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?*

Answer 1

I noticed a few interesting things regarding the rules of the grid world:

- The cab cannot violate traffic rules or create accidents. The car will simply stay where it is when attempting an illegal action
- Traffic lights seem to be randomized, they’re not always changing on the same interval
- There is not much traffic and the amount of other drivers is constant
- There is a hard limit of (100+deadline) steps even when enforce_deadline is set to False
- Because there is no way to cheat by f.ex. crossing red lights, the deadline variable is probably not very important for a learning algorithm and the optimal policy should be to simply follow the next_waypoint indicator

As a conclusion of my observations, it seems that randomized behavior should be able to reach the destination because accidents cannot happen and crossing red lights is impossible. The cab only gets a negative reward. I implemented a counter to

see how often the cab would reach the destination when moving randomly and found a 69.1% chance of the cab reaching the goal with the deadline turned off when running 1,000 iterations.
Another interesting observation was that even with the deadline turned on, running a 1,000 iterations, still resulted in an 18.9% chance of the cab reaching its destination by only taking random actions.

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 2

Unfortunately, there is no data available about the location of the car that can be used. So modeling states based on the position is not an option.
At first I tried to include what I know into the states. So I wrote a mapping from input data to a set of the states of the following form:
{go_right, go_left, go_forward, do_nothing, move_wrong_direction}
I removed the responsibility to learn when to go where from the learner and did it myself.
I think this idea is coming to mind easily because simply following the next_waypoint seems to be the optimal policy.
I then did some research in the forums and found out that a few other students did it the same way and failed their reviews.
As a conclusion I decided that using the Cartesian product of the input values as a set of states was a better choice. This way, the algorithm learns the traffic rules by itself instead of me telling it what to do. This of course resulted in a quite larger state space of 512 states.

I terms of code, I modelled my states as a list of lists where each list is a state:
S1: ['wp_left' 'green' 'on_forward' 'lf_forward' 'rg_forward']
S2: ['wp_left' 'green' 'on_forward' 'lf_forward' 'rg_left']
S3:['wp_left' 'green' 'on_forward' 'lf_forward' 'rg_right']
...,
S510: ['wp_none' 'red' 'on_none' 'lf_none' 'rg_left']
S511: ['wp_none' 'red' 'on_none' 'lf_none' 'rg_right']
S512: ['wp_none' 'red' 'on_none' 'lf_none' 'rg_none']]

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 In decisions about each state? Why or why not?*

Optional Answer

I think the total number of states is inifinite. This is because I could always encode past information as a new states and add them on the fly during learning. For example there could be a state s1 and I could expand the state space like this:
New_state= s1 and the last waypoint_direction was ‘forward’
Or information about how often the destination was reached could be added.
This is theoretically possible but I do not think it to be reasonable. With an infinite number of states the algorithm would not converge and could not be used.
So apart from using past information, I think using the Cartesian product of all available input data is the way to go. Like so:
next_waypoint=['wp_left', 'wp_right', 'wp_forward','wp_none']
traffic_light=['green','red']
oncoming=['on_forward','on_left','on_right','on_none']
left=['lf_forward','lf_left','lf_right','lf_none']
right=['rg_forward','rg_left','rg_right','rg_none']
These will result in 512 states. I did not include the deadline in the states because, as I observed earlier, it does not play any important role in the decision process. The agent cannot make “risky” decisions like crossing a red light based on a passing deadline because he simply is not able to.
With this state space, every important information is captured for the agent to make informed decisions.
One thing I am unsure about is, whether it is an option to remove some of the 512 states. Some of them obviously will never be reached or used because of the rules of the world. Removing them might increase the algorithms accuracy but I think it also removes its ability to generalize should the rules ever change. As a conclusion, I did not remove unreachable states.

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 3

The agent starts out making random moves because of the uniformly initiated Q Matrix and the exploration rate. Very quickly the algorithm learns the rules and proceeds directly to the destination with only a few wrong moves. There will always be wrong actions of course, when the exploration rate is >0. Maybe an idea for fine tuning the algorithm would be to reduce it to zero after a few thousand iterations.

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 4

I used a simple metric to determine success:
# of success divided by # of trials
My approach of searching optimal parameters was similar to the GridLearning algorithm from sklearn. I started out with a wide range of parameters like learning rate from 0 to 1 and narrowed it down for each parameter. I then chose the best results of each trial .
After searching the parameter space I found the following parameter set:
Learning rate: 0.19 discount factor: 0.25 exploration rate: 0.05
To achieve a total accuracy of 96.141% running 100,000 trials.
When running 100 trials success rates vary between 80%-96% because of the random nature of the algorithm which will make Q Learning converge at different rates for different trials. Without a deadline, the algorithm runs with 100% accuracy. I think that result is quite good.

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?*

I think my agent has found a near optimal policy. I still accumulates a lot of negative reward probably because it tries to cross red lights repeatedly. Since there is no strong negative consequence for this, I do not consider it a problem (contrary to a real life situation of course).