Project 4

Train a Smartcab to Drive

By Robert Lutz

April 23, 2016

Rev 1.0

1) Implement a Basic Driving Agent

This most basic driving agent makes no attempt to learn. It just executes some random action at each intersection. This agent, while not particularly useful, is a good starting point for building an agent that can actually learn about its environment.

This agent is implemented in *agent.py* as `class StochasticAgent`. At every intersection it just selects an action at random from a predefined list (a tuple actually). We code it as follows:

```
action = ('forward','left', 'right', None)[random.randrange(4)]
```

The agent as expected moves about randomly. It rarely (<10% of the time) reaches the destination before the deadline of 20 to 55 turns (the deadline varies from run to run). It often blows the deadline spectacularly, by a factor of 10x the allotted number of turns or more. But we expect this: it must stumble across the destination by chance, and there are 48 intersections for it to explore.

2) Identify and Update State

Now let's identify the most critical bits of information that we shall think of as the state in which our agent finds itself. The agent's sensor inputs of it's immediate surroundings at the intersection are perhaps the most important. The sensory inputs consist of the color of the traffic light and the location and intents of any other cars at the intersection. But are they all necessary? Clearly we need the color of the light, as running a red light would be very bad. What about oncoming cars? Well, if the light were green and the agent wanted to go left, knowledge of oncoming cars would be critical so we include it. If the light were red, and our agent wanted to make a right turn it would need to be aware of cars entering the intersection from its left and going forward, so we keep that information too. What about cars entering the intersection from its right? If the agent wants to go straight ahead, that would imply that it has a green light and the car to its right has a red and must in all cases wait for the agent to clear the intersection. If the agent wants to turn left, that would again imply it had a green light and the car to its right must either wait for the light to turn green or safely make a turn to the right. If our agent wants to turn right, then again the presence of the car at right is irrelevant as it does not interfere with our agent's progress. The car to the right does not matter to our agent, so we do not include it in the agent's state.

There are some other elements of information that are important in describing the state of the agent. Now the information available to the agent is insufficient to tell it precisely where it is in the world, or even where it is precisely with respect to the destination. But it does have a piece of information called a waypoint, i.e., an immediate destination that is one intersection away from the intersection at which the agent finds itself. The waypoints indicate to the agent the general direction in which it must travel to get to the destination. In fact, this is the only way the agent can get to the destination without randomly stumbling across it. Having said that, proceeding directly from waypoint to waypoint may not always be optimal strategy for the agent to follow.

Finally we must consider the deadline. Perhaps there will occur some situations in which the agent would be 'wise' to alter its behavior based upon the number of moves remaining. If we want to be as comprehensive as possible we would include the deadline in the state. But in doing so we will greatly increase the number of unique states by a factor of 55 or more and will most certainly slow down and most probably needlessly complicate the learning process. For this reason we choose not to include the deadline in the state.

This agent is implemented in *agent.py* as `class DimAwarenessAgent`. Like the `StochasticAgent` it still proceeds randomly, but this agent is superior in that it has explicit knowledge of that state it is in. The state is encoded as follows

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

Notice that we are encoding the state as a tuple rather than a list. This is because we will eventually want to look up the state in our agent's policy to determine its actions. We could conveniently encode the policy as a dictionary with the state as key; thus the state would need to be an immutable object. A tuple fulfills the immutability requirement whereas a list does not.

3) Implement Q-Learning

We're now going to attempt to have our agent learn by using an algorithm called Q-Learning. Q-Learning is a technique for enabling the agent to learn in real-time what to do in any situation it might encounter. The technique is predicated on the definition of the Q-value, which is a measure of the long-term reward for taking some action. To put it simply, when the agent finds itself in state *s* it will choose the action *a* that has the highest (estimated) Q-value. From this a policy can be specified that explicitly tells the agent what to do when in any state *s*. All of this is done in real-time as the agent moves about and acts in its world, so the concept of 'best thing to do' will change and in fact get closer to the ideal as the agent interacts with its environment.

The Q-Learning procedure is complex and its implementation requires a little explanation. The procedure for finding an estimate of Q is an iterated process defined by the following equation

$$\widehat{Q}(s,a) \leftarrow (1-\alpha)\,\widehat{Q}(s,a) \;+\; \alpha\left(r+\gamma\, max_{a'}\widehat{Q}(s',a')\right)$$

Where $\widehat{Q}(s,a)$ is the estimated Q-value for the current state and action (notice that it's the quantity that gets updated during the iteration), $\widehat{Q}(s',a')$ is the estimated Q-value for some action *a'* in the subsequent state *s'*, *r* is the reward obtained for performing action *a* in state *s*, γ is a factor that discounts the value of all future rewards such that the Q-value does not increase without bound, α is the learning rate. And $max_{a'}$ means we take the maximum of $\widehat{Q}(s',a')$ out of all possible actions *a'* . The Q-value is essentially the discounted expected reward available to the agent if it makes all the right moves from that point in time forward. By re-assessing Q every time the agent performs an action, it can eventually develop a highly accurate estimate of the true value of Q for taking any action *a* when in any state *s*.

