Project 4: Training a Smart Cab to Drive

Patrick Martin

In this project, I applied reinforcement learning techniques to design and simulate a smart cab agent operating.

Testing and Design Tasks

To ensure that my development environment was complete, I implemented an agent that randomly selects an action without any "smarts." This agent uses a discrete uniform distribution over the four possible actions: `None`, `left`, `right`, and `forward`. It is clear from this strategy that the agent will bounce around the board due to the randomness of its action selection and the randomness of the environment (traffic lights and other agents). The agent usually arrives at the final goal; however, there is no guarantee on timing performance. I let the agent run five times and tabulated the simulation deadline and the number of steps to completion:

Deadline	Steps to Completion
20	168
40	120
25	7
30	22
40	118

Figure 1: Tabulated data from several runs where the smart cab agent uses a full random action selection.

As Figure 1 illustrates, two out of the five runs completed before the deadline, but it is likely due to the right combinations of initial conditions. An open-loop algorithm will clearly not work for this task.

The driving agent state should be composed of environment variables that facilitate proper decision making. The chosen state must balance the information required to solve the problem with the space complexity of the Q-learning algorithm. Some input data may not be needed, which will create a more efficient learning algorithm.

I studied the available agent data and experimented with my implementation of the Q-learning algorithm. At a minimum the state should include the intersection light, {`green`, `red`}, the direction of the next waypoint, {`forward`, `left`, `right`}. I added a new implicit state, the intersection traffic, based on the status of each path available to the agent. A full state space would have used all three options for each direction; however, I chose to abstract them as a binary value: `true` := there is a vehicle on that path, `false` := the path is clear. Although I lose some information, the state space would have expanded to 384 states, which would take longer to explore. I created the state space by building a Cartesian product in my Python code, `itertools.product()`, resulting in 48 states.

Q-Learning Design and Implementation

The cornerstone of the Q-learning algorithm is the construction of a **Q-table**, which stores the value of each state-action pair. This table starts with all entries at 0.0. Every time the smart cab agent takes an action from a state, it earns a reward from the environment, R. Q-learning uses this reward and the current Q-table values to compute new values for the next iteration. More formally: given current and next states, $s, s' \in S$, an action $a \in \mathcal{A} = \{\texttt{None}, \texttt{left}, \texttt{right}, \texttt{forward}\}$, at time step k the estimate of the Q-value, $\tilde{Q}$ is computed by:

$$\tilde{Q}_{k+1}(s,a) = (1-\alpha_k)\tilde{Q}_k(s,a) + \alpha_k[R_k + \gamma \max_{a' \in \mathcal{A}} \tilde{Q}_k(s',a')].$$

This particular form of the algorithm assumes that I am using the adjustable learning rate, α_k, given by:

$$\alpha_{k+1} = \frac{1}{\alpha_k}, \text{ with } \alpha_0 = 1.$$

Adopting this learning rate adjustment equation was in class as a strategy to make approximate Q-values converge to their expected value.

The implementation of my Q-table is a dictionary data structure that maps state-action tuples to their Q-value entry. For example, one of my chosen entries in the Q-table would look like:

```
(('green', False, False, False, 'forward'), 'forward')
```

that represents an intersection with a green light, no traffic, a waypoint goal that is in front of the cab and the agent takes the `forward` action.

An agent running the Q-Learning algorithm stops taking random actions and takes smarter actions based on its prior performance. The data in Figure 1, demonstrated that the random selection policy had no reasonable guarantee on reaching the destination in the desired time.

Q-Learning Algorithm Tuning

Since my chosen state space has 48 states, I used the ϵ-greedy exploration selection strategy to explore as much of the state as possible using a changing exploration probability, ϵ. In the lectures, I learned that using this technique (with inifinite exploration) guarantees that $\tilde{Q} \to Q$ and the approximate policy, $\tilde{\pi}$, approaches the optimal policy, π^*. The ϵ-greedy exploration action selection algorithm takes the following form:

$$\tilde{\pi}(s) = \begin{cases} \arg\max_a \tilde{Q}(s,a), & 1-\epsilon \\ rand(\mathcal{A}), & \epsilon \end{cases}$$

where $rand(\mathcal{A})$ represents the random action selection used at the beginning. In reinforcement learning, there is always a trade off between exploration and exploitation of knowledge. If you do not explore, then you have limited knowledge; however, if all you do is explore, then you will not gather consistent enough knowledge for learning the optimal policy.

