smartcab

February 9, 2017

1 Machine Learning Engineer Nanodegree

1.1 Reinforcement Learning

1.2 Project: Train a Smartcab to Drive

Welcome to the fourth project of the Machine Learning Engineer Nanodegree! In this notebook, template code has already been provided for you to aid in your analysis of the *Smartcab* and your implemented learning algorithm. You will not need to modify the included code beyond what is requested. There will be questions that you must answer which relate to the project and the visualizations provided in the notebook. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide in `agent.py`.

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. In addition, Markdown cells can be edited by typically double-clicking the cell to enter edit mode.

1.3 Getting Started

In this project, you will work towards constructing an optimized Q-Learning driving agent that will navigate a *Smartcab* through its environment towards a goal. Since the *Smartcab* is expected to drive passengers from one location to another, the driving agent will be evaluated on two very important metrics: **Safety** and **Reliability**. A driving agent that gets the *Smartcab* to its destination while running red lights or narrowly avoiding accidents would be considered **unsafe**. Similarly, a driving agent that frequently fails to reach the destination in time would be considered **unreliable**. Maximizing the driving agent's **safety** and **reliability** would ensure that *Smartcabs* have a permanent place in the transportation industry.

Safety and **Reliability** are measured using a letter-grade system as follows:

Grade	Safety	Reliability
A+	Agent commits no traffic violations,and always chooses the correct action.	Agent reaches the destination in timefor 100% of trips.
A	Agent commits few minor traffic violations,such as failing to move on a green light.	Agent reaches the destination on timefor at least 90% of trips.
B	Agent commits frequent minor traffic violations,such as failing to move on a green light.	Agent reaches the destination on timefor at least 80% of trips.
C	Agent commits at least one major traffic violation, such as driving through a red light.	Agent reaches the destination on timefor at least 70% of trips.

Grade	Safety	Reliability
D	Agent causes at least one minor accident, such as turning left on green with oncoming traffic.	Agent reaches the destination on timefor at least 60% of trips.
F	Agent causes at least one major acci- dent,such as driving through a red light with cross- traffic.	Agent fails to reach the destination on timefor at least 60% of trips.

To assist evaluating these important metrics, you will need to load visualization code that will be used later on in the project. Run the code cell below to import this code which is required for your analysis.

```
In [1]: # Import the visualization code
        import visuals as vs

        # Pretty display for notebooks
        %matplotlib inline
```

1.3.1 Understand the World

Before starting to work on implementing your driving agent, it's necessary to first understand the world (environment) which the *Smartcab* and driving agent work in. One of the major components to building a self-learning agent is understanding the characteristics about the agent, which includes how the agent operates. To begin, simply run the `agent.py` agent code exactly how it is – no need to make any additions whatsoever. Let the resulting simulation run for some time to see the various working components. Note that in the visual simulation (if enabled), the **white vehicle** is the *Smartcab*.

1.3.2 Question 1

In a few sentences, describe what you observe during the simulation when running the default `agent.py` agent code. Some things you could consider: - *Does the Smartcab move at all during the simulation? - What kind of rewards is the driving agent receiving? - How does the light changing color affect the rewards?*

Hint: From the `/smartcab/` top-level directory (where this notebook is located), run the command

`'python smartcab/agent.py'`

Answer:
"Does the Smartcab move at all during the simulation?"
No. Smartcab's state isn't updated anytime when running the default agent.py.
"What kind of rewards is the driving agent receiving?"
The agent receives a positive or negative reward depending on the kind of action he took in its current state based on safety and reliability.
"How does the light changing color affect the rewards?"
It receives a positive reward when not moving toward a red light and a negative reward otherwise. More generally, the agent receives a positive reward when not commit traffic violations and a negative reward otherwise.

1.3.3 Understand the Code

In addition to understanding the world, it is also necessary to understand the code itself that governs how the world, simulation, and so on operate. Attempting to create a driving agent would be difficult without having at least explored the *"hidden"* devices that make everything work. In the `/smartcab/` top-level directory, there are two folders: `/logs/` (which will be used later) and `/smartcab/`. Open the `/smartcab/` folder and explore each Python file included, then answer the following question.

