Reinforcement Learning Project: Train a Smartcab How to Drive

Project Overview

In this project you will apply reinforcement learning techniques for a self-driving agent in a simplified world to aid it in effectively reaching its destinations in the allotted time. You will first investigate the environment the agent operates in by constructing a very basic driving implementation. Once your agent is successful at operating within the environment, you will then identify each possible state the agent can be in when considering such things as traffic lights and oncoming traffic at each intersection. With states identified, you will then implement a Q-Learning algorithm for the self-driving agent to guide the agent towards its destination within the allotted time. Finally, you will improve upon the Q-Learning algorithm to find the best configuration of learning and exploration factors to ensure the self-driving agent is reaching its destinations with consistently positive results.

Description

In the not-so-distant future, taxicab companies across the United States no longer employ human drivers to operate their fleet of vehicles. Instead, the taxicabs are operated by self-driving agents, known as **smartcabs**, to transport people from one location to another within the cities those companies operate. In major metropolitan areas, such as Chicago, New York City, and San Francisco, an increasing number of people have come to rely on **smartcabs** to get to where they need to go as safely and efficiently as possible. Although **smartcabs** have become the transport of choice, concerns have arose that a self-driving agent might not be as safe or efficient as human drivers, particularly when considering city traffic lights and other vehicles. To alleviate these concerns, your task as an employee for a national taxicab company is to use reinforcement learning techniques to construct a demonstration of a **smartcab** operating in real-time to prove that both safety and efficiency can be achieved.

Definitions

Environment

The **smartcab** operates in an ideal, grid-like city (similar to New York City), with roads going in the North-South and East-West directions. Other vehicles will certainly be present on the road, but there will be no pedestrians to be concerned with. At each intersection there is a traffic light that either allows traffic in the North-South direction or the East-West direction. U.S. Right-of-Way rules apply:
- On a green light, a left turn is permitted if there is no oncoming traffic making a right turn or coming straight through the intersection.
- On a red light, a right turn is permitted if no oncoming traffic is approaching from your left through the intersection.
To understand how to correctly yield to oncoming traffic when turning left, you may refer to [this official drivers? education video](https://www.youtube.com/watch?v=TW0Eq2Q-9Ac), or [this passionate exposition](https://www.youtube.com/watch?v=0EdkxI6NeuA).

Inputs and Outputs

Assume that the **smartcab** is assigned a route plan based on the passengers? starting location and destination. The route is split at each intersection into waypoints, and you may assume that the **smartcab**, at any instant, is at some intersection in the world. Therefore, the next waypoint to the destination, assuming the destination has not already been reached, is one intersection away in one direction (North, South, East, or West). The **smartcab** has only an egocentric view of the intersection it is at: It can determine the state of the traffic light for its direction of movement, and whether there is a vehicle at the intersection for each of the oncoming directions. For each action, the **smartcab** may either idle at the intersection, or drive to the next intersection to the left, right, or ahead of it. Finally, each trip has a time to reach the destination which decreases for each action taken (the passengers want to get there quickly). If the allotted time becomes zero before reaching the destination, the trip has failed.

Rewards and Goal

The **smartcab** receives a reward for each successfully completed trip, and also receives a smaller reward for each action it executes successfully that obeys traffic rules. The **smartcab** receives a small penalty for any incorrect action, and a larger penalty for any action that violates traffic rules or causes an accident with another vehicle. Based on the rewards and penalties the **smartcab** receives, the self-driving agent implementation should learn an optimal policy for driving on the city roads while obeying traffic rules, avoiding accidents, and reaching passengers? destinations in the allotted time.

Tasks

Implement a Basic Driving Agent

To begin, your only task is to get the **smartcab** to move around in the environment. At this point, you will not be concerned with any sort of optimal driving policy. Note that the driving agent is given the following information at each intersection:
- The next waypoint location relative to its current location and heading.
- The state of the traffic light at the intersection and the presence of oncoming vehicles from other directions.
- The current time left from the allotted deadline.

To complete this task, simply have your driving agent choose a random action from the set of possible actions (`None`, `'forward'`, `'left'`, `'right'`) at each intersection, disregarding the input information above. Set the simulation deadline enforcement, `enforce_deadline` to `False` and observe how it performs.

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?_

When the action taken by the smartcab is random, then it very rarely reaches the destination. Most often, it just times out the hard time limit.
Also the smartcab makes many very serious mistakes, like bumping into oncoming vehicles paying no heed to the traffic lights, etc.