Now it should start to become apparent how Q-value information might be useful. If the agent finds itself in state *s* with an number of potential actions to take, it could pick the one with the highest Q-value in order to maximize its long-term benefit. This is the essence of true learning

by experience, codified in a mathematical algorithm, and is the basis of our implementation of the agent's experience: the policy. We update the policy every time we update Q. If our agent, through experience, finds that some action is associated with a negative reward, like going forward when the light is red, the Q-value for that action will decrease--perhaps going negative--and some other action will overtake it as the policy default. Likewise, if an agent finds that in some state some action leads to a positive reward, then the agent will come to perform that action every time it finds itself in that state.

This agent is implemented in *agent.py* as `class QLearningAgent`. We begin by initializing Q for every possible state-action pair as well as choosing a default action for every state in the policy. We choose to initialize all Q values to 0, and have the agent move forward by default. We also choose without much consideration the discount factor γ to be 0.9 and the learning rate α to be 0.2.

The `QLearningAgent` definitely performs better than the `StochasticAgent`. It learns pretty quickly not to violate traffic laws, and is about thrice as likely as the `StochasticAgent` to reach the destination. This is what we expected Q-Learning to do. Actions leading to penalties are discouraged and actions leading to rewards are encouraged.

However we can do better. One of the problems is thus: the reward scheme is not optimized to compel the agent to find the destination. For example, if the agent simply turns right at every intersection it will be rewarded with something like 3.5 points each loop. There is no incentive to discontinue this behavior, so it could conceivably get stuck in an infinite loop. Or suppose it decides to do nothing at a green light; it gets one point for this, so through reinforced behavior it might sit there forever. There needs to be a penalty such that every action that does not bring the agent closer to its goal is (mildly) discouraged. That's how we'll improve the `QLearningAgent` in the next section.

4) Enhance the Driving Agent

We now wish to find the optimal policy. In order to find the optimal policy, we must first define it. First and foremost the optimal policy must ensure that the agent follows all traffic laws. As mentioned in the previous section, the `QLearningAgent` already does this quite well so no further action is required. The optimal policy must ensure that the agent arrives at its destination every trial, and that it arrive there as quickly as possible (forgoing violations of traffic law in the course of getting there, naturally). So improving the success rate, and minimizing the number of turns required to reach the destination are the areas we need to work on.

There are a number of things we could do to improve the agent's performance. We could introduce Greedy Exploration, which would encourage the agent to try new things at the beginning. Or we could introduce some randomness into the policy, so that the probability of selecting a given action was proportional to exp(Q). But it seems that, if we want to encourage more exploration, the simplest thing to do is to penalize the agent for any action that does not get it closer to its goal. So rather than getting 1 point for sitting at a red light, it should lose a little; a negative reward of -0.25 might encourage the agent to take a right turn instead on the chance it might get closer to its destination. Rather than going into the environment and

changing the reward structure (which would be cheating, by the way!), we simply dock the agent *p* points in every Q-Learning iteration like this

$$\widehat{Q}(s,a) \leftarrow (1-\alpha)\,\widehat{Q}(s,a) \;+\; \alpha\left(r - p + \gamma\, max_{a'}\widehat{Q}(s',a')\right)$$

Doing so will give many actions a negative effective reward. The idea is that this will encourage exploration without requiring an overhaul of the policy structure. If the Q-value for the default action in a certain state drops below zero, the agent is automatically encouraged to try something new: something which has a Q-value of zero because its never been tried; something that just might turn out to be the optimal action.

This agent is implemented in *agent.py* as `class TrueLearningAgent`. Aside from the introduction of the penalty factor, it is identical to the `QLearningAgent`.

Let's do some experiments to find the optimal value of the penalty factor *p*. We perform one run of the *agent.py* program running the `TrueLearningAgent` for each value of *p* listed in Table 1 below. Each run consists of 10 trials of variable length. The success rate is that fraction of the trials in which the agent has reached its destination. The Avg. Turns till Destination is the average number of turns taken by the agent on trials that were successful.

Table1: Effect of penalty factor on success rate and turns till destination

Penalty	Success Rate	Avg. Turns till Destination
0	0.3	29.0
0.25	0.3	16.7
0.5	0.3	25.0
0.75	0.7	13.4
1.0	1.0	16.6
1.25	**1.0**	**10.3**
1.5	1.0	12.8
1.75	0.9	13.6
2.0	0.8	16.4

We find that a penalty factor on the order of 1.25 points is near optimum for encouraging the agent to find its destination. This penalty factor leads to success approaching universal, and a relatively rapid convergence toward the destination in about 10 turns. Much below this value and the penalty is insufficient to compel the agent toward its goal. Much more than this and the penalty becomes so draconian that it indiscriminately discourages any and all actions. It is likely not a coincidence that the penalty factor of 1.25 lies between the reward of 1 point the agent receives for sitting at a red light (not really worthy of any reward at all, in the views of some),

and the reward of 2 points for making a move closer to its destination. Again, the penalty system is one option among many to improve performance, but we can see clearly that, without modification, the default reward system is one of too many carrots and not enough sticks.

Is the `TrueLearningAgent` a brilliant navigator? Can it fool someone into believing there's a sentient being behind the wheel? No, certainly not. It still does some stupid things, like making three consecutive right turns when it should have waited at a red light then gone left, or sitting idly at a red light when it could get closer to its destination by making a right turn. Greedy exploration might better unearth the optimal actions. But the simple introduction of a modest penalty factor has substantially improved the agent's performance--bringing it very near that which we would expect from the optimal policy--with very little modification to the code.

Bibliography

Udacity, www.udacity.com (see videos related to Machine Learning Nanodegree)