smartcab

October 24, 2016

1 Machine Learning Engineer Nanodegree

1.1 Reinforcement Learning

1.2 Project 4: Train a Smartcab How to Drive

1.3 Project Overview

In this project you will apply reinforcement learning techniques for a self-driving agent in a simplified world to aid it in effectively reaching its destinations in the allotted time. You will first investigate the environment the agent operates in by constructing a very basic driving implementation. Once your agent is successful at operating within the environment, you will then identify each possible state the agent can be in when considering such things as traffic lights and oncoming traffic at each intersection. With states identified, you will then implement a Q-Learning algorithm for the self-driving agent to guide the agent towards its destination within the allotted time. Finally, you will improve upon the Q-Learning algorithm to find the best configuration of learning and exploration factors to ensure the self-driving agent is reaching its destinations with consistently positive results.

1.4 Description

In the not-so-distant future, taxicab companies across the United States no longer employ human drivers to operate their fleet of vehicles. Instead, the taxicabs are operated by self-driving agents — known as smartcabs — to transport people from one location to another within the cities those companies operate. In major metropolitan areas, such as Chicago, New York City, and San Francisco, an increasing number of people have come to rely on smartcabs to get to where they need to go as safely and efficiently as possible. Although smartcabs have become the transport of choice, concerns have arose that a self-driving agent might not be as safe or efficient as human drivers, particularly when considering city traffic lights and other vehicles. To alleviate these concerns, your task as an employee for a national taxicab company is to use reinforcement learning techniques to construct a demonstration of a smartcab operating in real-time to prove that both safety and efficiency can be achieved.

1.5 Definitions

1.5.1 Environment

The smartcab operates in an ideal, grid-like city (similar to New York City), with roads going in the North-South and East-West directions. Other vehicles will certainly be present on the road,

but there will be no pedestrians to be concerned with. At each intersection there is a traffic light that either allows traffic in the North-South direction or the East-West direction. U.S. Right-of-Way rules apply:

- On a green light, a left turn is permitted if there is no oncoming traffic making a right turn or coming straight through the intersection.
- On a red light, a right turn is permitted if no oncoming traffic is approaching from your left through the intersection. To understand how to correctly yield to oncoming traffic when turning left, you may refer to this official drivers' education video, or this passionate exposition.

1.5.2 Inputs and Outputs

Assume that the smartcab is assigned a route plan based on the passengers' starting location and destination. The route is split at each intersection into waypoints, and you may assume that the smartcab, at any instant, is at some intersection in the world. Therefore, the next waypoint to the destination, assuming the destination has not already been reached, is one intersection away in one direction (North, South, East, or West). The smartcab has only an egocentric view of the intersection it is at: It can determine the state of the traffic light for its direction of movement, and whether there is a vehicle at the intersection for each of the oncoming directions. For each action, the smartcab may either idle at the intersection, or drive to the next intersection to the left, right, or ahead of it. Finally, each trip has a time to reach the destination which decreases for each action taken (the passengers want to get there quickly). If the allotted time becomes zero before reaching the destination, the trip has failed.

1.5.3 Rewards and Goal

The smartcab receives a reward for each successfully completed trip, and also receives a smaller reward for each action it executes successfully that obeys traffic rules. The smartcab receives a small penalty for any incorrect action, and a larger penalty for any action that violates traffic rules or causes an accident with another vehicle. Based on the rewards and penalties the smartcab receives, the self-driving agent implementation should learn an optimal policy for driving on the city roads while obeying traffic rules, avoiding accidents, and reaching passengers' destinations in the allotted time.

1.6 Tasks

1.6.1 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*.

1.6.2 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: When the agent takes random actions, it does not always make it to the destination. Though `enforce_deadline` is set to `False`, there is a `hard deadline` set, which can make the trial abort if it takes too long to reach the destination. It is possible to see that sometimes the **smartcab** doesn't comply with the *U.S. Right-of-Way rules*, either by running a red light or by taking an action in conflict with other agents at its intersection. It is important to note that by taking random actions, the **smartcab** doesn't learn anything about the environment, wasting the feedback (*reward* and the *next state*) returned by the interaction with the environment.

1.6.3 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: We could map each state as being the following tuple: `(inputs['light'], inputs['left'], inputs['oncoming'], self.next_waypoint)` where `inputs['light']` is `red` or `green`; and `inputs['left'], inputs['oncoming']` and `self.next_waypoint` are `None`, `left`, `oncoming` or `right`.

Following this definition, each state can be divided and summarized as:

1. `(inputs['light'], inputs['left'], inputs['oncoming'])` represents the current situation where the **smartcab** is. This information takes into account the traffic lights, the other **smartcabs** at the intersection, and in which direction they intend to go. Therefore, it describes the agent surroundings without taking into account the `next_waypoint`.
2. `self.next_waypoint` represents the action suggested by the `planner`. This information represents the *best action/direction* to take not considering the agent's surroundings (traffic lights and other agents).

