SMARTCAB REPORT

HENRY O. JACOBS

1. Implementing a basic driving agent

I chose to use the variable `next_waypoint` and the light color (taken from the sensor) as a state vector for the agent. The variable `next_waypoint` can take on three values, and the light color can be either red or green. This makes for a total of six states. The space of all six possible states is stored as a member variable `LearningAgent.states` in line 5 in the code-snippet below. I have made this choice because the next waypoint and the light color are the only information used in rewarding the agent (see member function `Environment.act` in `environment.py`). If the actions of the other agents had some bearing on the reward, I should include the other input variables as well. However, they are not, so I wont.

In this document we can symbolize all six states by

$$(\leftarrow, \bullet), (\leftarrow, \bullet), (\uparrow, \bullet), (\uparrow, \bullet), (\rightarrow, \bullet), (\rightarrow, \bullet).$$

My choice for the state-space is implemented in the member function `make_state` on line 49 in the code-snippet below and executed in the update function at line 32. The action at each time-step is chosen randomly line 36.

```
1  class LearningAgent(Agent):
2      #An agent that learns to drive in the smartcab world.
3      actions = (None, 'forward','left','right')
4      from itertools import product
5      states = set( product( ('forward','left','right') , ('red','green')))
6
7      # state-space consists of next_waypoint output and light color
8      def __init__(self, env):
9          super(LearningAgent, self).__init__(env)  # sets self.env = env,
10         #state = None, next_waypoint = None, and a default color
11         self.color = 'red'  # override color
12         self.planner = RoutePlanner(self.env, self)  # simple route planner to get next_waypoint
13         # TODO: Initialize any additional variables here
14         self.Q = {}
```

Date: May 23, 2016.

```
15          for state in self.states:
16              for action in LearningAgent.actions:
17                  self.Q[(state,action)] = randomn()
18          self.explored_state_action_pairs = []
19          self.time = 1 #life-time of agent
20          self.trial = 0 #how many trials have elapsed
21
22      def reset(self, destination=None):
23          self.planner.route_to(destination)
24          # TODO: Prepare for a new trip; reset any variables here, if required
25          self.trial += 1
26
27      def update(self, t):
28          # Gather inputs
29          self.next_waypoint = self.planner.next_waypoint()  # from route planner, also displa
30          inputs = self.env.sense(self)
31          deadline = self.env.get_deadline(self)
32
33          # TODO: Update state
34          self.state = self.make_state( inputs , self.next_waypoint)
35          self.time += 1
36
37          # TODO: Select action according to your policy
38          action = random.choice( self.actions )
39
40          # Execute action and get reward
41          reward = self.env.act(self, action)
42
43          # TODO: Learn policy based on state, action, reward
44          new_inputs = self.env.sense(self)
45          new_next_waypoint = self.planner.next_waypoint()
46          new_state = self.make_state( new_inputs, new_next_waypoint)
47          self.update_Q(  action , new_state , reward, t)
48          if self.time % 10 == 0:
49              self.print_Q()
50
51      def make_state( self, inputs, next_waypoint):
52          return (next_waypoint,inputs['light'] )
```

The output of running this (with `enforce_deadline=False`) is a car that randomly chooses a move at each time step and does a random walk around the city. A snapshot of this is shown in Figure 1.