1.3.4 Question 2

- *In the* `agent.py`* Python file, choose three flags that can be set and explain how they change the simulation.*
- *In the* `environment.py`* Python file, what Environment class function is called when an agent performs an action?*
- *In the* `simulator.py`* Python file, what is the difference between the* `'render_text()'` function and the `'render()'` function?*
- *In the* `planner.py`* Python file, will the* `'next_waypoint()` function consider the North-South or East-West direction first?*

Answer:
"*In the* `agent.py`* Python file, choose three flags that can be set and explain how they change the simulation.*"
The flag names are self-describing (good programming practice):

1) num_dummies: the number of agents in the environment. The greater the value of num_dummies, the greater the number of cars on screen.

2) grid_size: the number of streets in the screen.

3) learning: when True will make the agent use the Q-learning function.

   "In the `environment.py`* Python file, what Environment class function is called when an agent performs an action?*"
   act() function.
   "In the `simulator.py`* Python file, what is the difference between the `'render_text()'` function and the `'render()'` function?*"
   render_text() updates the command prompt while render() updates the game screen (with the cars, traffic, streets...).
   "In the `planner.py`* Python file, will the `'next_waypoint()` function consider the North-South or East-West direction first?*"
   It considers East-West. Firstly, it checks if dx is different from zero and take an action. Only if dx equals zero, it checks whether dy is different from zero to consider north-south.

1.4   Implement a Basic Driving Agent

The first step to creating an optimized Q-Learning driving agent is getting the agent to actually take valid actions. In this case, a valid action is one of `None`, (do nothing) `'Left'` (turn left), `'Right'` (turn right), or `'Forward'` (go forward). For your first implementation, navigate to the `'choose_action()'` agent function and make the driving agent randomly choose one of these actions. Note that you have access to several class variables that will help you write this functionality, such as `'self.learning'` and `'self.valid_actions'`. Once implemented, run the agent file and simulation briefly to confirm that your driving agent is taking a random action each time step.

1.4.1   Basic Agent Simulation Results

To obtain results from the initial simulation, you will need to adjust following flags: - `'enforce_deadline'` - Set this to `True` to force the driving agent to capture whether it reaches the destination in time. - `'update_delay'` - Set this to a small value (such as `0.01`) to reduce the time between steps in each trial. - `'log_metrics'` - Set this to `True` to log the simluation results as a `.csv` file in `/logs/`. - `'n_test'` - Set this to `'10'` to perform 10 testing trials.

Optionally, you may disable to the visual simulation (which can make the trials go faster) by setting the `'display'` flag to `False`. Flags that have been set here should be returned to their default setting when debugging. It is important that you understand what each flag does and how it affects the simulation!

Once you have successfully completed the initial simulation (there should have been 20 training trials and 10 testing trials), run the code cell below to visualize the results. Note that log files are overwritten when identical simulations are run, so be careful with what log file is being loaded! Run the agent.py file after setting the flags from projects/smartcab folder instead of projects/smartcab/smartcab.

```
In [2]:  # Load the 'sim_no-learning' log file from the initial simulation results
         vs.plot_trials('sim_no-learning.csv')
```
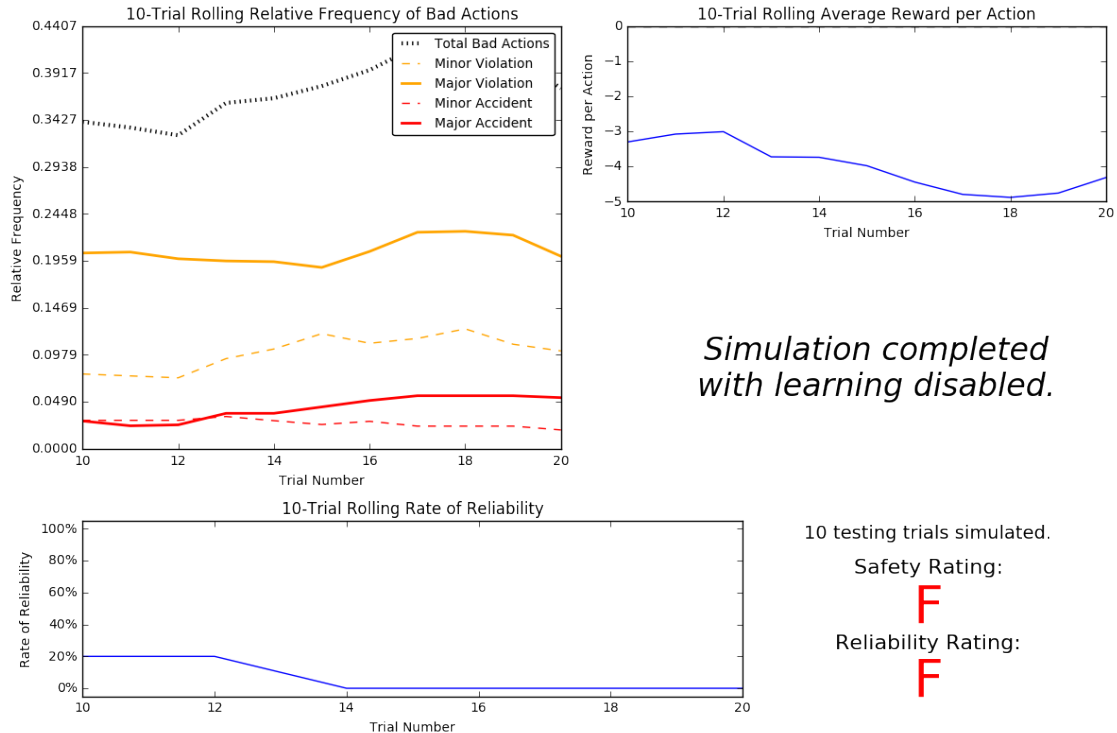
1.4.2 Question 3

