Train a Smartcab How to Drive

Uirá Caiado

August 4, 2016

Abstract

A smartcab is a self-driving car from the not-so-distant future that ferries people from one arbitrary location to another. In this project, I will design the AI driving agent for the smartcab using reinforcement learning. This area of machine learning[1] is inspired by behaviorist psychology and consists in training the agent by reward and punishment without needing to specify how the task is to be achieved. The agent should learn an optimal policy for driving on city roads, obeying traffic rules correctly, and trying to reach the destination within a goal time.

1 Introduction

In this section, I will present a brief introduction to reinforcement learning and to the problem addressed by this project.

1.1 Reinforcement Learning

As explained by [3], reinforcement learning is the study of planning and learning in a scenario where a learner (or agent) actively interacts with the environment to achieve a particular goal. The achievement of the agent's goal is typically measured by the reward he receives from the environment and which he seeks to maximize.

[1] state that the most significant difference between reinforcement learning and supervised learning is that there is no presentation of input/output pairs. Instead, they explained that after choosing an action, the agent is told the immediate reward and the following state, but is not told which action would have been in its best long-term interests. It is necessary for the agent to gather useful experience about the possible system states, actions, transitions and rewards actively to act optimally.

Defining a general formulation of the problem based on a Markov Decision Process (MDP), as proposed by [2], the agent can perceive a set S os distinct states of its environment and has a set A of actions that it can perform. So, at each discrete time step t, the agent senses the current state s_t and choose to take an action a_t. The environment responds by giving the agent a reward $r_t = r(s_t, a_t)$ and by producing the succeeding state $s_{t+1} = \delta(s_t, a_t)$. The functions r and δ only depend on the current state and action (it is memoryless[2]), are part of the environment and are not necessarily known to the agent.

[1] Source: `https://en.wikipedia.org/wiki/Reinforcement_learning`
[2] Source: `https://en.wikipedia.org/wiki/Markov_process`

The task of the agent is to learn a policy π that maps each state to an action ($\pi : S \rightarrow A$), selecting its next action a_t based solely on the current observed state s_t, that is $\pi(s_t) = a_t$. The optimal policy, or control strategy, is the one that produces the greatest possible cumulative reward over time. So, stating that:

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+1} + ... = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

Where $V^{\pi}(s_t)$ is also called the discounted cumulative reward and it represents the cumulative value achieved by following an policy π from an initial state s_t and $\gamma \in [0, 1]$ is a constant that determines the relative value of delayed versus immediate rewards. If we set $\gamma = 0$, only immediate rewards is considered. As $\gamma \rightarrow 1$, future rewards are given greater emphasis relative to immediate reward. The optimal policy π^* that will maximizes $V^{\pi}(s_t)$ for all states s can be written as:

$$\pi^* = \arg\max_{\pi} V^{\pi}(s) \quad , \forall s$$

As learning $\pi^* : S \rightarrow A$ directly is difficult because the available training data does not provide training examples of the form (s, a), in the section 3 I will implement the Q-learning algorithm for estimating the optimal policy.

1.2 What Will Be Done

The goal of this project[3] is to design the AI driving agent for the smartcab, that operates in an idealized grid-like city.

The smartcab is able to sense whether the traffic light is green for its direction of movement and whether there is a car at the intersection on each of the incoming roadways (and which direction they are trying to go). In addition to this, each trip has an associated timer that counts down every time step. If the timer is at 0 and the destination has not been reached, the trip is over, and a new one may start.

It should receive the inputs mentioned above at each time step t, and generate an output move, that consists on to stay put at the current intersection, move one block forward, one block left, or one block right (no backward movement).

The smartcab also should receive a reward for each successfully completed trip. A trip is considered "successfully completed" if the passenger is dropped off at the desired destination within a pre-specified time bound. It also gets a smaller reward for each correct move executed at an intersection. It gets a minor penalty for a wrong move and a larger penalty for violating traffic rules and/or causing an accident.

Based on the rewards and penalties it gets, the agent should learn an optimal policy for driving on city roads, obeying traffic rules correctly, and trying to reach the destination within a goal time.

[3]Source: `https://goo.gl/BZdyLo`

2 Implement a Basic Driving Agent

In this section, I will implement a basic driving agent that accepts specified inputs ans produces a valid output.

