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.

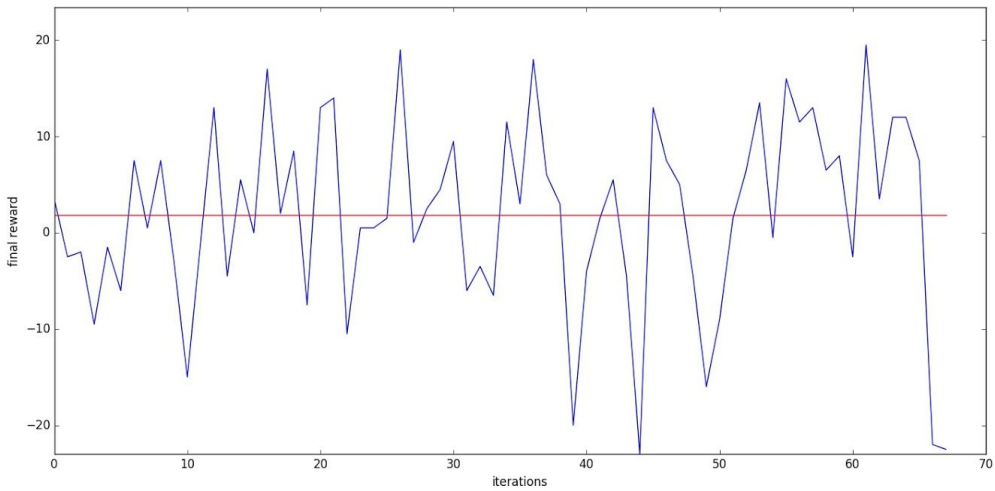
States	Next way point	Traffic light	Front car	Left car	Allows to go to next point
s0	Left	Green	Forward	Right | Left | Forward | None	NO
s1	Left	Green	Right | Left | None	Right | Left | Forward | None	YES
s2	Left	Red	Right | Left | Forward | None	Right | Left | Forward | None	NO
s3	Right	Green	Right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

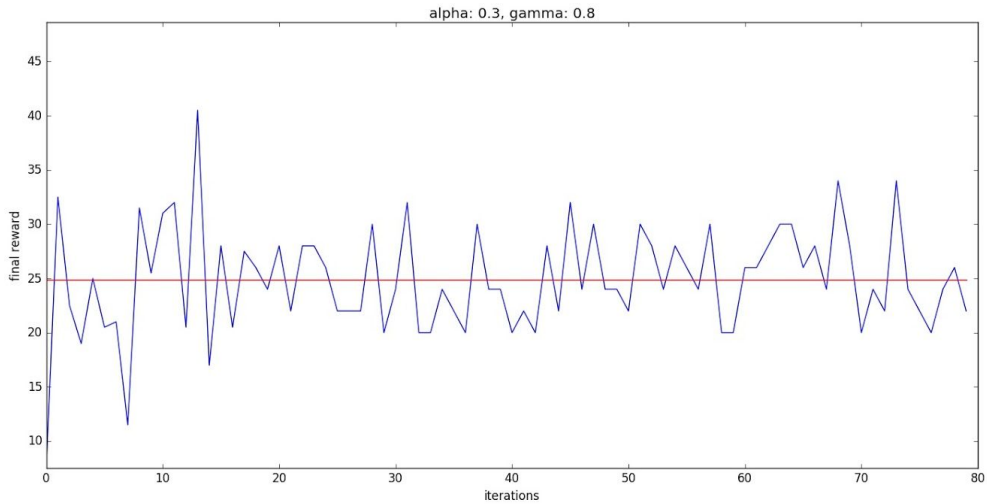
The table is self explanatory, it basically says there are two kind of states: when the agent can move to the next way point, and when it can not.