Using the visualization above that was produced from your initial simulation, provide an analysis and make several observations about the driving agent. Be sure that you are making at least one observation about each panel present in the visualization. Some things you could consider: - *How frequently is the driving agent making bad decisions? How many of those bad decisions cause accidents? - Given that the agent is driving randomly, does the rate of reliabilty make sense? - What kind of rewards is the agent receiving for its actions? Do the rewards suggest it has been penalized heavily? - As the number of trials increases, does the outcome of results change significantly? - Would this Smartcab be considered safe and/or reliable for its passengers? Why or why not?*

Answer:

- *How frequently is the driving agent making bad decisions? How many of those bad decisions cause accidents?*

The driving agent is making bad decisions constantly. In between 34% to 44% of its decisions were considered bad decisions. Accidents - considering minor and major - occurred in almost 8% of its decisions.

- *Given that the agent is driving randomly, does the rate of reliabilty make sense?*

It doesn't. Since the agent isn't learning anything, achieving the goal is purely random.

- *What kind of rewards is the agent receiving for its actions? Do the rewards suggest it has been penalized heavily?*

6

The agent receives a positive reward for good actions and negative rewards for bad actions taken. Observing the mean of reward per action taken in each trial, we can see that the agent is heavily penalized, and it is fair since a bad action taken is much more impacting to the agent than a positive action.

- *As the number of trials increases, does the outcome of results change significantly?*

No. Since the decisions of actions to take are randomly chosen, the results tend to converge to the mean.

- *Would this Smartcab be considered safe and/or reliable for its passengers? Why or why not?*

This Smartcab is neither safe or reliable. Isn't safe due to the number of accidents involved and not reliable since it doesn't achieve its goal almost every trial.

1.5   Inform the Driving Agent

The second step to creating an optimized Q-learning driving agent is defining a set of states that the agent can occupy in the environment. Depending on the input, sensory data, and additional variables available to the driving agent, a set of states can be defined for the agent so that it can eventually *learn* what action it should take when occupying a state. The condition of `'if state then action'` for each state is called a **policy**, and is ultimately what the driving agent is expected to learn. Without defining states, the driving agent would never understand which action is most optimal – or even what environmental variables and conditions it cares about!

1.5.1   Identify States

Inspecting the `'build_state()'` agent function shows that the driving agent is given the following data from the environment: - `'waypoint'`, which is the direction the *Smartcab* should drive leading to the destination, relative to the *Smartcab*'s heading. - `'inputs'`, which is the sensor data from the *Smartcab*. It includes - `'light'`, the color of the light. - `'left'`, the intended direction of travel for a vehicle to the *Smartcab*'s left. Returns `None` if no vehicle is present. - `'right'`, the intended direction of travel for a vehicle to the *Smartcab*'s right. Returns `None` if no vehicle is present. - `'oncoming'`, the intended direction of travel for a vehicle across the intersection from the *Smartcab*. Returns `None` if no vehicle is present. - `'deadline'`, which is the number of actions remaining for the *Smartcab* to reach the destination before running out of time.

