Project 4: Reinforcement Learning

Train a Smartcab How to Drive

Implement a Basic Driving Agent

> QUESTION: Observe what you see with the agent's behavior as it takes random actions. Does the smartcab eventually make it to the destination? Are there any other interesting observations to note?

Answer:

The current agent randomly takes actions (from a set of 4) with total disregard of its environment. Since the environment assigns a penalty to each negative action, such as going forward on a red light or when other cars are coming, the current learner keeps getting penalized and is likely to never reach the destination.

From this we can infer that our smart(er) agent has to take into account its state and the waypoints within each step of the episode. By following these updated rules, the agent can avoid incurring negative rewards and arrive the destination in a more direct / intelligent way.

Trips	Percentage
Successful Trips	13%
Failed Trips	87%

Note: See `outputs/output_3.csv` for the entire list of trials.

Inform the Driving Agent

> QUESTION: What states have you identified that are appropriate for modeling the smartcab and environment? Why do you believe each of these states to be appropriate for this problem?
>
> OPTIONAL: How many states in total exist for the smartcab in this environment? Does this number seem reasonable given that the goal of Q-Learning is to learn and make informed decisions about each state? Why or why not?

Answer:

In the current environment, the givens which we can use to determine the state are the following:

- The `next_waypoint`: a string denoting the direction of the destination
- The traffic `light`
- The other cars, including `oncoming` traffic as well as from the `left` and `right`

This implementation of the smart agent will build a state from a subset of that the information above. Namely, `next_waypoint` for directions, `light` for traffic light signal and `oncoming` traffic.

Note: This implementation chooses to ignore `right` because we can safely assume that other agents (cars) on the grid will follow the driving rules outlined. While ignoring `left` can cause the agent to learn that taking a left turn on a red light is an permitted action, we will assume that ideally this will rarely happen in the current simulation.

Also, we choose to leave out the `deadline` from the agent's state since it does not effect the policy in any way. If the environment allowed the learner to do actions such as drive faster when deadline is getting closer to 0 or perhaps an extra reward for maximizing the time remaining in a trip, only then would it effect the outcome and we would include it in the state.

Adding more features to the state will add to the learning time so we will chose to ignore features that are not relevant to the agent's policy. To this point, we can also choose to ignore cars from both ways (`right` and `left`) since there is no penalty for crashing into other cars.

In technical terms, adding more information to the state with require us to fill out a larger QTable.

Let's have a look at the numbers.

The Factors:

A summary of the values of the agent's state is as follows:

- Waypoints: the direction could be 'Left', 'Right' or 'Forward'. (3 possible values).
- Lights: the traffic lights could either be 'green' or 'red'. (2 possible values).
- Oncoming: is there oncoming traffic. (2 possible values).

As for the responses the agent can have to its state:

- Actions: the actions the agent can take can be None, 'forward', 'left', 'right'. (4 possible values).

If we multiply each of the following values by each other => `3 x 2 x 4 x 2` , we get **48**. This will constitute all the possible combinations

between the state and the actions for our agent to add to its QTable.

Looking into the `Environment` class, we notice that it multiplies the distance by 5. Since the distance is always 1 to 12 steps from the agent to its destination. We conclude that the `deadline` is always between 5 to a maximum of 60 (56 possible values).

If we multiply the deadlines possible values => `48 x 56`, we get **2,688** combinations of state to actions which the agent needs to train for. Note that we can decrease this number to half (**1,344**) by dropping `oncoming` from the state if our environment does not penalize crashes between cars.

Implement a Q-Learning Driving Agent

> QUESTION: What changes do you notice in the agent's behavior when compared to the basic driving agent when random actions were always taken? Why is this behavior occurring?

Once we implement a Q-Learning Driving Agent, we notice that it performs much better than taking random actions. We also notice that the agent starts to follow the rules of the road after some trials, while the random agent does not learn from them in any way thus ignores the traffic rules all together.

The Q-Learning agent also starts to follow recommendations from the planner based on previous experiences, this means during later trials, the agent's performance increases. The random agent on the other hand keeps taking actions without learning in any way (i.e. past experiences have no effect on future actions).

About the implementation:

To implement a QTable for the current smartcab agent, a `QLearner` class was created. An instance `QLearner` class is then instantiated by the agent upon starting. The QTable initializes all values to 1 (arbitrarily chosen).

Once a QTable is created, the agent tries to maximize its reward by picking the action with the highest Q value for that state. This however only works when the state the agent is at has been previously encountered. Otherwise, a random action is picked (from the 4 possible actions).

