Train a Smartcab to Drive

April 26, 2016

1 Introduction

A smartcab is a self-driving car from the not-so-distant future that ferries people from one arbitrary location to another. In this project, we will use reinforcement learning to train a smartcab how to drive.

1.1 Environment

Our smartcab operates in an idealized grid-like city, with roads going North-South and East-West. Other vehicles may be present on the roads, but no pedestrians. There is a traffic light at each intersection that can be in one of two states: North-South open or East-West open.

US right-of-way rules apply: On a green light, you can turn left only if there is no oncoming traffic at the intersection coming straight. On a red light, you can turn right if there is no oncoming traffic turning left or traffic from the left going straight.

1.2 Inputs

Assume that a higher-level planner assigns a route to the smartcab, splitting it into waypoints at each intersection. And time in this world is quantized. At any instant, the smartcab is at some intersection. Therefore, the next waypoint is always either one block straight ahead, one block left, one block right, one block back or exactly there (reached the destination).

The smartcab only has an egocentric view of the intersection it is currently at (sorry, no accurate GPS, no global location). It is able to sense whether the traffic light is green for its direction of movement (heading), and whether there is a car at the intersection on each of the incoming roadways (and which direction they are trying to go).

In addition to this, each trip has an associated timer that counts down every time step. If the timer is at 0 and the destination has not been reached, the trip is over, and a new one may start.

1.3 Outputs

At any instant, the smartcab can either stay put at the current intersection, move one block forward, one block left, or one block right (no backward movement).

1.4 Rewards

The smartcab gets a reward for each successfully completed trip. A trip is considered "successfully completed" if the passenger is dropped off at the desired destination (some intersection) within a pre-specified time bound (computed with a route plan).

It also gets a smaller reward for each correct move executed at an intersection. It gets a small penalty for an incorrect move, and a larger penalty for violating traffic rules and/or causing an accident.

1.5 Goal

Design the AI driving agent for the smartcab. It should receive the above-mentioned inputs at each time step t, and generate an output move. Based on the rewards and penalties it gets, the agent should learn an optimal policy for driving on city roads, obeying traffic rules correctly, and trying to reach the destination within a goal time.

2 Tasks

2.1 Setup

We need Python 2.7 and PyGame for this project. For help with installation, it is best to reach out to the pygame community [help page, Google group, reddit].

Here is the procedure I followed to install PyGame using brew:

```
brew install mercurial
brew install sdl sdl_image sdl_mixer sdl_ttf smpeg portmidi
pip install hg+http://bitbucket.org/pygame/pygame
```

2.2 Download

Download smartcab.zip, unzip and open the template Python file *agent.py* (do not modify any other file). Perform the following tasks to build our agent, referring to instructions mentioned in README.md as well as inline comments in *agent.py*.

2.3 Implement a basic driving agent

Implement the basic driving agent, which processes the following inputs at each time step:

- Next waypoint location, relative to its current location and heading,
- Intersection state (traffic light and presence of cars), and,
- Current deadline value (time steps remaining),

And produces some random move/action *(None, 'forward', 'left', 'right')*. Don't try to implement the correct strategy! That's exactly what our agent is supposed to learn.

Run this agent within the simulation environment with *enforce_deadline* set to *False* (see run function in *agent.py*), and observe how it performs. In this mode, the agent is given unlimited time to reach the destination. The current state, action taken by your agent and reward/penalty earned are shown in the simulator.

What can we see in the agent's behavior? Does it eventually make it to the target location?

Because the action produced is random, the agent's move is also random. Most of time, it cannot make it to the target location.

2.4 Identify and update state

Identify a set of states that are appropriate for modeling the driving agent. The main source of state variables are current inputs, but not all of them may be worth representing. Also, we can choose to explicitly define states, or use some combination (vector) of inputs as an implicit state.

At each time step, process the inputs and update the current state. Run it again (and as often as you need) to observe how the reported state changes through the run.

Justify why picked these set of states, and how they model the agent and its environment.

A set of states that I think are appropriate for modeling the driving agent is `(self.next_waypoint, inputs["light"], inputs["oncoming"], inputs["left"])`.

- "next_waypoint" can give us the next step, which is important for the location and relative position to the destination.

- “light” shows us the traffici lights status at each intersection, which tells whether we can take certain action or not.
- “oncomming” gives us the status of oncoming cars, which is important when we try to turn left if traffic light is green, and when we try to turn left when the traffic light is red.
- “left” gives us the status of left side cars, which is important when we try to turn right if the traffici light is red.

We don’t need the “right” category in inputs. Because acorrding to the rules, when the traffic lights are green, we only need to care about the “oncoming” cars when we try to turn left (“next_waypoint”). When the traffic lights are red, we only need to car about the “oncoming” and “left” cars.