It is interesting to note that we don't need to take into account the `inputs['right']` information because the **primary agent** have preference over an agent to its right. In other words, assuming all the agents conform to the *U.S. Right-of-Way rules*, the agent only needs to know about

the *left* and *oncoming* positions at the intersection (if there are other agents in that positions, and if so, in which directions they are going).

Another important fact is that we do not choose to include `deadline` into the agent state once it could grow the number of states to a large scale. If we did so, it would take a lot more iterations to train the agent making the agent training impractical.

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 decisions about each state? Why or why not?*

Answer: There are 96 possible states in total regarding the above state definition. Precisely, we have two possible states for `inputs['light']`, four for `inputs['left']` and `inputs['oncoming']`, and three for `self.next_waypoint` (`self.next_waypoint` is `None` only when the agent reaches its final destination, which is a state not recorded in the `q_matrix`). Given the size of the environment (*6x8* grid), this number seems reasonable in order of magnitude. As the model runs multiple times (100 times by default), the **smartcab** makes hundreds of moves, and as a result, it visits each state more than once most of the time.

1.6.4 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: When the actions are taken using a policy instead of randomness, it is possible to see that the agent learns over time. In the beginning, the **smartcab** takes random actions as if it didn't know yet the relationships between *states, actions, and rewards*. During the simulation the agent doesn't always choose the best action, making some mistakes along the way. For instance, the agent can run in circles in its exploration for the best action, it could not follow the rules of the road, it could take the wrong direction, etc. Once it has visited the same state twice or more times, the agent can better assess which direction to take. This behavior happens because the **smartcab** learns the optimal action to take given its current state (function known as *policy*). Specifically, the agent learns the best action in that state by choosing the one that maximizes the *quality function*, represented in the code by `self.q_mapping` and considered the *long-term reward* achieved by the agent from a state taking an action.

1.6.5 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: We can tune the learning rate (`alpha`), exploration rate (`epsilon`), and the discount factor (`gamma`) parameters using the following functions:

1. The learning rate (`alpha`) is always the rational function *1/x*.
2. The exploration rate (`epsilon`) is one of the constant functions *(0.1, 0.2, or 0.4)* or the rational function *1/x*.
3. The discount factor (`gamma`) is one of the constant functions *(0.0, or 1.0)* or it is one of the exponential functions *pow(0.8, t), pow(0.5, t), or pow(0.8, t)*. It is interesting to note that the zero function (constant function 0.0) represents a myopic learner, which chooses an action that returns immediate high rewards, instead of long-term high rewards (which is the constant function 1.0).

Another parameter we can tune is the initial *q-value* for the *q-function*. This parameter represents the *optimism* of the learner in the face of uncertainty, where higher initial values will lead to an exploratory-learning agent.

In total, we have 40 combinations of parameters, as we can see in the table below:

1. Low exploration rate : `epsilon=0.1` and `qvalue = 0.0`

Learning rate (`gamma`)	Aborted Trials	Successful Trials	Total Trials	Percentage Aborted
constant 0.0 (low)	5	95	100	5%
constant 1.0 (high)	29	71	100	29%
pow(0.25, t) (low)	3	97	100	3%
pow(0.5, t) (median)	5	95	100	5%
pow(0.8, t) (high)	4	96	100	4%

2. Low exploration rate : `epsilon=0.1` and `qvalue = 10.0` (optimistic)

Learning rate (`gamma`)	Aborted Trials	Successful Trials	Total Trials	Percentage Aborted
constant 0.0 (low)	13	87	100	13%
constant 1.0 (high)	15	85	100	15%
pow(0.25, t) (low)	11	89	100	11%

Learning rate (`gamma`)	Aborted Trials	Successful Trials	Total Trials	Percentage Aborted
pow(0.5, t) (median)	15	85	100	15%
pow(0.8, t) (high)	14	86	100	14%

3. Median exploration rate : `epsilon=0.2` and `qvalue = 0.0`

Learning rate (`gamma`)	Aborted Trials	Successful Trials	Total Trials	Percentage Aborted
constant 0.0 (low)	7	93	100	7%
constant 1.0 (high)	26	74	100	26%
pow(0.25, t) (low)	8	92	100	8%
pow(0.5, t) (median)	6	94	100	6%
pow(0.8, t) (high)	12	88	100	12%

4. Median exploration rate : `epsilon=0.2` and `qvalue = 10.0` (optimistic)

Learning rate (`gamma`)	Aborted Trials	Successful Trials	Total Trials	Percentage Aborted
constant 0.0 (low)	14	86	100	14%
constant 1.0 (high)	21	79	100	21%
pow(0.25, t) (low)	11	89	100	11%
pow(0.5, t) (median)	18	82	100	18%
pow(0.8, t) (high)	26	74	100	26%

