P4 – SmartCab Project Write-up

Implement a basic driving agent

See code base.

Identify and update state

State could be defined as the inputs into the agent at this time. An example of this is:

State = {'light': 'green', 'oncoming': None, 'right': None, 'left': None}

We also have an input which is the direction that the Planner is trying to send the Smart Cab for the next waypoint. This can be:

Left, right, forward, none

A combination of these could be used to create the state for the cab, for example:

State = {'light': 'green', 'oncoming': None, 'right': None, 'left': None, ‘waypoint’: ‘Left’}

However, in order to work well with Python dictionaries, the format of this state has been modified use tuples, such as:

State = (('lights', 'red'), ('oncoming', None), ('right', None), ('left', None), ('waypoint', 'forward'))

There are other possible environmental variables the could be included in the state, but these have been discounted for the following reasons:

Deadline: This is the time to the destination. This has not been used in the state because:

i) we do not wish for the agent to become reckless as the deadline approaches, for example running red lights
ii) it significantly increases the number of possible states making it difficult for the agent to learn the rules of the game in the typical duration of a game
iii) The rules of the road, or the rules by which the agent has to drive, do not vary based on proximity to the deadline
iv) It could be possible to incentivise the agent to run a red light if the deadline is approaching, but this will likely increase the chance of an accident (see point i).

For the smartcab, I have allowed the state model to emerge over time. I have not set up an initial state space complete with default values as many of these may never happen (for example, we are unlikely in this simulation to see 3 cars at a particular junction).

Implement Q-Learning

The learnt reward for each state are held in a Python dictionary called ‘q’. This has a key based on the state_id from the above table.

Each entry in the q dictionary contains 4 values based on the possible actions that the smart cab can take:

{'forward': -1.0, 'right': -0.5, None: 0.0, 'left': -1.0}

The function choose_action() determines if the state that the cab is in is one that has been seen before. If it isn’t, then a random action is picked from the possible actions (None, forward, left, right), and if it is, then a choice is made from the actions with the highest q value. For example, if left and forward both have a q value of 2, then the agent will pick one of those at random.

What is obvious from implementing this functionality is that the cab reaches the destination much quicker and more purposefully than before. Its direction is generally towards the destination, whereas previous to this functionality, it would wander aimlessly around the environment.

This change in behaviour is due to learning about the rules of the environment and picking the action with the best reward. As the environment rewards for following the direction to the next waypoint, and provides a negative “reward” for not following the rules of the road, the cab learns that the best possible action is to follow the direction of the next waypoint while avoiding traffic accidents.

Enhancing the Agent

Using the approach defined above, after maybe 2-3 runs, the smart cab agent is regularly able to arrive at the destination within the deadline and with a positive net reward.

This is based on the long term reward being predominantly based on taking immediate actions that obey the rules of the road (e.g. these are short term actions). Therefore, it is difficult to tune the algorithm to improve it. If the rewards are discounted, then they are still relative to the default value of 0 and therefore the agent would learn just as quick (for example -0.01 and +0.05 would affect the likely actions in the same way as -0.1 and +0.5 or -1 and +0.5.

However, the code has been updated to enable an explore/exploit model with a threshold of 0.2 (so 80% of actions are based on the best q-value, 20% are based on a random choice). This had the effect of slowing down learning, but it does ensure that the agent understands more of the environment over time.