2.1 My Grid-Like World

Let's start by testing the output produced by my current *'world'* that will be used to analyze the results of this project. This *'world'* is a grid-like city, with roads going North-South and East-West. Other vehicles may be present on the streets, but no pedestrians. There is a traffic light at each intersection that can be in one of two states: North-South open or East-West open. US right-of-way rules apply: On a green light, you can turn left only if there is no oncoming traffic at the intersection coming straight. On a red light, you can turn right if there is no oncoming traffic turning left or traffic from the left going straight.

We are told to assume that a higher-level planner assigns a route to the smartcab, splitting it into waypoints at each intersection. The time in this world is quantized and at any instant, the smartcab is at some intersection. Therefore, the next waypoint is always either one block straight ahead, one block left, one block right, one block back or exactly there (reached the destination).Below is a sample of the log file generated by the simulator.

```
<Time>;Environment.reset(): Trial set up with start=(4, 3)
                        ,destination=(8, 4), deadline = 25
<Time>;RoutePlanner.route_to(): destination = (8, 4)
<Time>;LearningAgent.update(): deadline = 25,
    inputs = {'light': 'green', 'oncoming': None,
              'right': 'forward', 'left': None},
    action = None, reward = 0.0
...
<Time>;Environment.step(): Primary agent ran out of time!
Trial aborted.
```

2.2 The Basic Agent

As mentioned before, I am going to implement a Smartcab that processes the following inputs at each time step:

- *Next waypoint location*: about its current position and heading
- *Intersection state*: traffic light and presence of cars
- *Current deadline value*: time steps remaining

For the purposes of this project, the agent should produce just some random move, like *'None'*, *'forward'*, *'left'* and *'right'*. I will not implement the correct strategy because it is precisely what my agent is supposed to learn.

For the purposes of this project, the first implementation of the agent should produce just some random move at each time t, such that $a_t \in (None, forward, left, right)$. I will not implement the correct strategy because it is precisely what my agent is supposed to learn. Below I am going to simulate 100 different

trials with the *'enforce_deadline'* set to False and will save the logs produced by the agent to observe how it performs. In this mode, the agent is given unlimited time to reach the destination.

```
Number of Trials: 100
Times that the agent reached the target location: 69
Times the agent reached the hard deadline: 31
Times the agent successfully reached the target: 21
```

As can be seen above, even using just random moves, the agent still was able to reach the target destination roughly 20% of the trials at the deadline stipulated by the Planner. If we considering all the times it reached the location, it was able to complete the route almost 70% of the trials. Now, I am going to produce same basic statistics about the times that it reached the target location.

	Number Of Steps
mean	59.00
std	36.06
min	2.00
25%	23.75
50%	60.00
75%	91.00
max	130.00

Table 1: Basic Agent Simulation Statistics

According to the table 1, It took 59 steps on average, and the variation was huge. The agent has taken from 2 steps to 130 to finish the route. In the section 4, I will try to improve that.

2.3 Identify and Update State

To complete the implementation of my Basic Agent, I still need to decide how it will represent its state internally. Considering the inputs that it receives before performs any action and, consequently, receives a reward, I initially will use a tuple using all of them to represent the current state of my agent, in the form (*inputs*, *next_waypoint*, *deadline*). I believe that it will represent a reasonable set of states that my agent could use to gather more information from the environment.

Inputs is a dictionary with the state of the traffic light for the agents' direction of movement and the direction of the random agents on each of the incoming roadways (if any). *next_waypoint* is the direction of the target location and *deadline*, a timer.

I believe that *next_waypoint* and *inputs* are natural choices, given that the first gives the direction of the target and the second provides information about the environment state. The inclusion of *deadline* might help the agent choose to explore the environment when it has more time available, for example. Later on, we will see how significant this single decision is when dealing with Reinforcement Learning problems.

