P4 Smartcab Project worksheet

In this project we'll implement an agent that is capable of learning how to drive inside a grid-like world. The traffic rules are known to us and the simulated environment, but not to the agent. The agent will utilize reinforcement learning to learn the rules. Specifically, we'll be using Q-learning to approximate the optimal policy.

Implementing a basic driving agent

After installing the necessary libraries to run the simulation, we implemented a random policy agent. The intent was to observe the general behaviour of the simulation. Without an explicit deadline the agent was able to reach the destination before hitting the hard time limit in approximately 80% of trials. The motion resembled brownian motion as expected.
I also observed that when the agent attempts to violate the traffic rules the agent is penalized but it does not in fact move.

Identify and update state

We elected to build a composite state representation, comprised of a two state stoplight variable, three four-state variables for traffic coming from each direction and one three state desired direction variable. That gives us a total number of 384 possible states.
The most significant state variable is of course the stoplight as that has a huge impact on the legality of individual moves. It's also the most common one that the agent must pay attention to to avoid running a red light.
Next we need to provide the agent with the information regarding the traffic situation around it. That is represented by the three four-state traffic variables. This traffic information could've been condensed but we elected not to do that as this is exactly the sort of behavior we want the agent to learn without preprogramming. The downside is that there are quite a lot of rare combinations that the agent is unlikely to encounter during training, such as a car coming in from all three directions. We made the choice to leave them in as they would eventually be learned given a longer training period. Preprogramming complex behavior would defeat the purpose of the exercise and preprogramming simplified behavior would cause the agent to occasionally perform traffic violations.
The last state variable is the desired direction as indicated by the planner. The agent needs to know what the desired direction is during each step, or it would be impossible for the agent to even attempt to drive better than the random agent.

We've elected to not model the None desired direction as the simulation will halt when that occurs, so there's no need to model that inside the agent.

We chose not to model the current timestep. If we had included the timestep in the state directly, we would've created a huge state space where it would be impossible to repeat a state during a single trial. That would have substantially slowed down learning and made it impossible for the agent to drive correctly if it ever encountered a larger time limit than it had seen during training. We could have implemented some sort of a discretized version of the timestep (say, by two states 'no rush' or 'rush') but that would've required us to hard-code such logic, and the relationship between remaining time and the distance to target isn't exactly clear-cut in an environment such as the grid-city.

Another variable we might've chosen to model is how long the agent has been waiting at a red-light. That could've yielded some benefits but would've expanded the state space.

I've encapsulated all state-related information to a separate class called State.

Implementing Q-learning

Compared to the randomly acting agent, the learning agent proved far superior. On average the agent started to reliably reach the goal from the third trial onwards, sometimes even from the first. By trial 100 it was getting an average reward of over +20

In the initial few trials the agent has a tendency to drift left because of the way that the action selection is implemented.

Enhance the driving agent

We implemented the learning rate and discount factor in the initial implementation as it seemed the obvious thing to do. To improve on the agent's learning ability we added Q^ initialization as an additional parameter and implemented greedy epsilon exploration.
The performance was functionally identical to the previous version. The agent consistently learns to reach the target within three trials and gradually improves over time. By trial 100 it maintains a positive cumulative reward near the +20 mark.

We fine-tuned the learning rate, discount factor and the initial Q^ value using a form of gridsearch, where performance was measured by the average reward per step taken by the agent over thirty trials. We ran the gridsearch algorithm a few times, and included the latest completed run as a separate file (gridsearch_results.csv). Based on that the optimal triplet of learning rate, discount factor and initial Q^ seems to be 0.21, 0.95, and 4, respectively. However, such a high discount factor proved to be ultimately detrimental to the agent's performance, it had learned to artifically inflate it's score by taking unnecessary loops whenever it ran into a red light. We opted for a more conservative 0.42 0.15 4 configuration based on an earlier run, which produced better results. An agent trained with this configuration is much more likely to simply wait at a red light rather than attempting a right-on-red.

Overall the agent seems to get close to the optimal policy. It does get the occasional minus score and sometimes it'll wait at a red light for a long time but other than that it seems to be performing well.

Appendix

Technical notes

We are not allowed to modify files other than the provided smartcab/agent.py. However, because of the way that the external code references images, we moved the images folder inside the smartcab folder. As it happened I was unable to get pygame working on my macbook so I wasn't able to run the simulation with graphics. As a result some observational aspects of the projects may be lacking.

Q-Learning implementation details

The implementation itself was fairly straightforward. During each simulation step we receive some input and use that to update our current state S. For that state we then find the action that has the best known Q^ value and return that as our chosen action a'. We then check what new state S' we're in and approximate the utility of that new state with max over a of Q^(a, S'), multiply that with our discount rate, add our reward and move our Q^(a', S) towards that value, modified by our learning rate. In other words exactly like in the reinforcement learning lecture.

There is a diminishing chance that the agent will act randomly rather than according to the Q^ value. We also adopted a policy of preferring to try untried options over previously tried options to speed up learning.