project_report

June 16, 2016

1 Project 4 Report

This is my report by Project 4 of the Machine Learning Engineer Nanodegree - **Teach a Smartcab How to Drive**. In this project I used reinforcement learning techniques to teach an engine how to play a simple game of reaching a destination on a grid-like world given some restrictions. The project description can be found here, and my code for the project is on this github repo.

1.1 Task 0: Implementing a Perfect Agent

Although this is not a requirement, it is relatively simple, given the inputs, to define a set of rules that result in an agent that will always take the right action. The next move is given by the `self.next_waypoint` variable inside the `update` method of our agent; the only thing to do is take care that the agent will only perform the next move when it can, following the right-of-way rules in the project description, reproduced below:

> US right-of-way rules apply: On a green light, you can turn left only if there is no oncoming traffic at the intersection coming straight. On a red light, you can turn right if there is no oncoming traffic turning left or traffic from the left going straight.

So, a simple set of rules should suffice to make our agent perfect:

```
In [1]: # this is a method of an Agent object
        def update(self, t):
            # Gather inputs
            self.next_waypoint = self.planner.next_waypoint()  # from route planner
            inputs = self.env.sense(self)
            deadline = self.env.get_deadline(self)

            # The next best move is given by the planner
            action = self.next_waypoint

            # On a red light, the agent can only turn right, and even so only if:
            # - no oncoming traffic is going left
            # - no traffic from the left is going forward
            if inputs['light'] == 'red':
                if (action != 'right') or (inputs['oncoming'] == 'left') or (inputs
                    action = None
```

```
            # On a green light, the agent cannot turn left if there is
            # oncoming traffic going forward
            elif inputs['oncoming'] == 'forward' and action == 'left':
                action = None
```

The "perfect agent" version of the code is stored in this Github commit.

However, the goal of the project is to build an agent that can *learn these rules by itself*. Let's follow the project's rubric and see how it goes.

1.2 Task 1: Implement a Basic Driving Agent

According to the project's instructions, the basic driving agent should "produce some random move/action." That's easy enough:

```
In [2]: import random

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

            # Do something random
            action = random.choice(['right', 'left', 'forward', None])
```

With this strategy, the agent may reach its destination on time, but only if it gets lucky. When the deadline is not enforced, the agent will eventually get there - but it can take a long time, since it's not at all actually aiming for it. The agent also gets lots of penalties for incurring in illegal moves, such as trying to go forward when the light is red.

The "random agent" version of the code is stored in this Github commit.

From this behavior (and the behavior of the perfect agent implemented before) we can begin to think about what information needs to be in the state for the agent to learn the appropriate behavior: it needs to know where the destination is, as well as information about its surroundings (if the light is green or red and whether right-of-way rules imply it should stay put).

1.3 Task 2: Identify and Update State

The following information is available for the agent at each update:

- *Light*: whether the light is red or green. As mentioned above, a green light means the agent can perform the next action, with the possible exception of a left turn, while a red light means the agent should stay put, with the possible exception of a right turn. So, it is important to add the light to the state.
- *Oncoming*: whether there is oncoming traffic, and which direction it is going. As mentioned above, oncoming traffic may mean the agent cannot turn left or right, so this information needs to be in the state as well.
- *Right*: whether there is traffic from the right of the agent, and which direction it is going. Right-of-way rules don't mention traffic to the right at any point, so this is unnecessary information that doesn't need to be in the state for the agent to learn the optimal policy.

- *Left*: whether there is traffic from the left of the agent, and which direction it is going. Traffic from the left going straight means the agent cannot turn right on a red light, so this needs to be in the state.
- *Next waypoint*: the direction the agent should go to reach the destination. Without this information, the agent does not know where to go next and might as well wonder around randomly, so this needs to go in the state.
- *Deadline*: how much time the agent has left to reach its destination. At first, I would say this is not meaningful information for the agent, since it doesn't change right-of-way rules nor the best route. I thought about adding it to the state anyway, but this would mean a large increase in the number of possible states. Using only `light` (red or green), `oncoming` (None, left, right, or forward), `left` (None, left, right, or forward) and `next_waypoint` (left, right, or forward), we have $2 \times 4 \times 4 \times 3 = 96$ possible states. Adding `deadline` would mean multiplying this number by 50, if not more. So I'll keep `deadline` off my state for now.

There is the possibility of combining inputs to create a state. Maybe I could define the state in such a way that the allowed actions would be immediately available for the agent; for instance, I could come up with a `turn_right` variable that would check if the agent can turn right, and add that variable to the state. At least for now, though, I'd rather see how the agent deals with the "raw" variables; I can tweak the state later to try and get the agent to find the optimal policy.

```
In [3]: import random

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

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

            # Do something random
            action = random.choice(['right', 'left', 'forward', None])
```

The version of the code with a state is stored in this Github commit.