1.5.2   Question 4

*Which features available to the agent are most relevant for learning both **safety** and **efficiency**? Why are these features appropriate for modeling the* Smartcab* in the environment? If you did not choose some features, why are those features* not *appropriate?*

Answer: - Which features available to the agent are most relevant for learning both **safety** and **efficiency**?

For safety and efficiency, the most important features are the input features: waypoint, lights, left, right and oncoming.

For only efficiency, the most important is waypoint. For only safety, the most important are the inputs.

- Why are these features appropriate for modeling the *Smartcab* in the environment?

Safety cares about the protection of the agent. It doesn't matter if the agent achieved its goal of going to some place/state, it commits exclusively to learn how to avoid problems like traffic violations and accidents, and the inputs feature are related to that.

Efficiency, in another hand, cares about the agent reach its destination. So, knowing which direction to take is important.

1.5.3 Define a State Space

When defining a set of states that the agent can occupy, it is necessary to consider the *size* of the state space. That is to say, if you expect the driving agent to learn a **policy** for each state, you would need to have an optimal action for *every* state the agent can occupy. If the number of all possible states is very large, it might be the case that the driving agent never learns what to do in some states, which can lead to uninformed decisions. For example, consider a case where the following features are used to define the state of the *Smartcab*:

```
('is_raining', 'is_foggy', 'is_red_light', 'turn_left',
'no_traffic', 'previous_turn_left', 'time_of_day').
```

How frequently would the agent occupy a state like (`False`, `True`, `True`, `True`, `False`, `False`, `'3AM'`)? Without a near-infinite amount of time for training, it's doubtful the agent would ever learn the proper action!

1.5.4 Question 5

*If a state is defined using the features you've selected from **Question 4**, what would be the size of the state space? Given what you know about the evironment and how it is simulated, do you think the driving agent could learn a policy for each possible state within a reasonable number of training trials?*
Hint: Consider the *combinations* of features to calculate the total number of states!

Answer: Selecting waypoint, light, left, right and oncoming and assuming that 1) any combination is allowed, and 2) each feature, with exception of light, can assume 4 values (left, right, forward and None), we obtain 512 different possible states.

1.5.5 Update the Driving Agent State

For your second implementation, navigate to the `'build_state()'` agent function. With the justification you've provided in **Question 4**, you will now set the `'state'` variable to a tuple of all the features necessary for Q-Learning. Confirm your driving agent is updating its state by running the agent file and simulation briefly and note whether the state is displaying. If the visual simulation is used, confirm that the updated state corresponds with what is seen in the simulation.

Note: Remember to reset simulation flags to their default setting when making this observation!

1.6 Implement a Q-Learning Driving Agent

The third step to creating an optimized Q-Learning agent is to begin implementing the functionality of Q-Learning itself. The concept of Q-Learning is fairly straightforward: For every state the agent visits, create an entry in the Q-table for all state-action pairs available. Then, when the agent

encounters a state and performs an action, update the Q-value associated with that state-action pair based on the reward received and the interative update rule implemented. Of course, additional benefits come from Q-Learning, such that we can have the agent choose the *best* action for each state based on the Q-values of each state-action pair possible. For this project, you will be implementing a *decaying, ϵ-greedy* Q-learning algorithm with *no* discount factor. Follow the implementation instructions under each **TODO** in the agent functions.

Note that the agent attribute `self.Q` is a dictionary: This is how the Q-table will be formed. Each state will be a key of the `self.Q` dictionary, and each value will then be another dictionary that holds the *action* and *Q-value*. Here is an example:

```
{ 'state-1': {
    'action-1' : Qvalue-1,
    'action-2' : Qvalue-2,
     ...
   },
  'state-2': {
    'action-1' : Qvalue-1,
     ...
   },
   ...
}
```

Furthermore, note that you are expected to use a *decaying ϵ (exploration) factor*. Hence, as the number of trials increases, ϵ should decrease towards 0. This is because the agent is expected to learn from its behavior and begin acting on its learned behavior. Additionally, The agent will be tested on what it has learned after ϵ has passed a certain threshold (the default threshold is 0.01). For the initial Q-Learning implementation, you will be implementing a linear decaying function for ϵ.

