Smartcab Training

Reinforcement Learning Project

Ihab Sultan

Contents

1. Introduction

1.1 Project Overview

The goal of this project is to use reinforcement learning to train a smartcab to make the right decisions at intersections, and to reach its goal before timer expiry.

1.2 Software and Libraries

The following SW was used in the first part of the project:

- Python 2.7
- pygame

The Legrand Orange Book by Mathias Legrand used under CC BY-NC-SA 3.0
Cab by Ours Blanc used under CC BY-ND 2.0
New York Cabs by Richard Schneider used under CC BY-ND 2.0

2. Project Description

2.1 Problem Setup

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.

2.2 Inputs

- **Route**: Waypoints at each intersection, where next waypoint is always either one block straight ahead, one block left, one block right, one block back or exactly there (reached the destination).
- **Traffic lights**: To check if green for the direction of movement (heading)
- **Cars at the intersection**: Includes direction they want to go.
- **Trip timer**: 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.

2.3 Outputs

- **Action**:At any instant, decide whether smartcab should stay put at the current intersection, move one block forward, one block left, or one block right (no backward movement).

2.4 Rewards

- **Large reward**: successfully completed trip - passenger is dropped off at the desired destination (some intersection) within a pre-specified time bound (computed with a route plan).
- **Small reward**: correct move executed at an intersection.
- **Small penalty**: incorrect move.
- **Large penalty**: violating traffic rules and/or causing an accident.

3. Project Report

3.1 Implement a basic driving agent

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

Applying random actions using:

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

makes the cab move in random directions. Cab manages to find target after a long series of trials, and losing rewards here and there.

Given that the car moves on a 6×8 grid, there is $1/48$ chance that the car will be at the right position. If enforce_deadline was enabled, the cab would fail to find its targets on most of the trials.

If, instead, only the direction towards new waypoint was applied:

```
action = self.next_waypoint
```

The cab would reach its target in fewer steps, albeit it shall lose multiple rewards along the way by not performing the optimal action.

3.2 Identify and update state

Identify a set of states that you think are appropriate for modeling the driving agent. 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.

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

States contain updated info needed by the agent to identify the best action to be taken. As such, states should contain all of the following:

- **Next waypoint**: This state models the driving angent, and has the following values: one block straight ahead, one block left, one block right, one block back or exactly there (reached the destination).
  Next waypoint helps the agent identify next step towards the goal, ideally this would be the optimal action to take if there were no other constraints.
- **Traffic lights**: This state is related to environment surrounding the agent (not the agent itself), and has two values: green or red.
  Traffic light needs to be included in states for the agent to learn whether it should obey the traffic light or not, and what would be the reward or penalty for such actions.
- **Cars at the intersection**: This state models the environment surrounding the agent, and includes the direction other cars at the intersection want to go.
  Including this information in the state will help the agent learn how it should react when a different agent behaves in a certain way at the intersection (ie. road rules).

Following is how the state is represented in code:

```
self.state = (self.next_waypoint, inputs['light'], inputs['oncoming'],
                  inputs['left'], inputs['right'])
```

3.3 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.

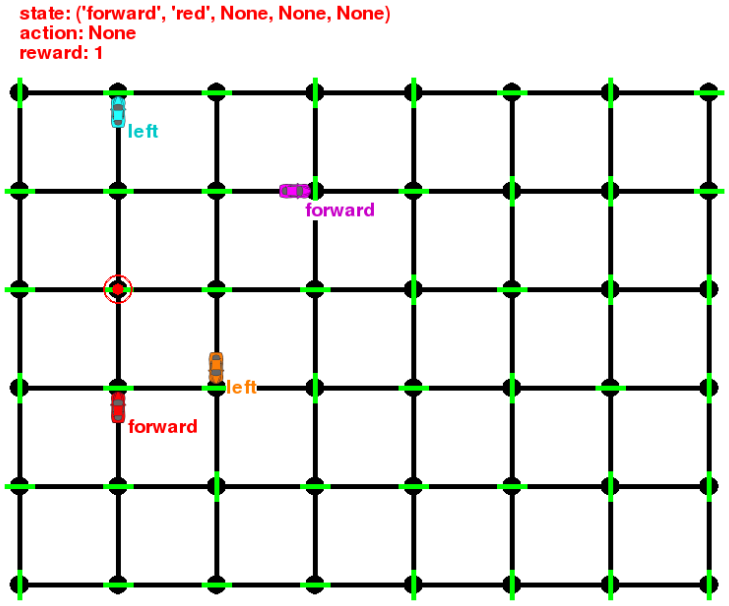
What changes do you notice in the agent's behavior?

