report

October 30, 2016

1 Report - Reinforced Learning

1.1 1. Implement a Basic Driving Agent

Task: To begin, your only task is to get the smartcab to move around in the environment. At this point, you will not be concerned with any sort of optimal driving policy. Note that the driving agent is given the following information at each intersection: • The next waypoint location relative to its current location and heading. • The state of the traffic light at the intersection and the presence of oncoming vehicles from other directions. • The current time left from the allotted deadline. To complete this task, simply have your driving agent choose a random action from the set of possible actions (None, 'forward', 'left', 'right') at each intersection, disregarding the input information above. Set the simulation deadline enforcement, enforce_deadline to False and observe how it performs.

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

Anwser: I implemented a random action generator:

```
In [ ]: # TODO: Select action according to your policy
        possible_directions = (None, 'forward', 'left', 'right')
        action = random.choice(possible_directions)
```

The red car stops at every red light correctly and chooses a random action at a green light (None, 'forward', 'left', 'right'). Eventually, it reaches the destination by luck after a long time. All the other cars seem to move randomly too.

1.2 2. Inform the Driving Agent

Now that your driving agent is capable of moving around in the environment, your next task is to identify a set of states that are appropriate for modeling the smartcab and environment. The main source of state variables are the current inputs at the intersection, but not all may require representation. You may choose to explicitly define states, or use some combination of inputs as an implicit state. At each time step, process the inputs and update the agent's current state using the self.state variable. Continue with the simulation deadline enforcementenforce_deadline being set to False, and observe how your driving agent now reports the change in state as the simulation progresses.

Solution:

```
In [ ]: # TODO: Update state
        state = {}
        state['light'] = inputs['light']
        state['oncoming'] = inputs['oncoming']
        state['left'] = inputs['left']
        state['next_waypoint'] = self.get_next_waypoint()
        self.state = state
```

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

Answer: I chose 4 states in total: - light - oncoming - left - next_waypoint

I did not include "right", because it does not affect our agent: - if our light is "green", the right car's light is "red" (or if it wants to turn right on a red light, it has to give us the right of way) - if our light is "red" and we want to turn right, the right car can not affect us (I was not sure about the US-traffic rules. It seems that there would be a exception, if the right car wants to make a u-turn. http://arstechnica.com/civis/viewtopic.php?t=67324 . But this is not included in our simulation, so it does not matter)

OPTIONAL: How many states in total exist for the smartcab in this environment? Does this number seem reasonable given that the goal of Q-Learning is to learn and make informed decisions about each state? Why or why not?

Anwer: - light {red, green} -> 2 - oncoming {None, 'forward', 'left', 'right'} -> 4 - left {None, 'forward', 'left', 'right'} -> 4 - next_waypoint {'forward', 'left', 'right'} -> 3

2x4x4x3 = **96**

I did not include deadline, because it would increase the size of all possible states by **a factor of 50** while not helping just a little. **This big increase of dimensionality would slow down the learning process enormously.**

Nonetheless, adding 'deadline' could help a little bit for situations, where the deadline is almost reached and a few small negative rewards (e.g. run a red light) are tolerated in exchange for getting a big reward for reaching the goal. However this special situation does not justify increasing the possible states by a factor of 50. One solution could be to just add a **boolean "deadline_almost_reached"-state** with a fixed threshold.

1.3 3. Implement a Q-Learning Driving Agent

With your driving agent being capable of interpreting the input information and having a mapping of environmental states, your next task is to implement the Q-Learning algorithm for your driving agent to choose the best action at each time step, based on the Q-values for the current state and action. Each action taken by the smartcab will produce a reward which depends on the state of the environment. The Q-Learning driving agent will need to consider these rewards when updating the Q-values. Once implemented, set the simulation deadline enforcement enforce_deadline to True. Run the simulation and observe how the smartcab moves about the environment in each trial.

The formulas for updating Q-values can be found in this video.