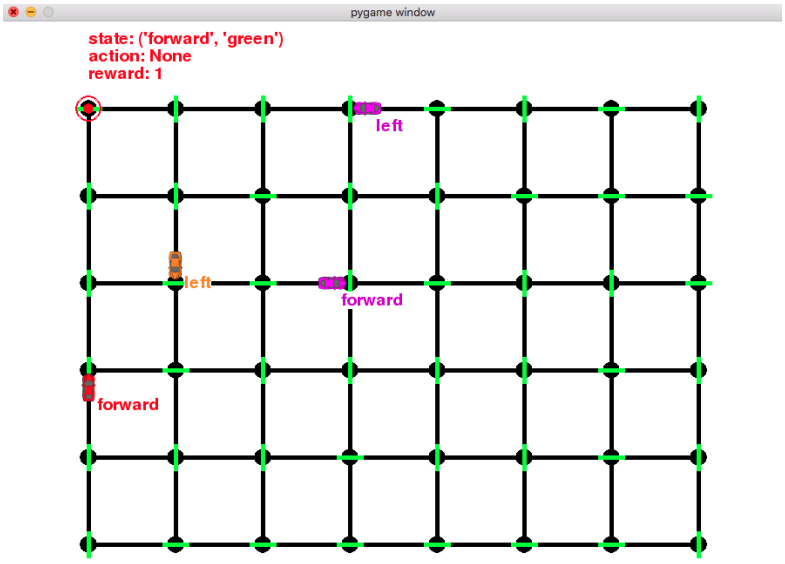

FIGURE 1. A screen shot

The primary agent is colored red. The red text at the top of the image tells us the state, action taken, and the resulting reward at the current time-step. The text next to each car is the `next_waypoint`for each car. In this case, the primary agent is in state ($\uparrow$, $\bullet$) and the variable `next_waypoint` suggests that he go forward. The action he has chosen is `None`, which means he just sits there and does nothing at this time-step. For this he recieves 1 reward (units = Chilean dollars). This random walk strategy "works" only because random walks are ergodic on finite state space and there is no-deadline being enforced. One sample trial of this method took 135 time-steps for the smartcab to reach the destination. This smartcab is pretty dumb.

2. IMPLEMENTING Q-LEARNING

In this section I discuss my implementation of the Q-learning algorithm. We store the Q-function as a hash-table. The keys are (state,action) tuples and the values are real numbers. We initialize all these values randomly in lines 14-17 of the code-snippet above.

To implement Q-learning we store the time in the member variable `time`, on line 19. In this note, we will refer to time as t. This time represents the life time

of the agent and does *not* reset when a new trial begins. It is incremented every time the update function is called (line 33).

We use a learning rate of $\alpha_t = T/(t-T)$ with $T = 20$ and a retention rate of $\gamma = 0.1$. Later we will tune the parameter T. At each time step, we can update the Q-function as

$$Q(s,a) \leftarrow (1-\alpha_t)Q(s,a) + \alpha_t(R(s,a) + \gamma \max_{a'} Q(s',a')) \tag{1}$$

where s' is the state which results from executing a in state s and $R(s,a)$ is the reward at the current time.

In order for $Q(\cdot,\cdot)$ to converge we must execute each action at each state multiple times. This is the crux of the exploration-exploitation dillemma, and it suggests we should spend time exploring the space of state-action pairs before implementing our policy.

2.1. **Exploration vs Exploitation.** A typical approach to the exploration-exploitation dilemma is to define a time-dependent parameter $\varepsilon_t \in [0,1]$ which vanishes as $t \to \infty$. Then at each time, we force the agent to execute an exploration step by randomly choosing an action (as opposed to choosing an action which maximizes our estimate of Q). This is a natural thing to do, even when you are not given any explicit training phase like we are given.

In this assignment an explicitly training period is defined for us. We are asked find a trust-worthy policy within 100 trials. This suggests that after 100 trials our policy should be very good, and so we should focus on exploration during the first 100 trials. The Q-learning algorithm I've chosen to implement uses this freedom to explore by setting

$$\varepsilon_t = \begin{cases} 1 \text{ if } trial < 100 \\ 0 \text{else} \end{cases}$$

In words, we choose our actions randomly for the first 100 trials. After 100 trials we use the optimal policy

$$\pi(s) = \arg\max_a Q(s,a). \tag{2}$$

to determine the action.

The policy (including the random exploration phase) is implemented in the member function `LearningAgent.policy`. The update for Q is implemented in the member function `LearningAgent.update_Q` and is executed in `LearningAgent.update`. The code-snippet below shows my implementation.

```
1     def policy( self , deterministic = False):
2         if self.trial > 100 or deterministic:
3             U = lambda action : self.Q[( self.state , action)]
4             return max( self.actions , key = U )
```

```
        else:
            for action in LearningAgent.actions:
                if (self.state,action) not in self.explored_state_action_pairs:
                    self.explored_state_action_pairs.append( (self.state,action) )
                    return action
            return random.choice( self.actions )

    def update_Q( self,  a , s_new, r,t):
        gamma = 0.1
        time_scale = 20.0
        alpha = time_scale / (self.time+time_scale) #learning rate
        Q_old = self.Q[(self.state,a)]
        Q_future = max( [ self.Q[(s_new,a_new)] for a_new in self.actions] )
        self.Q[(self.state,a)] = Q_old + alpha*(r + gamma*Q_future - Q_old)
```

2.2. **Changes in behavior.** After the hundredth trial, the agent follows the policy π given in (2). At this point in time the agent will always reach the destination before the deadline. We see this from figure 2. During the training phase, the cab usually does not make it to the destination on time (these are the red curves). However, after the first 100 trials, when the cab switches to using the optimal policy π, we observe that the cab always makes it to the destination on time, and the reward never falls below 1 (the blue curves).

