Task1:

Implement the basic driving agent. In your report, mention what you see in the agent's behavior. Does it eventually make it to the target location?

I have observed that when the agent produces a random choice, it does eventually reach to the destination. However it may take a short time or a long time. Here are some example runs and how long it took: 118, 43, 87, 20, 28, and 146. Most of the time, the agent would not reach the destination on time, if there was deadline.

Identify a set of states that you think are appropriate for modeling the driving agent. Justify why you picked these set of states, and how they model the agent and its environment.

Next I decided to teach the agent how to observe the traffice rules, and at the same time to follow the next way point. I defined 6 different states based on the combination of traffic lights and the next waypoints. There are 3 choices for next waypoint and 2 choices for the light, therefore there are 6 combinations.

There are actually more inputs; for example incoming traffic. However, I did not include these as part of a state for two reasons; one is that it was extremely rare to see oncoming traffic with 3 other cars. Second reason is that since there were six combinations of traffic, now I had to include 6x6=36 states, and the agent would not reach optimal performance within 100 trials.

Here is my Q table defining the states 0 to 5:

Q table:

States			Actions			
State_Number	State Description		Forward	Right	Left	None
	Light	Next_waypoint				
0	red	forward				
1	red	right				
2	red	left				
3	green	forward				
4	green	right				
5	green	left				

Task 2:

Implement the Q-Learning algorithm by initializing and updating a table/mapping of Q-values at each time step.

I applied basic Q learning algorithm according to formula.

q-new = (1- alpha)*q-old + alpha*[reward + gamma * q-max]

Q values tell us the best action to choose for any particular states. However we start randomly with all Q values equal to 0. And at each step we move alpha away from the old q value, based on what happened taking any action. We also have the parameter gamma. What gamma tells us is that whenever we take an action we should not only consider immediate reward but also the possible future reward (utility) of landing in a state after taking a particular action.

As you can see during my implementation, in the smartcab project we do not know which state we will land after taking a particular action. Therefore I update the Q value not when I am in state S, but I am in the S'. Therefore I update the Q value according to the formula:

New_Q(previous state, previous action) = (1- alpha)*Old_Q(previous state, previous action) + alpha*[previous_reward + gamma *Q_Max(current state]

I have got this idea from the following forum post:

https://discussions.udacity.com/t/next-state-action-pair/44902

Apply the reinforcement learning techniques you have learnt, and tweak the parameters (e.g. learning rate, discount factor, action selection method, etc.), to improve the performance of your agent.

Initialization of Q values:

Initially, I have tried initializing the Q-value with all zeros. However after reviewer's suggestion, I have set the Q value to all 2's. Indeed, I have observed that this results in a much better performance with the agent reaching the destination almost 98 out of 100 times. The reason is that setting Q values close to high reward encourages trying actions that the agent hasn't tried before, and it is much more effective than slow decay of epsilon in terms of exploration.

Alpha: learning rate, i.e. the rate at which Q table is updated:

I varied alpha with fixed epsilon=random action selection, fixed gamma=0.01:
Alpha =0 means no learning. Q table remains 0:

```
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
```

[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]]

Alpha =1 full learning: In this scenario, we have very fast learning but we have no memory of the past. Q table I got was pretty close to optimal Q table, however there were more variations in performance:

[[-0.98989899   0.52020201 -0.97989899   1.01010101]
 [-0.97979788   2.01010102 -0.97989899   1.02020212]
 [-0.97979799   0.52020212 -0.98989799   1.02010101]
 [2.01010101   0.52010101   0.52010101   1.02010101]
 [0.52020212 12.01010101   0.51010101   1.02020212]
 [0.51020101   0.51010101   2.02010101   1.02010101]

Although each action we chose results in the same reward deterministically, setting alpha value = 1 results in more variation in performance than alpha=0.8. I think it is because initial random choices may result in bad states resulting in high reward. i.e. reaching the destination. Therefore alpha=0.8 is the optimal value.

Gamma: Discount factor, i.e. the utility of the future reward in the next state:

I varied gama with fixed epsilon=random action selection, fixed alpha=0.8:

Gamma 3. My Q table was ridiculous with no resemblance to optimal Q table:

[[3.31551343e+236   8.96484569e+239   9.17474972e+239   3.81426810e+239]
 [5.20415482e+242   2.31808577e+240   8.74694151e+241   3.07850807e+245]
 [5.27478209e+242   7.38841937e+245   1.28228794e+243   3.07750559e+243]
 [1.38990098e+241   5.71121054e+240   3.07600460e+245   9.46210330e+239]
 [2.05394856e+244   6.48987575e+242   3.03949466e+242   3.33233817e+245]
 [5.03241988e+242   4.50957622e+236   1.41828155e+241   6.75257533e+237]]

Gamma =0 produced a table very close to optimal(or maybe optimal?)

[[-1.   0.5 -1.   1.]
 [-1.   2. -1.   1.]
 [-1.   0.5 -1.   1.]
 [2.   0.5 0.5 1.]
 [0.5 2.   0.5 1.]
 [0.5 0.5 2.   1.]]

I think in this case gamma should be 0, the reason is that there is very little reason to believe that taking a particular action puts the agent in a more or less advantageous S'. In other words, because we do not have global information about the environment, in the next state we do not know whether we are any closer to the destination. Therefore high gamma messes up the learning.

Epsilon: Action selection method. i.e. exploration exploitation dilemma:

Epsilon tells us whether we select actions randomly or based on the Q table. I chose the epsilon that decays with each time step. As the epsilon decays, so does the probability of random action selection.

I have tried different decay rate for epsilon(experiment was done in triplicates):

epsilon = 1/(self.time).

Q table starts not converging correctly. Reaches destination 60 or15 or 11 times.

epsilon = 1/(self.time**(1/2.)).

Variation in performance. Q table not converging correctly. Reaches destination 94 or 69 or 61 times.

epsilon = 1/(self.time**(1/4.)).

Close to optimal Q table. Reaches destination 93 or 95 or 95 times.

epsilon = 1/(self.time**(1/16.)).

Close to optimal Q table. Reaches destination 70 or 61 or 62 times. Epsilon is not decaying fast enough. Too much random choice initially.

Based on the experiments above, I picked epsilon as epsilon = 1/(self.time**(1/4.)), as it gave the best performance.

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 varied initialization of Q-table, alpha, gamma and epsilon and settled on alpha = 0.8, gamma=0, epsilon = 1/(self.time**(1/4.)). My agent reaches the destination on average 98 out of 100 times.

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?

My agent finds the optimal policy according to reward structure in the environment. If the lights are green it always follows the next waypoint. If the light are red, it takes no action if the next waypoint is forward or left. If the lights are red, and the next way point is right, it turns right. These q values are highlighted in the table below: Here is my Q table after 1000 trials:

States			Actions			
State_Number	State Description		Forward	Right	Left	None
	Light	Next_waypoint				
0	red	forward	-1	0.5	-1	1
1	red	right	-1	2.000001	-1	1
2	red	left	-1	0.5	-1	1
3	green	forward	2	0.5	0.5	1
4	green	right	0.5	2.000512	0.500512	1
5	green	left	0.5	0.5	2	1

As the time progresses the agent acquires less and less negative rewards. However, even though epsilon decays there remains a possibility that the agent takes a random step even towards the later trials. To prevent this problem, I added an additional condition on random action choice. Once trial number is more than 25, we will act only according to the optimal policy.