5. High exploration rate : `epsilon=0.4` and `qvalue = 0.0`

Learning rate (`gamma`)	Aborted Trials	Successful Trials	Total Trials	Percentage Aborted
constant 0.0 (low)	10	90	100	10%
constant 1.0 (high)	28	72	100	28%
pow(0.25, t) (low)	22	78	100	22%
pow(0.5, t) (median)	22	78	100	22%
pow(0.8, t) (high)	24	76	100	24%

6. High exploration rate : `epsilon=0.4` and `qvalue = 10.0` (optimistic)

Learning rate (`gamma`)	Aborted Trials	Successful Trials	Total Trials	Percentage Aborted
constant 0.0 (low)	29	71	100	29%

Learning rate (`gamma`)	Aborted Trials	Successful Trials	Total Trials	Percentage Aborted
constant 1.0 (high)	28	72	100	28%
pow(0.25, t) (low)	33	67	100	33%
pow(0.5, t) (median)	31	69	100	31%
pow(0.8, t) (high)	28	72	100	28%

7. "Rational" exploration rate : `epsilon=1/x` and `qvalue = 0.0`

Learning rate (`gamma`)	Aborted Trials	Successful Trials	Total Trials	Percentage Aborted
constant 0.0 (low)	3	97	100	3%
constant 1.0 (high)	25	75	100	25%
pow(0.25, t) (low)	3	97	100	3%
***pow(0.5, t)* (median)**	2	98	100	2%
pow(0.8, t) (high)	3	97	100	3%

8. "Rational" exploration rate : `epsilon=1/x` and `qvalue = 10.0` (optimistic)

Learning rate (`gamma`)	Aborted Trials	Successful Trials	Total Trials	Percentage Aborted
constant 0.0 (low)	10	90	100	10%
constant 1.0 (high)	29	71	100	29%
pow(0.25, t) (low)	3	97	100	3%
pow(0.5, t) (median)	5	95	100	5%
pow(0.8, t) (high)	4	96	100	4%

As we can see from the tables above, an *optimistic* learner (`qvalue = 10.0`) performs worst than the *non-optimistic* (`qvalue = 0.0`). Among the *non-optimistic* learners, the best results are achieved when `epsilon in (0.1, 1/x)` and `gamma in (pow(0.25, t), pow(0.5, t), pow(0.8, t))`. As the results above (*tables 1 and 7*) are very close to each other, let's analyze these combinations for 1,000 trials:

1'. Low exploration rate : `epsilon=0.1` and `n_trials = 1000`

Learning rate (`gamma`)	Aborted Trials	Successful Trials	Total Trials	Percentage Aborted
pow(0.25, t) (low)	24	976	1000	2.4%
***pow(0.5, t)* (median)**	21	979	1000	2.1%
pow(0.8, t) (high)	41	959	1000	4.1%

7'. "Rational" exploration rate : `epsilon=1/x` and `n_trials = 1000`

Learning rate (`gamma`)	Aborted Trials	Successful Trials	Total Trials	Percentage Aborted
pow(0.25, t) (low)	26	974	1000	2.6%
pow(0.5, t) (median)	40	960	1000	4.0%
pow(0.8, t) (high)	65	935	1000	6.5%

We can see clearly that `epsilon = 0.1` delivers better learners than "rational" exploration rate. So, a good combination for `(qvalue, alpha, epsilon, gamma)` would be `(0.0, 1/x, 0.1, pow(0.5, t))`.

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?*

Answer: The agent gets close to find the optimal policy, which is reaching the destination in the minimum possible time, and not incurring any penalties. Running the simulation using the parameters defined above, we get the following penalties for the last trials in the simulation (a suitable evaluation metric to assess this optimal policy situation):

1. List of penalties for the 10 last trials when `n_trials = 1,000`

Iteration	Penalties List	Length	Penalties Sum
990	`[ ]`	0	0
991	`[-1.0]`	1	-1.0
992	`[ ]`	0	0
993	`[ ]`	0	0
994	`[ ]`	0	0
995	`[-1.0, -1.0]`	2	-2.0
996	`[ ]`	0	0
997	`[-1.0, -1.0, -1.0]`	3	-3.0
998	`[-0.5]`	1	-0.5
999	`[ ]`	0	0

From the table above, we can see that our agent is able to reach the destination optimally with 0.0 total penalty most of the time (6 out of the 10 trials). However, it still makes some mistakes along the way (either by not following the *U.S. Right-of-Way rules* (penalty of -1.0) or taking the wrong direction (penalty of -0.5).

Interestingly the optimal policy doesn't always agree with the `self.planner.next_waypoint()` function. When the function returns a `next_waypoint` that respects the *U.S. Right-of-Way rules*, this is the best action to take regarding the optimal policy. However, when this action does not follow the rules of the road, the best action is `None`. `None` will not incur any penalty, while any other action will return a penalty either -0.5 or -1.0. So, it

won't take long so that the **smartcab** can finally follow the `next_waypoint` suggestion in the next iterations when its state changes.