1.6.1 Q-Learning Simulation Results

To obtain results from the initial Q-Learning implementation, you will need to adjust the following flags and setup: - `'enforce_deadline'` - Set this to `True` to force the driving agent to capture whether it reaches the destination in time. - `'update_delay'` - Set this to a small value (such as `0.01`) to reduce the time between steps in each trial. - `'log_metrics'` - Set this to `True` to log the simluation results as a `.csv` file and the Q-table as a `.txt` file in `/logs/`. - `'n_test'` - Set this to `'10'` to perform 10 testing trials. - `'learning'` - Set this to `'True'` to tell the driving agent to use your Q-Learning implementation.
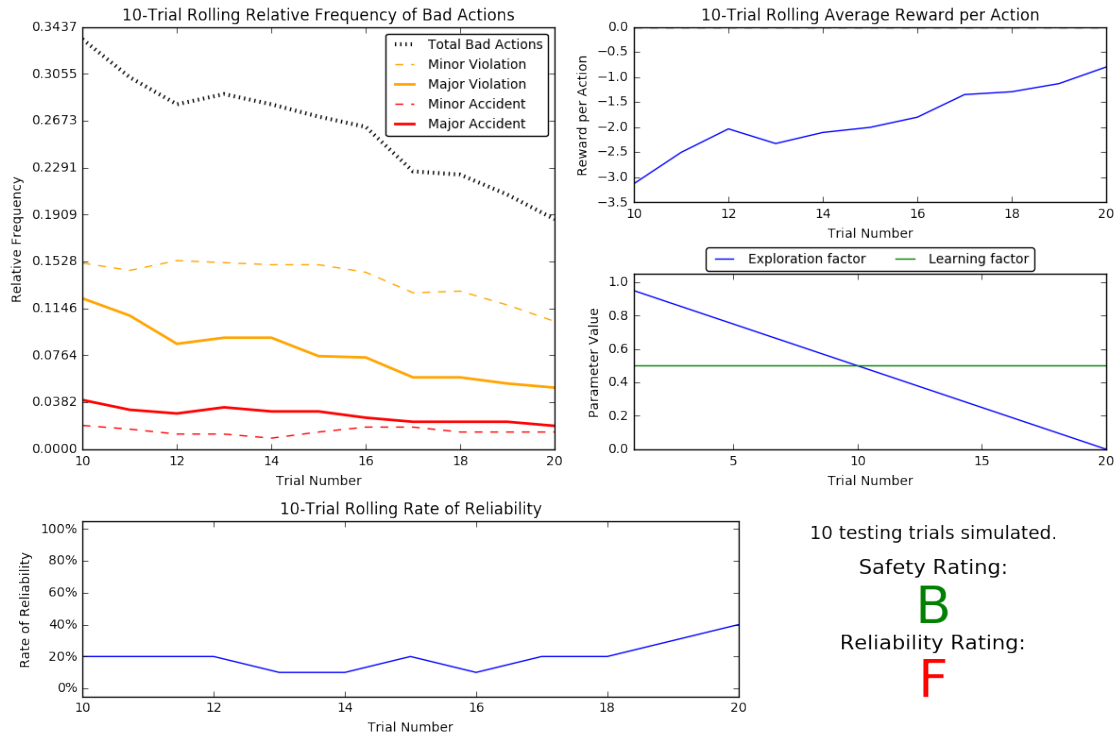
In addition, use the following decay function for ϵ:

$$\epsilon_{t+1} = \epsilon_t - 0.05, \quad \text{for trial number } t$$

If you have difficulty getting your implementation to work, try setting the `'verbose'` flag to `True` to help debug. Flags that have been set here should be returned to their default setting when debugging. It is important that you understand what each flag does and how it affects the simulation!

Once you have successfully completed the initial Q-Learning simulation, run the code cell below to visualize the results. Note that log files are overwritten when identical simulations are run, so be careful with what log file is being loaded!

1.6.2 Question 6

Using the visualization above that was produced from your default Q-Learning simulation, provide an analysis and make observations about the driving agent like in **Question 3**. Note that the simulation should have also produced the Q-table in a text file which can help you make observations about the agent's learning. Some additional things you could consider:
- Are there any observations that are similar between the basic driving agent and the default Q-Learning agent? - Approximately how many training trials did the driving agent require before testing? Does that number make sense given the epsilon-tolerance? - Is the decaying function you implemented for ϵ (the exploration factor) accurately represented in the parameters panel? - As the number of training trials increased, did the number of bad actions decrease? Did the average reward increase? - How does the safety and reliability rating compare to the initial driving agent?

