smartcab

October 29, 2016

1 Report: Smartcab Project

by Oliver Tacke (October 29, 2016)

1.1 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.

1.1.1 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?*

1.1.2 ANSWER

As we would expect, the agent showed completely erratic behavior. Still, even with random actions represented by $\alpha = 1$, $\gamma = 0$ and $\epsilon = 1$ averaged over 100 runs each with 100 training trials (`n_times=100`), the agent can reach its destination quite often. In its best run it fails in only about 31 cases by hitting the hard time limit (-100):

```
RESULTS FOR 100 RUNS WITH 100 TRIALS EACH
Highest success rate is 0.6821 for
    alpha=1,
    gamma=0, and
    epsilon=1.
with average traffic violations per cab ride: 21.30
with total net reward over all 100 trials: -278.94
with average moves above optimum per cab ride: 73.74
```

We should also note the high average number of traffic violations per cab ride: 21.30. Good for a Hollywood movie, but probably not for a smart cab. And the violations didn't even pay off, because our total reward after all the trials is negative.

Anyway, given $8 \times 6 = 48$ intersections and the option to do nothing, I expected the cab to fail more often.

Naturally, the results deteriorate if we set `enforce_deadline` to `True`. The cab would reach the destination in about 21% of the cases at best:

```
RESULTS FOR 100 RUNS WITH 100 TRIALS EACH
Highest success rate is 0.2141 for
    alpha=1,
    gamma=0, and
    epsilon=1.
with average traffic violations per cab ride: 7.41
with total net reward over all 100 trials: 40.78
with average moves above optimum per cab ride: 21.59
```

Judging by the other data that I checked, we can see that we're doing our cab a favor with pulling it off the street.

1.2 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.

1.2.1 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?*

1.2.2 ANSWER

Naively, we could use all the `inputs` containing information about the light and possible cars and their directions, the next `waypoint`, and also the `deadline` to constitute states. Since `deadline` can take up lots of different values, it would dramatically increase the number of states and thus training time. If we'd like to increase the size of the grid later, we'd even be in a worse situation.

Of course, we could also think of something more complicated like $\lfloor log_3(deadline) \rfloor$. This approach would not only reduce the number of states resulting by changes in `deadline`, but we could also tackle the importance of the remaining time: the state would change more often the smaller the remaining time becomes. Anyway, this is should also be possible by adjusting α, γ, and ϵ according to the `deadline`.

The next best thing would be 5 variables with 2 or 4 possible values:

- $light \in \{red, green\}$
- $left \in \{None, forward, left, right\}$
- $oncoming \in \{None, forward, left, right\}$
- $right \in \{None, forward, left, right\}$
- $waypoint \in \{None, forward, left, right\}$

Having a closer look at the traffic laws given for this project, we can see that `right` doesn't give us important information: - If there's a green light (given that other cars obey the traffic rules), we'd only have to check for oncoming traffic that might cross our path if we're turing left. - If there's a red light and we go left or forward, then we will definitely be punished regardless of other cars. - If there's a red light and we turn right, then only cars from our left would be relevant.

In conclusion, `right` is redundant and we can stick with `light`, `left`, `oncoming` and `waypoint`.

1.2.3 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?*

1.2.4 ANSWER

The approach taken using `light`, `left`, `oncoming` and `waypoint` generates $2 \times 4 \times 4 \times 4 = 128$ different states possible. This will allow the agent to make good decisions, but it will also learn quite slowly. For Q-Learning, there are many states that have to be visited multiple times in order to learn the value of each action possible, so we must also take these into account, resulting in 512 different fields for our Q table.

Still, given the 100 trials set for this assignment, it should work quite well. The variable `light` has only to possible values but gives us a lot of indicators for driving well. Combining it with `waypoint` with 4 different values for moving quickly will also give us good directions. They're our "principal components" that will do most of the work within the algorithm. For our environment with only three other cars, we could probably even get away with only these using 8 states thus learning quickly, but with more cars the probability of crashes would increase and we'd probably want to avoid these at all costs (at least in Germany which seems to become technophobe judging by the reactions after the resent accient by a Tesla car).

The most extreme alternative (besides driving randomly) would probably be to ignore everything but `waypoint`. We'd only have 4 states, and given the small amount of traffic this might even work - but I'd probably not use such a smartcab in real life, and we better don't come across a cop…

1.3 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.

1.3.1 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?

1.3.2 ANSWER

To me it is not completely clear how α, γ and ϵ should be set for this first run, so I simply used `random.random()` to generate random values for each of them. I also had a look at the results before setting `enforce_deadline` to `True`. In most cases, even with these random values the performance increased dramatically. In some cases, the agent wouldn't fail anymore. That was to be expected, because now the agent learns over time and uses the rewards as guidance for his actions instead of driving randomly. Furthermore, the agent violates traffic rules less frequently and moves to the destination quicker. Learning is also accountable for this, because the agent will receive large negative feedback for violating rules and also negative feedback for deviating from the next waypoint suggested by the planner. I also noticed that the agent tended to only fail in eary trials if at all. That's because it takes some time to fill in all the "Q values" for a state, but after that's done: boom. We could achieve the same with increasing the number of trials, of course, which would give the algorithm more time for exploring all possible states and computing the utilty for those depending on the action we'd take.