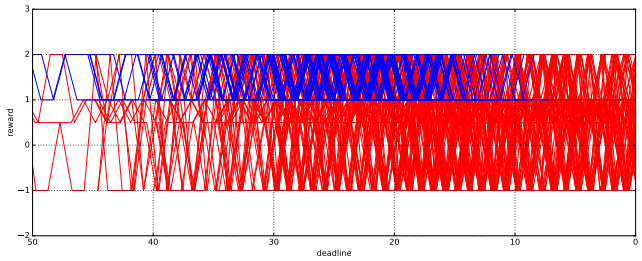

FIGURE 2. A time series of the rewards plotted against time to deadline. Red trajectories denote the training phase, blue denote the trials where the policy (2) is used.

We can glean the policy in (2) from glancing at the Q-function on the 100th trial. Below is a table printed using Python's `tabulate` module. The left column lists the states, and the header lists actions. The entries of the table are the Q-values.

	None	forward	left	right
('forward', 'red')	1.12716	-0.880853	-0.883921	0.579113
('right', 'green')	1.21164	0.709319	0.667372	2.1733

('right', 'red')	1.21819	-0.781176	-0.781479	2.18639
('left', 'green')	1.111	0.543032	1.90831	0.704114
('left', 'red')	0.910603	-0.793709	-0.889615	0.674029
('forward', 'green')	1.09022	2.12282	0.671177	0.584748

We observe that the action which maximizes Q is to follow the directions of `next_waypoint` when the light is green, or when `next_waypoint='right'`, and to choose `None` otherwise.

As γ is small, this outcome makes sense. When γ is small the Q function does not retain much information about nearby decisions in space and time. In this small γ regime, maximizing Q is (roughly) equivalent to maximizing the reward at the current time-step. The reward function, in words, will penalize moving forward or turning left on a red (-1) and will reward legally following the directions given by `next_waypoint` ($+2$). The function has smaller penalties and rewards for obeying the law and ignoring `next_waypoint`. Optimizing this reward function matches the policy induced by the above Q table.

This Q-table might drift, but the policy π is a stable behavior of the system. For example, if we were to perturb the Q-function so that π suggested going forward in state (forward,red), then the reward function would output -1. This negative term would appear in the update for Q, and make $Q((forward, red), forward)$ decrease. We could expect this decrease to continue until π no longer suggests going forward on red-lights.

It is not clear to me that this is the only stable policy. Perhaps there are others. It is also not clear how we should choose the parameters γ and α_t.

3. Parameter tuning

We've already found a policy which meets the requirements of the assignment with $\gamma = 0.1$ and $\alpha_t = 20/(t + 20)$. These parameters allows π to converge to a policy after 100 trials which guides the cab in a way that successfully delivers clients before the deadline every time. However, criticism can be lodged at the fact that I used all 100 trials to train on. Could we train with fewer trials: say 50, or 20... 10?

The optimal policy, π^*, under my interpretation is

$$\pi^*(\uparrow, \bullet) = \emptyset$$

$$\pi^*(\uparrow, \bullet) = \uparrow$$

$$\pi^*(\rightarrow, \bullet) = \rightarrow$$

$$\pi^*(\rightarrow, \bullet) = \rightarrow$$

$$\pi^*(\leftarrow, \bullet) = \emptyset$$

$$\pi^*(\leftarrow, \bullet) = \leftarrow$$

We should be able to see how many trials it takes π to converge to π^* for a given γ. In the following table we count how many trials, n_{trials}, it takes. As this is a random variable, (because the training and initialization of Q are stochastic), I will report the average and standard deviation over 10 trials. If any trial fails to converge for some γ after 100 trials, we mark the data associated to that γ as non-convergent.

γ	n_{trials}	$\sigma_{n_{trials}}$
0.0	6.50	4.86
0.1	4.30	2.00
0.2	5.50	3.44
0.3	11.70	10.56
0.4	not convergent	N/A
0.5	not convergent	N/A

It appears that $\gamma = 0.1$ does a relatively good job and converges quite a bit earlier than the allotted 100 trials given to me.

During this parameter sweep of γ we held the parameter T constant. The parameter T controls the learning rate, α_t, and should also affect how quickly we observe convergence. Qualitatively, α_t controls how large the corrections to the Q-function are at each time-step. When α_t is large these corrections might yield faster convergence, but large jumps also might overshoot a stationary Q, or even yield an unstable algorithm (one where bits of Q to go infinity). So α_t is also something that one should tune, and we can do so by tuning the parameter T in the same way we tuned γ. I won't do this here, but in principle it could only help.