For the purposes of quantifying how well the agent does, I define a metric called success_rate which indicates the percentage of trials the agent successfully reaches the destination within the deadline.
I also output the parameter values of the experiment:

- random_action: indicates whether the agent does only random actions or does it do Q-learning. If this parameter value is true, then the agent takes only random actions. If the value is false, then the agent does Q-learning.

- alpha, gamma, epsilon: the parameters of Q-learning. These parameters are valid only in the case where random_action is false.
- success_rate: In what percentage of trials was the learning agent able to successfully reach the destination within the allotted time (a high value, closer to 1 indicates that our agent is doing well)
- avg_reward: what is the average reward that our agent is getting averaged over all the trials (note that penalties are negative rewards). A high value indicates that our agent is doing well (i.e getting more positive rewards or getting lesser penalties).

An output will look like this:

random_action=True, alpha=None, gamma=None, eps=None, success_rate=0.18, avg_reward=1.035

Note that the above output tells us that the agent took random actions and only 18% of the time did it reach the destination within the allotted time. This is an actual output from my experiment when random action was set to true. So it tells us that the agent does indeed perform very poorly with random actions. Also we can see that the average reward per trial is only 1.035, so this is rather poor and very likely the agent commits many mistakes so it gets many negative rewards.

Inform the Driving Agent

Now that your driving agent is capable of moving around in the environment, your next task is to identify a set of states that are appropriate for modeling the **smartcab** and environment. The main source of state variables are the current inputs at the intersection, but not all may require representation. You may choose to explicitly define states, or use some combination of inputs as an implicit state. At each time step, process the inputs and update the agent's current state using the `self.state` variable. Continue with the simulation deadline enforcement `enforce_deadline` being set to `False`, and observe how your driving agent now reports the change in state as the simulation progresses.

*****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?_**

I have identified the state as a tuple of {inputs, next_waypoint, deadline }. I am maintaining a tuple for this so that the state can be nicely indexed into a dictionary.

Description of the sub-items used in the state:
Inputs:
At each intersection, the agent can sense the environment and the environment contains details like what is the signal light at the intersection, where the oncoming vehicles are headed and also where the vehicle to the left and the right are headed. As an example, If the vehicle to the left is headed straight vs turning right, our current agent might have to take different actions to optimize the reward. So the inputs are each intersection form an ideal candidate for state.

Note that we do not include the entire input information as part of the current state. This is because, the other agents in the right do not have any impact on the action that our agent takes. For example, we can take a free right, depending only upon the traffic light and the oncoming and left coming agents. So we do not include the next way point of the right agents as part of our current state.

Next way point:
The next way point tells where the agent is headed next given the current circumstances. In the Q-learning, based on the state, we want to decide on an optimal action given that the agent is in the current state. Since the next way point of the agent decides in which direction the agent will go next, it also forms part of the state, because if it is going in the correct direction, then we want or q-learning algorithm to complement that direction by saying that the optimal action is indeed the direction that the agent decides to take next. On the other hand, if

the next way point that the agent planner decides is not optimal at all, then we want our q-learning algorithm to say that a different action would be optimal in this case.

Deadline:
We also consider the deadline as a binary variable, i.e is our agent more than 10 time steps away or less than 10 time steps away. Based on that, the agent may choose different actions that are optimal.
(Note that we do not use a continuous value for the deadline so that we can avoid the explosion of states that would result because of that)

*****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?_**
To calculate the total number of states, we need to know the total number of values taken by the following variables:

- Light : 2 values
- Oncoming agents : 4 values (none, forward, left and right)
- Left agents : 4 values
- Next way point : 3 values (left, right and forward)
- Deadline : 2 values (since it is a binary value in our case)

So totally, there can be 2*4*4*3*2 states that is 192 states. This number is reasonable and easily within the realm of efficient Q value computation for each state provided we train over 100s of trials. (Note that the Q table will contain 4 action values for each of the 384 states; i.e values for action = none, forward, left and right, so there would be 768 entries in the Q table)

Implement a Q-Learning Driving Agent

With your driving agent being capable of interpreting the input information and having a mapping of environmental states, your next task is to implement the Q-Learning algorithm for your driving agent to choose the *best* action at each time step, based on the Q-values for the current state and action. Each action taken by the **smartcab** will produce a reward which depends on the state of the environment. The Q-Learning driving agent will need to consider these rewards when updating the Q-values. Once implemented, set the simulation deadline enforcement `enforce_deadline` to `True`. Run the simulation and observe how the **smartcab** moves about the environment in each trial.

