	0.1	False	0.86
0.2	0.4	0.3	False	0.86
0.2	0.6	0.3	False	0.86
0.5	0.8	0.1	False	0.83
0.4	0.8	0.3	False	0.77
0.4	0.6	0.3	False	0.74
0.4	0.4	0.3	False	0.73
0.5	0.6	0.3	False	0.73
0.5	0.4	0.3	False	0.72
0.4	0.8	0.5	False	0.72
0.2	0.8	0.5	False	0.70
0.2	0.4	0.5	False	0.69
0.2	0.6	0.5	False	0.67

0.5	0.8	0.3	False	0.61
0.5	0.6	0.5	False	0.57
0.4	0.6	0.5	False	0.55
0.5	0.4	0.5	False	0.55
0.4	0.4	0.5	False	0.52
0.5	0.8	0.5	False	0.50

As can be seen, the algorithm performs best for: * A small value of γ: As discussed briefly above, this is expected given that the only real information we have about the future is the `next_waypoint` variable. So, the algorithm does well without assigning high weights to future rewards. * Any vale of α: As can be seen in the second column below, the values of α do not have an effect on the performance. Every level of performance has a representation of every value of α. * A low value of ϵ: Our algorithm prefers exploitation over exploration. This is also evident by how quickly the optimal policy is learned. Looking at the simulations, within a few iteration, the car is already making its way to the destination without any trouble. * $\epsilon_{Decay} = True$: Our algorithm prefers a decaying ϵ rather than a constant one. This one was an obvious choice. In fact, looking at the table below, the top half of the performing paramter tuples almost all have decaying values of ϵ! Our learner learns really quickly and really wants to explore!

From this analysis, the learner does seem to pretty quickly move to an optimal strategy. As can be seen below, the `net_rewards` is always positive and the `avg_net_reward` which represents the average net rewards converges to around 22, showing that the algorithm has converged to a strategy. Looking the performance on the simulation, the car does make a few mistakes, so the strategy may not be **fully** optimal, but the destination is reached every time.



Performance metrics for the optimal policy learned