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 which seems a bit high. We'll see how the number of states affects our learner performance, if it turns out to be too complicated we'll reduce the number of traffic states to just two states for when there is traffic and when there isn't. Traffic in the simulation seems a very unlikely occurrence, so the learner can afford to be suboptimal in those cases.

Given that violating the traffic rules carries only penalties and no rewards (ie getting closer to the goal), we don't need to track the current timestep. The reason why we might've wanted to track that as an input it would've been to only engage in risky behavior if we're running out of time, but since said risky behavior carries no potential benefit, there is no point.

I've also 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.

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. For the sake of making some enhancements we added a small, decaying chance that the agent acts randomly rather than according to the Q^ values. This is an effort to ensure that the Q^ sequence converges to Q and doesn't get stuck in local optima. 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 and discount factor using a form of gridsearch, where performance was measured by the average reward per step taken by the agent over thirty trials. Based on that the optimal learning rate and discount factor combination seems to be 0.42 and 0.15.

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.