Answer: *- Are there any observations that are similar between the basic driving agent and the default Q-Learning agent?*

They both have a low reliability rating.

- *Approximately how many training trials did the driving agent require before testing? Does that number make sense given the epsilon-tolerance?*

It required 20 trials. It does make sense. The initial value for epsilon is 1, it decreases by 0.05 in each trial and the tolerance is 0.05, then 20 trials is just the expected number.

10

- *Is the decaying function you implemented for ϵ (the exploration factor) accurately represented in the parameters panel?*

Yes. The blue line represents this decaying function.

- *As the number of training trials increased, did the number of bad actions decrease? Did the average reward increase?*

The number of bad actions decreased and the average reward increased as long as the number of training trials increased.

- *How does the safety and reliability rating compare to the initial driving agent?*

Both ratings increased, mainly the safety rating who now gets a B. The reliability rating, besides gets an F, increased its value a lot and now achieves around 40%.

1.7   Improve the Q-Learning Driving Agent

The third step to creating an optimized Q-Learning agent is to perform the optimization! Now that the Q-Learning algorithm is implemented and the driving agent is successfully learning, it's necessary to tune settings and adjust learning paramaters so the driving agent learns both **safety** and **efficiency**. Typically this step will require a lot of trial and error, as some settings will invariably make the learning worse. One thing to keep in mind is the act of learning itself and the time that this takes: In theory, we could allow the agent to learn for an incredibly long amount of time; however, another goal of Q-Learning is to *transition from experimenting with unlearned behavior to acting on learned behavior*. For example, always allowing the agent to perform a random action during training (if $\epsilon = 1$ and never decays) will certainly make it *learn*, but never let it *act*. When improving on your Q-Learning implementation, consider the impliciations it creates and whether it is logistically sensible to make a particular adjustment.

1.7.1   Improved Q-Learning Simulation Results

To obtain results from the initial Q-Learning implementation, you will need to adjust the following flags and setup: - `'enforce_deadline'` - Set this to `True` to force the driving agent to capture whether it reaches the destination in time. - `'update_delay'` - Set this to a small value (such as `0.01`) to reduce the time between steps in each trial. - `'log_metrics'` - Set this to `True` to log the simluation results as a `.csv` file and the Q-table as a `.txt` file in `/logs/`. - `'learning'` - Set this to `'True'` to tell the driving agent to use your Q-Learning implementation. - `'optimized'` - Set this to `'True'` to tell the driving agent you are performing an optimized version of the Q-Learning implementation.

Additional flags that can be adjusted as part of optimizing the Q-Learning agent: - `'n_test'` - Set this to some positive number (previously 10) to perform that many testing trials. - `'alpha'` - Set this to a real number between 0 - 1 to adjust the learning rate of the Q-Learning algorithm. - `'epsilon'` - Set this to a real number between 0 - 1 to adjust the starting exploration factor of the Q-Learning algorithm. - `'tolerance'` - set this to some small value larger than 0 (default was 0.05) to set the epsilon threshold for testing.

Furthermore, use a decaying function of your choice for ϵ (the exploration factor). Note that whichever function you use, it **must decay to** `'tolerance'`** at a reasonable rate**. The Q-Learning agent will not begin testing until this occurs. Some example decaying functions (for t, the number of trials):

$$\epsilon = a^t, \text{for } 0 < a < 1 \qquad \epsilon = \frac{1}{t^2} \qquad \epsilon = e^{-at}, \text{for } 0 < a < 1 \qquad \epsilon = \cos(at), \text{for } 0 < a < 1$$

You may also use a decaying function for α (the learning rate) if you so choose, however this is typically less common. If you do so, be sure that it adheres to the inequality $0 \le \alpha \le 1$.

If you have difficulty getting your implementation to work, try setting the `'verbose'` flag to `True` to help debug. Flags that have been set here should be returned to their default setting when debugging. It is important that you understand what each flag does and how it affects the simulation!