My particular ϵ-greedy algorithm adjusts the value of ϵ as each trial runs according to:

$$\epsilon_{k+1} = \frac{1}{\sqrt{k}}, \; k = 1, 2, \cdots, n_d,$$

where n_d is the deadline of a trial, i.e. 30, 40, etc. The only value that is fixed for a simulation run is the discount factor, γ. I used this parameter to tune the learning of the agent over several runs. Figure 2 plots the success rate of my learning agent (using 100 trials per run) over the discount factor. It is clear from the plot that lower γ values result in better average success rates.

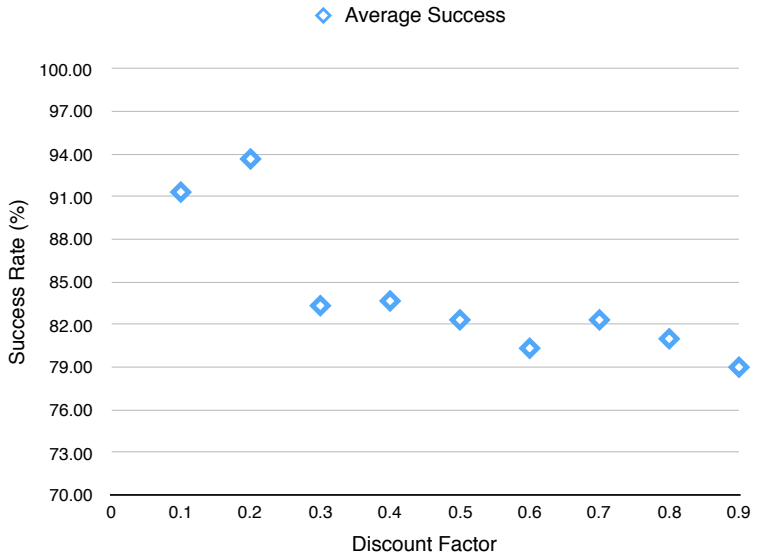

Figure 2: This plot shows how the success rate for my learning algorithm depended on γ. Each data point is the average success rate for 3 simulations runs, where each run had 100 trials.

Using this data, I chose my final discount factor to be $\gamma = 0.2$. This value, in combination with my ϵ-greedy strategy with "fading" α, resulted in a smart cab that consistently performs above 90% for getting to the destination before the deadline. Figure 3 shows that my algorithm maintains a positive reward after an initial exploration period caused by the ϵ-greedy strategy.

The optimal policy should always take the direction of the next waypoint, unless there is traffic in that direction or if the traffic light is `red`. Performing that action would minimize the distance of the smart cab from the destination. If the traffic light is `red` or there is traffic, the agent should take a legal action, such as right turn on red. I examined the final Q-tables (after the 100^{th} trial) and sought out some key elements that represent the optimal policy. For example, a smart cab with a green light, no traffic, and a waypoint straight ahead should choose `forward`. Figure 4 shows selected Q-table rows that demonstrate two key examples from the optimal policy. In both of these entries, the $\arg\max \tilde{Q}(s, a)$ would result in behavior like the optimal policy.

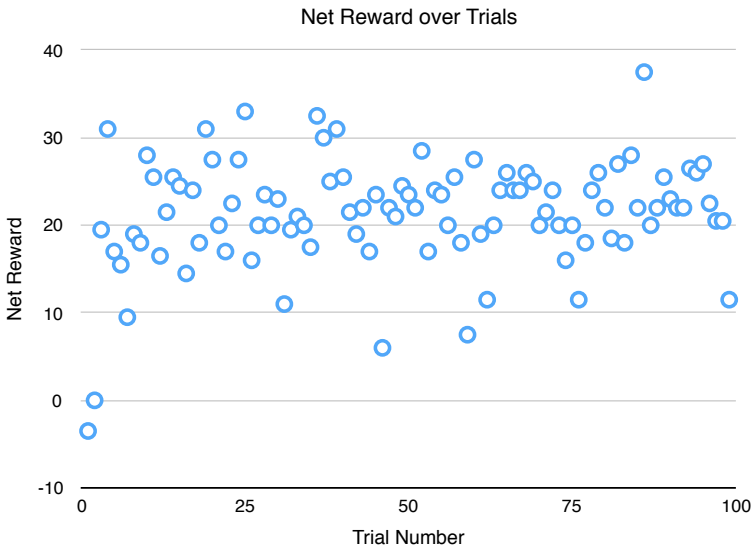

Figure 3: The data in this plot shows the net reward of the agent over the 100 trials. It consistently scores above 0.0 after the exploration period.

	None	forward	left	right
(red,F,F,F,right)	0.521	−0.419	−0.457	3.537
(green,F,F,F,forward)	0.647	4.378	−0.269	−0.356

Figure 4: Selected results of the final Q-table using my learning algorithm.