Train a Smartcab to Drive: Project Report

Shilpak Banerjee

July 5, 2016

1 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'). Dont try to implement the correct strategy! Thats 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.

TO DO: In your report, mention what you see in the agents behavior. Does it eventually make it to the target location?

Answer: At the very begining I made the agent choose a random action at each intersection. I achieved this via the command

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

which in our set-up is equivalent to the following command suggested by the reviewer,

```
action = random.choice(self.env.valid_actions)
```

Predictably the agent in this case almost never reaches the destination (see table 1, last row) when

```
enforce_deadline = True
```

However if one sets the above to False, the performance of the agent improves somewhat however in most of the cases the agent is still unable to reach its destination as the algorithm hits the hard time limit of -100.

One may note that in the existing code one can replicate the the aforementioned situation by setting $\alpha =$ anything, $\gamma =$ anything, $\epsilon = 1$ and enforce_deadline = False.

2 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.

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

Answer: I choose the combination of *next_waypoint, traffic light status, heading/presence of oncoming traffic and heading/presence of traffic to the left of the smartcab* as my state. The traffic to the right of the smartcab do no affect its decision making process in any way, so I skipped it. A valid argument can be made to include the deadline as an indicator of state. However the following two reasons persuaded me to avoid using it:

- Including the deadline will greatly increase the number of states and hence the agent will visit each state with less frequency (100 trials is not enough) and hence it wont train very well.
- I do not want my smartcab to disobey a traffic rule or crash [1] to make it to the destination in time. May be the fare can also be a indicator and if Bill Gates is paying, this indicator should be the only indicator of the state :D.

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.

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.

TO DO: What changes do you notice in the agents behavior?

Answer: With Q-learning implemented, the agent takes an action and then updates the Q value and then the policy based on the reward obtain and on the next intersection if uses the policy with probability $1 - \epsilon_t$ and hence this produces a much better opposed response than just randomly taking an action.

For a detailed analysis we refer to table 1 where we use the following to measure the efficacy of our choice of parameters:

[1] Also after completing the project, it seems to me that there is no severe penalty for crashing or it does not happen very often because of low traffic/low number of trails. So next waypint and traffic light will probably make a better choice of states since this lower number of states will also increase the number of times each state is visited during the trails.

- **Average reward/trial**: We run 100 trials of our algorithm and then record the net reward for the last 10 of those trials and then repeat this entire procedure for 10 times and take the average. This gives a good measure of the expected reward earned by the agent.
- **Average penalty/trial**: We run the entire 100 trials of our algorithm and then record the negative rewards (penalties) for the last 10 of those trials and then repeat this entire procedure for 10 times and take the average. This gives a measure of the expected penalty earned by the agent.
- **Average number of moves/trial**: We run the entire 100 trials of our algorithm and then record the average number of moves for the last 10 of those trials and then repeat this entire procedure for 10 times and take the average. This gives a measure of the expected number of moves executed by the agent.
- **Average number of success out of last 10 trials**: We run the entire 100 trials of our algorithm and then record the no of times the smartcab reaches its destination out of the last 10 of those trials and then repeat this entire procedure for 10 times and take the average. This gives a measure of the success rate of the agent.

So now we explain how the parameter were chosen in the Q-learning algorithm. Since we measure our success from the last 10 trials, it is appropriate to consider the first 90 trials to be the training part of the algorithm. So we set $\alpha = \epsilon = 1$ for the first 90 trials and then for the last 10 trials or the evaluation phase of the algorithm we use a very fast decaying α and ϵ [2]. This way we switch from heavy exploration to heavy exploitation very quickly. So the only parameter that remains to be tuned is γ and we use table 1 to select $\gamma = 0.10$ for its superior performance in the category of rewards and success rate.

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.

TO DO: 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?

Answer: I implemented Q-Learning using the following update formula:

$$Q(s_{t-1}, a_{t-1}) = (1 - \alpha_{t-1})Q(s_{t-1}, a_{t-1}) + \alpha\big(R + \gamma \max_{a_t \in A(s_t)} Q(s_t, a_t)\big)$$

Here s_{t-1} is the state at time $t-1$, a_{t-1} is the action taken at time $t-1$. $A(s_t)$ is the list of all available actions when agent is at state s_t. In our case, $A(s_t) = \{\text{None, forward, left, right}\}$. This way the agent is able to learn a policy within the allocated 100 trials. See Table 1 and the discussion is the last paragraph of the previous section for a detailed discussion of how we obtained an optimal set of parameters.

[2] Thanks to the reviewer for making this suggestion

γ	Reward/trial	Penalty/trial	No of moves/trial	No of success out of last 10 trials
0.90	21.435	2.925	20.600	6.2
0.70	23.150	3.060	20.030	6.0
0.50	27.770	2.760	16.940	8.1
0.30	28.425	2.065	15.080	9.2
0.10	23.185	0.390	12.640	9.9
0.00	21.245	0.295	12.500	9.2
random	0.430	11.570	27.030	1.8

Table 1: Parameter tweaking table: I played aroud with different values of the parameter γ. Also in the last row I listed the values when the agent acts randomly with enforce_deadline set to True. Note that the 2nd, 3rd and 4th column is measured after the taking the average of the last 10 trials and the then those averages are further averaged by running the algorithm 10 more times. Last column is averaged over 10 runs of the algorithm.

TO DO: 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?

Answer: Yes it does comes close to finding an optimal policy. An optimal policy would be to go directly to the next waypoint without violating any traffic rules. The reason our smartcab behaves almost optimally is that the most important states seems to be the (next_waypoint, traffic light) combination which the agent learns very well (8 possibilities and 4 actions, 100 trials with average deadline of 20 or so seems enough for visiting 32 possibilities several times). Since the number of traffic is very low and combined with the fact that the number of trials is also vey low, our smartcab does not always learn not to violate traffic rules and crash in another car. Also I do think that penalty of violating a traffic rule in presence of another traffic (higher accident chance) should be set much higher. Another funny thing I noted: Since our city is a torus, right turn on red is on average the correct decision.

However one should note that there are times when our smartcab behaves suboptimally, infact one may consider this to be a fatal mistake. For example on one run of the program, I got

$$\pi^*\big(('\text{left}', '\text{green}', '\text{forward}', \text{None}), '\text{left}'\big) = '\text{left}'$$

which means that the optimal policy we found dictated the smartcab to turn left when an oncoming car has a forward heading. The q-value for this was -1 and the reason this happened is because this state was visited exactly once during all of the 100 trials and hence the agent did not have time to learn.