Once you have successfully completed the improved Q-Learning simulation, run the code cell below to visualize the results. Note that log files are overwritten when identical simulations are run, so be careful with what log file is being loaded!

```
In [4]: # Load the 'sim_improved-learning' file from the improved Q-Learning simula
        vs.plot_trials('sim_improved-learning.csv')
```
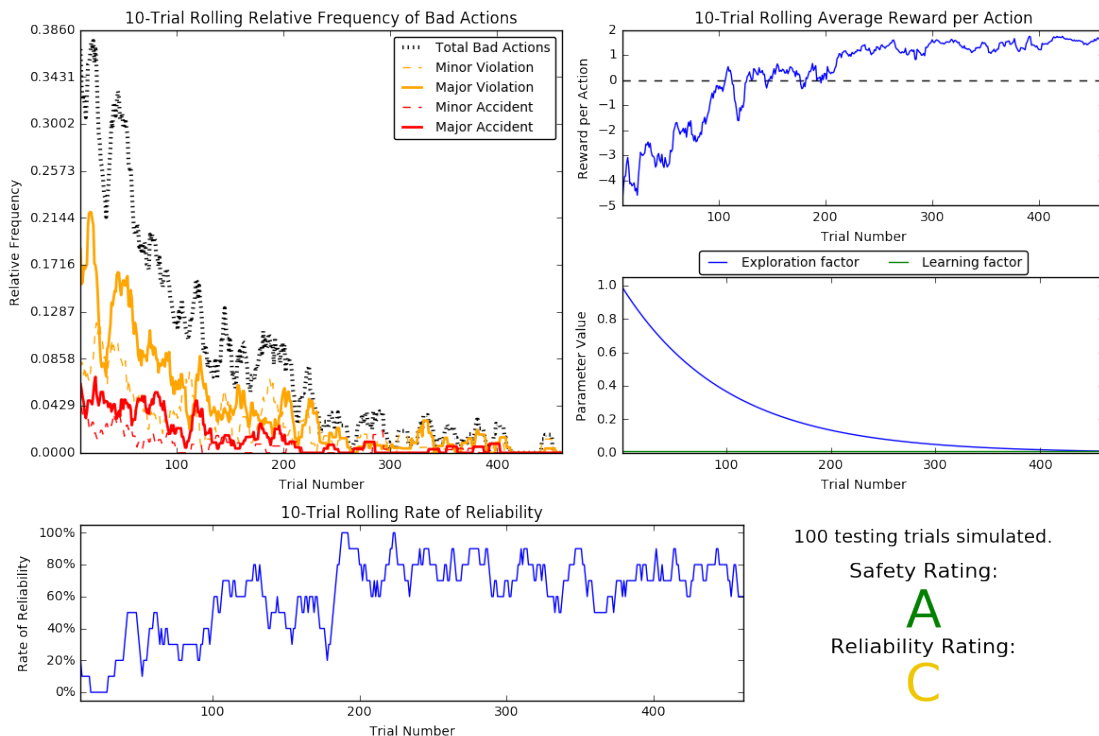

1.7.2 Question 7

Using the visualization above that was produced from your improved Q-Learning simulation, provide a final analysis and make observations about the improved driving agent like in **Question**

6. Questions you should answer:
- What decaying function was used for epsilon (the exploration factor)? - Approximately how many training trials were needed for your agent before begining testing? - What epsilon-tolerance and alpha (learning rate) did you use? Why did you use them? - How much improvement was made with this Q-Learner when compared to the default Q-Learner from the previous section? - Would you say that the Q-Learner results show that your driving agent successfully learned an appropriate policy? - Are you satisfied with the safety and reliability ratings of the Smartcab?

Answer:

- *What decaying function was used for epsilon (the exploration factor)?*

$$\epsilon = e^{-at}, \text{for } 0 < a < 1$$

- *Approximately how many training trials were needed for your agent before begining testing?*

461 training trials.

- *What epsilon-tolerance and alpha (learning rate) did you use? Why did you use them?*

epsilon-tolerance = 0.01, alpha = 0.01.
Alpha represent how much information about the past the algorithm will override, then, a alpha value equals to 1 means that the algorithm will desconsider all the past states and the closer to 0, means the algorithm will consider more past informations. 0.01 was chosen in the will to use the more information possible about the past.

The tolerance 0.01 was chosen due to the number of training trials. We wanted to raise the number of training trials to guarantee the model learn how to behave.

- *How much improvement was made with this Q-Learner when compared to the default Q-Learner from the previous section?*

A lot improvement was made. The reliability rate now achieves a C grade - that means the model is achieving its goal more than 70% - while the safety achieved an A.