2 The most important Code parts:

How to choose an action:

```
In [ ]: def select_action(self):
            # With a chance of epsilon take a random action, otherwise the argmax
            if random.random() < self.epsilon:
                # random action
                action = random.choice(Environment.valid_actions)
                max_value = self.get_q_value(self.state, action)
            else:
                action = ''
                # pick argmax ^Q(s,a)
                max_value = -99999
                for a in Environment.valid_actions:
                    q = self.get_q_value(self.state, a)
                    if q > max_value:
                        max_value = q
                        action = a

            return max_value, action
```

Updating Q-Table:

```
In [ ]: # TODO: Learn policy based on state, action, reward
        if self.prev_state is not None:
            # Q(s,a) = (1 - alpha) * Q(s,a) + alpha * ( r + gamma * max_over_a'[ Q
                  if self.prev_state is not None:
                  # Q(s,a) = (1 - alpha) * Q(s,a) + alpha * ( r + gamma * max_ove
                  self.Q[(self.prev_state, self.prev_action)] = (1 - self.alpha)
                      self.prev_state, self.prev_action) + self.alpha * \
                                                           (self.prev_reward + se

        # set current states as previous states
        self.prev_state = self.state
        self.prev_action = action
        self.prev_reward = reward
        self.score_counter += 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?

Answer: The car seems to chose randomly first, however it is learning over time! After the first few rounds it starts to behaving really good: Stopping at red light and following the "next_path" advices.

The q-learning algorithm works by writing values into the q-table. These values Q(s,a) represent how well an action a is in regard of a certain state s. After each action and receiving a reward, theses values are updated with the formula:

** Q(s,a) = (1 - alpha) * Q(s,a) + alpha * (r + gamma * max_over_a'[Q(s', a')]) * alpha:** alpha is the learning rate. It determines the ratio of the old value vs. the new learned experience. * **r:** Reward of the action in state s. * **max_over_a'[Q(s', a')]:** Gives a value for the biggest possible qvalue for next move. This is reasonable, so the algorithm don't only look for quick wins, but for more steps! * **gamma:** discount factor for max_over_a'[Q(s', a')]

But what is a good strategy to chose the next move? It is appropriate to make a mix of exploration vs. exploitation. * **Exploration** means, that with the probability of epsilon the algorithm chooses a random action. * **Exploitation** means, that the algorithm uses the already learned q-values and chooses the best possible action.

2.1 4. Improve the Q-Learning Driving Agent

Your final task for this project is to enhance your driving agent so that, after sufficient training, the smartcab is able to reach the destination within the allotted time safely and efficiently. Parameters in the Q-Learning algorithm, such as the learning rate (alpha), the discount factor (gamma) and the exploration rate (epsilon) all contribute to the driving agent's ability to learn the best action for each state. To improve on the success of your smartcab:

Set the number of trials, n_trials, in the simulation to 100. Run the simulation with the deadline enforcement enforce_deadline set to True (you will need to reduce the update delay update_delay and set the display to False). Observe the driving agent's learning and smartcab's success rate, particularly during the later trials. Adjust one or several of the above parameters and iterate this process. This task is complete once you have arrived at what you determine is the best combination of parameters required for your driving agent to learn successfully.

