Tasks

Project Report

You will be required to submit a project report along with your modified agent code as part of your submission. As you complete the tasks below, include thorough, detailed answers to each question *provided in italics*.

Implement a Basic Driving Agent

To begin, your only task is to get the **smartcab** to move around in the environment. At this point, you will not be concerned with any sort of optimal driving policy. Note that the driving agent is given the following information at each intersection:

- The next waypoint location relative to its current location and heading.
- The state of the traffic light at the intersection and the presence of oncoming vehicles from other directions.
- The current time left from the allotted deadline.

To complete this task, simply have your driving agent choose a random action from the set of possible actions (`None`, `'forward'`, `'left'`, `'right'`) at each intersection, disregarding the input information above. Set the simulation deadline enforcement, `enforce_deadline` to `False` and observe how it performs.

QUESTION: *Observe what you see with the agent's behavior as it takes random actions. Does the* ***smartcab*** *eventually make it to the destination? Are there any other interesting observations to note?*

Yes. The cab eventually makes it to the destination.
One interesting observation is that there is not real bound for the city. For example, if the cab is on the east edge, it can go further east and reach the west edge of the city.

Inform the Driving Agent

Now that your driving agent is capable of moving around in the environment, your next task is to identify a set of states that are appropriate for modeling the **smartcab** and environment. The main source of state variables are the current inputs at the intersection, but not all may require representation. You may choose to explicitly define states, or use some combination of inputs as an implicit state. At each time step, process the inputs and update the agent's current state using the `self.state` variable. Continue with the simulation deadline

enforcement`enforce_deadline` being set to `False`, and observe how your driving agent now reports the change in state as the simulation progresses.

QUESTION: *What states have you identified that are appropriate for modeling the* ***smartcab*** *and environment? Why do you believe each of these states to be appropriate for this problem?*

ANSWER:

I implemented a class "LearningAgent_v0" in "agent.py".

I directly used all input as state including whether light is green or red, whether there is any cab on the intersection and there are they going. The reason is that both the light and information for other cabs gives information on which direction I can go if I want to obey the driving rules.

Moreover, I also included the "next_waypoint" in states, because it works like a "short sighted" GPS and can provide me partial information of which direction is optimal (without considering traffic and light) if I want to go to destination as soon as possible.

I also tried to include "deadline" as state. The reason is maybe the cab need to take different strategy when there is little time left. I devided deadline to 3 categories and added it in to state. However, the success rate didn't increase and the average reward also doesn't change significantly. So I didn't use deadline here.

*The reviewer asked "*Therefore make sure you assign your desired state to `self.state` so it updates while running in the pygame window*".*

Response: "self.state" was implemented in "LearningAgent_v0" and "LearningAgent_v1". But not in "LearningAgent_v2". Now I implemented "self.state" in "LearningAgent_v2". In order to run different agent, simply change the code section "a = e.create_agent(LearningAgent_v2) " in the "run()" function.

OPTIONAL: *How many states in total exist for the* ***smartcab*** *in this environment? Does this number seem reasonable given that the goal of Q-Learning is to learn and make informed decisions about each state? Why or why not?*

Answer:

*In my choice of state, I have 2 state for light, 4 states for right side traffic, 4 states for left side traffic, 4 states for oncoming traffic, and 3 states for the information for recommended action from planner (the GPS). So total state is 2*4*4*4*3=384.*

The number seems pretty large; however, some state might not happen. For example, when the light is red, the oncoming traffic cannot go forward or make left turn. I will leave this for the cab to learn. I implement a dictionary for the cab. Every time the cab encountered a new state, the cab will add several new entries (each state corresponds to 4 state-action pair) to the dictionary. The key of the dict is combination of state, and possible actions, while the value is Q value for the state and action.
*I run the simulation to train the cab with 500 trials. At the end, the cab need to learn 166 parameters (number of state-action pairs) for the Q-values look-up table, much less than 384*4.*

Implement a Q-Learning Driving Agent

With your driving agent being capable of interpreting the input information and having a mapping of environmental states, your next task is to implement the Q-Learning algorithm for your driving agent to choose the *best* action at each time step, based on the Q-values for the current state and action. Each action taken by the **smartcab** will produce a reward which depends on the state of the environment. The Q-Learning driving agent will need to consider these rewards when updating the Q-values. Once implemented, set the simulation deadline enforcement `enforce_deadline` to`True`. Run the simulation and observe how the **smartcab** moves about the environment in each trial.
The formulas for updating Q-values can be found in **this** video.
QUESTION: *What changes do you notice in the agent's behavior when compared to the basic driving agent when random actions were always taken? Why is this behavior occurring?*

Answer:
The cab start to drive around following some pattern, instead of randomly wondering around the city. One interesting pattern is that the cab often making clock-wise circle in the city.
This is maybe because at each intersection, turning right on average has the highest possibility to get positive reward. For example, if there is no traffic and no destination, a cab can always turn right without violation driving rules. Even if there is traffic, a cab can always turn right given the cab can turn left or go straight.

Improve the Q-Learning Driving Agent

Your final task for this project is to enhance your driving agent so that, after sufficient training, the**smartcab** is able to reach the destination within the allotted time safely and efficiently. Parameters in the Q-Learning algorithm, such as the learning rate (`alpha`), the discount factor

(`gamma`) and the exploration rate (`epsilon`) all contribute to the driving agent's ability to learn the best action for each state. To improve on the success of your **smartcab**:

- Set the number of trials, `n_trials`, in the simulation to 100.
- Run the simulation with the deadline enforcement `enforce_deadline` set to `True` (you will need to reduce the update delay `update_delay` and set the `display` to `False`).
- Observe the driving agent's learning and **smartcab's** success rate, particularly during the later trials.
- Adjust one or several of the above parameters and iterate this process.

