Smartcab Project Report

by Moritz Bendiek

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.

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?

ANSWER: The requested behavior (random choice of action disregarding any input information) is realized by adding the following code to the `update` method:

```
action_list = [None, 'forward', 'left', 'right']
action = action_list[random.randint(0,3)]
```

Note that `random` should be initialized (via `seed`) first.

The agent executes a "random walk". Due to the limited size of the world model (6*8=48 positions on the grid), the smartcab will arrive at its destination within a reasonably short time span. I executed 100 test runs and found that the smartcab arrived at the destination before the hard time limit (-100) was reached 63 times, and 37 times the agent ran into the hard time limit.

When switching `enforce_deadline` to `True` and running 100 sets of runs, I find an **cumulative success rate of 21.8%**. This should be a baseline for future improvement of the model.

The smartcab apparently is influenced by the traffic lights, insofar as it does not always move when it sees a "red" light. However, judging from the visualization, I am not sure that this behavior is consistent. Also, the movement of the smartcab somedoes does not seem to fit to the chosen action (e.g. displays 'left' as action, but drives forward).

Inform the driving agent

TASK: 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.

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?

ANSWER: While intuitively the position of the smartcab on the grid should be part of its state, I think I can omit it as in order to learn to drive it is not relevant for the smartcab where it is, as all traffic rules apply equally in all places.

To differentiate between situations that require different behavior of the agent, while minimizing the explicit input of domain knowledge (traffic rules), it is instructive to use the inputs as "seen" by the smartcab.

The problem I want to avoid here is to model an excessive number of states, as this might prohibit the agent to learn sufficiently fast in the given situation. Therefore I'd like to balance my choice of states between allowing general enough learning and keeping the number of states reasonably small.

My choice is to use the following information to generate states:

- In which direction shall the smartcab move? Here, we can use `self.next_waypoint` directly. It can be any of `['left', 'forward', 'right']`
- What does the traffic light show? (`inputs['light']: 'red', 'green'`)
- What traffic is coming onward (`inputs['oncoming']: None, 'left', 'forward', 'right'`)
- What traffic is coming from the left (`inputs['left']: None, 'left', 'forward', 'right'`)

This yields 3 *2* 4 * 4 = 96 states, which already seems like a lot to me in the given situation. Note that I did not include information on traffic coming from the right (`inputs['right']`), as it is never needed to make a decision. I realize this is already a piece of domain knowledge, that I put "manually" into the model, but in terms of the above mentioned trade-off, I see this as justified.

If I were to reduce the number of states further, I would also want to get rid of the `inputs['left']` and `inputs['oncoming']` information, as they should play a role much more infrequently than the other two items.

```
self.state = (self.next_waypoint, inputs['light'], inputs['oncoming'], inputs['l
eft'])
```

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?

ANSWER: The total number of states is the Cartesian Product of all the dimensions, in this case 3 * 2 * 4 * 4 = 96 states.

The goal of Q-Learning is to move through all states infinitely (read: *very*) often. As in the model there are 100 test runs, and each test run consists of a capped number of individual moves (even with `enforce_deadline` being set to `False`, a run stops after hitting the hard deadline of -100), we can broadly assume that with 50 moves per run, we will visit 5000 states. However it is likely that the probability to be in a specific state can vary (i.e. is not uniform). In particular, some combinations will most likely never appear (like traffic from both oncoming and left is more unlikely than no traffic from any direction.

Implement a Q-Learning Driving Agent

TASK: 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.

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?

ANSWER: Out of 100 test runs (alpha = 0.3, alphadecay = 0.3, gamma = 0.3, epsilon = 0.05, epsilondecay = 0.5 - see below section for an explanation of the paramters!) the smartcab reached its destination in *most* of the cases. The result is quite stable between sets of runs. While I find on average (based on 100 sets of runs) the smartcab reaches its destination in **96.4%** of the runs.

As I analyzed 100 sets of test runs (with the parameter setting stated above), I found that

- in the first 10 runs (of each set) the average success rate is **86.6%**
- in the first 50 runs (of each set) the average success rate is **94.6%**
- in the last 50 runs the average success rate is **98.1%**.
- in the last 10 runs the average success rate is **100%**

This indicates that indeed a learning effect is present.

When looking at the change of behavior within one set of 100 test runs, it becomes apparent that zero rewards are apparently common in early runs, while the become rarer in later runs. This behavior is consistent with the notion that the Q-Matrix is initialized with a uniform value which in early stages effects a more or less random choice of actions, which often have a zero reward, whereas in later stages enough learning has happened to pick positively rewarded actions more frequently.

Analyzing the occurrence of (average) negative rewards per trial I see (with the above parameter settings) that...

- in the first trial, on average 9.25 negative rewards occur per trial
- in the first 10 trials, on average 1.60 negative rewards occur per trial
- in the last 10 trials, on average 0.455 negative rewards occur per trial

This shows that negative rewards are common in very early learning phases, but quickly the agent learn to avoid them.