Implement Q-Learning

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

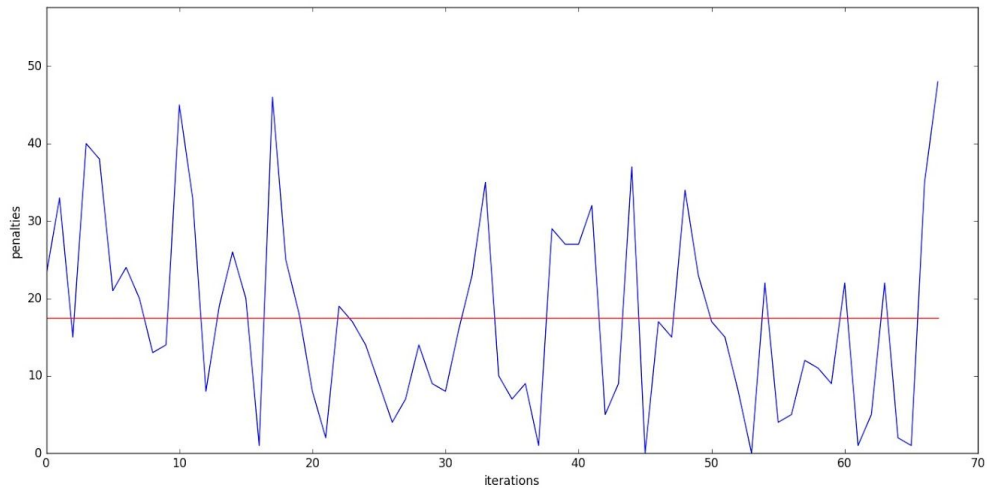
After running the simulation several times, the first thing I noticed is the agent was able to reach the destination in almost every trial. The success percentage is noticeable higher (~97%) compared to the no learning process (~68%). Moreover, it was able to reach the destination before the time limit and with a positive reward. The next images show this situation.

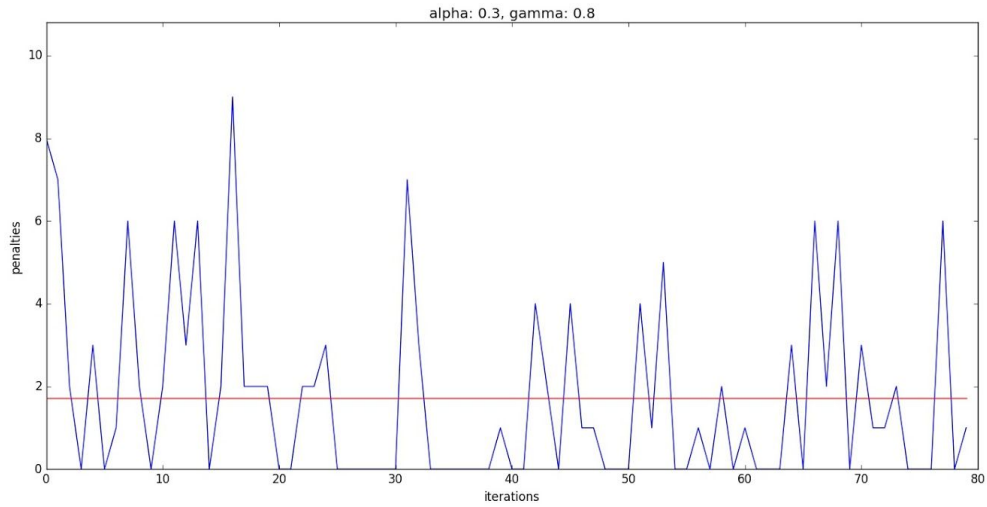




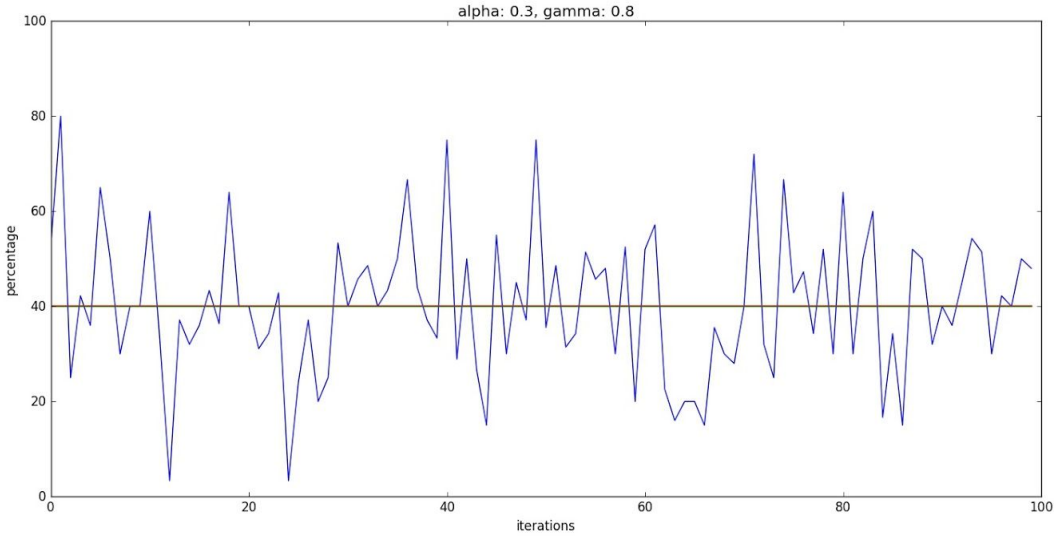
The images show the final reward when reaching the destination. The first one was obtained when the agent selected actions randomly; here it is clear that in many occasions the reward was negative. The second image was obtained when the Q-Learning process was implemented; it is noticeable how the agent reached the destination with positive rewards in every trial; the average reward is also much higher (~25) than the former one (~2).

