# P4 – SmartCab Project Write-up

## Implement a basic driving agent

## The initial code base makes decisions randomly as to which way the Smartcab should drive. This of course means that it rarely reaches its intended destination within the allotted time. The available actions for the cab are none (no action), forward, turn left and turn right. Picking these directions at random also means that the cab doesn’t pay attention to other cars or whether lights are red or green. It therefore also drives erratically and if this was a real car, it would cause many accidents!

In this simulation, there are times when the smart cab reaches its destination within the allotted time, but this is a very low percentage of times.

## 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'))

The rationale for including these in the state are:

1. The state of the lights indicates whether it is safe to move from the current location, and which cars are important to pay attention to. For example, in a green light it is safe to drive forwards irrespective of other cars, but we should check for cars coming from ahead if we want to turn left).
2. ‘oncoming, ‘left’ and ‘right’ indicate whether there are other cars approaching this location, and if so which direction they are travelling or likely to travel. This, in combination with the status of the lights, gives us an indication as to whether it is safe to move from this location.
3. The waypoint is the next direction that the planner is trying to send the smart cab. As the aim of this simulation is to get the cab to its destination, it is obviously important to bear in mind the direction that the cab needs to go. Without this, the cab would look to take the safest driving action, but it would still be directionless. In effect without this as part of the state, the cab would pick a random direction out of all of the driving actions with the best score (e.g. The safest actions)

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:

1. we do not wish for the agent to become reckless as the deadline approaches, for example running red lights
2. 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. In effect, the state-space would be too sparse and we would find it difficult to get q-values to converge within the timescales of the simulation
3. The rules of the road, or the rules by which the agent has to drive, do not vary based on proximity to the deadline
4. 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

Initial runs of the code only took the immediate reward as the q-value. Although this enabled the agent to learn the rules of the game very quickly, it was only ever motivated by the short term reward of its actions (e.g. did it follow the right directions from the planner and avoid any traffic violations). Although this can be seen as providing a suitable outcome, it does not enable the agent to take action based on future potential rewards. Obviously there is a balance between an immediate reward (or penalty) and a longer term reward (or penalty).

The code was then modified to provide the following variables that can be tuned to affect the way the agent learns:

1. Alpha – This is the learning rate of the cab and determines how much of the reward for a particular action and any potential future rewards is taken into account. A value of 1 means that the entirety of this reward and future rewards are used, and a value of 0 means that the agent won’t learn (as in won’t update its q-values).
2. Gamma – This enables tuning of how much of a potential future reward is used in modifying the q-values. A value of 0 means only the current reward is used, whereas a value of 1 means a greater importance is put on future rewards.
3. Epsilon – This determines whether the agent goes for a random action all of the time, goes with learned actions (decisions based on q-values) only, or a mix of the 2. During early runs of the project it is typically useful to explore the world more thoroughly, and therefore a higher percentage of actions are chosen randomly. However, as the agent understands more of the rules of its environment and q-values converge, it is possible to reduce the percentage of actions that are chosen randomly.
4. Policy - I have used this variable as an easy way to compare tuned parameters with a completely random approach. This variable can be set to ‘q’ to use q-values (default) or any other value to force completely random actions

The following section describes the results of various testing runs

### A Random Policy

An initial test was taken using a purely random approach to picking actions. As can be seen from the graphs below, the cumulative success rate over the 100 trials is around 25%. What is more striking is that net reward over this run, with a significant number of trials receiving a negative net reward. This could be caused by not following the planner’s directions or by traffic violations. The final graph shows the duration of each trial (so how long it takes the agent to reach the destination, if at all). There is no pattern to this, and it is obvious that the agent does not improve it’s performance over time.

For these reasons, it is obvious that a random approach is not a feasible policy.





