Smartcab project

Implement a basic driving agent

Mention what you see in the agent's behavior. Does it eventually make it to the target location?

In this first part, I saw the agent is moving randomly, not following any policy. **The actions were executed with no concern about the reward to obtain**. The agent was learning nothing. For this reason, the agent got a lot of penalties in every trial, it was not doing any better.

Eventually, the agent reached the destination, in most of the trials. This is because of the lack of time limit (there is a hard time limit from the environment to avoid the simulation run indefinitely).

Identify and update state

Justify why you picked these set of states, and how they model the agent and its environment.

In order to select the states, first I analyzed the data available from the environment and the rules to drive.

These are the variables in the environment and their possible values:

- Next waypoint: Left, Right, Forward
- Traffic light: Green, Red
- Oncoming (Front) car: Left, Right, Forward, None
- Left car: Left, Right, Forward, None
- Right car: Left, Right, Forward, None
- Time steps remaining: [Start value – hard time limit | 0]

First, we can get rid of **Time steps** remaining because the range of values for this variable is high which creates a lot of useless states.

Next, based on the rules for driving, there is no rule applied to turn right when there is a car in the right side, therefore I put **Right car** aside.

This gives me four variables to analyze and found the possible states. The easiest way to do so is create a table of all possible combinations.

States	Next way point	Traffic light	Front car	Left car
s0	Left	Green	Left	Forward
s1	Left	Green	Left	Right | Left | None
s2	Left	Green	Forward	Forward
s3	Left	Green	Forward	Right | Left | None
s4	Left	Green	Right | None	Forward
s5	Left	Green	Right | None	Right | Left | None
s6	Left	Red	Left	Forward
s7	Left	Red	Left	Right | Left | None
s8	Left	Red	Forward	Forward
s9	Left	Red	Forward	Right | Left | None
s10	Left	Red	Right | None	Forward
s11	Left	Red	Right | None	Right | Left | None
s12	Right	Green	Left	Forward
s13	Right	Green	Left	Right | Left | None
s14	Right	Green	Forward	Forward
s15	Right	Green	Forward	Right | Left | None
s16	Right	Green	Right | None	Forward
s17	Right	Green	Right | None	Right | Left | None
s18	Right	Red	Left	Forward
s19	Right	Red	Left	Right | Left | None
s20	Right	Red	Forward	Forward
s21	Right	Red	Forward	Right | Left | None
s22	Right	Red	Right | None	Forward
s23	Right	Red	Right | None	Right | Left | None
s24	Forward	Green	Left	Forward
s25	Forward	Green	Left	Right | Left | None
s26	Forward	Green	Forward	Forward
s27	Forward	Green	Forward	Right | Left | None
s28	Forward	Green	Right | None	Forward
s29	Forward	Green	Right | None	Right | Left | None
s30	Forward	Red	Left	Forward
s31	Forward	Red	Left	Right | Left | None
s32	Forward	Red	Forward	Forward
s33	Forward	Red	Forward	Right | Left | None
s34	Forward	Red	Right | None	Forward
s35	Forward	Red	Right | None	Right | Left | None

As shown in the table, all possible combinations give 36 states. This table is already reduced otherwise, it would have have 96 states.

There are four possible values for Front car and Left car (Left, Right, Forward, None), but the table shows only three values (Left, Forward, Right | None) for the former one, and just two values for the latter (Forward, Right | Left | None). In both cases this comes from the driving rules. In the first situation, we only care about forward and left directions because those are the ones used for our agent to turn left or right respectively. The absence of a car (None) or a right direction can be treated as one value. The same is true for left car, we just care about its forward direction to turn right.

We can reduce this table even more by analyzing the 36 possible states and merge some of them.

States	Next way point	Traffic light	Front car	Left car	Allows to go to next point
s0	Left	Green	Forward	Right | Left | Forward | None	NO
s1	Left	Green	Right | Left | None	Right | Left | Forward | None	YES
s2	Left	Red	Right | Left | Forward | None	Right | Left | Forward | None	NO
s3	Right	Green	Right | Left | Forward | None	Right | Left | Forward | None	YES
s4	Right	Red	Right | Left | Forward | None	Forward	NO
s5	Right	Red	Left	Right | Left | Forward | None	NO
s6	Right	Red	Right | Forward | None	Right | Left | None	YES
s7	Forward	Green	Right | Left | Forward | None	Right | Left | Forward | None	YES
s8	Forward	Red	Right | Left | Forward | None	Right | Left | Forward | None	NO