Along with the newly implemented `QLearner`, the current implementation sets the following parameters:

- Number of trials is set to **100**
- The `enforce_deadline` is set to **True**
- The initial Q value for all actions at a new state, `Q_init` is set to **1**
- The discounting rate, `gamma`, is set to **0.5**
- The learning rate, `alpha`, is set to **0.5**
- The exploration rate, `epsilon`, is set to **0.5**

The results (see `outputs/output_5.csv`) of the current implementation are below:

Trips	Percentage
Successful Trips	60%
Failed Trips	40%

Since the values of `gamma`, `alpha` and `epsilon` are not optimized, we can conclude these values need to be changed in order to achieve a higher success rate.

Update:

Please note that in this implementation, I've chosen to set `epsilon` as the value of random action NOT taking place which I understand is unusual. For the sake of keeping the results in this report consistent, I will keep the current implementation but I intend to change it in the future.

Improve the Q-Learning Driving Agent

> QUESTION: Report the different values for the parameters tuned in your basic implementation of Q-Learning. For which set of parameters does the agent perform best? How well does the final driving agent perform?

In this sections, we will discuss how we can optimize the values following values for the agent to get closer to finding the optimal policy. For example, `epsilon` is the exploration rate, a higher value indicates less randomness in response to states the agent has already seen.

Below is a table of the 3 main learning values (`epsilon` , `alpha` and `gamma`) and their respective successful trips percentage once we have set the `update_delay` value to 0.1:

Experiment #	Success Rate	alpha	epsilon	gamma	Q_init
1	64%	0.5	0.5	0.5	2
2	57%	0.5	0.5	0.8	2
3	65%	0.5	0.5	0.25	2
4	71%	0.75	0.5	0.25	2
5	86%	0.75	0.75	0.25	2
6	91%	0.75	0.85	0.5	2

7	95%	0.75	0.85	0.4	2

During the first experiment, we set `alpha`, `gamma` and `epsilon` are to 0.5 arbitrarily as an exploratory step to set a benchmark with our Q-Learning implementation.

In **experiment #2** above (see `outputs/output_9.csv`), we notice that having a higher `gamma` value causes a drop in rate of successful trips for the agent.

In **experiment #4** (see `outputs/output_11.csv`), we set the learning rate, `alpha`, to 0.75 while setting `gamma` to 0.25 and we see an increase in our success rate. We could argue that this is due to our environment being predictable, i.e. the agent can repeat previously rewarding actions at a given state without having to consider possibly changing environment factors.

In **experiment #5** (see `outputs/output_12.csv`), we increase the predictability of how the agent will behave given that it has experienced this state previously by increasing `epsilon`. We note that achieve 86% success rate in this experiment.

We then keep changing those 3 variables until we converge to a success rate that is higher than 90% which we can consider acceptable for our learning agent. The final values are see above in **experiment #7** (see `outputs/output_15.csv`).

> QUESTION: 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? How would you describe an optimal policy for this problem?

Before answering the question, let's discuss the meaning of an optimal

policy for the current environment.

From the responses above, we can conclude that our **current environment does not care if the agent arrives with 1 step left in the `deadline` or 10 or 12**. This means that although having more steps remaining in the `deadline` is logically a better policy, it actually has no effect on the performance of the policy since the reward for arriving is always 12.

The optimal policy for this problem can be outlined by an agent's policy which ensures the following:

- Traffic rules are always obeyed.
- Planner suggestions are always followed.

While we can get close to the optimal policy, the agent might take certain actions with negative rewards if it hasn't been trained on these states often or at all (see case below from Trial #92). For example, since the agent rarely encounters other cars at traffic lights then it might end up taking the wrong actions, i.e. if a given state hasn't been visited by the agent, then the liklihood that it will take the correct action is lower.

```
State: (('directions', 'forward'), ('light', 'red'), ('oncom
Action: left, Reward: -1.0
Q-Values: {'forward': -0.7593967550529443, 'right': -0.02401
```

Towards to the last trials, the learning agent's QTable contains most states that are important for making the optimal decision. We observe by looking at the same state as the example above being encountered by the agent at a later trial (#93) but it now takes the correct action of stopping at the red light when the action is forward:

```
State: (('directions', 'forward'), ('light', 'red'), ('oncom
Action: None, Reward: 0.0
Q-Values: {'forward': -0.7593967550529443, 'right': -0.02401
```