So, let's count the number of stated that I get when I enforce deadline in my simulation on 100 trials.

```
number of states in the state space: 891
```

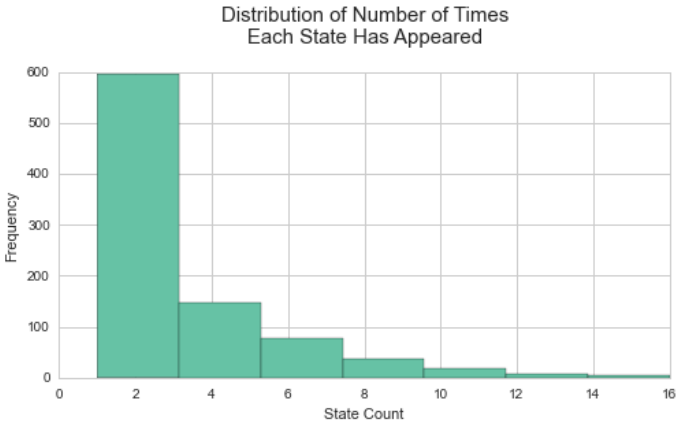

Figure 1: Count Of Times That Each State Appeared

The figure 1 presents the frequency of the counting of each state. For instance, around 600 states appeared just two times, more than 100 appeared four times and so on. Curiously the agent's behavior distribution looks like a log-normal distribution.

3 Implement Q-Learning

In this section, I will briefly explained the concept of Q-learning and implement an algorithm to learn the Q function.

3.1 The Q-Function

As mention before, to learn an optimal policy π^*, I can't learn a function $\pi^* : S \rightarrow A$ that maps a state to the optimal action directly. There is no such information upfront to be used as training data. Instead, as [2] explained, the only available information is the sequence of immediate rewards $r(s_i, a_i)$ for $i = 1, 2, 3, ...$

So, as we are trying to maximize the cumulative rewards $V^*(s_t)$ for all states s, the agent should prefer s_1 over s_2 wherever $V^*(s_1) > V^*(s_2)$. Given that the agent must choose among actions and not states, and it isn't able to perfectly predict the immediate reward and immediate successor for every possible state-action transition, we also must learn V^* indirectly.

To solve that, we define a function $Q(s, a)$ such that its value is the maximum discounted cumulative reward that can be achieved starting from state s and applying action a as the first action. So, we can write:

$$Q(s, a) = r(s, a) + \gamma V^*(\delta(s, a))$$

As $\delta(s, a)$ is the state resulting from applying action a to state s (the successor) chosen by following the optimal policy, V^* is the cumulative value of the

immediate successor state discounted by a factor γ. Thus, what we are trying to achieve is

$$\pi^*(s) = \arg\max_a Q(s, a)$$

Thus, the optimal policy can be obtained even if the agent just uses the current action a and state s and chooses the action that maximizes $Q(s, a)$. Also, it is important to notice that the function above implies that the agent can select optimal actions even when it has no knowledge of the functions r and δ. In the next subsection, we will see how [2] defined a reliable way to estimate training values for Q, given only a sequence of immediate rewards r.

3.2 Learning Q

As we have seen, learning the Q function corresponds to learning the optimal policy. According to [3], the optimal state-action value function Q^* is defined for all $(s, a) \in S \times A$ as the expected return for taking the action $a \in A$ at the state $s \in S$, following the optimal policy. So, it can be written as [2] suggested:

$$V^*(s) = \arg\max_{a'} Q(s, a')$$

Using this relationship, we can write a recursive definition of Q function, such that:

$$Q(s, a) = r(s, a) + \gamma \max_{a'} Q(\delta(s, a), a')$$

The recursive nature of the function above implies that our agent doesn't know the actual Q function. It just can estimate Q, that we will refer as $\hat{Q}$. It will represents is hypothesis $\hat{Q}$ as a large table that attributes each pair (s, a) to a value for $\hat{Q}(s, a)$ - the current hypothesis about the actual but unknown value $Q(s, a)$. I will initialize this table with zeros, but it could be filled with random numbers, according to [2]. Still according to him, the agent repeatedly should do the following:

Algorithm 1 Update Q-table

1: **loop** Observe the current state s and:
2: Choose some action a and execute it
3: Receive the immediate reward $r = r(s, a)$
4: initialize the table entry $\hat{Q}(s, a)$ to zero if there is no entry (s, a)
5: Observe the new state $s' = \delta(s, a)$
6: Updates the table entry for $\hat{Q}(s, a)$ following:
$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$
7: $s \leftarrow s'$

Note this training rule is suited to a deterministic Markov decision process (where $r(s, a)$ and $\delta(s, a)$ are deterministic) and uses the agent's current $\hat{Q}$ values for the new state s' to refine its estimate of $\hat{Q}(s, a)$ for the previous state s. It is tricky. Let's take a look at its performance.

```
Number of Trials: 100
Times that the agent reached the target location: 55
Times the agent reached the hard deadline: 45
Times the agent successfully reached the target: 55
```

