Machine Learning Engineer Nanodegree

Reinforcement Learning

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.

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 time for 100% of trips.
A	Agent commits few minor traffic violations, such as failing to move on a green light.	Agent reaches the destination on time for 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 time for 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 time for at least 70% of trips.
D	Agent causes at least one minor accident, such as turning left on green with oncoming traffic.	Agent reaches the destination on time for at least 60% of trips.
F	Agent causes at least one major accident, such as driving through a red light with cross-traffic.	Agent fails to reach the destination on time for 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
```

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*.

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:

- The Smartcab does not move at all during the simulation.
- The kind of rewards is "Agent idled at a light".
- The agent receives positive (negative) rewards when the light changes from green (red) to red (green).

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.

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:

(1) Questions about agent.py:

The following three flags in agent.py can be set:

- learning
- epsilon
- alpha

The flag learning (True/False) is to force (True) or not force (False) the driving agent to use Q-learning.

The flag epsilon is to implement the epislon-greedy policy, that is, at each step it chooses an action randomly with probability epislon, and greedily choose the action with the highest Q-value with probability (1 - epislon).

The flag alpha is to set the learning rate for the Q-Learning algorithm. A small value of alpha is to change the Q-value for a given pair of state and action slowly.

(2) Question about environment.py:

The Enrironment class function act() is called when an agent performs an action.

(3) Question about simulator.py:

The difference between render_text() and render() is that render_text() displays simulation result in textual format in the terminal/command prompt while render() displays simulation result in GUI.

(4) Question about planner.py:

the next_waypoint() function will consider the East-West direction first.

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.

Basic Agent Simulation Results

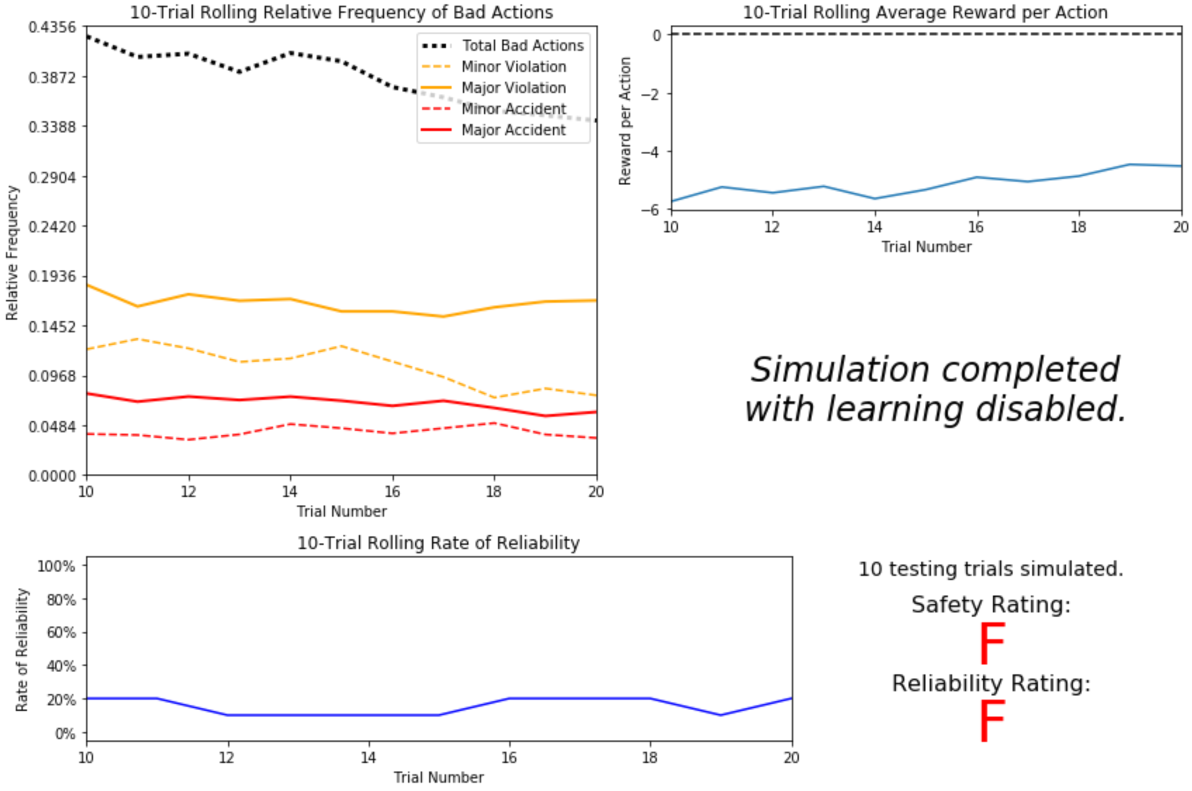
To obtain results from the initial simulation, you will need to adjust following flags:

- `'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 [3]: # Load the 'sim_no-learning' log file from the initial simulation result
        s
        vs.plot_trials('sim_no-learning.csv')
```



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 reliability 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:

- The driving agent made about 43.56% bad decisions out of all the decisons at the beginning. The frequence of bad decisions came down to about 33.88% after 20 tries.
- Given that the agent is driving randomly, the rate of reliability does not make sense because the agent will reach its destination randomly.
- The agent received significant negative rewards in the following cases:

(a) It received -40.98 reward when it attempted driving forward through a red light with traffic and cause a major accident. (b) It received -20.36 reward when it attempted driving right through traffic and cause a minor accident. (c) It received -11.00 and -9.42 rewards when it attempted driving forward and left through a red light respectively.

These negative rewards show that the agent has been penalized heavily for major traffic violations and minor/major accidents.