A similar situation happened with the number of penalties, but in the opposite way. In the first case, the average number of penalties was higher (~18) than in the second one (~2). This can be seen clearer in the next images.





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?

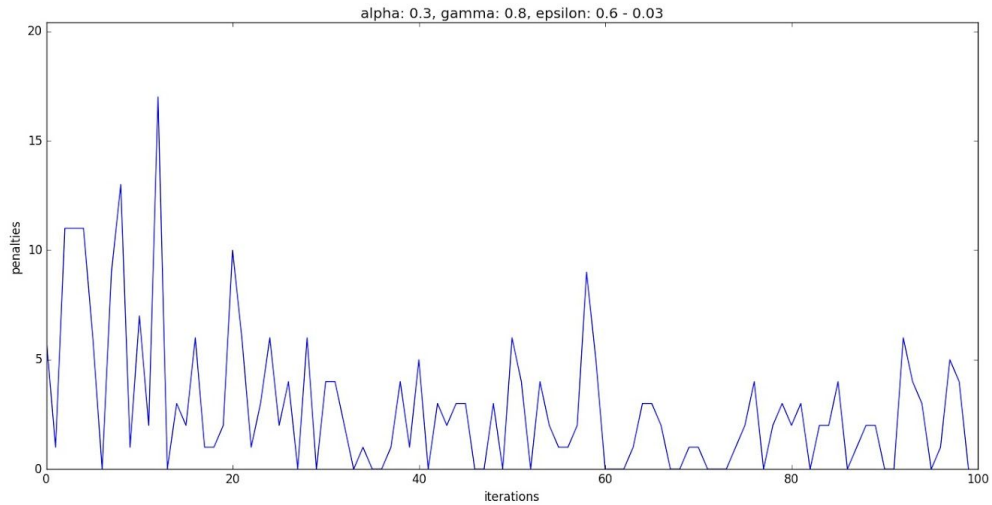
I implemented two changes:

1. Exploration
2. Initialize Q

Exploration

Exploration is accomplished by adding a variable called epsilon. This variable allows exploration and varies from 0 to 1. A value of zero, means that no exploration is done by the agent, it decides the next action based on previous experience. In the other hand, a value of one means the chooses the next action randomly.

First, 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.

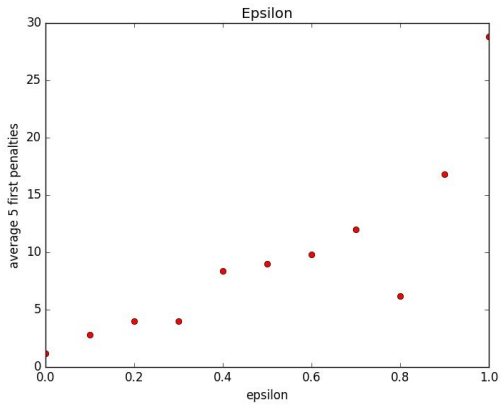
In order to see how the agent behaves for different values of epsilon, I ran the simulation several times keeping the value of epsilon constant, I got the next results.