The formulas for updating Q-values can be found in
[this](https://classroom.udacity.com/nanodegrees/nd009/parts/0091345409/modules/e64f9a65-fdb5-4e60-81a9-72813beebb7e/lessons/5446820041/concepts/6348990570923) video.

*****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?_**
I fixed the parameters of the Q learning as follows:
Alpha = 0.1
Gamma = 0.9
Epsilon = 0.1
This results in the following success rate:
alpha=0.1, gamma=0.9, eps=0.105, success_rate=0.54, avg_reward=21.84
As we can see this is an improvement over the previous random action success rate of 18%, however this is far from ideal and we would like to tune the parameters further to select the optimally performing params.

In the next section, we will see how to tune these parameters and identify the best values. For now we will go with the above values. Now the agent starts performing better after a few initial trials have been run and it has learnt the Q values to some extent.
Note that in the initial trials, the agent still performs poorly; this is because the Q values take some time to converge based on the parameters and so the initial trials still look a lot like random actions. But over time and particularly towards the later trials, the agent starts performing rather well and in some cases also takes the optimal route to the destination.
Note that the Q value update formula is given by:
Q(s,a) ← (alpha) ---- R + Gamma (max_a' :: Q(s', a'))

i.e Let's say that from state s, after taking action a we arrive at new state s'.
Now, to update the Q value at state s and action a, we take the existing Q value at that place and multiply it by (1-alpha). To that we add, alpha * (current_reward + max Q values at state s').
So this is basically, current reward + max expeted utility at next state.

So over time, the Q value at a given state and action will converge to the true utility of taking that action when in that state.

Improve the Q-Learning Driving Agent

Your final task for this project is to enhance your driving agent so that, after sufficient training, the **smartcab** is able to reach the destination within the allotted time safely and efficiently. Parameters in the Q-Learning algorithm, such as the learning rate (`alpha`), the discount factor (`gamma`) and the exploration rate (`epsilon`) all contribute to the driving agent?s ability to learn the best action for each state. To improve on the success of your **smartcab**:
- Set the number of trials, `n_trials`, in the simulation to 100.
- Run the simulation with the deadline enforcement `enforce_deadline` set to `True` (you will need to reduce the update delay `update_delay` and set the `display` to `False`).
- Observe the driving agent?s learning and **smartcab?s** success rate, particularly during the later trials.
- Adjust one or several of the above parameters and iterate this process.

This task is complete once you have arrived at what you determine is the best combination of parameters required for your driving agent to learn successfully.

*****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?_**
I tried various values for alpha, gamma and epsilon for this problem. Refer the below code snippet for how this was achieved.

```
for alpha in np.linspace(0.1, 0.9, num=5):
    for gamma in np.linspace(0.1, 0.9, num=5):
        for eps in np.linspace(0.01, 0.2, num=5):
            e = Environment()
            kwargs = {'alpha': alpha, 'gamma': gamma, 'eps': eps}
            a = e.create_agent(LearningAgent, **kwargs)
            e.set_primary_agent(a, enforce_deadline=True)
            sim = Simulator(e, update_delay=0.0, display=True)
            sim.run(n_trials=100)
```

Various values for the above 3 parameters (i.e alpha, gamma and epsilon) gave pretty good results, but in the end I choose the values alpha = 0.3, gamma = 0.3, epsilon = 0.01 as this converged to the close to optimal solution rather quickly. In the latter part of the trials, this achieves close to a 96% completion rate within the allotted time and with very minimal penalties for wrong/dangerous actions. A note on the 3 parameters,

Alpha tells us how much we want to keep the old Q value when we update the Q value. If alpha is equal to 0, then we will keep the entirety of the old Q value hence not learning anything at all. If alpha is equal to 1, then we completely ignore the old Q value and only take the new learned value. Both of these approaches will not yield a good solution as we either do not learn at all or we ignore completely what we reinforced previously through several iterations. A better approach would be to use a value of alpha that tries to learn new things but also reinforces/consolidates what it previous learnt. In this case 0.3 seems to do quite well.

Gamma tells us how much we want to decay the Q value that we will see in the future states. If gamma is close to 1, then we don't decay the future weights at all. i.e we give equal importance to the future anticipated rewards as we give to the current actual reward. If gamma is close to 0, then we almost ignore the future rewards and only choose an action based on the current immediate reward. In this case gamma of 0.3 gives us good experimental results. i.e we don't completely zero out the future rewards, but rather decay them by a fair bit. We can argue that in this smartcab case, it seems reasonable to do so, because in the future states, other vehicles may reach our intersection and it will be difficult to forsee too much into the future, so we will be conservative and choose the action by giving a much larger weight to the immediate surrounding.