- *Would you say that the Q-Learner results show that your driving agent successfully learned an appropriate policy?*

I would say the driving agent learned an appropriate policy but not the best policy. Its grades must be greater than A and C.

- *Are you satisfied with the safety and reliability ratings of the* Smartcab?

No. A model like this couldn't be used in real life. This model will be satisfactory when achieving an A+ and A+.

1.7.3 Define an Optimal Policy

Sometimes, the answer to the important question *"what am I trying to get my agent to learn?"* only has a theoretical answer and cannot be concretely described. Here, however, you can concretely define what it is the agent is trying to learn, and that is the U.S. right-of-way traffic laws. Since these laws are known information, you can further define, for each state the *Smartcab* is occupying, the optimal action for the driving agent based on these laws. In that case, we call the set of optimal state-action pairs an **optimal policy**. Hence, unlike some theoretical answers, it is clear whether the agent is acting "incorrectly" not only by the reward (penalty) it receives, but also by pure observation. If the agent drives through a red light, we both see it receive a negative reward but also know that it is not the correct behavior. This can be used to your advantage for verifying whether the **policy** your driving agent has learned is the correct one, or if it is a **suboptimal policy**.

1.7.4 Question 8

Provide a few examples (using the states you've defined) of what an optimal policy for this problem would look like. Afterwards, investigate the `'sim_improved-learning.txt'` text file to see the results of your improved Q-Learning algorithm. *For each state that has been recorded from the simulation, is the **policy** (the action with the highest value) correct for the given state? Are there any states where the policy is different than what would be expected from an optimal policy?* Provide an example of a state and all state-action rewards recorded, and explain why it is the correct policy.

Answer:

First of all, we have to remember that a state for our agent is composed of waypoint, light, left, right and oncoming, in this sequence.

Examples of optimal policy (structured as state -> optimal action):

1) ('left', 'red', None, 'left', 'forward') -> None. For any state where light is red, the optimal action should be None. 2) ('left', 'green', 'forward', 'forward', 'forward') -> forward. In this case, take the action of turning left could lead to an accident, than, the optimal action would be countinue forward. 3) ('forward', 'green', 'right', 'left', 'left') -> forward. 4) ('right', 'green', 'left', 'right', None) -> right. The light is green and the cars by smartcab's left and right are standing still in red lights.

"For each state that has been recorded from the simulation, is the policy (the action with the highest value) correct for the given state? Are there any states where the policy is different than what would be expected from an optimal policy?"

No. There are some states with incorrect policies. I.e.:

('forward', 'red', None, 'left', 'right') – forward : 0.00 – right : 0.07 – None : 0.00 – left : 0.00

It means the policy for that state is "right" but it should be "None".

"Provide an example of a state and all state-action rewards recorded, and explain why it is the correct policy."

('left', 'red', 'right', None, 'left') – forward : 0.00 – right : 0.00 – None : 0.07 – left : -0.41

As already mentioned in the previous answer, for any state, if the light is red, the right action is None. If it is different from None, the smartcab will at least be committed to a traffic violation and will be at great risk of being involved in an accident.

1.7.5 Optional: Future Rewards - Discount Factor, `gamma`

Curiously, as part of the Q-Learning algorithm, you were asked to **not** use the discount factor, `gamma` in the implementation. Including future rewards in the algorithm is used to aid in propogating positive rewards backwards from a future state to the current state. Essentially, if the driving agent is given the option to make several actions to arrive at different states, including future rewards will bias the agent towards states that could provide even more rewards. An example of this would be the driving agent moving towards a goal: With all actions and rewards equal, moving towards the goal would theoretically yield better rewards if there is an additional reward for reaching the goal. However, even though in this project, the driving agent is trying to reach a destination in the allotted time, including future rewards will not benefit the agent. In fact, if the agent were given many trials to learn, it could negatively affect Q-values!

1.7.6 Optional Question 9

There are two characteristics about the project that invalidate the use of future rewards in the Q-Learning algorithm. One characteristic has to do with the Smartcab* itself, and the other has to do with the environment. Can you figure out what they are and why future rewards won't work for this project?*

Answer:

Note: Once you have completed all of the code implementations and successfully answered each question above, you may finalize your work by exporting the iPython Notebook as an HTML document. You can do this by using the menu above and navigating to
File -> Download as -> HTML (.html). Include the finished document along with this notebook as your submission.