The agent is reaching its destination much more frequently than before. When the agent selected its action randomly, it reached its destination just 21% of the times. Now, it is reaching 55%. Now the agent is always taking the "safest action": takes the action with the biggest reward. It is not bad, but looking at the number of explored actions by state in the Q-table produced by this simulation, we can notice something curious

Number of Explored Actions	
1	56.28%
2	21.10%
3	15.83%
4	6.78%

Table 2: Explored Actions in The Q-Table

The agent explored just one possible action in 56% of the states in the Q-Table. According to [2], there are some conditions to ensure that the algorithm converges toward a $\hat{Q}$ equal to the true Q function. One of them is that the system should be a deterministic MDP. Another is that the agent must select actions in a way that it visits every possible state-action pair infinitely often. Looking at the number of times this implementation of the agent explored all 4 possibles actions (less than 7% of the states), I would say that it isn't true. In the next section, I will try to improve te agent's performance.

4 Enhance the Driving Agent

In this section, I will try to improve the driving agent to make it consistently reaches the destination within allotted time and the net reward positive.

4.1 Experimentation Strategies

One of the issues of the current strategy is that the agent could over-commit to actions that presented positive $\hat{Q}$ values early in the simulation, failing to explore other actions that could present even higher values. [2] proposed to use a probabilistic approach to select actions, assigning higher probabilities to action with high $\hat{Q}$ values, but given to every action at least a nonzero probability. So, I will implement the following relation:

$$P(a_i \mid s) = \frac{k^{\hat{Q}(s,a_i)}}{\sum_j k^{\hat{Q}(s,a_j)}}$$

Where $P(a_i \mid s)$ is the probability of selecting the action a_i given the state s. The constant k is positive and determines how strongly the selection favors action with high $\hat{Q}$ values. In the figure 2 we can see the number of times the algorithm choose to explore when we vary the value of the k.

Large values of k assigned a higher probability of the agent exploits what it already has learned. Small values have made the agent favor the exploration of new actions. In the figure 3 we can check how the different values of k impacted on the times the agent completed the routes.

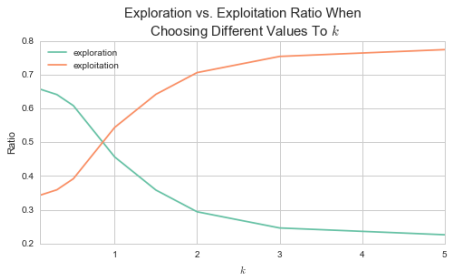

Figure 2: Exploration vs Exploitation

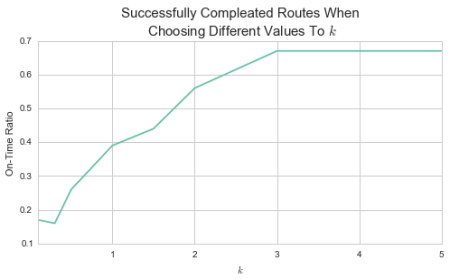

Figure 3: On Time Ratio

The agent presented a very steep improvement as k was approaching 3. Above this value, the completed trips ratio just showed a slight growth. Now, let's check how the number of trials changed the average reward received by the agent.

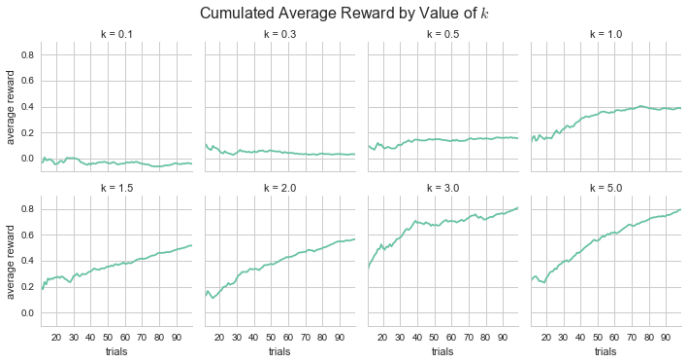

Figure 4: Average Cumulative Reward By Value of k

Clearly, small values of k are worst than large values, as can be seen at the figure 4. As I want that my agent has some probability of exploring new actions, and it seems that $k = 3$ is the best choice (the "slope" is not so steep, and the start value is pretty high), I will use this parameter to perform the next simulations.