After some random actions, the agent is able to learn proper behavior. The speed at which

proper action for a given state is discovered depends on the number of times the agent has visited the state.

Visiting the same state frequently gives the agent more opportunities to explore different actions from that state, and hence the agent will be able to learn the optimal action to take from there.

As an example, one can notice that agent quickly discovers that it should stop at a red traffic light if it wants to move forward to the next waypoint.



3.4 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?

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?

- **Learning Rate**:

  First tried a fixed value for learning rate, and it went surprisingly well, though this causes Q estimate to not converge to actual Q since $\sum_{t=1}^{\inf} \alpha^2$ is infinite (ie our estimate is biased).

This could be understood if we assume that finding the optimal policy does not mandate an unbiased estimate of Q value.

Final implementation of the project used a proper learning rate that is the reciprocal of number of updates per (state, action) pair. This in return requires storing the number of updates per (state, action) in a separate python dictionary.

- **Discount factor**:

  Modifying discount factor gives some interesting results. This is because in this project, the largest reward is delayed tell the end of the game, when the agent arrives at the destination.

  At high discount factors (≤ 0.5), the agent could capture unexpected behaviors such as turning right at a traffic light instead of waiting when next waypoint is forward. At the end, the agent does reach the goal but not in expected ways, missing on few rewards along the way.

  Final implementation of the project used a discount factor of 0.1, this worked nicely and avoided unexpected behaviors mentioned above.

- **Action selection**:

  At first trial, actions were selected probabilistically according to latest Q value estimates. The results were wrong since suboptimal actions would have a considerable Q value even after many iteration.

  Last implementation picks the action with the largest Q value. All state,action pairs were initialized with a large Q value in order to force the agent to explore at the beginning, and exploit later.

  It would be interesting to apply simulated-annealing here, and randomly choose whether to explore or exploit at each step with decaying random action probability (ε), this is the "Greedy Limit with Infinite Exploration" method. However the results were satisfactory with the method mentioned above.

- **Initial Q-value**:

  As mentioned above, this value needs to be high to force the agent to explore different actions. Multiple values were tried, and found that results were not specially sensitive to this value (which is to be expected given the rewards options in this specific project), final chosen initial value was the maximum reward of 10.

  In general, one could leave the agent running with random actions in the environment for a while, and then initialize Q value to be larger than the maximum observed reward for any (state, action) pair.

3.5 Final agent performance

We expected the optimal policy for the agent to be like: Move towards target while obeying the rules of the road.
As an example, the agent should stop at red traffic lights if it wants to move forward or left, it should move towards target considering the proposed next waypoint, and it should give the right of the way to other drivers whenever applicable.

Final agent policy matched the expected optimal policy mentioned above, and agent was able to reach target successfully with maximum rewards.

4. Q value Analysis

This is not part of the project, but an addendum for analysis of Q-values after running the Q-learning algorithm for some time

4.1 Sample analysis: Green light

Below is a sample for Q-values for a state containing a green light, and next waypoint is in-front of the agent as well, we expect best action to be moving forward, which agrees with the Q-values captured below:

Waypoint	Light	Oncoming	Left	Right	Action	Qvalue	Iterations
forward	green	None	None	None	None	1.50	2
forward	green	None	None	None	left	1.00	2
forward	green	None	None	None	**forward**	3.69	29
forward	green	None	None	None	right	1.00	2

One can notice from the table above that a single iteration was enough to rule out remaining actions in this case.

The second example below shows a similar result with left waypoint:

Waypoint	Light	Oncoming	Left	Right	Action	Qvalue	Iterations
left	green	None	None	None	forward	1.00	2
left	green	None	None	None	right	0.83	2
left	green	None	None	None	**left**	4.28	6
left	green	None	None	None	None	1.50	2

4.2 Sample analysis: Red light

Our following analysis is for Q-values in case of red light:

Waypoint	Light	Oncoming	Left	Right	Action	Qvalue	Iterations
left	red	None	None	None	left	-0.50	2
left	red	None	None	None	right	1.00	2
left	red	None	None	None	forward	-0.50	2
left	red	None	None	None	**None**	1.15	13

And the final one captures another interesting observation, that our agent knows it can turn right on a red traffic light.

Waypoint	Light	Oncoming	Left	Right	Action	Qvalue	Iterations
right	red	None	None	None	forward	-0.50	2
right	red	None	None	None	left	-0.50	2
right	red	None	None	None	None	1.50	2
right	red	None	None	None	**right**	2.21	9