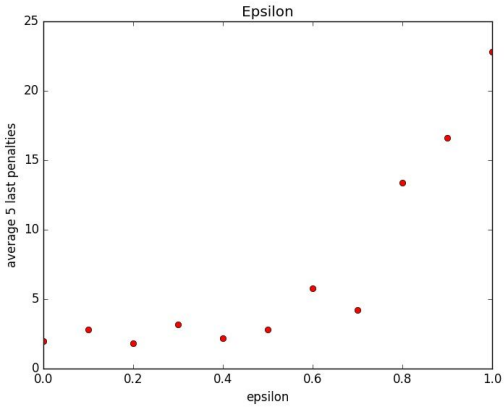
epsilonStart	epsilonEnd	average 5 first penalties	average 5 last penalties	average 5 first rewards	average 5 last rewards
0	0	1.2	2	28.5	26
0.1	0.1	2.8	2.8	25.7	30.2
0.2	0.2	4	1.8	22.8	22.9
0.3	0.3	4	3.2	22.6	25.9
0.4	0.4	8.4	2.2	16.7	22.8
0.5	0.5	9	2.8	17.7	18.7
0.6	0.6	9.8	5.8	14.1	17.6
0.7	0.7	12	4.2	18.6	21.7
0.8	0.8	6.2	13.4	14.3	16.5
0.9	0.9	16.8	16.6	6.4	7.9
1	1	28.8	22.8	-2.3	5.3

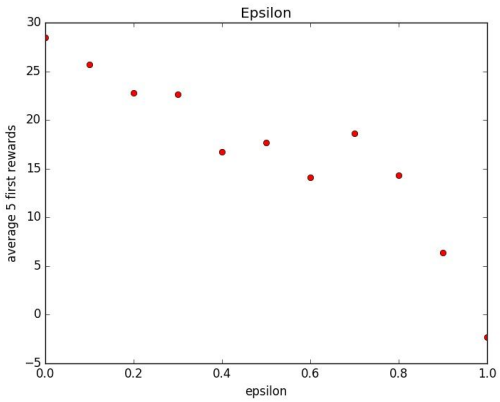
There are clear relationships between epsilon and penalties. Average penalties at start and at the end increase along with the value of epsilon. This happens because the lower the value of epsilon the lower the exploration the agent does, therefore it chooses the next action based on previous experience mainly, therefore reducing the amount of incorrect actions taken.

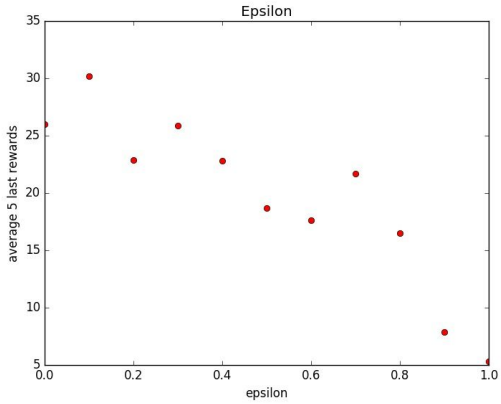
The same happens with rewards, but in the opposite way, they decrease when epsilon increases. This means that, the less the exploration, the higher the reward.

These relationships can be described as f(x) = mx + b, with a positive slope (m) for penalties, and negative slope for rewards. The next graphics best show the relationships between epsilon and the other variables.