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?

Answer I tuned the parameters Alpha, Gamma and Epsilon manually (GridSearch would be an alternative) and focused on the result "Win/Los"-Ratio and "Average Net Rewards". I found that the following Parameters give really good results!

2.1.1 Final Parameter

Parameter: * Alpha: 0.9 * Gamma: 0.2 * Epsilon 0.02:

Results * Average of Steps to reach goal: 13.47 * Win/Lose Ratio: 98/2 * Average of Net Rewards in total: 22.035 * Average of Positive Rewards in total: 6.32 * Average of Negative Rewards in total: 7.15

3 I did some additional tests to understand the parameter:

3.0.1 What happens for low Alpha (Learning Rate)?

Parameter: * Alpha: 0.0001 * Gamma: 0.2 * Epsilon 0.02:

Results * Average of Steps to reach goal: 14.94 * Win/Lose Ratio: 92/8 * Average of Net Rewards in total: 21.62 * Average of Positive Rewards in total: 7.82 * Average of Negative Rewards in total: 7.12

Interpretation: The results are way better than expected, but the most interesting chart is the comparison between the net results. As you can see, the Q-Learning needs longer to learn with low alpha. With Alpha=0.9 it is a lot steeper.

3.0.2 What happens for high Epsilon (Chance for a random actionchoice - Exploration vs. Exploitation)?

Parameter: * Alpha: 0.0001 * Gamma: 0.2 * Epsilon 0.9:

Results * Average of Steps to reach goal: 27.15 * Win/Lose Ratio: 16/84 * Average of Net Rewards in total: 3.985 * Average of Positive Rewards in total: 5.99 * Average of Negative Rewards in total: 21.16

Interpretation: The results are bad and it seems like the Q-Learner is not really learning, but this is not the case! The Q-Learning algorithm does learn, but choses in 90% of all cases to take a random chance anyway (Exploration).

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?

3.0.3 The optimal policy would be:

- Always follow the navigation
- Do not run a red light, except for turning right (and the left one does not go 'forward', or the oncoming does not go 'left')
- If it is green and the car goes left, check for oncoming cars.

To see if the agent has learned the rules, it's interesting to take a look at some values in the Q-Table:

4 Summary of optimal policy vs. Q-Table

4.0.1 Always follow the navigation

- ((('left', None), ('light', 'green'), (**'next_waypoint', 'right'**), ('oncoming', None)), **'right'**): **6.7689057895232745,**
- ((('left', None), ('light', 'green'), (**'next_waypoint', 'right'**), ('oncoming', None)), **'left'**): **-0.32101988749999993,**

4.0.2 Do not run a red light, except for turning right (and the left one does not go 'forward', or the oncoming does not go 'left')

- ((('left', None), (**'light', 'red'**), ('next_waypoint', 'left'), ('oncoming', None)), **'left'**): **-1.1631927499999999,**
- ((('left', None), (**'light', 'red'**), ('next_waypoint', 'left'), ('oncoming', None)), **'right'**): **-0.11149999999999993,**
- ((('left', 'right'), (**'light', 'red'**), ('next_waypoint', 'forward'), ('oncoming', None)), **None**): **0.9171506871347347,**
- ((('left', None), (**'light', 'red'), ('next_waypoint','right'**), ('oncoming', None)), **'right'**): **3.1364968585282127,**

4.0.3 If it is green and the car goes left, check for oncoming cars.

- I did not find this rule in the Q-Table. Probably because it does not happen very often.

4.1 Result: The agent has learned most of these rules!