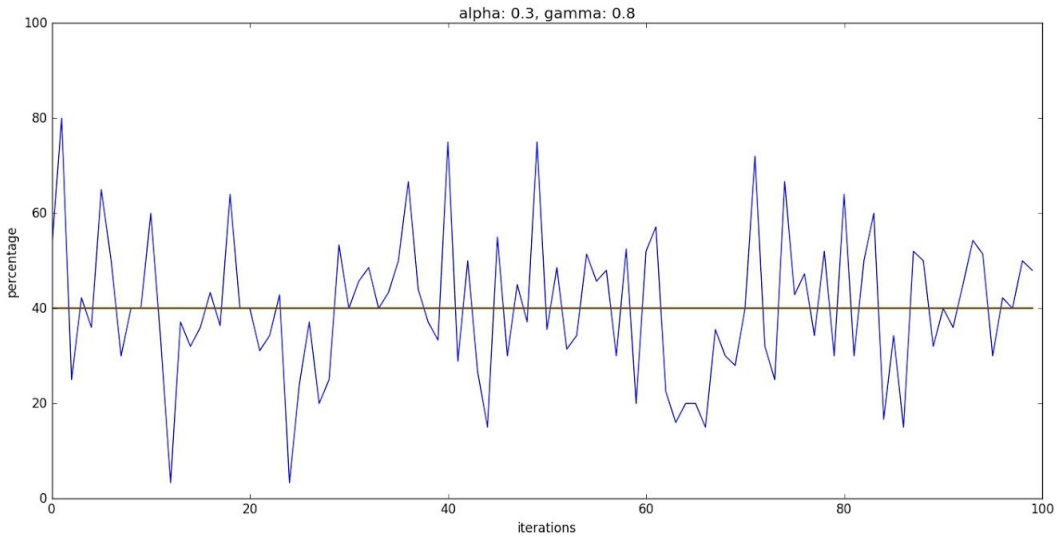
The table is self explanatory, it basically says there are two kind of states: when the agent can move to the next way point, and when it can not.

Implement Q-Learning

What changes do you notice in the agent's behavior?

After running the simulation several times, the first thing I noticed is the agent was able to reach the destination in every trial in the majority of the simulations. Moreover, it was able to do so before the time limit. There are very few cases where it gets to the destination out of time or the trial is aborted. So, I can say confidently enough that the agent is learning.

I have to mention that one of my expectations was that the agent would improve the time (number of steps) to reach the destination. However, after further analysis I realized this is unlikely to happen because the agent is not set to reach the destination based on the time limit. Moreover, time (number of steps) is not a variable used to define the states of the agent. The next graphic shows this situation



The image above depicts the variation of the percentage of steps used to reach the destination. The percentage is the relation between the number of steps needed to get to the destination and the maximum number of steps to do so. It is clear that there's no tendency to increase or decrease this value (according to my expectation, it should decrease)

Enhance the driving agent

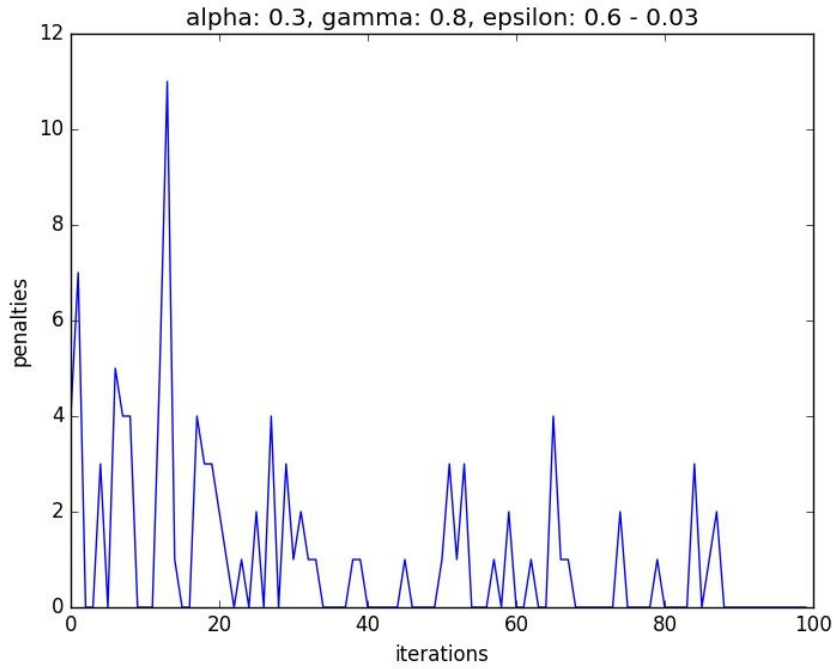
Report what changes you made to your basic implementation of Q-Learning to achieve the final version of the agent. How well does it perform?