As an example, here's just one result for different random combinations of α, γ and ϵ also averaged over 100 runs:

```
RESULTS FOR 100 RUNS WITH 100 TRIALS EACH
Highest success rate is 0.9889 for
    alpha=0.813522964366,
    gamma=0.699908332339, and
    epsilon=0.442052994264.
with average traffic violations per cab ride: 4.13
with total net reward over all 100 trials: 2377.79
with average moves above optimum per cab ride: 26.74
```

After these trials, I set `enforce_deadline` to `True` and simply set α and γ to 0.5 and ϵ to 0.25, so the agent would still randomly cruise the streets in about 25 % of all moves, moderately consider previous visits and moderately take into account the utility of the next state. In 100 runs with 100 iterations each, the average success rate was about 80 % - still better than the random approach even though now we have a deadline:

```
RESULTS FOR 100 RUNS WITH 100 TRIALS EACH
Highest success rate is 0.7988 for
    alpha=0.5,
    gamma=0.5, and
    epsilon=0.25.
with average traffic violations per cab ride: 1.29
with total net reward over all 100 trials: 2198.30
with average moves above optimum per cab ride: 12.71
```

1.4 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.

1.4.1 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?

1.4.2 ANSWER

In order to find the best combination of α, γ and ϵ, I quickly implemented a grid search. I lazily started it with a set of $\{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$ for each parameter and averaged the success rate of 100 runs for each combination, each with 100 trials for training (`n_times=100`). A feasible problem like ours can still be tackled fairly well this way ;-) After quite some time of waiting, the result was:

```
RESULTS FOR 100 RUNS WITH 100 TRIALS EACH
Highest success rate is 0.8718 for
    alpha=0.1,
    gamma=0.1, and
    epsilon=0.1.
```

```
with average traffic violations per cab ride: 0.49
with total net reward over all 100 trials: 2040.79
with average moves above optimum per cab ride: 10.79
```

Afterwards, I explored the space around this solution a little more diligently by setting $\alpha \in \{0.03, 0.07, 0.1, 0.13, 0.17\}$, $\gamma \in \{0.03, 0.07, 0.1, 0.13, 0.17\}$ and $\epsilon \in \{0.03, 0.07, 0.1, 0.13, 0.17\}$. The final result was:

```
RESULTS FOR 100 RUNS WITH 100 TRIALS EACH
Highest success rate is 0.871166666667 for
    alpha=0.07,
    gamma=0.1, and
    epsilon=0.13.
with average traffic violations per cab ride: 0.50
with total net reward over all 100 trials: 2035.64
with average moves above optimum per cab ride: 10.86
```

Since the results are even slightly worse (there's still some variation in the results even with averaging over 100 runs), "I call it an agent". Not considering the way we're measuring sucess (see final answer for more on that), with $\alpha = 0.1$, $\gamma = 0.1$, and $\epsilon = 0.1$, we have a success rate of roughly 0.87, average traffic violations of 0.49 per cab ride and 10.79 moves above optimum per cab ride.

Would have been much easier to just increase the number of trials some more and use the final q tables, but we were restricted to `n_trials=100`.

1.4.3 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?

1.4.4 ANSWER

An optimal policy could be "Safety first, but then be quick!", and I think the agent is pretty close to that. It could need some more training than "only" 100 trials, but the states chosen will guarantee that following a "safety first" mindset can be achieved. To make sure about that, ϵ should be set to 0 either after a certain number of trials or by using a function that converges to 0 instead of a fixed value. This way, there really wouldn't be any random driving after a certain amount of time, but the agent would solely rely on the q tables from the rewards it got. We should probably use this agent "version" to judge the success rate.

One could argue that the average penalty per cab ride is really small but not zero. However, a closer look at the data tells us that those penalties primarily occur in early trials when the agent is still learning and sometimes later on when he chooses to explore state space by a random move. This could also be prevented by letting ϵ converge to zero after a certain amount of time. Again, we should probably use the fully trained agent to judge the number of penalties per cab ride.

Also, one might argue that it is not yet optimal because the number of moves the agent needs still exceed the distance between two intersections, but we must consider red lights that force us to wait or use a detour. In consequence, reaching the target in minimum time is really only possible if there's no red light, so this is hardly possible.

I found that even an agent perfectly obeying the traffic rules would fail sometimes because of bad luck with the traffic lights. Interesting… But we probably wouldn't want to risk ignoring a red light even when time is short and there's no traffic, would we?