1.4 Task 3: Implement Q-Learning

In this step our agent begins to learn from its actions. The project specifically says to "pick the best action available from the current state based on Q-values," so I'll leave the exploration-exploitation dilemma for the next section.

1.4.1 Deciding on the Appropriate Q-learning Function

The general form of the Q-function is:

$$Q(s,a) = R(s) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q(s',a')$$

That is, the Q-value for a given `(state, action)` pair is the the reward for that state, $R(s)$, plus the discounted expected value of Q for the next state the agent lands in, considering the transition function $T(s, a, s') = \Pr(s' \mid s, a)$ (the probability of landing on state s' coming from state s and performing action a) and that, whatever s' is, the agent will maximize Q from there on.

The Q-learning update function is given by:

$\hat{Q}_t(s, a) = (1 - \alpha_t)\hat{Q}_{t-1}(s, a) + \alpha_t(r + \gamma \max_{a'} \hat{Q}_{t-1}(s', a'))$

That is, our estimate of the Q-value for the `(state, action)` pair is updated with the learning rate (α_t, which varies over time) by the observed reward (r) and our previous estimate of the Q-value for the observed next state (s'), discounted by the discount factor (γ) and considering the agent will pick the action a' that maximizes Q from the next state on.

However, in this case there's no need to worry about the future state, since the agent gets an immediate reward for doing the right thing. According to the project description:

> The smartcab gets a reward for each successfully completed trip. A trip is considered "successfully completed" if the passenger is dropped off at the desired destination (some intersection) within a pre-specified time bound (computed with a route plan).
>
> It also gets a smaller reward for each correct move executed at an intersection. It gets a small penalty for an incorrect move, and a larger penalty for violating traffic rules and/or causing an accident.

So, even though the larger reward is only reaped once the agent reaches its destination, there are smaller rewards for following the correct path, and penalties for not doing so. This should be enough for the agent to learn the best policy.

Granted, ignoring the agent's future decisions means I'm not using some information that could be of help. But the upside is a simplification of the problem: it's as if the agent is playing a 1-round game over and over, with immediate rewards for immediate actions. I expect this simplification more than compensates ignoring long-term rewards in this particular setting.

1.4.2 Q-learning implementation

This means I won't actually bother with keeping track of the state the agent ends up in after performing an action (or, to be more technical, I'm setting the discount factor γ to zero). My update function will then simply be $\hat{Q}_t(s, a) = (1 - \alpha_t)\hat{Q}_{t-1}(s, a) + \alpha_t r$.

Here's what I'll do: - Initialize `agent.qvals` as an empty dictionary and `agent.time` as 0 when the agent is initialized. - Define a `best_action()` method that takes a state and returns the best action (or one of the best actions) given the current Q-values:

```
In [4]: # agent method
        def best_action(self, state):
            """
            Returns the best action (the one with the maximum Q-value)
            or one of the best actions, given a state.
            """
            # get all possible q-values for the state
            all_qvals = {action: self.qvals.get((state, action), 0)
                         for action in self.possible_actions}
```

```
            # pick the actions that yield the largest q-value for the state
            best_actions = [action for action in self.possible_actions
                            if all_qvals[action] == max(all_qvals.values())]

            # return one of the best actions at random
            return random.choice(best_actions)
```

- In the `update()` agent method:
  - increment the time by 1
  - set the learning rate as 1/time
  - pick the action using `best_action()`
  - update the value for the `(state, action)` pair in the `qvals` with the reward.

```
In [5]: # agente method
        def update(self, t):
            # Gather inputs
            self.next_waypoint = self.planner.next_waypoint()  # from route planner
            inputs = self.env.sense(self)
            deadline = self.env.get_deadline(self)

            # update time and learning rate
            self.time += 1
            learn_rate = 1.0 / self.time

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

            # Pick the best known action
            action = self.best_action(self.state)

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

            # Update the q-value of the (state, action) pair
            self.qvals[(self.state, action)] = \
                (1 - learn_rate) * self.qvals.get((self.state, action), 0) + \
                learn_rate * reward
```

This version of the code is stored in this Github commit.

1.5 Results

The last task, according to the project description, is:

> Apply the reinforcement learning techniques you have learnt, and tweak the parameters (e.g. learning rate, discount factor, action selection method, etc.), to improve the performance of your agent. Your goal is to get it to a point so that within 100 trials,

the agent is able to learn a feasible policy - i.e. reach the destination within the allotted time, with net reward remaining positive.

This goal is reached with the implementation presented above. Although there are instances in which, with 90+ trials, the agent will pick a wrong action from time to time, in my tests it never failed to reach the destination after 60+ trials.