Train a Smartcab to Drive

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, you will use reinforcement learning to train a smartcab how to drive.

Environment

Your 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.

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.

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.

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).

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.

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.

Tasks

Setup

You need Python 2.7 and **pygame** for this project: https://www.pygame.org/wiki/GettingStarted For help with installation, it is best to reach out to the pygame community [help page, Google group, reddit].

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 your agent, referring to instructions mentioned in `README.md` as well as inline comments in `agent.py`.

Also create a project report (e.g. Word or Google doc), and start addressing the questions indicated in *italics*below. When you have finished the project, save/download the report as a PDF and turn it in with your code.

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 your 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.

In your report, mention what you see in the agent's behavior. Does it eventually make it to the target location?

Since the action was randomly chosen, the agent wasn't optimizing the future action from the past rewarding actions. If we set the 'enforce_dealine' to be false, it will reach the target finally, but with too many unnecessary steps. If we set the 'enforce_dealine' to be true. The agent always failed running out of times.

Identify and update state

Identify a set of states that you think 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, you 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.

There are three places where we could find the state update. The first is next waypoint output from planner, as one action is complete, and then the next waypoint begins. The second is often in the process of intersection. A cab needs to handle traffic lights and oncoming traffic so as to choose the right path and avoid accident. The third is counting down the deadline.

In the agent2.py, we are going to update with traffic rules in the intersection. We could notice that it agent is able to reach the target following the rules. Besides, the reward for each state is always 0 or positive.

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). Your agent should take this into account when updating Q-values. Run it again, and observe the behavior.

What changes do you notice in the agent's behavior?

The Q_Learning implementation is based on Travis DeWolf in https://github.com/studywolf

In the agent3.py, we replaced the update of the traffic rules with the Qlearn algorithm. The learning process happens in the iterative update of Q value between the current state and the next state.

Unlike the agent2.py where each state always has a positive reward, agent3 makes some mistakes by having negative rewards. But the agent3 could reach the target with the help of Q-Learning update.

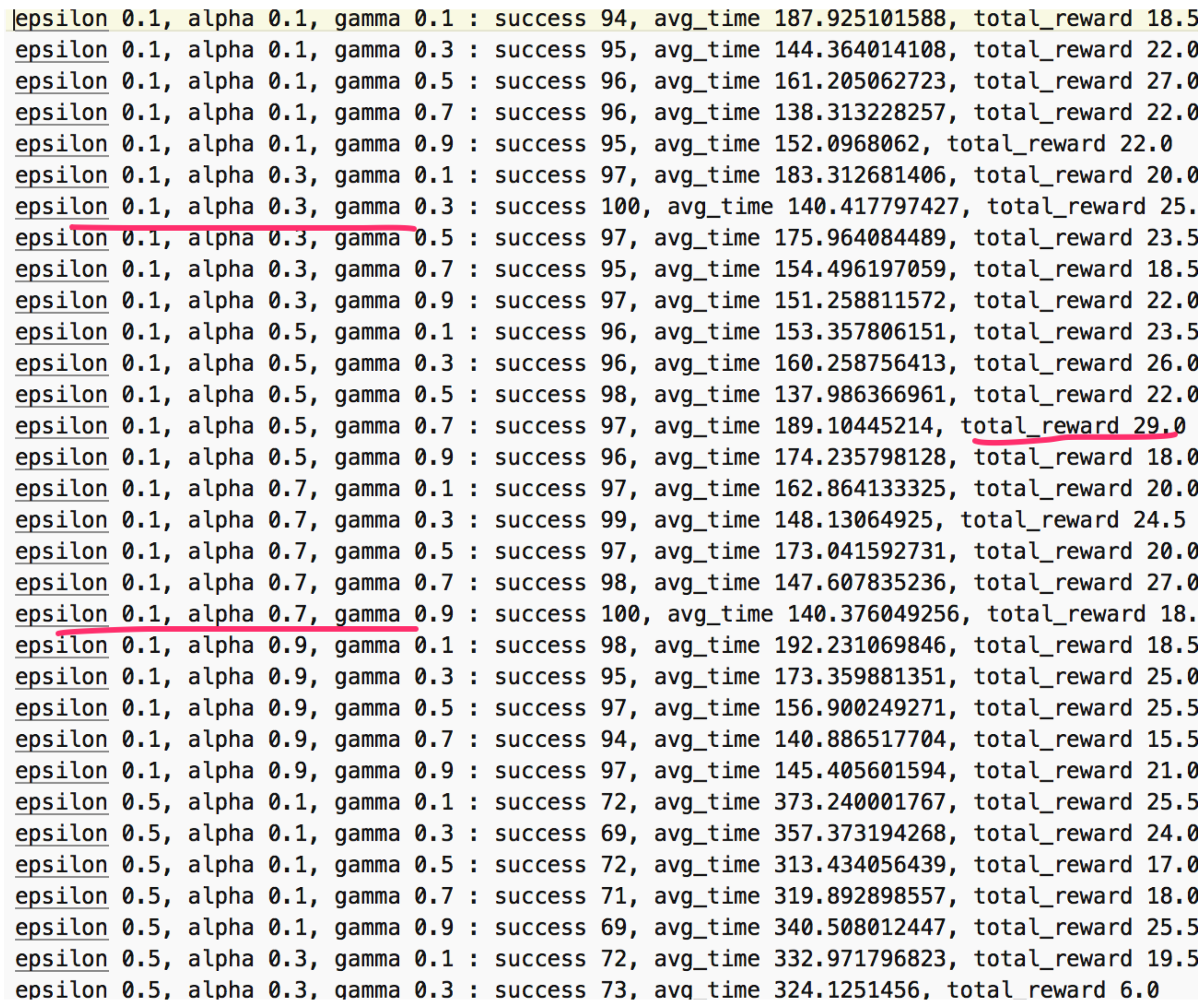
Enhance the driving agent

Apply the reinforcement learning techniques you have learnt, and tweak the parameters (e.g. learning rate, discount factor, action selection method, etc.), to improve the performance of your agent. Your 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.