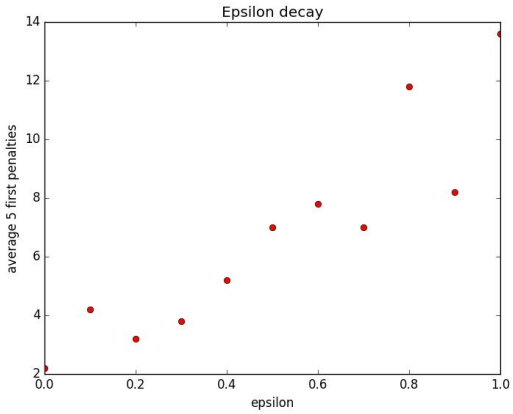


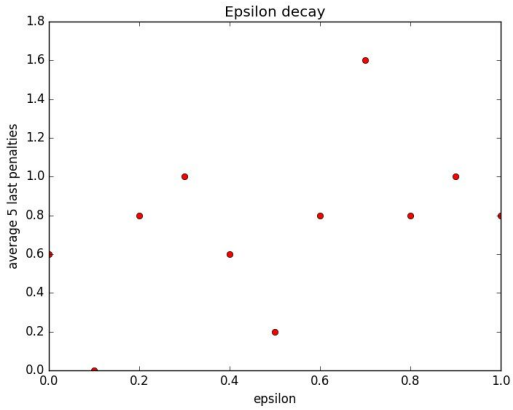


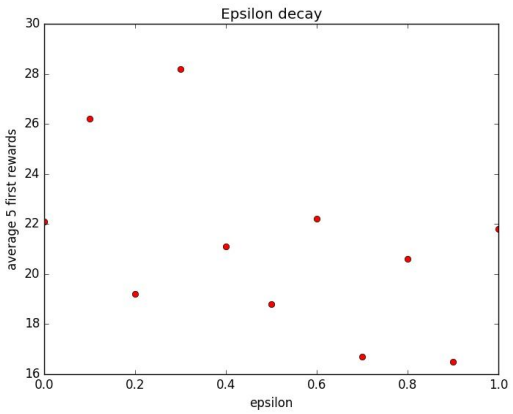
Then, I ran the simulation again but this time decreasing the value of epsilon. I got the next results.

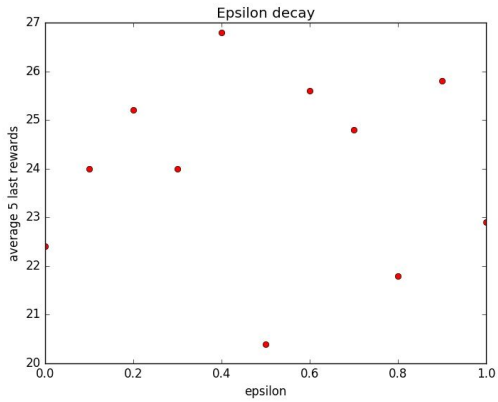
epsilonStart	epsilonEnd	average 5 first penalties	average 5 last penalties	average 5 first rewards	average 5 last rewards
0	0	2.2	0.6	22.1	22.4
0.1	0.0135085172	4.2	0	26.2	24
0.2	0.0177258762	3.2	0.8	19.2	25.2
0.3	0.0295431271	3.8	1	28.2	24
0.4	0.0354517525	5.2	0.6	21.1	26.8
0.5	0.0398832215	7	0.2	18.8	20.4
0.6	0.0478598658	7.8	0.8	22.2	25.6
0.7	0.0452275732	7	1.6	16.7	24.8
0.8	0.0465197896	11.8	0.8	20.6	21.8
0.9	0.0381520424	8.2	1	16.5	25.8
1	0.0523347633	13.6	0.8	21.8	22.9

There’s a clear difference with the previous results. To have a better analysis, let’s first see the corresponding plots.









For the first penalties, the relationship f(x) = mx + b is still valid but with a lower value for slope (m). For the last penalties, the relationship is quite different, with values ranging from 0 to 1.6 and a mean of 0.75, the relationship can be defined as f(x) = b. The same is true for the first and last rewards.

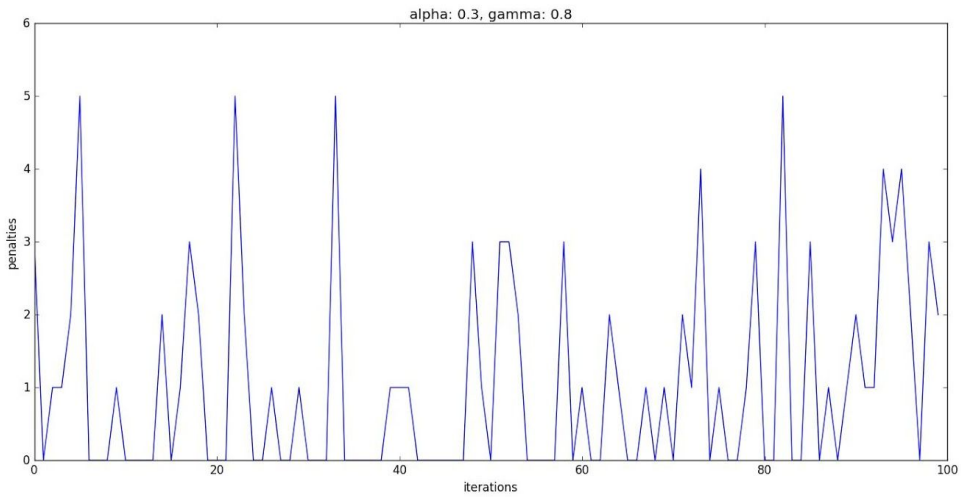
From the previous analysis I can say that exploration with epsilon decay gives better results than just epsilon. In both cases, the number of initial penalties increases along with epsilon. However, by decreasing epsilon during the learning process (that is the more experience the agent gets, the less exploration it does) we get a consistent amount of penalties and rewards at the end of the learning process.