Epsilon tells us how much we want to explore, i.e instead of following the Q table, choose a completely random action. In our case a 1% exploration seems to do just fine.

I tried various values for the 3 parameters, overall I tried around 125 different combinations of values and in each combination, I ran 100 trials. Following are some of the intermediate values for various values of alpha, gamma and epsilon:

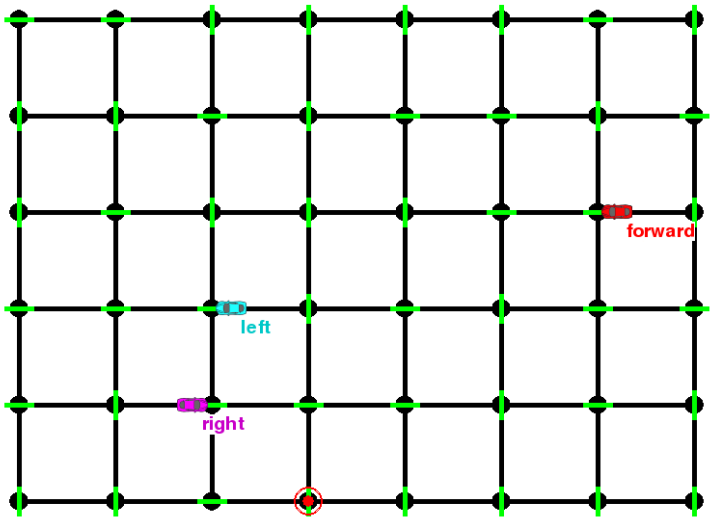
alpha	gamma	epsilon	Success_rate	Avg_reward
0.1	0.9	0.1	54 %	21.84
0.1	0.9	0.01	31 %	20.6
0.3	0.7	0.2	68 %	22.2
0.3	**0.3**	**0.01**	**96 %**	**23.9**
0.5	0.3	0.2	81 %	20.77
0.5	0.7	0.05	83 %	24.7
0.7	0.1	0.1	92 %	21.2
0.7	0.7	0.01	86 %	24.5
0.9	0.9	0.15	90 %	23.38
0.9	0.9	0.2	58 %	19.86

So based on the above, we can select the parameters as alpha=0.3, gamma=0.3 and exploration_rate=1%

*****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?_**

The agent does indeed follow an almost optimal policy after it has learned the Q values over a large number of iterations. As an example, consider the following sequence of moves for the given initial state that is pictured below: (Note that before this particular example, the agent was trained for 100 runs with parameters 'alpha': 0.3, 'gamma': 0.3, 'eps': 0.01.

state: ({'light': 'green', 'oncoming': None, 'right': None, 'left': None}, 'forward')
action: forward
reward: 12.0



The moves are

- inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None}, action = left
- inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = right
- inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = None
- inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = None
- inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = None
- inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None}, action = forward
- inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = None
- inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = None
- inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = None
- inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = None
- inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None}, action = forward
- inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None}, action = left
- inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = None
- inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = None
- inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = None
- inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None}, action = forward

As we can see above, the agent stops when the light is red and it needs to move in the forward direction. Also the agent takes a free right, even when the light is red because there is no traffic coming from the left. Also the route that the agent follows from the initial state is the optimal route in this case.

How to describe an optimal policy for the learning agent:

An optimal policy for this problem would be the agent following the shortest possible route given the state of the traffic lights and the presence of other agents and their states. But since we cannot guess the movement of the other agents multiple time steps into the future, an optimal policy for our agent would be a greedy approach, i.e take the best possible move given the current state of the traffic intersection and the other agents that are present at that intersection.

So following is a greedy way of calculating the optimal number of steps from source to destination:

- Lower bound : We can count the number of intersections in the manhattan distance between the starting point and the destination. This would form a lower bound for the optimal route, meaning we cannot go below this.
- Upper bound: However this cannot always be followed, for instance when there the signal is showing red, the agent has to stop if it is moving forward. So if we add the manhattan distance to the number of times the signal shows red as the agent moves along the manhatan path, then this would form a reasonable estimate (modulo the free right rule) for the optimal distance from source to destination. (Ideally, to the number obtained previously, we can subtract the number of times our agent takes a free right even on red signal and we can also add the number of times our agent is unable to take a left on a green signal due to oncoming traffic to get a more accurate upper bound on the optimal distance)