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

In the agent4.py, we would test combinations of learning rate alpha, discount factor gamma and exploration epsilon. In order to count the successful trials, a 'success' tag is added in the 'enivronment.py' to record if one trial successful reaches one target. And then a counter is added in the 'simulator.py' to count the number of successful trials of 100 trials with each combination of parameters. A running time counter is also added in the 'simulator.py' to count the average running time. We also calculate net rewards for each 100 trials.

The tuning results are output into the file 'qlearning_tuning_report.txt'. The top combinations of parameters are shown as follows:



```
epsilon 0.1, alpha 0.1, gamma 0.1 : success 94, avg_time 187.925101588, total_reward 18.5
epsilon 0.1, alpha 0.1, gamma 0.3 : success 95, avg_time 144.364014108, total_reward 22.0
epsilon 0.1, alpha 0.1, gamma 0.5 : success 96, avg_time 161.205062723, total_reward 27.0
epsilon 0.1, alpha 0.1, gamma 0.7 : success 96, avg_time 138.313228257, total_reward 22.0
epsilon 0.1, alpha 0.1, gamma 0.9 : success 95, avg_time 152.0968062, total_reward 22.0
epsilon 0.1, alpha 0.3, gamma 0.1 : success 97, avg_time 183.312681406, total_reward 20.0
epsilon 0.1, alpha 0.3, gamma 0.3 : success 100, avg_time 140.417797427, total_reward 25.
epsilon 0.1, alpha 0.3, gamma 0.5 : success 97, avg_time 175.964084489, total_reward 23.5
epsilon 0.1, alpha 0.3, gamma 0.7 : success 95, avg_time 154.496197059, total_reward 18.5
epsilon 0.1, alpha 0.3, gamma 0.9 : success 97, avg_time 151.258811572, total_reward 22.0
epsilon 0.1, alpha 0.5, gamma 0.1 : success 96, avg_time 153.357806151, total_reward 23.5
epsilon 0.1, alpha 0.5, gamma 0.3 : success 96, avg_time 160.258756413, total_reward 26.0
epsilon 0.1, alpha 0.5, gamma 0.5 : success 98, avg_time 137.986366961, total_reward 22.0
epsilon 0.1, alpha 0.5, gamma 0.7 : success 97, avg_time 189.10445214, total_reward 29.0
epsilon 0.1, alpha 0.5, gamma 0.9 : success 96, avg_time 174.235798128, total_reward 18.0
epsilon 0.1, alpha 0.7, gamma 0.1 : success 97, avg_time 162.864133325, total_reward 20.0
epsilon 0.1, alpha 0.7, gamma 0.3 : success 99, avg_time 148.13064925, total_reward 24.5
epsilon 0.1, alpha 0.7, gamma 0.5 : success 97, avg_time 173.041592731, total_reward 20.0
epsilon 0.1, alpha 0.7, gamma 0.7 : success 98, avg_time 147.607835236, total_reward 27.0
epsilon 0.1, alpha 0.7, gamma 0.9 : success 100, avg_time 140.376049256, total_reward 18.
epsilon 0.1, alpha 0.9, gamma 0.1 : success 98, avg_time 192.231069846, total_reward 18.5
epsilon 0.1, alpha 0.9, gamma 0.3 : success 95, avg_time 173.359881351, total_reward 25.0
epsilon 0.1, alpha 0.9, gamma 0.5 : success 97, avg_time 156.900249271, total_reward 25.5
epsilon 0.1, alpha 0.9, gamma 0.7 : success 94, avg_time 140.886517704, total_reward 15.5
epsilon 0.1, alpha 0.9, gamma 0.9 : success 97, avg_time 145.405601594, total_reward 21.0
epsilon 0.5, alpha 0.1, gamma 0.1 : success 72, avg_time 373.240001767, total_reward 25.5
epsilon 0.5, alpha 0.1, gamma 0.3 : success 69, avg_time 357.373194268, total_reward 24.0
epsilon 0.5, alpha 0.1, gamma 0.5 : success 72, avg_time 313.434056439, total_reward 17.0
epsilon 0.5, alpha 0.1, gamma 0.7 : success 71, avg_time 319.892898557, total_reward 18.0
epsilon 0.5, alpha 0.1, gamma 0.9 : success 69, avg_time 340.508012447, total_reward 25.5
epsilon 0.5, alpha 0.3, gamma 0.1 : success 72, avg_time 332.971796823, total_reward 19.5
epsilon 0.5, alpha 0.3, gamma 0.3 : success 73, avg_time 324.1251456, total_reward 6.0
```

The performance varied according to the combinations of parameters. But the combination with the lowest epsilon of 0.1 has the highest the success rates, lowest average running time, and the highest net rewards.

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?

I think my model finds an optimal policy as shown in the previous question. There are two combinations of parameters having the 100% success rates. The highest reward is 29. The net rewards are almost positive.