4.2 Non-deterministic Rewards and Actions

The last modification that I want to introduce is when the environment is non-deterministic. As I said before, the algorithm used is appropriate to the deterministic case. On a nondeterministic environment, the reward function $r(s, a)$ and action transition function $\delta(s, a)$ may have probabilistic outcomes. As there are other agents in the Smartcab world, I believe that it is a proper assumption.

As explained by [2], to handle nondeterministic MDPs, it is needed to redefine the value V^* to be expressed as a expected value. So, jumping to the Q function, we can re-express Q recursively as:

$$\begin{aligned} Q(s,\, a) &= E\left[r(s,a)\right] + E\left[\gamma \max_{a'} Q(\delta(s,\, a),\, a')\right] \\ &= E\left[r(s,a)\right] + \gamma\, E\left[\max_{a'} Q(\delta(s,\, a),\, a')\right] \\ &= E\left[r(s,a)\right] + \gamma \sum_{s'} P\left(s' \mid s, a\right) \max_{a'} Q(\delta(s,\, a),\, a') \end{aligned} \tag{1}$$

Where $P\left(s' \mid s, a\right)$ is the probability that taking action a in state s will produce the next state s'. To implement that, we need to assume that the $\hat{Q}$ values in our table may be wrong, due to the non-deterministic behavior of our environment. So, [2] suggested to modifying the training rule by introducing a decaying weighted average factor to the current $\hat{Q}$ value and to the revised estimate, such that:

$$\hat{Q}(s,\, a) \leftarrow (1 - \alpha_n)\, \hat{Q}_{n-1}(s,\, a) + \alpha_n \left[r + \gamma \max_{a'} \hat{Q}_{n-1}(s',\, a')\right]$$

Where $\alpha_n = \frac{1}{1 + visits_n(s,\, a)}$ is the decaying factor and $visits_n$ is the total number of times this state-action $(s,\, a)$ pair has been visited. So, let's put it in practice.

```
Number of Trials: 100
Times that the agent reached the target location: 58
Times the agent reached the hard deadline: 42
Times the agent successfully reached the target: 58
```

Looking just at the times the agent has completed the route, it doesn't look like we made much progress. I will compare the performance of all the agents created so far in the last section.

4.3 The last parameter to Tune: Gamma

There is just one remain parameter of the model that was not explored yet: the γ. As already explained, the γ in the $\hat{Q}$ function is a factor used to discount the immediate successor state to be used to update $\hat{Q}$ table. As γ approaches 0, less "memory" the agent has. So, let's check what happens when we vary this parameter about the average cumulative reward of the learning agent:

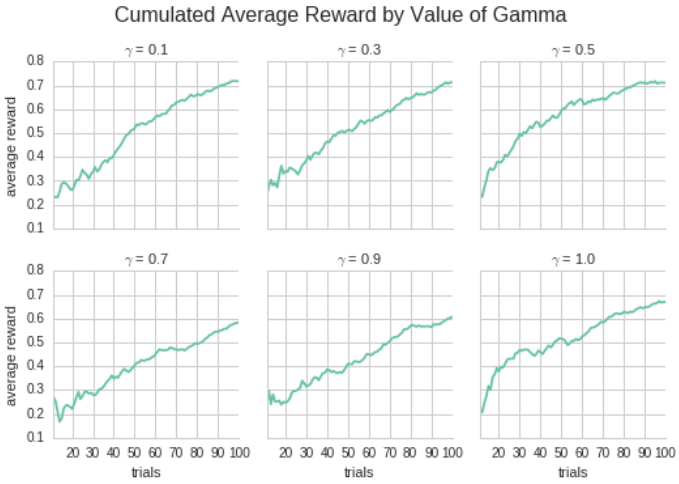

Figure 5: Average Cumulative Reward By Value of γ

In the last section, I will use $\gamma = 0.5$ to perform the final tests.

5 Conclusion

In this section, I will compare the performance of the algorithms implemented so far and then I will show how a little change can make a huge difference in a reinforcement learning problem.

5.1 Final Remarks

The goal of the whole project was to build an agent that obeys traffic rules correctly and reachs the destination within an allotted time. So, the "perfect" agent would be the one that always come to the destination in the goal time without any penalty.