```
In [ ]: # Complete Q-Table
        Q: {
```

```
((('left', None), ('light', 'red'), ('next_waypoint', 'forward'), ('oncomin
((('left', None), ('light', 'red'), ('next_waypoint', 'right'), ('oncoming'
((('left', 'right'), ('light', 'red'), ('next_waypoint', 'left'), ('oncomin
((('left', None), ('light', 'green'), ('next_waypoint', 'right'), ('oncomin
 ((('left', None), ('light', 'green'), ('next_waypoint', 'right'), ('oncomi
 ((('left', 'right'), ('light', 'red'), ('next_waypoint', 'left'), ('oncomi
 ((('left', None), ('light', 'red'), ('next_waypoint', 'right'), ('oncoming
 ((('left', 'forward'), ('light', 'red'), ('next_waypoint', 'right'), ('onc
 ((('left', None), ('light', 'green'), ('next_waypoint', 'forward'), ('onco
 ((('left', None), ('light', 'red'), ('next_waypoint', 'forward'), ('oncomi
 ((('left', None), ('light', 'green'), ('next_waypoint', 'right'), ('oncomi
 ((('left', None), ('light', 'green'), ('next_waypoint', 'right'), ('oncomi
 ((('left', None), ('light', 'red'), ('next_waypoint', 'forward'), ('oncomi
 ((('left', None), ('light', 'green'), ('next_waypoint', 'right'), ('oncomi
 ((('left', 'left'), ('light', 'green'), ('next_waypoint', 'forward'), ('on
 ((('left', None), ('light', 'red'), ('next_waypoint', 'left'), ('oncoming'
 ((('left', None), ('light', 'green'), ('next_waypoint', 'forward'), ('onco
 ((('left', 'left'), ('light', 'green'), ('next_waypoint', 'forward'), ('on
 ((('left', 'left'), ('light', 'green'), ('next_waypoint', 'forward'), ('on
 ((('left', None), ('light', 'green'), ('next_waypoint', 'forward'), ('onco
 ((('left', None), ('light', 'red'), ('next_waypoint', 'forward'), ('oncomi
 ((('left', 'forward'), ('light', 'red'), ('next_waypoint', 'right'), ('onc
 ((('left', 'forward'), ('light', 'red'), ('next_waypoint', 'right'), ('onc
 ((('left', None), ('light', 'red'), ('next_waypoint', 'forward'), ('oncomi
 ((('left', None), ('light', 'green'), ('next_waypoint', 'forward'), ('onco
 ((('left', None), ('light', 'red'), ('next_waypoint', 'forward'), ('oncomi
 ((('left', 'left'), ('light', 'green'), ('next_waypoint', 'forward'), ('on
 ((('left', None), ('light', 'red'), ('next_waypoint', 'left'), ('oncoming'
 ((('left', None), ('light', 'red'), ('next_waypoint', 'forward'), ('oncomi
 ((('left', None), ('light', 'red'), ('next_waypoint', 'right'), ('oncoming
 ((('left', None), ('light', 'red'), ('next_waypoint', 'left'), ('oncoming'
 ((('left', None), ('light', 'green'), ('next_waypoint', 'right'), ('oncomi
 ((('left', 'right'), ('light', 'red'), ('next_waypoint', 'forward'), ('onc
 ((('left', None), ('light', 'red'), ('next_waypoint', 'forward'), ('oncomi
 ((('left', None), ('light', 'green'), ('next_waypoint', 'right'), ('oncomi
 ((('left', None), ('light', 'green'), ('next_waypoint', 'left'), ('oncomin
 ((('left', None), ('light', 'red'), ('next_waypoint', 'right'), ('oncoming
 ((('left', 'right'), ('light', 'red'), ('next_waypoint', 'forward'), ('onc
 ((('left', None), ('light', 'red'), ('next_waypoint', 'left'), ('oncoming'
 ((('left', None), ('light', 'green'), ('next_waypoint', 'right'), ('oncomi
 ((('left', None), ('light', 'red'), ('next_waypoint', 'right'), ('oncoming
 ((('left', None), ('light', 'green'), ('next_waypoint', 'right'), ('oncomi
 ((('left', None), ('light', 'green'), ('next_waypoint', 'left'), ('oncomin
 ((('left', None), ('light', 'green'), ('next_waypoint', 'right'), ('oncomi
 ((('left', None), ('light', 'red'), ('next_waypoint','right'), ('oncoming'
 ((('left', None), ('light', 'red'), ('next_waypoint', 'forward'), ('oncomi
 ((('left', None), ('light', 'green'), ('next_waypoint', 'left'), ('oncomin
 ((('left', 'right'), ('light', 'red'), ('next_waypoint', 'left'), ('oncomi
((('left', 'forward'), ('light', 'red'), ('next_waypoint', 'right'), ('onc
((('left', 'right'), ('light', 'red'), ('next_waypoint', 'left'), ('oncomi
((('left', None), ('light', 'red'), ('next_waypoint', 'right'), ('oncoming
((('left', None), ('light', 'red'), ('next_waypoint', 'forward'), ('oncomi
((('left', None), ('light', 'green'), ('next_waypoint', 'forward'), ('onco
((('left', 'right'), ('light', 'red'), ('next_waypoint', 'forward'), ('onc
((('left', None), ('light', 'green'), ('next_waypoint', 'forward'), ('onco
((('left', None), ('light', 'red'), ('next_waypoint', 'right'), ('oncoming
((('left', None), ('light', 'green'), ('next_waypoint', 'forward'), ('onco
((('left', 'right'), ('light', 'red'), ('next_waypoint', 'forward'), ('onc
((('left', None), ('light', 'green'), ('next_waypoint', 'right'), ('oncomi
((('left', None), ('light', 'red'), ('next_waypoint', 'forward'), ('oncomi
((('left', None), ('light', 'green'), ('next_waypoint', 'right'), ('oncomi
((('left', None), ('light', 'red'), ('next_waypoint', 'forward'), ('oncomi
((('left', None), ('light', 'green'), ('next_waypoint', 'forward'), ('onco
((('left', None), ('light', 'green'), ('next_waypoint', 'left'), ('oncomin
```