Train a Smartcab How to Drive

Fernando Hernandez

Udacity - Machine Learning Engineer - Project 4

Implement a basic driving agent

Implement the basic driving agent, which processes the following inputs at each time step:

- *Next waypoint location, relative to its current location and heading,*
- *Intersection state (traffic light and presence of cars), and*
- *Current deadline value (time steps remaining),*
  And produces some `random move/action` *(None, 'forward', 'left', 'right'). Don't try to implement the correct strategy! That's exactly what your agent is supposed to learn.*

Run this agent within the simulation environment with enforce_deadline set to False (see run function in `agent.py`*), and observe how it performs. In this mode, the agent is given unlimited time to reach the destination. The current state, action taken by your agent and reward/penalty earned are shown in the simulator.*

In your report, mention what you see in the agent's behavior. Does it eventually make it to the target location?

With random behavior, the agent does eventually reach the target, but very rarely makes it if a deadline is enforced.

Identify and update state

Identify a set of states that you think are appropriate for modeling the driving agent. The main source of state variables are current inputs, but not all of them may be worth representing. Also, you can choose to explicitly define states, or use some combination (vector) of inputs as an implicit state.

At each time step, process the inputs and update the current state. Run it again (and as often as you need) to observe how the reported state changes through the run.

Justify why you picked these set of states, and how they model the agent and its environment.

The states of the system were represented by:

1. The state of the traffic light: ‘green’ or ‘red’
2. The next waypoint direction: A valid direction returned by ‘next_waypoint()’.

With these two variables, we can represent where we are anywhere in the map relative to the target location, and the state of the biggest source of negative scores - running red lights (-1).

If the world were not round (going all the way right teleports us left) and we could access our location (and destination location), then Manhattan distance to destination might be used.

With these representing the states of the system, we should be able to learn to follow the way point direction, while following the current implementation of traffic laws. This should eschew the need to explicitly use deadline in our representation. We should also be able to reasonably cover most instances of state/action pairs enough times within 100 trials (given some stochastic choice encouragement when appropriate.)

- 2 (lights) * 3 (waypoint directions) = 6 different states
- 6 (states) * 4 (actions) = 24 different state/action pairs

Implement Q-Learning

Implement the Q-Learning algorithm by initializing and updating a table/mapping of Q-values at each time step. Now, instead of randomly selecting an action, pick the best action available from the current state based on Q-values, and return that.

Each action generates a corresponding numeric reward or penalty (which may be zero). Your agent should take this into account when updating Q-values. Run it again, and observe the behavior.

What changes do you notice in the agent's behavior?

When Q-Values were initialized to 0, our model had trouble getting past just gobbling up the small rewards for simply making right turns. This is presumably because there was always a positive reward associated since right turns are always legal at red lights for some positive reinforcement. There is currently no state-space representation associated with deadline time left, so as far as the model was concerned, it was happy just collecting guaranteed small rewards ad infinitum.

Rather than try to model each of these possible deadline state values multiple times to learn (and seriously prolong learning) , we can learn to chase the goal state implicitly without the need for an impending timer of doom. (We can also avoid trying to create custom features such as 'deadline < 5', etc.)

To accomplish this, a few conditions were needed in our Q-learning updates.

The sample-based Q-value iteration was based on a running average of samples and is defined as follows:

$$sample = R(s, a, s') + \gamma * max_{a'}\left(Q(s', a')\right)$$

$$Q(s, a) = (1 - \alpha)Q(s, a) + (\alpha)[sample]$$

Where:

$R(s, a, s')$ is the reward for taking an action a from state s to s'.

$max_{a'}(Q(s', a'))$is the max Q-value of the actions we can take from the next state s'

α is our learning rate

γ is our discounted reward rate for future rewards

Our critical parameters used were:

1. α - Our learning rate is set to around 0.1 We want to slowly incorporate our new knowledge into our view of the world (our Q-values) without discounting past experiences too heavily.
2. γ - Our discount factor for future rewards is set to around 0.9. In much the same vein, we want our past values to remain fairly vivid, and phase out slowly as we gain knowledge about the way the world works.
3. Optimistic Initial Conditions - These values were set to around 10 (at or near our reward for reaching the target destination) and turned out to be critical for learning quickly.

α and γ are kind of two sides of the same supporting our Optimistic Initial Conditions (OIC). The OIC turned out to be useful for overcoming our initial problem of getting stuck chasing the first tiny rewards seen (right turn loops).

With higher OIC, we are encouraged to explore in the beginning since no action's rewards can be as good as the initial Q-values. Then once we start to chase the right actions, our high scores are reinforced and strengthened, while chasing the wrong actions cause those Q-values to continually drop.

As this happens, we start to quickly favor chasing the right actions and the Q-values should, in the limit, converge on the optimal values. But in the meantime, we can work out an optimal policy long before this convergence in optimal values.

With Q-learning (with OIC) implemented, the agent learned a fair bit in 100 trials. Once we changed to Initial Optimistic Conditions, the agent learned some optimal behaviors within 1-2 trials.

Quick Learning

For instance, upon observing the Q-value table after 100 trials, the agent quickly learned to follow the waypoint on all green lights, and if the light was red, the agent learned to follow the waypoint if it pointed right.

Slower learning

But if the light was red and not the ‘right’ direction waypoint, the agent (more slowly) learned to either:

- Make right turns anyways (for small safe reward that is always positive) then immediately start following the waypoint direction again.
- Wait for the light to change, then continue following the waypoint straight or left

---

Enhance the driving agent

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.

Report what changes you made to your basic implementation of Q-Learning to achieve the final version of the agent. How well does it perform?

As discussed earlier, adjusting the learning rate to 0.1, discount factor to 0.9, and Initial Optimistic Q-values to 10 allowed our agent to explore at first, and learn quickly.

In our post-mortem, it turns out that our agent learned to mimic the ‘next waypoint’ action sending it into the rewarding target area. This turned out to not be surprising since, once inspected, our agent is rewarded with +2 for following the ‘next waypoint’ and only +0.5 otherwise (unless waiting, +1).

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?

Our agent does seem to find the optimal policy very quickly, rarely not making it to the destination. Our agent is also learning sub-optimal actions Q-values dropping below the

default value of 10, and optimal action Q-values rising above 10. This has the added benefit of our agent learning to avoid incurring penalties fairly well.

Finally, our agent seems to learn an optimal policy/actions much more quickly than converging on optimal Q-values, which is a good thing.