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.

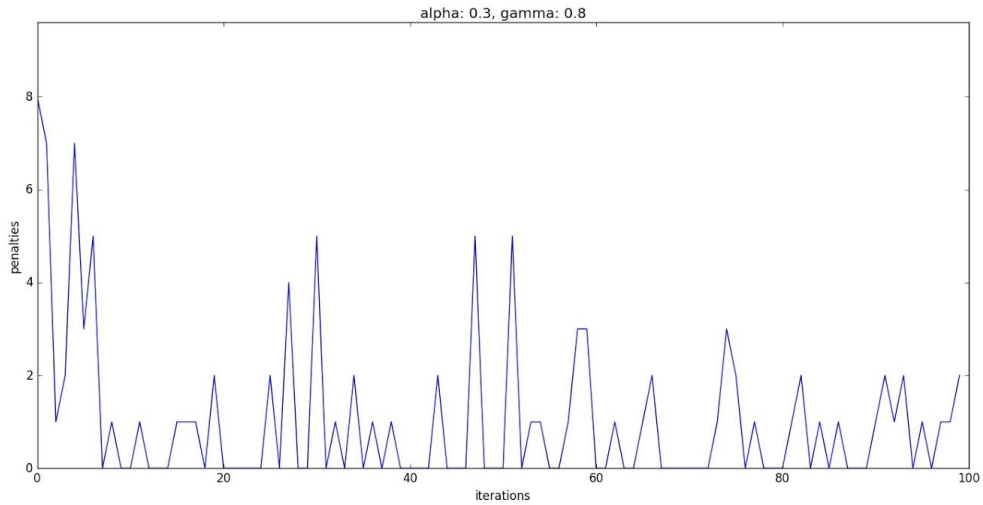


Here, the agent got a consistent number of penalties through every trial. This is far from being good, because the agent is learning nothing since it always selects the right action. I can say that this is a planning situation with no benefits at all.

Next thing I did set all the previous values to one, as 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

This gave the next results for penalties. Unlike the previous case, here the amount of penalties is reduced through the iterations. That means the agent is able to correct the initial error (moving to next waypoint when it shouldn't). It learns much more than the previous situation. I can say that it learns more from error than from success.



I also set the values of Q table to be the opposite to the first situation. So, the agent moves when it shouldn't and it does not move when it should, as shown in the next table

	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

In this case, the agent never reaches the destination. This happens because, the agent eventually gets to the situation where the best action to choose is None, that is stay in the same position forever.

For instance, in s0, the best action is go to next waypoint when it shouldn't, the agent will get a penalty; eventually the value for this action will be reduced through the iterations. Then it can choose Forward, Right or None. For the first two actions it will get a penalty too, so these values will be reduced in every trial too. Finally, the agent will choose None, but this action gives no penalty no reward, so it keeps zero.

In the opposite case, s1 for instance, the best action is not to go to next waypoint when it should. The agent will choose among Forward, Right and None. Similarly as the previous case, it will a penalty for the first two actions. Eventually, values for these actions will be negative, the value for the right action is already negative, and value for None will be zero.

At the end, values for actions Forward, Left and Right will be negative in every state, and values for None will be zero; therefore the best action to choose is None.