We're close to the end now… ;-) If we use the best combination of parameters that we have found so far for our best solution (see Appendix A), let the agent train in 1 run with 100 trials for training that yields really good results (mind the variation), then store the q table as default, and finally set ϵ to 0, we should now be measuring the "final" performance after 100 trials and it should be good:

```
RESULTS FOR 1 RUNS WITH 100 TRIALS EACH
Highest success rate is 1.0 for
    alpha=0.1,
    gamma=0.1, and
    epsilon=0.
with average traffic violations per cab ride: 0.00
with total net reward over all 100 trials: 2224.00
with average moves above optimum per cab ride: 6.60
```

Pretty neat smartcab! Still, it could fail sometimes… Just increase `n_runs` and see for yourself… Bad luck with traffic lights can mess up your ride.

1.5 Appendix A (q table of best solution)

This is the q table that was found using alpha=0.1, gamma=0.1, epsilon=0.1, and n_times=100.

It's a dictionary that uses a tupel of a state (tupel of `light`, `oncoming`, `left`, and `waypoint`) and an action as key.

```
{
(('red', None, None, 'right'), 'left'): -0.9461915040873616,
(('green', 'left', 'forward', 'forward'), None): 0.0,
(('green', 'left', None, 'forward'), 'forward'): 2.0769662559368993,
(('green', None, 'left', 'right'), None): 0.0,
(('red', 'left', None, 'forward'), None): 0.0,
(('green', None, None, 'right'), None): 0.06003990223563961,
(('red', None, None, 'forward'), 'forward'): -1.0,
(('red', 'right', None, 'forward'), None): 0.0,
(('green', None, None, 'forward'), 'left'): -0.3825676325917002,
(('red', 'forward', None, 'right'), 'forward'): -1.0,
(('green', 'left', 'left', 'forward'), None): 0.0,
(('red', None, 'forward', 'right'), None): 0.0,
(('red', 'left', None, 'forward'), 'forward'): -1.0,
(('red', None, None, 'forward'), 'right'): -0.28984611501191493,
(('red', None, 'forward', 'forward'), None): 0.0,
(('red', None, None, 'left'), 'forward'): -1.0,
(('green', None, 'forward', 'forward'), None): 0.0,
(('green', None, 'right', 'forward'), 'right'): -0.5,
(('green', 'left', None, 'right'), None): 0.0,
(('red', None, None, 'right'), None): 0.06664317301401293,
(('red', None, 'left', 'forward'), None): 0.0,
(('red', 'left', None, 'right'), None): 0.0,
(('green', 'left', None, 'forward'), None): 0.0,
(('green', None, None, 'right'), 'forward'): -0.45476794876879145,
(('red', None, None, 'left'), 'right'): -0.4623040027650762,
(('red', None, None, 'left'), 'left'): -1.0,
(('red', None, None, 'left'), None): 0.0,
(('green', None, None, 'left'), None): 0.14350871184045882,
(('red', None, None, 'right'), 'right'): 3.2047056750995457,
(('red', None, 'right', 'left'), None): 0.0,
(('green', 'left', None, 'forward'), 'right'): -0.5,
(('red', 'right', None, 'right'), None): 0.0,
(('green', None, None, 'forward'), None): 0.22886213824736912,
(('green', None, None, 'forward'), 'forward'): 4.162929168271038,
(('green', 'right', None, 'left'), None): 0.0,
(('green', None, None, 'forward'), 'right'): -0.4182159915856784,
(('red', None, 'right', 'forward'), None): 0.0,
(('red', 'left', None, 'forward'), 'left'): -1.0,
(('red', None, None, 'right'), 'forward'): -0.9204180100584439,
(('green', None, None, 'left'), 'right'): -0.4639989927242393,
(('green', None, None, 'left'), 'forward'): -0.482,
(('red', 'left', 'right', 'forward'), None): 0.0,
(('green', None, 'forward', 'left'), 'right'): -0.5,
(('green', None, 'left', 'forward'), None): 0.0,
(('green', None, None, 'right'), 'right'): 2.6992010514084797,
(('green', 'left', None, 'right'), 'forward'): -0.5,
(('red', 'forward', None, 'right'), 'left'): -1.0,
(('green', 'forward', None, 'left'), None): 0.0,
(('red', None, None, 'forward'), 'left'): -1.0,
(('red', 'forward', None, 'forward'), None): 0.0,
(('green', None, None, 'left'), 'left'): 5.428300368904872,
(('green', None, None, 'right'), 'left'): -0.45395326785025214,
(('red', None, None, 'forward'), None): 0.0,
(('green', 'right', None, 'forward'), 'right'): -0.42548484225145505,
(('red', 'forward', None, 'right'), None): 0.0,
(('green', 'left', None, 'left'), 'left'): 2.022234215001157,
(('red', 'right', None, 'left'), None): 0.0,
(('green', 'left', 'right', 'forward'), 'right'): -0.5
}
```