Shashank Srivastava

1. Implement a basic driving agent
Implement the basic driving agent, which processes the following inputs at each time step:

- Next waypoint location, relative to its current location and heading,
- Intersection state (traffic light and presence of cars), and,
- Current deadline value (time steps remaining)

And produces some random move/action (None, 'forward', 'left', 'right').
Don't try to implement the correct strategy that's exactly what your agent is supposed to learn.

Run this agent within the simulation environment with enforce_deadline set to False (see run function in agent.py), and observe how it performs. In this mode, the agent is given unlimited time to reach the destination. The current state, action taken by your agent and reward/penalty earned are shown in the simulator.

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

The implementation of a basic driving agent is in agent.py. The random agent eventually reaches the target location, but the actions are randomly chosen, which means the agent does not learn from the previous behavior and improve the action. So as I observed, sometimes the agent's action is not optimal, for instance, the agent keep going forward when there is a red light, or remains in the same position even there is no other oncoming cars or red light. So the reward is terrible. To check the success rate of reaching the destination during the deadline, I set update_delay=0.01, enforce_deadline = True and n_trails = 100, then implemented the basic agent 5 times, the average success rate is 19%.

2. 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.

Before implementing Q-learning algorithm, the first task was to identify the possible states, here are some possible options:

Intersection state (traffic light and presence of cars): I chose intersection state as one of state variables because we want to train the cab to perform legal moves and obey the US right-of-way rules.

Next waypoint location: I chose this to model the destination relatives to the current location, we cannot directly choose destination and location as variables since destination changes all the time and location will cause an issue of having large amount of states and taking a long time for q values to converge, so we choose next waypoint location instead.

Deadline: I do not include this variable as state variable here because if there

are too many state to visit, it will make q values a long time to converge and the q-tables will be very large in memory.

So finally I considered Intersection state(traffic light and presence of cars) and Next waypoint location as two state variables.

3. 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?

Q-Learning Implementation is in Qlearning.py. Here are the steps for implementing Q-Learning Algorithm:

1. Initialize the Q-values: I initialized q-values as 5.0
2. Observe the current state: I considered the following state variables:
    - next waypoint
    - Intersection state (traffic light, (oncoming, left, right))
3. For the current state, choose an action based on the selection policy (epsilon-greedy),which is available in get_action function. The epsilon-greedy approach chooses an action with the highest Q value with a probability of (1-epsilon) and explores with a probability epsilon, given the state, which is like flipping a coin at every time step and then deciding to make either a random action(exploration) or follow the optimal policy given the inequality.
4. Take the action, and observe the reward and as the new state
5. Update the Q-value for the state using the observed reward the maximum reward possible for the next state.

Changes noticed in Agent's behavior
Q-Learning is implemented in its own class Qlearning.py. I noticed that Qlearning gives better appreciation of the world.
Agent is learning the traffic light rules and trying to reach the destination.
I see that agent does not choose to stay in position anymore in case of no traffic light is green. This can be considered an improvement. I also notice that agent is not choosing the red-light direction once it learns the bad impact of the choice.

Performance:
Alpha value (learning rate) was set to 0.9, Gamma value (discount factor) was set to 0.2, Epsilon was set to 0.05, I set these values by trying many possible values and compare the performances, the details will be shown later.
Implemented the Qlearning.py after determining these variables. The agent has reached the destination before deadline 95 times out of 100 runs.

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 tuned parameters (alpha, gamma and epsilon) by trying different combinations, here are the performances:

alpha	gamma	epilson	Cum_reward(n=100	Number of success moves(n=100)

0.5	0.06	0.9	10.6	23	25/100
0.5	0.05	0.5	13.0	10	55/100
0.5	0.2	0.5	15.0	15	62/100
0.7	0.2	0.05	11.0	16	68/100
0.9	0.6	0.05	27.6	20	92/100
0.7	0.2	0.05	21	16	93/100
0.5	0.2	0.05	35	28	91/100
0.9	0.05	0.05	21.5	15	95/100
0.9	0.4	0.05	23	20	96/100
0.9	0.2	0.05	16.5	12	97/100

So the best combination is Alpha = 0.9, Gamma = 0.2 and Epsilon = 0.05. Now we compared the q-learning agent with random agent, the success rate is around 95% which is much better than the basic random agent (19%).

Intuitively, the optimal policy for the agent would be as follows:
1) Obey traffic laws, and then 2) move in the direction of the way point provided. If traffic laws occlude the way point direction, do not move. The Learning Agent generated from the agent.py program accomplishes this policy through Q-Learning iteration. I used two strategies to increase the efficiency by which the agent learned the optimal policy. The first version initialized Qˆ to zero, and then employed monotonic functions for epsilon and alpha to control the rates at which the agent explored the state-action space versus exploiting knowledge gained. The discount value for future reward was found to be optimal near the limit of 0. This is due to egocentric

nature of the agent, as well as the random aspects of state that cannot be predicted ahead of time. Were the simulation changed to an allcentric view, where the agent could sense where it was in relation to the destination, etc, then gamma term would be a more important parameter for optimal performance and policy generation. This program version generally took ˜20 trials to converge to an optimal policy. If one knew the simulation would incorporate new features at some point, I would choose to use this version of the program, as it is most adaptable to increased state-action space. However, for the simulation as it stands, the strategy that appeared most efficient was to set the Qˆ initialization to an overly optimistic value that exceeded any possible single action reward. This change, concomitant with subsequent changes to make alpha constant (set to 1) and a full discount factor (gamma = 0) generated a highly efficient learning agent that almost always found the destination within 5-7 trials. Notably, this strategy was only successful because of the limited state-action space that existed in the simulation. Where I to have included more inputs for the state, or additional variables were added to the simulation, it is likely that this strategy would take exponentially longer to converge to optimal policy. Also helping the efficiency of this strategy is that certain states were much more likely to be encountered than others, e.g. no traffic leaving just traffic light and way point direction. If there were more traffic, then many more training examples would need to be observed by the agent before optimal actions were consistently evident.
In conclusion, I have used various strategies to build a self-learning smartcab that is able to obey traffic rules and efficiently reach a destination. It is reliant on the ability to sense the immediate intersection environment, and is also dependent on information provided by the way point planner. Given these variables an optimal policy for self-driving was generated using Q-learning iteration. The best performing strategy was to initialize Qˆ values at an unobtainable reward level, however it is probable that the addition of features or the incorporation of probabilistic outcomes would require a more fine-tuned strategy similar to first strategy discussed here.