I implemented two changes:

1. Exploration
2. Initialize Q

Exploration

I used the technique of epsilon decay, the next image shows the number of penalties the agent gets in every iteration:



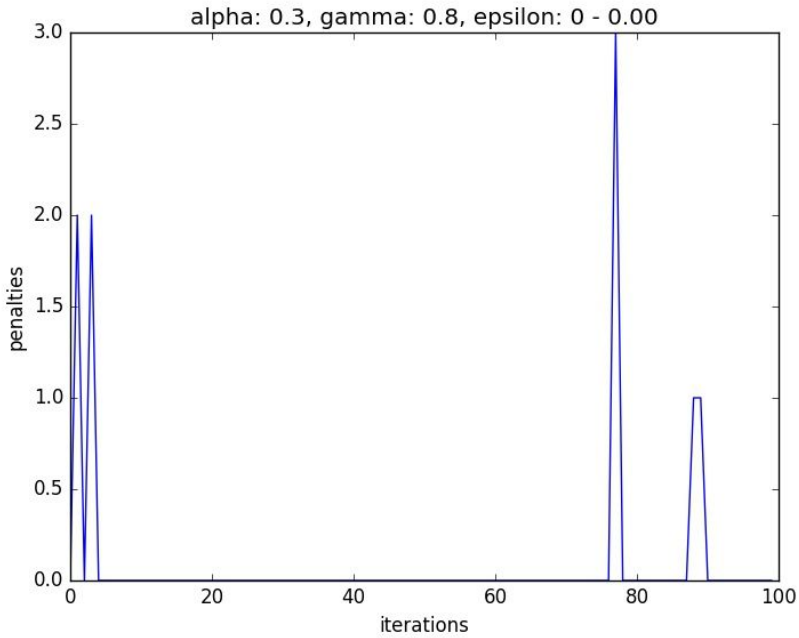
It is clear that the number of penalties reduced with the number of trials reaching a point where no penalties are gotten to reach the destination

Initialize Q

I set the values of Q table at beginning. There are nine steps, five of them does not allow to go to next waypoint, four of them do. I set Q table so when the agent is in a state where it can't move to next waypoint, it gets a penalty; whereas in a state in which it can move to next waypoint, it gets a reward. The initial Q table is shown next:

	None	Forward	Left	Right
s0	0	0	-1	0
s1	0	0	1	0
s2	0	0	-1	0
s3	0	0	0	1
s4	0	0	0	-1
s5	0	0	0	-1
s6	0	0	0	1
s7	0	1	0	0
s8	0	-1	0	0

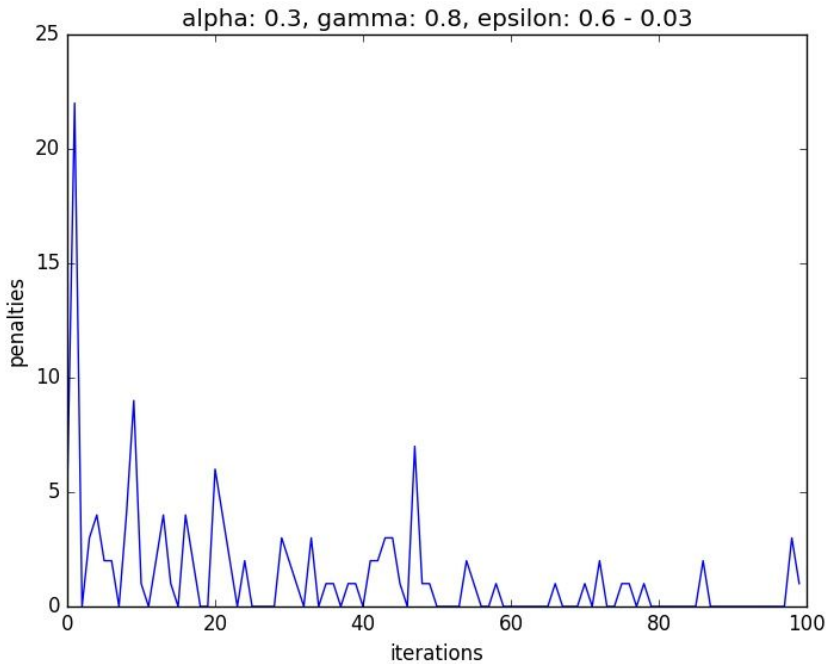
Using that Q table, we get the next images for the number of penalties.



Here, the agent got no penalties in the majority of the trials. This is far from being good, because the agent is learning nothing since it always selects the right action. I can say that this is a planning situation with no benefits at all.

Does your agent get close to finding an optimal policy, i.e. reach the destination in the minimum possible time, and not incur any penalties?

Based on the previous approaches, I decided to use both options at the same time, getting the next results.



This approach gets the best of both worlds, where there is a plan for the agent to follow, but also an exploration situation that helps it to learn.

Finally, the next image shows the amount of penalties where there is no learning process, and the agent selects the next action randomly. It is clear how different the agent behaves.