

Training a Smartcab How to Drive

General

This project applies reinforcement learning techniques for a self-driving agent in a simplified world to aid it in effectively reaching its destinations in the allotted time. The agent implements a Q-Learning algorithm for the self-driving agent to guide the agent towards its destination within the allotted time.

Implementing a Basic Driving Agent

With the goal of creating a simple, albeit naive, driving agent, I started by creating a stochastic agent with random walk: it chose randomly from valid actions at each cycle. The core logic was implemented in the `update` definition via:

```
action = random.choice([None, 'forward', 'left', 'right'])
```

When disabling the deadline via `enforce_deadline=False`, the agent was observed to eventually reach its destination. Qualitatively speaking, it typically reached its destination quicker if the starting position was closer. The agent characteristics are that of random walk agents: there is no intelligence and it disregards any notions of risk or reward. The agent reaches the goal purely by chance.

Informing the Driving Agent

The states I chose map directly to the provided inputs, with the addition of a `next` parameter, denoting the next waypoint location. In all, the state is completely defined by:

Measure	Description
light	Whether the light is green or red
next	The next waypoint location relative to the agents current location and heading
oncoming	The heading of a car on the opposite side of the intersection, if present
left	The heading of a car on the left side of the intersection, if present
right	The heading of a car on the right side of the intersection, if present

It's important to note that this state is defined by low-level measures; yet, they completely describe the conditions at the intersection (via light, oncoming, left, right) and a heuristic for which action should be taken (via next). I reasoned that this would be appropriate because of the nature of the problem: the wrapping grid world with no obstacles meant that no higher-order path-planning needed to be conducted. There is no need to sense the agent's absolute location, absolute heading, nor relative location to the goal. While this data might have been obtainable via dead-reckoning, I chose not to incorporate it for both simplicity and the notion that scaling to a non-deterministic world would have been impossible without faculties to sense its location or heading.

For all intents and purposes, each intersection was equivalent. In that regard, the agent could adopt a seemingly myopic and egocentric view of

the world, only "caring" about what happens at the intersection it is currently at, and where the next waypoint resides.

Implementing a Q-Learning Driving Agent

After implementing a basic Q-Learner in the driving agent architecture, I noticed significant reduction in the time needed to reach the destination. Specifically, the agent had bias towards movement to the next waypoint (as opposed to randomly). After 50 trials, the average running reward was 22.18 and the average number of steps taken was 13.38. The agent was incredibly quick to learn and develop a robust policy layer; in comparison, the first trial (learning from scratch) received less than half the reward (9.5) and took more than twice as long to reach the destination (28 steps). Most importantly, the agent achieved a 100% success rate in reaching its destination.

After the first trial, the agent quickly learned the "rules of the road", particularly for traffic signs. Throughout the remaining trials, less common encounters with other cars on the roads added increased, albeit marginal, knowledge and proficiency to the agent. Comparing the metrics from the first half to the last half, I observed a 3% increase in reward and 4% decrease in steps taken. I believe this to be from learning from less common states.

Of important note, the agent is blind to both its current location and the deadline. An optimal agent would have held a "holding pattern", accruing reward points for correctly obeying traffic rules, and only landing on the destination at the last possible cycle. However, given the inputs and how the agent was built, neither location nor deadline was incorporated into the state. As such, the agent is myopic in its handling of rewards: it

maximizes its short-term reward with little regard for delayed, future rewards.

Improving the Q-Learning Driving Agent

After tuning, I found the following parameters to be the optimal combination:

Parameter	Value
alpha (learning rate)	0.9
epsilon (exploration)	0.1
gamma (discount)	0.20

The agent still maintained a 100% success rate, but achieved a marginal increase in reward (average of 23.01) and a slight increase in time taken (14.9).

I believe that the agent, with its immediate and "greedy" behavior, converges onto a policy that reaches the destination in minimum possible time without incurring penalties. As alluded to earlier, the myopic state policies ensures that the agent produces the optimal step at each intersection. However, in my opinion, this is not necessarily the optimal policy. In an extreme case, if a traffic light were "broken" and stayed red forever, the agent would have stayed at the intersection ad infinitum. A notion of history and memory is needed in the state (e.g. cycles stopped at an intersection) to correctly balance the delayed and long-term rewards. In that same example, the agent could have "escaped" the local minima by making a right turn, or simply passing through the red light (incurring a penalty, but resulting in a long-term reward). This optimal

policy is highly-sensitive to the rewards and penalties of collisions, running red lights, and deadlines. Nonetheless, there's still room for improvement!

Appendix

Installation

This project requires **Python 2.7** with the [pygame](#) library installed.

Running

Run:

```
python smartcab/agent.py
```