With this set of states, the agent can get a correct understanding of its status in the environment, which is important for avoiding potential accidents. Therefore, after implementing a set of states, the reported changes give us more positive rewards than the random case.

2.5 Implement Q-Learning

Implement the Q-Learning algorithm by initializing and updating a table/mapping of Q-values at each time step. Now, instead of randomly selecting an action, pick the best action available from the current state based on Q-values, and return that.

Each action generates a corresponding numeric reward or penalty (which may be zero). Our agent should take this into account when updating Q-values. Run it again, and observe the behavior.

What changes can we notice in the agent’s behavior?

The agent’s behavoir depends on the initial Q values. If I set `self.Q[(state, x)] = 0.0`, the agent tends to repeat its trajectories, and never reach the target point. If I set `self.Q[(state, x)] = 0.5`, the agent tends to repeat its trajectories at most time, while also explores other paths. After 25 trials, the success rate of the agent reaching target within deadline is about 0.2, which implies we need the help of other methods (e.g. ϵ-greedy) to help the agent to learn. When the defualt Q values are set to be 2.0, the agent can learn very fast with a success rate of over 0.9. The effect of default Q values is explained in the following session. The above results are based on the setting that both learning rate and discount factor are 0.1.

2.6 Enhance the driving agent

Apply the reinforcement learning techniques that have learnt, and tweak the parameters (e.g. learning rate, discount factor, action selection method, etc.), to improve the performance of our agent. Our goal is to get it to a point so that within 100 trials, the agent is able to learn a feasible policy - i.e. reach the destination within the allotted time, with net reward remaining positive. The formulas for updating Q-values can be found in this video.

Report what changes made to our basic implementation of Q-Learning to achieve the final version of the agent. How well does it perform?

To enhance the driving agent, I used the ϵ-greedy method to the basic Q-Learning method. Basically, the ϵ-greedy method uses a probability to balance the exploration and exploitation. With probability ϵ, the agent chooses a random action to explore new state. With probability $1 - \epsilon$, the agent remains its greedy exploitation that chooses the action with the maximum rewards. Several parameters were also explored, such as `learning_rate`, `discount_factor`, `epsilon`, and inital Q values, within 30 trails.

sucess rate	epsilon	learning rate	discount factor	initial Q
0.20	0	0.1	0.1	0.5
0.03	0	0.1	0.3	0.5
0.07	0	0.1	0.9	0.5
0.77	0	0.3	0.1	0.5
0.70	0	0.3	0.3	0.5
0.37	0	0.3	0.9	0.5
0.85	0	0.9	0.1	0.5

sucess rate	epsilon	learning rate	discount factor	initial Q
0.45	0	0.9	0.3	0.5
0.03	0	0.9	0.9	0.5
0.82	0.1	0.1	0.1	0.5
0.33	0.1	0.1	0.3	0.5
0.38	0.1	0.1	0.9	0.5
0.73	0.1	0.3	0.1	0.5
0.82	0.1	0.3	0.3	0.5
0.30	0.1	0.3	0.9	0.5
0.83	0.1	0.9	0.1	0.5
0.30	0.1	0.9	0.3	0.5
0.38	0.1	0.9	0.9	0.5
0.90	0.1	0.1	0.1	3.0
0.90	0.1	0.1	0.3	3.0
0.30	0.1	0.1	0.9	3.0
0.90	0.1	0.3	0.1	3.0
0.87	0.1	0.3	0.3	3.0
0.40	0.1	0.3	0.9	3.0
0.90	0.1	0.9	0.1	3.0
0.93	0.1	0.9	0.3	3.0
0.27	0.1	0.9	0.9	3.0

`learn_rate` and `discount_factor` are important parameters for Q-Learning, because they are related to the converage rate and reward acculation. Actually, the default Q-value is also important. Starting with a very high Q-value for actions means that the agent will basically always try out actions it hasn't tried before a couple of times, ensuring that it quickly explores its state space and then has a good picture of the environment. The high learning rate means this won't set it back very far. Having an initial Q-value that is slightly higher than the highest possible reward should be ideal (say, 3 in this problem) though since the environment may be unknown setting it very high and having a high learning rate that slowly decays over time is generally ideal.

After the above experiments, I choose the parameter sets to be 0.1, 0.9, 0.3, and 3.0 for epsilon, learning rate, discount factor, and initial Q values.

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?

The agent can get to the target point within 100 trails. Actually, the agent performs pretty well, and can learn very fast. Generally, the can get close to finding an optimal policy and not incur any penalties.

3 Summary

In this project, an AI driving agent is designed for the smartcab. It receives inputs such as next waypoint location, traffici lights status, and dealine time, at each time step t, and generate an output move. Based on the rewards and penalties it gets, the agent learns an optimal policy for driving on city roads, obeying traffic rules correctly, and tries to reach the destination within a goal time. During this process, the Q-Learning and ϵ-greedy methods are implemented to help the agent reaches the above ability.