1) Basics

pygame downloaded, working on Python 2.7

2) Implement a basic driving agent

Task A) Algorithm produces some random move/action `(None, 'forward', 'left', 'right')`.

Implemented by using random choice from list (None, forward, left, right).

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

→The actions are chosen by random, so eventfully the agent will make it to the target location. There is a hard time limit implemented of 100 time steps, which are reached relatively often before the agent reaches the target. But there are also cases in which the agent does reach the target before those are done.

Running 100 trials with random movements for a time resulted values of between 11 and 19 for reaching the destination. Obviously, since there is no feed back (or learner) there is also no improvement

3) Identify and update state

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

→ The chosen state is defined as a tuple (so it can be a dictionary-key) consisting of the dictionary from inputs = self.env.sense(self) transformed into a tuple and a string for the waypoint from self.next_waypoint

Those values are chosen because the surrounding (cars, traffic lights from sens) and where I want to go (next waypoint) determine what I do (action).

The deadline is not included, because it would blow up the possible states too much and then the learning process would be very slow because we would not visit all the states often enough. A possibility would be to define an extra variable like a binary value for deadline bigger a certain threshold (for example 8 more steps) and smaller that threshold, thereby the policy could be split in one where it is best to move as quickly as possible to the goal, and one where it is best to circle around and collect rewards for correct driving for a long time.

All the sensor inputs are relevant. In detail they are: Traffic light color → has a big influence on the rewards we are getting, and then three values → for the three directions cars can come from (and can block the smartcab from driving in the desired direction).

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.

→ Mapping done through a dictionaty the state is the key and the values are Lists of actions (as string) and reward (as value)

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?

→ With the Q_learning enabled there are way more trials reaching the destination. Running it for a few trials of 100 trials we observe values of between 96 and 87agents reaching the destination.

This was done using a learning rate of 0.17 which was fixed bt randomly set

Enhance the driving agent

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?

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?

Changing the learning rate to a very low value of 0.01 leads to the following result 92, 91,89
Values of the learning rate of 1 lead to values like 95, 93, 92
It does seem to have a big influence in the given set up.
Learning rate of 1 with epsilon of 0 --> has a strong random element. There are if by coincidence there are a lot of good steps taken at the beginning, it is possible that there is a reasonable outcome. But most runs still perform poor.
With a learning rate of 0.5 and an epsilon of 0 the algorithm performs better then random but still not as great was with the higher epsilon observed above. The entire spectrum of trials was observed, from no agent hits the destination to 97 of the 100.
With a learning rate of 0.5 and an epsilon of 1.5 there are very constant and good results achieved.
Out of 100 trials the goal is reached between 96 and 99 times.

The chosen policy is close to optimal given the reward structure. Since it is a greedy implementation, the reward structure leads to a policy where driving over a red light (reward of -1) to reach the goal (+10) is a good thing. That should not happen in the real world. Since the simple_route_planner and teh act()-function lead to positive rewards if moving towards the goal there is the general tendency to move closer to the destination, this could probably be optimized though in speed.