This task is complete once you have arrived at what you determine is the best combination of parameters required for your driving agent to learn successfully.

QUESTION: *Report the different values for the parameters tuned in your basic implementation of Q-Learning. For which set of parameters does the agent perform best? How well does the final driving agent perform?*

Answer:

I tried severl combinations, alpha={0.1,0.5,0.9}, gamma={0.2,0.8}, epsilon={0.2,0.8}.(combination = 12) For each parameter combination, I run for 500 trials. I monitored success rate for every recent 100 trials as well as average reward for every recent 400 moves. Based on success rate and average reward, I choose the combination of alpha=0.1, gamma=0.2, and epsilon=0.2. This combination of setting gives success rate of 0.86 (last 100 trials) and average reward 1.29(last 400 moves).

The reviewer asked to provide more information on performance of other parameters.
Here I listed comparisons of some other parameter settings.

alpha	gamma	epsilon	Success rate (last 100 trails)	Average reward (last 400 moves)
0.1	0.2	0.2	***0.86***	***1.29***
0.1	0.2	0.8	*0.31*	*0.30*
0.1	0.8	0.2	*0.65*	*1.17*
0.5	0.2	0.2	*0.25*	*0.59*
0.9	0.2	0.2	*0.35*	*0.80*

Reviewer comments that I should use number of trials equals 100 instead of 500.

*Response: The reason that I use 500 trails is that **1)** the success rate and average reward may not converge to its optimal value with 100 trials, for example, when n_trail=100, the success rate may be 0.59, but it may drop to around 0.2 when n_trail=200;**2)** the success rate and average reward have large variance when n_trials=100 compared with n_trails=500(see tables below)**3)** if use the results from n_trail=100 to choose optimal parameter, the parameter chosen may be not be optimal because of the large variance.*

500 trials (run 3 times have different success rate and average reward)

alpha	*gamma*	*epsilon*	*Success rate*	*Average reward*
0.1	*0.2*	*0.2*	*0.83*	*1.26*
			0.86	*1.47*
			0.83	*1.28*

100 trials(run 3 times have different success rate and average reward)

alpha	*gamma*	*epsilon*	*Success rate*	*Average reward*
0.1	*0.2*	*0.2*	*0.59*	*0.92*
			0.36	*0.75*
			0.43	*0.94*

***QUESTION:** 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? How would you describe an optimal policy for this problem?*

Answer:

Observing that there are almost 200 parameters in agent v0, another things to improve might be to extract features from the original state setting.

So I designed another class "learningagent_v1". Its state contains four features: "right_ok", "left_ok", "forward_ok" and "next_waypoint", where the first 3 represent whether it is ok to go to different direction depending on current light condition and traffic condition, and the last on give the direction the planner gives. Using the same parameter setting, i.e., alpha=0.1, gamma=0.2, and epsilon=0.2, "learningagent_v1" gives success rate 0.88 and average reward 1.45.

The above two classes use a look-up table (python dictionary) to store Q-value for state-action pairs. Another approach would be to parametrize the Q-function and then use supervised learning to approximate Q-function. Here I just use a simple linear regression to approximate Q-function. I use the state and action in "learningagent_v1" and the interaction (their product terms) between the state and action as feature space for linear regression. The Q-value for each state-action pair serves as outcome variable for linear regression. The regression is learned via stochastic gradient decent.

I implement the above approach in another class "learningagent_v2". With parameter gamma= 0.2 and epsilon=0.1, it gives ***success rate 0.99*** *and* ***average reward 1.74*** *after trained for 500 trials.*

More details of parameter settings for "learningagent_v2", such as learning rate and regularization for stochastic gradient decent and linear regression are in "agent.py"

From my opinion, the agent should first learn not to violate the driving rules. Then the agent should be able to learn, 1) if the planner's direction next move does not violate the driving rule, go to that direction, 2) if the planner's direction conflict with driving rule, make decision on what to do for next step. Under linear regression approximation of Q-value function, this should be reflected on the weights for different parameters. I listed the final regression model in the following table.

For example, if GPS shows that next action should go right, the weight for the interection terms "next_right" and action "right" (i.e., "next_right_action_right") should be greater than "next_right_action_left" and "next_right_action_forward". (see green color highlighted in table).

We can observe similar trend when GPS shows next actin should be "forward" or "left".

feature	weight
Bias term (intercept)	0.55509872
right_no	-0.188396
forward_no	-0.842783
left_no	-0.115965
next_right	0.441135
next_forward	0.825841
next_left	0.594713
right	0.046
forward	0.850888
left	0.31337
right_no_action_right	-0.172964
right_no_action_forward	-0.015616
right_no_action_left	0.00041
forward_no_action_right	0.5013
forward_no_action_forward	-2.13408
forward_no_action_left	-0.803357
left_no_action_right	0.498663

left_no_action_forward	-1.269164
left_no_action_left	-0.93597
next_right_action_right	1.808271
next_right_action_forward	-0.892596
next_right_action_left	-0.634323
next_forward_action_right	-1.200404
next_forward_action_forward	2.111136
next_forward_action_left	-0.404125
next_left_action_right	-0.561841
next_left_action_forward	-0.36763
next_left_action_left	1.351861