A similar situation happens when the values of Q table are set to -1, as 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

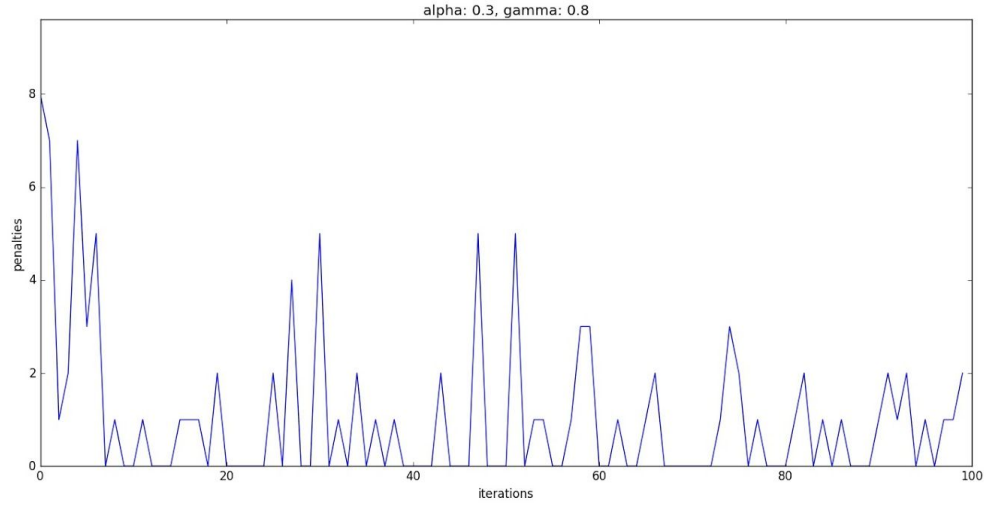
In this case, the agent never reaches the destination either, because the agent will choose the wrong action at every step, getting a Q Table, where all values for the actions will be negative except for None (which will be zero), in every state.

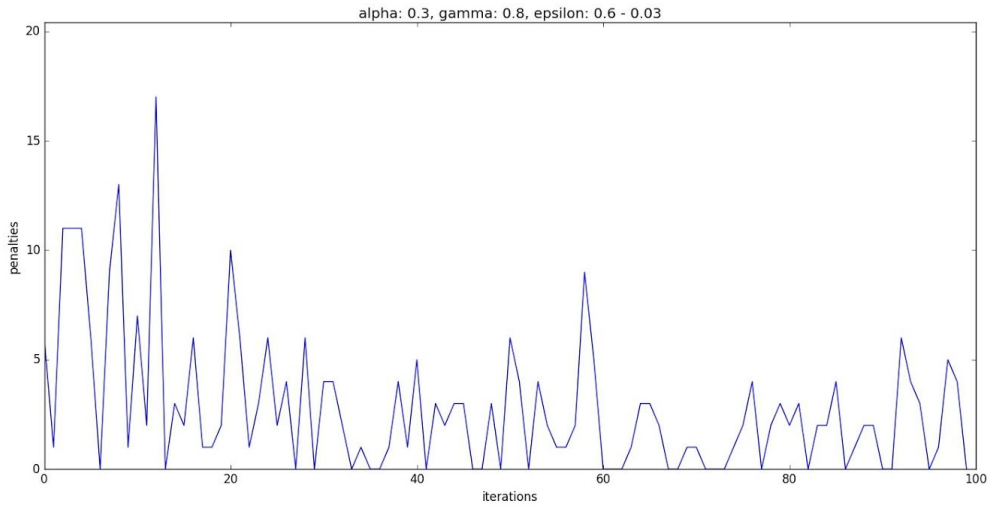
Based on the previous analysis, the best initial values for Q Table are when the agent moves to the next waypoint when it should, and when it shouldn't. Here the agent is able to learn and correct the error and improve their behaviour.

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 results, the best policy for this case is initialize Q Table so that at first the agent moves to the next waypoint when it should and when it shouldn't. In this case, the agent learns much faster because it makes mistakes and corrects them as soon as possible.

Comparing this approach to Epsilon Decay, the difference is quite clear. First, the agent will eventually choose the wrong actions, it will get penalties and correct them too. But it takes more time (trials) to do so. Also, the amount of penalties is higher, reaching a peak of 16, whereas with initial Q Table the maximum value is 8. So, this approach is more efficient because the agent needs to make fewer mistakes to learn the correct behaviour. This is something crucial, because if we were training a real car we would want it to learn as fast as possible with the fewer amount of inconveniences (penalties) due to the limited amount of resources.





Now, after the training, the agent still gets some penalties (1 - 2) before reaching the destination. This happens because the agent prioritize the long-term reward (gamma = 0.8). For instance, the agent learnt that when the next waypoint is ‘forward’ and traffic light is ‘red’, it is better to go ‘right’ than stay there (None), even though this causes a penalty.