Training a Smart Cab

Implement a Basic Driving Agent

After Random Action implementation

QUESTION: Observe what you see with the agent's behavior as it takes random actions. Does the smart cab eventually make it to the destination? Are there any other interesting observations to note?

There is only a change of code for action to choose an action randomly through the list valid_actions = [None, 'forward', 'left', 'right']. The cab initially moves about at random. Sometimes it happens upon a waypoint, and with sufficient running time and no obstacles it will always eventually hit the waypoint given infinite time, but there may be no guarantee that the agent will do the same given finite time no matter how large the given time constraint may be.

After State Storage

QUESTION: What states have you identified that are appropriate for modeling the smart cab and environment? Why do you believe each of these states to be appropriate for this problem?

The parameters that go in local state for the agent should be bits of data that are useful in deciding the next best course of action. In addition to the desired direction of travel from the planner, almost every input qualifies with the exception of what any car to the right in the current intersection is planning to do.

We require the direction of travel (next_waypoint) because the direction of the next waypoint tells the agent which way we would generally prefer to travel; without this information we wouldn't have a reason to turn left (for instance) instead of going straight, or any reason to travel any particular direction really.

We need to know whether the light is green or red because that limits whether we can take our desired action right now (we'd prefer to travel forward, for example, but if we do so when the light is red that's a traffic infraction with negative reward).

We have to know the status of any cars at the intersection oncoming or to the left, because they can interfere with a desired action. If we want to travel right, we can do so on a red light as long as there are no cars from the left traveling through. If we want to travel left we can do so as long as there is no oncoming car traveling straight across.

I can't think of a scenario in which we care about cars to the right, though. If we want to go straight, then if the light is green we can go, and if red we can't. If we want to go right, then on green we just go, on red we care what the car from the left wants to do. If we want to go left, then on red we stop, and on green we care what the oncoming car wants to do. In no case do we care about the intentions of the car to the right (unless that car is not a reliable rule follower, something we might actually want to consider in a more realistic simulation).

We also don't really care about deadline, since regardless of how long is left on the clock we still want to take the optimal action at each step (we would not want to make a car that would break safety laws when in a hurry, this wouldn't be a lot better than a human driver). Additionally, there are a lot of possible deadline values, which would explode the number of states we need to account for in the learning matrix.

After Q-Learning stage 1

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?

At first the behavior is not very different at all, for at least the first episode or two. Quickly the agent begins to manage to get to the waypoint before the deadline, though at least for this iteration it's behavior is a bit strange (more on that in a minute). Over time the agent also begins to drive more safely and legally, though not always. This is occurring because it examines it's history of behavior and what sort of rewards have occurred when it's taken actions in given states before. When deciding whether or not to go "forward" when the light is red, it notices that it's been penalized for this before and that taking the "None" action in this state is better, for example.

This is not totally as you would expect, though. For one thing, the agent appears to prefer to keep driving rather than hold still; so if it can't drive forward like it would prefer because of a red light, but it can turn right, it seems to do so even though it will get hit with a -0.5 reward. I suspect this is because I've set the discount rate for future rewards to be 0.8, and it's valuing the future 2.0 rewards it will get for driving back towards the waypoint too highly. I think I should reduce the discount rate significantly.

Additionally, it still takes illegal actions sometimes because I'm using an exploration value of 10% and so even when it knows the right thing to do, 10% of the time it takes a random action, often an unsafe or illegal one. I think I need to have some exploration path that doesn't just pick a random value; maybe we start with a higher epsilon value and reduce it to 0 over time by reducing it each time the exploration path is hit by a tiny amount.

During Q-learning parameter iteration:

Learning Rate: 0.5 Discount Value: 0.8 Initial Epsilon Value: 0.1 Epsilon Degradation Rate: 0.0 Trials: 100 Deadlines Missed: 23 Successful Arrivals: 77 Traffic Infractions: 71

attempts to make epsilon value degrade over time

Learning Rate: 0.5 Discount Value: 0.8 Initial Epsilon Value: 0.15 Epsilon Degradation Rate: 0.01 Trials: 100 Deadlines Missed: 62 Successful Arrivals: 38 Traffic Infractions: 5

Learning Rate: 0.5 Discount Value: 0.8 Initial Epsilon Value: 0.1 Epsilon Degradation Rate: 0.001 Trials: 100 Deadlines Missed: 26 Successful Arrivals: 74 Traffic Infractions: 25

decrease discount value to make strange looping behaviors reduce

Learning Rate: 0.5 Discount Value: 0.3 Initial Epsilon Value: 0.1 Epsilon Degradation Rate: 0.001 Trials: 100

