Project 4: Train a Smartcab how to drive

Submitted by Benjamin Tanz
benjamintanz@gmx.de
December 25, 2015

1) IMPLEMENT A BASIC DRIVING AGENT

I implemented the basic driving agent in the function 'choose_action'. This function returns an action using one of the implemented decision rules. The caller can request a specific decision rule by setting an indicator (planner_type). Implemented decision rules are random choices, a greedy decision rule, a randomized decision rule and a rule for the 'naive reward matrix learner' (Discussed in the appendix). This function has access to the class variables that hold the state, actions and the Q-matrix (as well as the reward matrix for the reward matrix learner).

Observing the random agent, it is no surprise the Smartcab sometimes does and sometimes does not obey the traffic rules. Also, by chance, it hits the destination from time to time. The random agent is also useful for observing the rewards of various actions across various states. For example, an action that obeys the rules and chooses the right direction generally receives a reward of 2. An action that obeys the rules but fails to go in the right direction receives a reward of 0.5 and staying put generally leads to a reward of 1. Not obeying the rules results in negative rewards.

My numerical simulations showed that the agent hits the destination on average after 117 moves (standard deviation of estimate 11.4; 100 trials).

2) IDENTIFY AND UPDATE THE STATE

To represent the state of the Smartcab, I used the Cartesian product of the following current inputs:

- **Current waymark**
- **Traffic light color**
- **Direction oncoming traffic**
- **Direction left traffic**

The current waymark is used because the optimal decision in each state should clearly depend on this waymark. Also, the agent needs to know the color of the traffic light as well as the direction of cars oncoming and left to be able to choose/learn an action that obeys traffic rules.
Perhaps more interesting is the discussion of variables not contained in the state. First of all, keeping the state space small generally has great benefits in terms of both time and space requirements for the algorithm and is therefore desirable (curse of dimensionality). Hence, it is good to keep the state space as small as possible. First, the direction of traffic from the right is not included in the state space. Based on the traffic rules, cars from the right never impact the agent in any way and hence don't need to be included in the state (assuming those cars obey the traffic rules). Moreover, the problem does not have a determined finite horizon, since it is not clear when the agent will reach the target. Hence,

the optimal policy will be stationary and the time/deadline value does not need to go into the state.
As far as implementation is concerned, I implemented the Q-matrix as a numpy matrix with each element of the aforementioned Cartesian product as row and each feasible action as column.

3) IMPLEMENT Q-LEARNING

I implemented Q-learning by setting up the Q-matrix as discussed in the last section. I initialize the Q-matrix with a value of five for each state-action combination. For choosing an action, I used a simple rule that just returns the best action at a given state according to the Q-matrix. I used the numpy argmax function to retrieve the best action. This function breaks ties automatically (by choosing the first of a set of equal values). For updating the Q-matrix, I used the ruled derived in the lecture.

Even though the agent's behavior still looks random initially, one can observe that the agent slowly starts to learn over time and stays put if the traffic light is red and the direction not right etc. For any state that is visited first, the agent always chooses the first action in the set ('forward' in this case). When the state is visited again, initially, it will almost certainly choose a different action because of the observed reward and the updated Q-matrix. After some time, the agent converges to the optimal action and always chooses this action in the future. Also, the implementation reveals that convergence may take some time, since some states are rarely visited (and others, of course, very often).

4) ENHANCE THE DRIVING AGENT

I did two sets of adjustments for the driving agent. One set deals with the parameters of Q-learning and is discussed here. The other set is implementing a method that learns the rewards matrix (assuming the problem is not a dynamic decision making problem) and is discussed in the appendix.

To make sure the agent is actually exploring in the first iterations, I initialized the Q-values to 30 for each state-action combination. This strategy is often termed 'optimism in the face of uncertainty' and ensures sufficient exploration of all available actions in any state. I then run experiments with various settings for learning rate and discount factor. From the experiments, I learned that a rather low discount factor works best (0.05). With regard to the learning rate, I observed that a high learning rate leads to rapid learning in the first trials, but also leads to several "late" (number of trials > 70) trials where the agent does not reach the destination in the allotted time. With a low learning rate, it takes longer until the algorithm stabilizes, but learning appears to be more robust (agent almost always reaches destination in later trials). Further exploring this tradeoff, I found that a learning rate of 0.35 works well.

Since the agent rarely reaches the destination in time in the first trials, but is almost always on time in the last trials, it is clear that learning occurs. However, from time to time, there are cases where the agent does not make it on time. I tried two tweak parameters, but these cases did not disappear. If the trials are random, this can for example happen when the agent is stuck in front of a red traffic light and can not move

towards the goal for a longer time. Also, it may be that the modelled state omits something important about the problem. For example, if the state would contain some notion of location relative or distance to the final destination and the deadline, the agent may start to sacrifice compliance with traffic rules in order to move faster to the final destination. In any case, overall the agent learns and starts to almost always reach the destination on time after some trials.

APPENDIX: A NAIVE REWARD MATRIX LEARNER

Based on the hypothesis that the Smartcab-problem is not actually a dynamic decision problem that needs to be modelled with an MDP (a question I asked on the forum, but was unanswered at the time of submission: https://discussions.udacity.com/t/utility-of-next-state-in-q-learning/42030/5 (entry nr. 6)), I implemented a naive reward matrix learner that ignores the dynamics of the problem. The algorithm works as follows:

- **Initialize matrixes R (rewards) and V (visited state-action combinations)**
- **Do until goal reached:**
    - **Let nature draw state**
    - **Lookup state in R and V**
    - **If rewards have been collected for all actions at this state:**
        - **Choose optimally given current rewards**
    - **Else**
        - **Choose the next action that was not yet explored in this state and store reward in R and update V**

It turns out that this algorithm converges extremely quickly to a solution that almost always reaches the destination in time. In the code, this planner can be activated setting the planner_type variable equal to 4.