# P4: TRAIN A SMARTCAB TO DRIVE

ANDY PEREZ

## 1. Implementation of a basic driving agent

Since we are initially setting the move/action to be entirely random (using python's random number generation to choose one of the four possible actions), it's not surpising that the car moves rather randomly and erratically. Turns are as likely as moving forward, so it tends to snake around quite a lot. It can take a considerable amount of time, but the car does eventually reach the target location, although it never reached it on time during my runs.

## 2. Identify and update state

The set of states I chose to model the environment are as follows, along with a justifaction for each.

- Orientation of traffic light at intersection (N-S or E-W)

  When at an intersection and making a decision as to whether proceed or wait, it's vitally important to know whether the light is red (and hence you have to wait else incur a negative reward) or green (in which case you can proceed forward). There are no other variables besides inputs['light'] that can convey this information, thus it's vital to include it.

- Whether there is oncoming traffic, traffic to the left, or traffic to the right

  These are all related so instead of considering them separately I've put them all under one item. It is important to know the position of other cars at an intersection before proceeding wth an action. According to the rules of the road, one needs to be aware of other cars at the intersection before making a decision so that they can be given the right of way if necessary, particularly when one is attempting to make a turn.

- The next waypoint

  There needs to be some way of including in the status variables some sort of guidance as to which direction the car is to go, or else it'll have no reason to move towards the destination square as opposed to some random square. This is the direction we want to go and that we want the car to go unless there's a good reason not to, so hence it should be included.

- Whether the deadline is imminent

  This is another factor that affects how one should drive. Since the reward of reaching the desired end state is larger than the possible positive or negative rewards in absolute value at any intersection, it may be the case that when a car is near the goal but almost out of time it should be willing to incur short-term negative rewards for a better shot at the large reward of reaching the goal. Since the negative reward is $-1$, I chose to use whether

the deadline is within 6 or not, for the environment isn't very large, and there's little gain to be had to start rushing and incurring negative rewards before, lest the negative rewards incurred outweigh the positive end reward.

The environment is modeled as an ever-changing intersection with the use of these variables. Cars appear, lights change, and waypoints change as given by the actual full environment outside these variables. However, this myopic worldview is sufficient, since the car does not need to see past the string of intersections or know it's position on the map in order to make its decisions except for figuring out how to get to the destination. However, that is already taken care of by the planner, which takes into consideration the bearing and the relative positions of the car and the destination. There isn't any other data that we can use unless we have control over the planner too and use reinforcement learning on the planner as well.

## 3. Implement Q-Learning

The agent is much more effective at reaching the target after implmenting Q-learning, at least after going through a large amount of rounds of learning. Initially, it's still going to be mostly random, with the relatively high Q initialization values causing it to be very exploratory. It still doesn't reach the target a lot of the time, but that's still a massive improvement. It seems to go in circles sometimes for no particular reasion, making four right turns in a row sometimes.

## 4. Enhancing the Driving Agent

First of all, the *deadline_approaching* variable was removed from the list of states that Q learning operates on. The reason being that there were already too many states, and having this extra state doubled the number of possible states. It would be worth using it, possibly, if there was a way to couple states where *deadline_approaching* was True/False so that learning Q in one affected Q in the other. If it's a good idea to normally make a right turn in a certain situation, it's at least a decent idea to do so when a deadline is approcahing as well. The car arrived at its destination quite before the target time almost always though, so there wasn't much learning on the proper Q values for states with this variable set to True, so I just removed it.

Furthermore, I reduced the initialized default values for Q from 10 to 4. It turned out that setting them to 10 slowed down the convergence of Q to it's actual values by too much. Because the learning rate went as $\alpha = 1/t$, by the time some of the rarer state-action pairs were being accessed, the learning rate was too slow to reduce the 10s to anywhere near their actual values. However, when Q is nearer the actual expected values of states, which tend to be around 2-3, then we create a model that is still very exploratory, but not too much so to the point of not learning the good choices in time. I reduced $\epsilon$ , the frequency with which random actions are taken, from .1 to .06, although this didn't have much effect except to make it so that there are less random actions in the model once it's been trained. Having epsilon go even lower but start higher by allowing it to vary with $t$ would have been appropriate as well, but just having it set to a constant sufficed for our purposes.

Rare state-action pairs were, well, rare. And because of this, since $\alpha$ was a globally set variable, sometimes $\alpha$ would decrease too quickly to let us meaningfully learn how to act on rare states. This was especially a problem when Q was initialized

to 10, and although this problem was mitigated when Q was initialized to a much lower value, it is still something that was able to be improved. Thus, I made $\alpha = \alpha(S, A)$, the learning rate was made to be a function of the state-action pair, where it went as $1/5t$, except where $t$ here was the amount of times the state-action pair $(S, A)$ was visited as opposed to how many times states were visited in general. It makes no sense not to learn how to handle a novel state from the first occurance of a novel state just because it happened to occur 2000 t steps in. I multiplied $t$ by a constant, however, so that for common states there wasn't too much of a big difference from how it was before.

My agent definitely gets quite close to finding an optimal policy! After the 100 trials of learning, it gets to the destination very efficiently (at least as efficiently it can reach the destination using the built-in planner) and rarely incurs penalties. From many runs of the final code, it has never failed to reach the destination in time with positive reward. Typically, in the 10 test runs after the 100 trials, it only makes a suboptimal stop maybe four to five times, occasionally because of the constant $\epsilon$ value resulting in a random wrong action, on other occasions because of a novel state that not much learning has occured on. There are 1536 state-action pairs, and most of them are uncommon (involving multiple cars at an intersection), so we can't expect the agent to learn every one of them, mostly just the more common ones.