Implement a basic driving agent

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

The agent (red car) randomly goes forward, left, right or stay put, ignoring traffic light and other agents. The label of next way point given by the planner is shown next to the agent but the actual action taken is not controlled by that. The agent may eventually make it to the target location after a long twisted trip, which by chance reaches target intersection.

Identify and update state

Identify a set of states that you think are appropriate for modeling the driving agent. The main source of state variables are current inputs, but not all of them may be worth representing. Also, you can choose to explicitly define states, or use some combination (vector) of inputs as an implicit state.

At each time step, process the inputs and update the current state. Run it again (and as often as you need) to observe how the reported state changes through the run.

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

I picked these inputs to construct state for primary agent:
`(inputs['light'], next_waypoint)`

Possible combinations of the state are:
`('red', 'forward'), ('red', 'left'), ('red', 'right')`
`('green', 'forward'), ('green', 'left'), ('green', 'right')`

`inputs['light']` contains information about whether the traffic light at intersection is red or green, which is strongly tied to rewards. For example, if it's red and agent takes action of forward, a negative reward will be given. The reward could happen at every time point because every intersection is associated with either green or red light status with 0.5 probability. Hence the agent's state is strongly influenced by this property therefore it's picked.

`next_waypoint` gives heading information to the agent assuming it is in an ideal situation (no traffic light, no other agents, only road and destination). This input greatly helps agent reach destination quickly. It is also available at each time point to the agent. Hence the agent's state is strongly influenced by this property therefore it's picked.

The reason to not picking sensor data of oncoming, left and right agent headings is that there are too few other agents in this grid city (3 of them). There is a very low chance that another agent being at the same intersection as the primary agent is. In another word, at most times sensor data of oncoming, left and right agent would be `None`. After excluding these information from state tuple, it might sometimes gets negative reward when primary agent violates traffic rule of giving ways to the other agent, however, it will have a very small impact on the overall performance due to it being a low probability incident.

Deadline information is only useful when there is a heavy traffic on the road and when deadline is close, agent will learn to take another way to avoid violating too many rules of way giving. It should be used in conjunction with sensor data of oncoming, left and right agent headings. Since neither there is a heavy traffic nor sensor data is used, deadline information is not picked either.

Implement Q-Learning

Implement the Q-Learning algorithm by initializing and updating a table/mapping of Q-values at each time step. Now, instead of randomly selecting an action, pick the best action available from the current state based on Q-values, and return that.

Each action generates a corresponding numeric reward or penalty (which may be zero). Your agent should take this into account when updating Q-values. Run it again, and observe the behavior.

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

The agent starts with some random movements for the first trial. It would violate traffic rule by going forward when it is red light and getting some negative rewards. And it may go away from destination and end up exceeding deadline. This is expected because at the beginning, ε is large (1, 1/2, 1/3, …) which results in high probability for the agent to do exploration by taking random actions. The more exploration is done, the more rules (e.g., crossing red light results in penalty) the agent learns, and the more converging Q values get populated.

As trials go on, the agent is getting less and less negative rewards. This is because as trial number gets larger and larger, ε is decaying and results in larger probability that agent to act based on policy learnt that far. This indicates agent has entered exploitation phase, meaning Q value table is more or less converged and ready for agent to query for optimal actions. Agent is also reaching destination more quickly because Q value table has learnt to teach agent to follow more planner's instructions while still obeying traffic rules.

A few negative rewards are still observed in later trials. This is because gamma (0.8) is too large and future utility of an action is taking too much weight for calculating Q values, which results in agent choosing to violate some rules, willing to get small penalties and receive big reaching-destination bonus reward in the end.

Compare to random action selection in the first question, implementing Q-Learning has significantly improved the agent's performance. By recording success trial rate (count one success trial if agent reaches destination before deadline and achieves positive reward in the end), it is easy to tell how much there is to performance improvement.

Success Trial Rate Comparison

Simulator run (100 trials per run)	All random actions	Q-Learning α=0.5, γ=0.8, decay ε
1	19%	80%
2	24%	86%
3	27%	91%
4	23%	93%
5	28%	93%
6	16%	86%
7	13%	89%
8	19%	85%
9	15%	83%
10	21%	89%

As shown in the comparison table, for each of the 10 simulator runs, Q-Learning achieves much better success trial rate than random action does. Therefore, Q-Learning has helped agent perform much better.

Enhance the driving agent

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. Your goal is to get it to a point so that within 100 trials, the agent is able to learn a feasible policy - i.e. reach the destination within the allotted time, with net reward remaining positive.

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 tweaked parameters by choosing different discount factors (gamma γ) and observed different performance represented by success trial rate, average penalty per trial and average positive reward per trial.

Performance Comparison of Gamma Tuning

Simulator run (100 trials per run)	Q-Learning α=0.5, γ=0.2, decay ε			Q-Learning α=0.5, γ=0.5, decay ε			Q-Learning α=0.5, γ=0.8, decay ε		
	Success Trial Rate	Average Penalty	Average Reward	Success Trial Rate	Average Penalty	Average Reward	Success Trial Rate	Average Penalty	Average Reward
1	96%	-0.47	22.135	93%	-1.635	26.84	83%	-2.84	32.255
2	99%	-0.575	22.395	94%	-2.595	30.775	89%	-3.485	32.07
3	99%	-0.78	22.48	90%	-2.285	30.115	85%	-3.335	33.2
4	98%	-1.025	22.9	96%	-0.98	24.175	95%	-1.99	25.7
5	98%	-0.93	22.26	93%	-2.1	29.765	82%	-5.315	39.29
Average	**98%**	**-0.756**	**22.43**	**93%**	**-1.919**	**28.33**	**87%**	**-3.393**	**32.50**

	Success Trial Rate	Average Penalty	Average Reward
γ=0.2	98%	-0.756	22.43
γ=0.5	93%	-1.919	28.33
γ=0.8	87%	-3.393	32.50

It is shown that when gamma set to 0.2, the agent has the highest success trial rate, lowest penalty and lowest reward.

Gamma determines how much weight the future utilities carry when calculating Q values. The larger the gamma gets, the Q values are more influenced by future utilities. Large gamma tends to tell agent to ignore penalties at moment and aim for bigger rewards in the future, which results in making agent too eager for future rewards. That explains why gamma of 0.8 gives the highest positive reward as well as the highest penalty. However, in this use case, reward is merely a virtual incentive to train agents while minimising penalty is the actual goal since we want to obey traffic rules whenever possible.

Therefore, it can be concluded that gamma of 0.2 gives the best performance.

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?

An optimal policy can guide agents not only to reach destination before deadline with positive rewards, but also to find optimal route based on real-time traffic condition. For example, the optimal policy would guide agent on saving time by following an equal distance route with minimum red lights. This requires information of all related traffic light conditions or at least adjacent junction's traffic light condition. It also teaches agents to obey all traffic rules including give ways to other agents when they should.

The final agent is close to the optimal policy in the sense that reaching destination within desired time limit and maintaining positive rewards under light traffic condition. It is also good at obeying rule of 'red light stop'. With an average of 98% success trial rate, it is considered doing an excellent job given first 1 or 2 trials out of 100 are purposely used for exploration.

However, the final agent lacks of ability to find minimum red light route because there is no traffic light information of other intersections. It is also unable to give ways to other agents due to state tuple excluding sensory data of other agents, which would not perform well if there is heavy traffic on the grid city roads. Without incorporating deadline information into the state tuple, the agent may go circles while following traffic rules in order to gain higher reward.