Deadlines Missed: 12 Successful Arrivals: 88 Traffic Infractions: 27

increase learning rate to value immediate rewards more highly

Learning Rate: 0.65 Discount Value: 0.3 Initial Epsilon Value: 0.1 Epsilon Degradation Rate: 0.001 Trials: 100

Deadlines Missed: 14 Successful Arrivals: 86 Traffic Infractions: 22

Deadlines Missed: 11 Successful Arrivals: 89 Traffic Infractions: 22

crank down the discount value further, that made a big impact on success rates

Learning Rate: 0.65 Discount Value: 0.2 Initial Epsilon Value: 0.1 Epsilon Degradation Rate: 0.001 Trials: 100

Deadlines Missed: 35 Successful Arrivals: 65 Traffic Infractions: 22

actually that was worse, bring discount value up a bit and learning rate down

Learning Rate: 0.6 Discount Value: 0.25 Initial Epsilon Value: 0.1 Epsilon Degradation Rate: 0.001 Trials: 100

Deadlines Missed: 11 Successful Arrivals: 89 Traffic Infractions: 26

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?

The numbers are available above. The last set performs best, of the sampled parameter combinations, and behaves more rationally than the initial agent when it's route is blocked. Once it's gone through around 12 iterations, it begins to behave reliably, and by the time it's gone through the first 30 episodes it's traffic infraction rate drops significantly. I suspect over time it performs quite well, so I'm going to run another set of 5,000 trials because I suspect it has converged already and will perform more or less successfully from this point forward.

Learning Rate: 0.6 Discount Value: 0.25 Initial Epsilon Value: 0.1 Epsilon Degradation Rate: 0.001 Trials: 5000 Deadlines Missed: 331 Successful Arrivals: 4619 Traffic Infractions: 24

This approach is working ok, but there are some edge cases. When we run into another car at an intersection in the oncoming lane, we tend to lock up. It seems we've learned that it's simply safest to let other cars go, and if both agents decide that then we get stuck. We could up the experimentation rate, but that would result in more infractions and we don't want that. I think the next step might be to have the epsilon value scale out the current q values by random amounts rather than just go away entirely so that if we're stuck for long enough the car will pick another "valid" direction eventually. Another possibility might be to attach a slight negative reward to holding still; less than the penalty of committing an infraction, of course, but high enough to encourage us out of just being parked and waiting. Increasing the discount rate would also help with this. That's probably the easiest to mess with so we'll try that one first.

Learning Rate: 0.6 Discount Value: 0.4 Initial Epsilon Value: 0.1 Epsilon Degradation Rate: 0.001 Trials: 100 Deadlines Missed: 14 Successful Arrivals: 86 Traffic Infractions: 23

This doesn't seem to be helping enough when I scale it up over a large number of trials, we still suffer failure > 10% of the time because of locking up in certain cases. I'm going to revert the discount value and try instead accumulating a small negative reward for staying still.

Learning Rate: 0.6 REWARD FOR STAYING STILL: -0.1 Discount Value: 0.25 Initial Epsilon Value: 0.1 Epsilon Degradation Rate: 0.001 Trials: 100 Deadlines Missed: 4 Successful Arrivals: 96 Traffic Infractions: 28

That seems way better. Let's see how it does scaled up over 500 trials.

Learning Rate: 0.6 REWARD FOR STAYING STILL: -0.1 Discount Value: 0.25 Initial Epsilon Value: 0.1 Epsilon Degradation Rate: 0.001 Trials: 500 Deadlines Missed: 38 Successful Arrivals: 462 Traffic Infractions: [illegible]

Infractions: 26

We're still missing some deadlines unnecessarily, but fewer, and the infractions are stable. This is probably good enough.

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?

I believe this is fairly close, but that an optimum policy would probably make some cleverer decisions when blocked. I think that by re- implementing the epsilon degradation to instead have an action randomly selected from the actions known to not be illegal that could be slightly better. Still, this is fairly close specifically because we care about both not breaking the law/hurting people and getting there on time, but far more about the first than the second thing. If we can get there on time 90% of the time, and never break the law or crash, that's better than a higher time-efficiency rating at the expense of causing traffic accidents occasionally. The most optimal policy (if it exists) would arrive on time 99.9% of the time (when deadline is reasonable) is never crash or break the law.

It's hard to calculate the optimal policy. A move that seems optimal at the time might cause unforeseen delays later depending on the unpredictable behavior of other cars. Nevertheless, since you can't predict the behavior of other agents, you can reasonably define the optimal policy to be to always go to the next way-point, while, when necessary, waiting for lights and traffic so as to not incur any penalties. As we saw earlier, our cab has learned this type of optimal policy after many trials.