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.

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?

ANSWER: In order to optimize the model, there are several hyper-parameters to tune. In particular, I have implemented a variable learning rate `alpha`, and a mechanism that decays `alpha` over time course of the trial runs. The `decay_alpha` parameter (`curr_alpha = alpha / ((cycle_count)**decay_alpha)`) is also tuned. The identical decay mechanism is implemented for `epsilon`, the additional hyper-parameter is called `decay_epsilon`.

The tuning for the parameters `alpha, decay_alpha, gamma, epsilon, decay_epsilon` is done by writing some loops around the exisiting code in the `run()` method, which in essence performs a grid search:

```
for alpha in [0.1, 0.2, 0.3]:
        for decay_alpha in [0.0, 0.3, 0.6  ]:
            for gamma in [0.3, 0.4]:
                for epsilon in [0.05, 0.10]:
                    for decay_epsilon in [0.0, 0.5,]:
```

In addition, to obtain some meaningful statistics, every combination is repeated 100 times (so for any given hyper-parameter combination, there are 100 * 100 test runs).

The analysis was done by writing the output of `agent.py` to a file and using a python script to determine success rate for each hyper-paramter combination. For the result, only the last 10 of 100 runs was used. The assumption is, that after 90 runs, the model has learned enough to achieve good results.

The result is shown in below table (only worst and best results). Note that the maximum of the success rate appears with `alpha=0.3; decay_alpha=0.3; epsilon=0.05; decay_epsilon=0.5; gamma=0.3` and is 100% success rate. Even the poorest result is well over 90% (92.3%) success rate.

It is not very clear from the results whether decaying alpha and/or epsilon really helps to increase performance, as there are hyper-parameter settings with disable either one or the other and still achieve near-perfect results (>99% success rate).

See below table:

alpha decay_alpha decay_epsilon trial cumul_reward gamma negative rewards success rate

	alpha	decay_alpha	decay_epsilon	epsilon	gamma	negative rewards	success rate
62	0.3	0.3	0.0	0.10	0.4	1700	0.923
54	0.3	0.0	0.0	0.10	0.4	1548	0.944
48	0.3	0.0	0.0	0.05	0.3	805	0.950
10	0.1	0.3	0.0	0.10	0.3	1231	0.953
59	0.3	0.3	0.5	0.10	0.3	632	0.954
8	0.1	0.3	0.0	0.05	0.3	738	0.955
30	0.2	0.0	0.0	0.10	0.4	1353	0.958
...	...	...	...	....	...	...	...
3	0.1	0.0	0.5	0.10	0.3	642	0.991
60	0.3	0.3	0.0	0.05	0.4	622	0.993
42	0.2	0.6	0.0	0.10	0.3	1120	0.994
55	0.3	0.0	0.5	0.10	0.4	893	0.995
32	0.2	0.3	0.0	0.05	0.3	501	0.995
21	0.1	0.6	0.5	0.05	0.4	290	0.995
35	0.2	0.3	0.5	0.10	0.3	646	0.996
40	0.2	0.6	0.0	0.05	0.3	863	0.997
27	0.2	0.0	0.5	0.10	0.3	828	0.999
57	0.3	0.3	0.5	0.05	0.3	455	1.000

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?

ANSWER: The agent *does* indeed get close to finding an optimal policy. The general success rate should be close to 100% and the occurrence of negative rewards should be close to zero. Both does hold true, which suggests that the learning happended as desired. From a visual inspection of the agent's behavior, I also found that the agent moves toward the destination whenever traffic rules allow it.

The average cumulated reward per trial is **13.1** in the first trial, and it rises very quickly (see below table, which was obtained with hyper-parameter settings as shown above and averaging with 100 sets of trials). In the last 10 trials the average cumulated reward per trial is **22.6**.
(see also above section with detailed reports on negative rewards).

trial	cumul_reward
1	13.080
2	18.775
3	20.765
4	18.375
5	20.870

trial	cumul_reward
6	16.385
7	20.665
8	18.840
9	19.385
10	18.885
11	20.965
12	19.875
13	20.940
14	22.355
15	20.965
16	21.695
17	23.810
18	21.085
19	21.330
...	...
93	22.680
94	22.165
95	22.755
96	23.300
97	22.210
98	22.610
99	23.685
100	21.605

An optimal policy should be to move in a direction which reduces the distance to the target and to obey all traffic rules (i.e. don't move when not allowed to move in *any* direction that reduces distance to target.)

An improvement to the reward mechanism might be to reward the agent more for obeying the traffic laws than to reach its destination when time is low (as it is certainly more desirable to reduce the risk of an accident while accepting a delay, that to risk an accident only to get to the destination on time).

```
In [ ]:
```