Machine Learning Engineer Nanodegree

Train a Smart Cab Report

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.

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

By picking a random action out of the 4 available ones, the agent behaves randomly at any given intersection. However, it **does obey** the traffic light rules as that is enforced by the environment. There is also an existing set of rewards that is given by the environment, which is not being learned at all by the agent.

Without enforcing the deadline, it is guaranteed to be able to reach the destination *eventually* as it is capable of traversing to every single intersection under infinite time. It would just average out to take a really, really long time to do so.

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.

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

The set (tuple) of states that I feel is appropriate is listed below with their possible number of variations. The total number of (state, action) pairs in the Q table would be **2048**.

- **Next waypoint location (4):** This is included because we need the model to be able to learn that conforming to the next waypoint given by the planner will result in a higher reward. E.g. left on a *planned left* will yield more than .. right on a *planned left.*
- **Traffic light status (2):** To avoid going against the traffic rules and incurring penalties (going during red light etc), we need the model to be able to differentiate between between the 2, or 8 (state, action) pairs.
  E.g. going during red light < stopping during red light
- **Oncoming/left/right traffic status (4*4*4):** All of these are included. Otherwise, the model would be incapable of differentiating scenarios where there are vehicles with right of way (and action would be to stay put etc) vs scenarios where there isn't. Yes the qTable will be significantly harder to fill and the model may perform less optimally in this environment.
  But the fact that the environment does not penalize right of way violations does not mean that these information should be removed. Our optimal policy, discussed later in the report, should reduce penalties as much as possible so that if it is placed in an environment where right of way is penalized, the cab would still perform well.

- **Deadline** was not added into the states because the resulting q-Table would be way too many big and impossible to effectively fill up for the model.

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.

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

For the basic Q-learning algorithm, I started with a learning rate of **0.8** and discount rate of **0.2** ***at first.*** I've also just started off with a fixed exploration rate (epsilon) value of **0.2** so that the smart cab might sometimes explore instead of always exploiting. The results were definitely better in certain metrics; it was reaching the destination more reliably.

However, it was still incurring penalties. This could be because of the fixed epsilon value. 20% of the time it is still exploring, even if there is already sufficient visits in the q-Table to properly guide the cab.

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.

Question: 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?

It is important to first define *an optimal policy* to establish the basis of a well-trained model;

It should:

- Have low/no penalties
- Reach destination reliably below time limit,
- With a good value of reward to action ratio

I have prioritised '*no penalties*' as the most important criteria because I feel that, in the real world, a penalty should be avoided at all costs because it could result in an accident/crash.

I found that there are two ways to improve the model.

1st method: **Removing inputs *left and right* from the state**

- The removal of those inputs leads to a much smaller Q-table and thus requiring significantly less visits to train the Q-table. This will reduce the number of runs the cab requires to learn all the possible scenarios.
- However, it is not ideal. The number of runs (per simulation) that incurs penalties *should* increase. Intuitively, the car is unable to sense cars in adjacent squares due to the lack of information in the state.
- **The current environment may not show this effect** but with more cars or in an unique environment where there are more right of way rules, this model would not be as robust.
- *I also initially thought that the number of runs (per simulation) that incurs penalties should increase since the car is unable to sense cars in adjacent squares due to the lack of information in the state but it turns out that the environment does not give penalties for right-of-way violations so in this current environment, the penalty-runs don't actually increase even if I stripped these.*

