Reynard Hilman
February 7, 2016

Training a Smartcab to Drive

Basic Driving Agent

Basic driving agent is an agent that randomly choose an action (none, left, right, or forward). It eventually reaches the destination, sometimes within the deadline when it's lucky. In the worst case it reaches destination after the deadline reaches -250. Because the agent's behavior is random, it gets penalized (-1) a lot with occasional rewards of 0.5 and 1 and a few 2. In some lucky cases, it gets 10.5 reward when it reached the destination before the deadline.

Identify and Update State

To begin with, I chose the following 4 values to represent the state, because they are the most important things the agent need to know to make a right decision following traffic rules and right of way on an intersection:

- **next_waypoint**
  The agent needs to know where it is supposed to go to reach the destination
- **light**
  The agent obviously needs to know the state of the traffic light (red or green) when the next_waypoint tells it to go forward or left.
- **oncoming**
  The agent needs to know if there is any oncoming car when the next_waypoint tells it to go left.
- **left**
  The agent needs to know if there is any car from left when the next_waypoint tells it to go right.

input["right"] was left out because it is not a determining factor to make the right decision on an intersection. Deadline is also not included in the state, because it means the agent will only see every state once, which is not helpful for learning.

After some observation and reading the source code of environment.py line 164-187, it is apparent that the right of way is not implemented in the environment source code. From the source code, the only things that affect the reward are just the light, and the next waypoint.
If the agent doesn't move, in any case it gets reward of 1. If the agent moves when the light is red it gets -1 except when it turns right. If it moves without getting penalized and the action matches the next waypoint, the agent gets a reward of 2, otherwise it gets 0.5.

So in a real world where the right of way applies, the agent should use the following input as state: **next_waypoint**, **light**, **oncoming**, and **left**. However, in this simulated world only **next_waypoint**, and **light** matters in making the right decision. I experimented using 4 input as state vs only 2 inputs (next_waypoint, light). The result is that the agent learn much faster (much less failures) with just **next_waypoint**, and **light** as the state.

Implement Q-Learning

Q-Learning is implemented with the following formula:

$$Q(s,a) = (1 - \alpha)\, Q(s,a) + \alpha\, (R + \gamma \max_{a'} Q(s',a'))$$

gamma = 0.5
alpha (learning rate) = 0.9.

Agent always chooses the action with the maximum Q value for that state.

Observed Behavior
The agent keeps choosing the first action that gives it a positive reward and does not explore the other possible actions at the expense of not reaching destination on time.

Improvement: Always Explore New Things First

Have the agent try an action that it has never tried before. By default Q value is 0 for all states and actions. So instead of taking the action with the maximum Q value, the agent always take an action where Q(s', a) = 0 if there exists such an action. If there are no actions with Q(s', a) = 0, the agent takes the action with the maximum Q.

Choosing action where Q(s', a) = 0 does not always work on all environment, but it works really well in this case because there is always a positive or negative reward for every action. When it is not possible to rely on Q = 0 for non-explored action, we can implement a different way to track when an agent has tried an action or not. For example by implementing Q function that returns None for non-explored actions, or use a separate hash table to keep track of explored actions.

Observed Behavior

- This cause the agent to perform really poorly the first time around, but it learns really quickly afterward.
- Very good performance: Out of 100 iterations, most of the times it only failed once to hit the deadline (usually on the first iteration).
- Agent's does not try an action more than once if it had non-optimal experience the first time. After 100 iterations, some values in the Q-table contains value that looks exactly

like the reward it gets the first time it tried that action. For example:
`Q('green,left', 'forward') = 0.5`
Which means when the state is: green light, and next waypoint is left, the Q value for going forward is 0.5 which is exactly the reward it gets for going forward.

Improvement: Introduce Exploration vs Exploitation Randomness

Introduce a random exploration once a while, even when the agent has tried all possible actions in a given state. This caused the agent to once a while choose the non-optimum action just to reinforce that it really does not like the action. Exploration probability of 0.1 was used.

Observed Behavior

- Still performs well at getting to destination before the deadline.
- More developed Q-table. On one of the runs, here is the Q-table values at the end of 100 iterations:

```
red,forward:   {'forward': 0.65, 'right': 1.74, None: 2.51, 'left': 0.71}
green,right:   {'forward': 2.27, 'right': 3.39, None: 2.82, 'left': 1.75}
red,right:     {'forward': 0.78, 'right': 3.94, None: 2.94, 'left': 0.70}
green,forward: {'forward': 3.77, 'right': 2.42, None: 2.57, 'left': 2.31}
red,left:      {'forward': 0.24, 'right': 2.45, None: 2.09, 'left': 0.27}
green,left:    {'forward': 2.40, 'right': 2.33, None: 2.64, 'left': 3.74}
```

Note: the state is a combination of (light, next_waypoint)

Agent Learning Summary (after 120 iterations)
Gamma: 0.5
Alpha: 0.9
Exploration vs Exploitation ratio: 1.0 : 9.0
Total failures: 4 times it did not reach destination within deadline
Total penalties: 72 times it got -1
Total wrong action: 91 times it did not follow the direction (got 0.5)
Iteration where it does not meet deadline: {0: -69, 2: -7, 54: -1, 46: -9}

I also tried a few different gamma values. When gamma = 0.8, the agent has a lot more failures to hit deadline (10+ failures for 100 iterations). Gamma = 0.2 performs better with about 3-4 failures. However Gamma = 0.5 seems to consistently have the best result with average failure to hit deadline only around 2. After running this enough times, there was rare cases where gamma=0.5 gives way more than 2 failures. In all of these cases, the agent is stuck in multiple red lights for multiple turns for each red light.

Conclusion

I pick gamma = 0.5 and exploration vs exploitation ratio of 1:9. Combined with the strategy of always explore new things first in the beginning, this turns out to be a very good learning algorithm for the agent. The average failure to hit deadline from 100 iterations is only 2, and most happen early in the iteration.