Implementation

In [ ]:

```
import random
from environment import Agent, Environment
from planner import RoutePlanner
from simulator import Simulator

class LearningAgent(Agent):
    """An agent that learns to drive in the smartcab world."""

    def __init__(self, env):
        super(LearningAgent, self).__init__(env)  # sets self.env = env, 
state = None, next_waypoint = None, and a default color
        self.color = 'red'  # override color
        self.planner = RoutePlanner(self.env, self)  # simple route planner 
to get next_waypoint
        self.state = {}
        self.learning_rate = 0.6
        self.exploration_rate = 0.1
        self.exploration_degradation_rate = 0.001
        self.discount_rate = 0.4
        self.q_values = {}
        self.valid_actions = [None, 'forward', 'left', 'right']

    def reset(self, destination=None):
        self.planner.route_to(destination)
        # TODO: Prepare for a new trip; reset any variables here, if requir
ed

    def update(self, t):
        # Gather inputs
        self.next_waypoint = self.planner.next_waypoint()  # from route
planner, also displayed by simulator
        inputs = self.env.sense(self)
        deadline = self.env.get_deadline(self)

        self.state = self.build_state(inputs)

        # TODO: Select action according to your policy
        action = self.choose_action_from_policy(self.state)

        # Execute action and get reward
        reward = self.env.act(self, action)

        # TODO: Learn policy based on state, action, reward
        self.update_q_value(self.state, action, reward)

        #print "LearningAgent.update(): deadline = {}, inputs = {}, action
= {}, reward = {}".format(deadline, inputs, action, reward)  # [debug]

    def build_state(self, inputs):
      return {
        "light": inputs["light"],
        "oncoming": inputs["oncoming"],
        "left": inputs["left"],
        "direction": self.next_waypoint
      }

    def choose_action_from_policy(self, state):
        if random.random() < self.exploration_rate:
            self.exploration_rate -= self.exploration_degradation_rate
            return random.choice(self.valid_actions)
        best_action = self.valid_actions[0]
        best_value = 0
        for action in self.valid_actions:
            cur_value = self.q_value_for(state, action)
            if cur_value > best_value:
                best_action = action
                best_value = cur_value
            elif cur_value == best_value:
                best_action = random.choice([best_action, action])
        return best_action

    def max_q_value(self, state):
        max_value = None
        for action in self.valid_actions:
            cur_value = self.q_value_for(state, action)
            if max_value is None or cur_value > max_value:
                max_value = cur_value
        return max_value

    def q_value_for(self, state, action):
        q_key = self.q_key_for(state, action)
        if q_key in self.q_values:
            return self.q_values[q_key]
        return 0

    def update_q_value(self, state, action, reward):
        q_key = self.q_key_for(state, action)
        cur_value = self.q_value_for(state, action)
        inputs = self.env.sense(self)
        self.next_waypoint = self.planner.next_waypoint()
        new_state = self.build_state(inputs)
        learned_value = reward + (self.discount_rate * self.max_q_value(new_
state))
        new_q_value = cur_value + (self.learning_rate * (learned_value - cur
_value))
        self.q_values[q_key] = new_q_value

    def q_key_for(self, state, action):
        return "{}|{}|{}|{}|{}".format(state["light"], state["direction"], s
tate["oncoming"], state["left"], action)



def run():
    """Run the agent for a finite number of trials."""

    # Set up environment and agent
    e = Environment()  # create environment (also adds some dummy traffic)
    a = e.create_agent(LearningAgent)  # create agent
    e.set_primary_agent(a, enforce_deadline=True)  # specify agent to track
    # NOTE: You can set enforce_deadline=False while debugging to allow lon
ger trials

    # Now simulate it
    sim = Simulator(e, update_delay=0.0, display=True)  # create simulator
(uses pygame when display=True, if available)
    # NOTE: To speed up simulation, reduce update_delay and/or set display=
False

    sim.run(n_trials=100)  # run for a specified number of trials
    # NOTE: To quit midway, press Esc or close pygame window, or hit Ctrl+C
on the command-line
    print "CONCLUSION REPORT"
    print "WINS: {}".format(e.wins)
    print "LOSSES: {}".format(e.losses)
    print "INFRACTIONS: {}".format(e.infractions)


if __name__ == '__main__':
    run()
```

Results: Learning Rate: 0.6 REWARD FOR STAYING STILL: -0.1 Discount Value: 0.25 Initial Epsilon Value: 0.1 Epsilon Degradation Rate: 0.001 Trials: 100 Deadlines Missed: 1 Successful Arrivals: 99 Traffic Infractions: 39