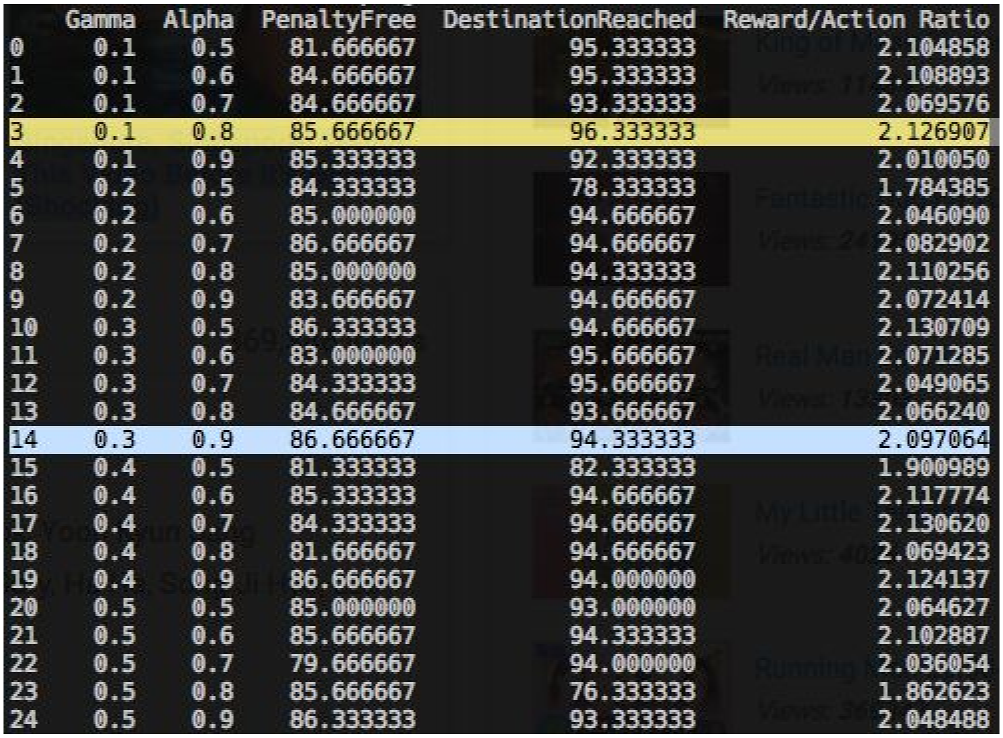
2nd method: **A dynamic epsilon value**

- Intuitively, when the Q table is relatively empty, it should favor exploration over exploitation. As the Q table fills up over time, it should slowly exploit more often.

- There are again 2 ways to achieve this: 1) setting a high default Q value such that unfilled (state, action) pairs in the Q-table will always be preferred because of the high default **OR** 2) a decaying Epsilon value.
- **I've chose to use a decaying epsilon.**
- Starting from an **initial value of 0.9**, the epsilon would **decay by a factor of ~0.83** *(0.9/1.2^run_index)*.
- This method yields a good performance. Out of 100 runs, it would typically reach the destination ~88-90 runs and be penalty-free ~85-90 runs. It may be slower to train but the fact that it remains penalty-free in more runs is more important.

To find the best possible combination of learning rate and discount rate values for the model, I ran each of the combinations through the model recording the metrics of each combination below:

- Number of runs (0-100) with no penalties
- Number of runs (0-100) reaching the destination
- Rewards to action ratio



```
    Gamma  Alpha  PenaltyFree  DestinationReached  Reward/Action Ratio
0     0.1    0.5    81.666667           95.333333             2.104858
1     0.1    0.6    84.666667           95.333333             2.108893
2     0.1    0.7    84.666667           93.333333             2.069576
3     0.1    0.8    85.666667           96.333333             2.126907
4     0.1    0.9    85.333333           92.333333             2.010050
5     0.2    0.5    84.333333           78.333333             1.784385
6     0.2    0.6    85.000000           94.666667             2.046090
7     0.2    0.7    86.666667           94.666667             2.082902
8     0.2    0.8    85.000000           94.333333             2.110256
9     0.2    0.9    83.666667           94.666667             2.072414
10    0.3    0.5    86.333333           94.666667             2.130709
11    0.3    0.6    83.000000           95.666667             2.071285
12    0.3    0.7    84.333333           95.666667             2.049065
13    0.3    0.8    84.666667           93.666667             2.066240
14    0.3    0.9    86.666667           94.333333             2.097064
15    0.4    0.5    81.333333           82.333333             1.900989
16    0.4    0.6    85.333333           94.666667             2.117774
17    0.4    0.7    84.333333           94.666667             2.130620
18    0.4    0.8    81.666667           94.666667             2.069423
19    0.4    0.9    86.666667           94.000000             2.124137
20    0.5    0.5    85.000000           93.000000             2.064627
21    0.5    0.6    85.666667           94.333333             2.102887
22    0.5    0.7    79.666667           94.000000             2.036054
23    0.5    0.8    85.666667           76.333333             1.862623
24    0.5    0.9    86.333333           93.333333             2.048488
```

All of the above run with a **decaying epsilon starting from 0.9**.

From the values above, we can see that the following 2 combinations work best:

- **Discount rate - Gamma: 0.1,   Learning rate - Alpha: 0.8**
- Discount rate - Gamma: 0.3,   Learning rate - Alpha: 0.9

Out of 100 runs, the final model effectively runs **penalty free for ~87 runs,** and reaches the destination under the allowed time **94-96 times** which I'd say is a rather good performance.

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?

The model takes only about 5-7 trials *in this environment* to learn that in a state where the traffic light allows movement, following the next_waypoint yields the most reward which would naturally motivate it to follow the planned route most of the time. I'll argue that it is *close* to a policy that gets to the destination in the shortest time.

However, when it is in a state where it senses surrounding cars, it is still making mistakes (not following the way planner). It would usually take a direction that was tried in an earlier run that obeys the traffic light which would've yielded 0.5 reward. This causes it to sometimes go in a roundabout fashion that does not incur a red/green light penalty but still isn't in the correct direction.

In terms of learning and obeying the rights of way traffic rules and avoiding penalties, I'd say that it is lacking. The reasons that lead to a suboptimal policy in terms of avoiding penalties are:

- A huge number of states that needs to be visited due to all the inputs being considered and with only 2 other cars, the chances of getting in the states required to learn the right of way traffic rules, are very very slim.
- No negative score for violating the right of way (only for green/red lights)

To improve the model for this environment, we could consider increasing training runs exponentially, increasing the number of cars in training, OR stripping the surrounding cars from the states. But as discussed, I feel that the latter would lead to a model that doesn't generalize as well to other environments as it would be incapable of differentiating scenarios where it should move towards the direction, vs one where it should give way to other cars.