However, as our environment is non-deterministic, to always reach the destination on time is not reasonable. Also, as the environment penalizes the agent even when it makes a correct move, but in a distinct direction from the target location, it isn't fair to expect that the perfect agent never incurs any penalty.

So, the best one would be the one that learns a reasonable policy fast and receives the maximum average cumulative reward possible. Average because we should weight the rewards received by the number of steps used to avoid discrepancies. In this sense, let's first compare how the modifications made on this project reflected on our agent when we ensure deadline. The improved agent will be set with $\gamma = 0.5$ and $k = 3$.

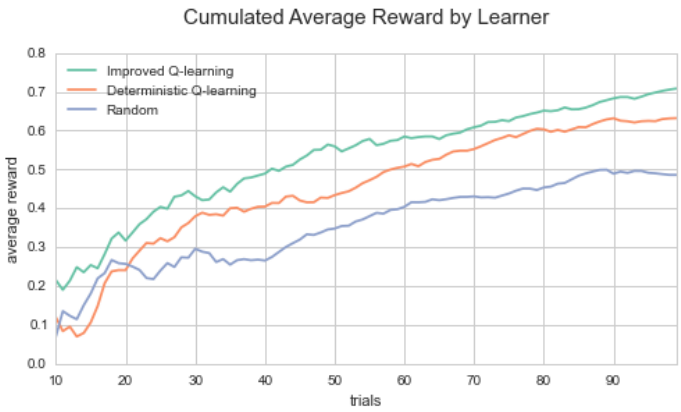

Figure 6: Cumulative Reward By Agent

As can be seen above, the improved agent did better than the other implementations. It reached the destination 64% of the trials on time and the final average reward by action was greater than the others. It was achieved by tuning the parameters, including an experimentation strategy and changing the $\hat{Q}$ function to deal with the non-deterministic environment, as explained before. The random agent reached the destination on time just 42% of the trials and the agent that used the deterministic version of $\hat{Q}$ function, 55%. Although the final agent is still incurring penalties, it is closer to a perfect agent than it was when using just random moves. In the next subsection, I will show how to change the behavior of the agent radically.

5.2 Blowing Up Everything: Changing The Intern State

As pointed out by the Udacity Reviewer, something that I should have taken into account was the size of state space resulted by my choice of agents' intern state representation. I included an Experimentation Strategy and addressed a non-deterministic MDP environment exactly due to the number of states that the agent would need to learn. It turned out being the right choice, as demonstrated in the last subsection.

However, I believe that I underestimated this single decision. My current state is represented by the tuple (*inputs*, *next_waypoint*, *deadline*). The *deadline* is assigned by the Environment in a non-deterministic way, what makes my state space unlikely to learn in 100 trials. In the figure 7 is possible to see what happens If I discard this single variable in the performance of all my *LearningAgent*.

Using this inter-state representation, the final version of the agent is much closer to a perfect agent. It reached the destination at 90% of the trials and learned most of what need to learn in less than 60 trials.

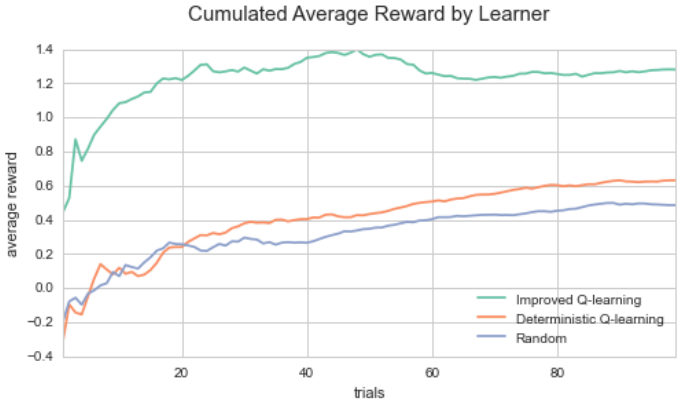

Figure 7: Cumulative Reward By Agent

6 Reflection

Something really valuable that I learned from reviewers is the importance of the state space representation and how it affects the performance of the learning agent.

Still, it's hard to say how better would be the performance of a perfect agent to contrast to the current implementation, given the stochastic environment. An option would be to implement a learning agent that already knows the traffic rules and just need to learn how to find his way to the destination on time.

References

[1] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[2] T.M. Mitchell. *Machine Learning*. McGraw-Hill International Editions. McGraw-Hill, 1997.

[3] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012.