- As the number of trials increases, the relative frequencies of bad decisions and minor violations decrease. However the relative frequenceis of major violations and minor/major accidents do not change significantly.
- This Smartcab would not be considered safe and/or reliable for its passengers because the visulation results showed that both of them are F.

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!

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.

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? Please note that whatever features you eventually choose for your agent's state, must be argued for here. That is: your code in agent.py should reflect the features chosen in this answer.*

NOTE: You are not allowed to engineer new features for the smartcab.

Answer:

The following features are most relevant for learning both safety and efficiency:

- the next waypoint
- the intersection inputs

The next waypoint determines the direction in which the Smartcab moves. Thus this feature directly impacts both safety and efficiency.

The intersection inputs provide information about potential cars that may be on the left, right or oncoming of our smartcab. Thus this feature is relevant for learning both safety and efficiency as well.

I have omitted the deadline for the following reason. The deadline is not a well defined state in that it keeps changing with the selected destination for each trial and/or testing. If I include this feature in the state, one of the potential flaws is that the undeterministic nature of deadline will decrease the Smartcab's learning capability.

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!

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 environment 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:

The next waypoint has three possible values:

- forward
- left
- right

The intersection inputs (sense) can take the following possible combinations of light colors and actions as values:

The light can take two possible values:

- green
- red

Given a light color, there are three possible inputs:

- inputs['oncoming']
- inputs['left']
- inputs['right']

Each input type of option can take one of four possible values:

- oncoming
- turn left
- turn right
- none (idle)

The number of possible values of sense is 2 x (4 x 4 x 4) = 128.

Thus the size of state space = 3 x 128 = 384.

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!

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 iterative 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 0.05). For the initial Q-Learning implementation, you will be implementing a linear decaying function for ϵ.

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.

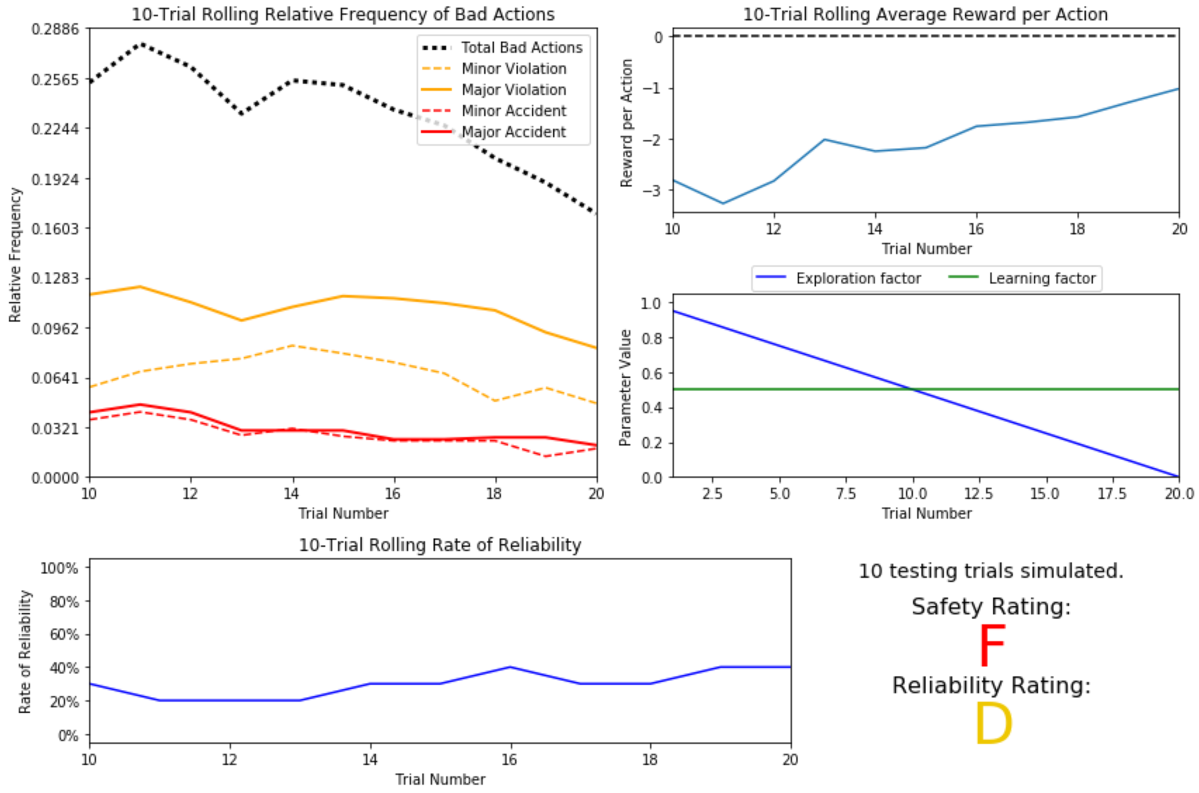
In addition, use the following decay function for ϵ:

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

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!

```
In [6]: # Load the 'sim_default-learning' file from the default Q-Learning simul
        ation
        vs.plot_trials('sim_default-learning.csv')
```



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:

- There are observations that are similar between the basic driving agent and the default Q-Learning agent. For example, the frequence of bad decisions came down with the number of tries, and the average reward per action went up.
- The driving agent required about 20 training trials before testing. That number makes sense given the epsilon 1.0, tolerance 0.05, and linear decay function (epsilon = 1.0 - 0.05 * t), where t is the number of tries.
- The decaying function (epsilon = 1.0 - 0.05 * t) I implemented for ϵϵ (the exploration factor) was accurately represented in the parameters panel.
- Yes, as the number of training trials increased, the number of bad actions decreased and the average reward increased over all.
- The safety rating was the same, while the reliability rating improved compared to the initial driving agent.

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 implications it creates and whether it is logistically sensible to make a particular adjustment.

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):

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

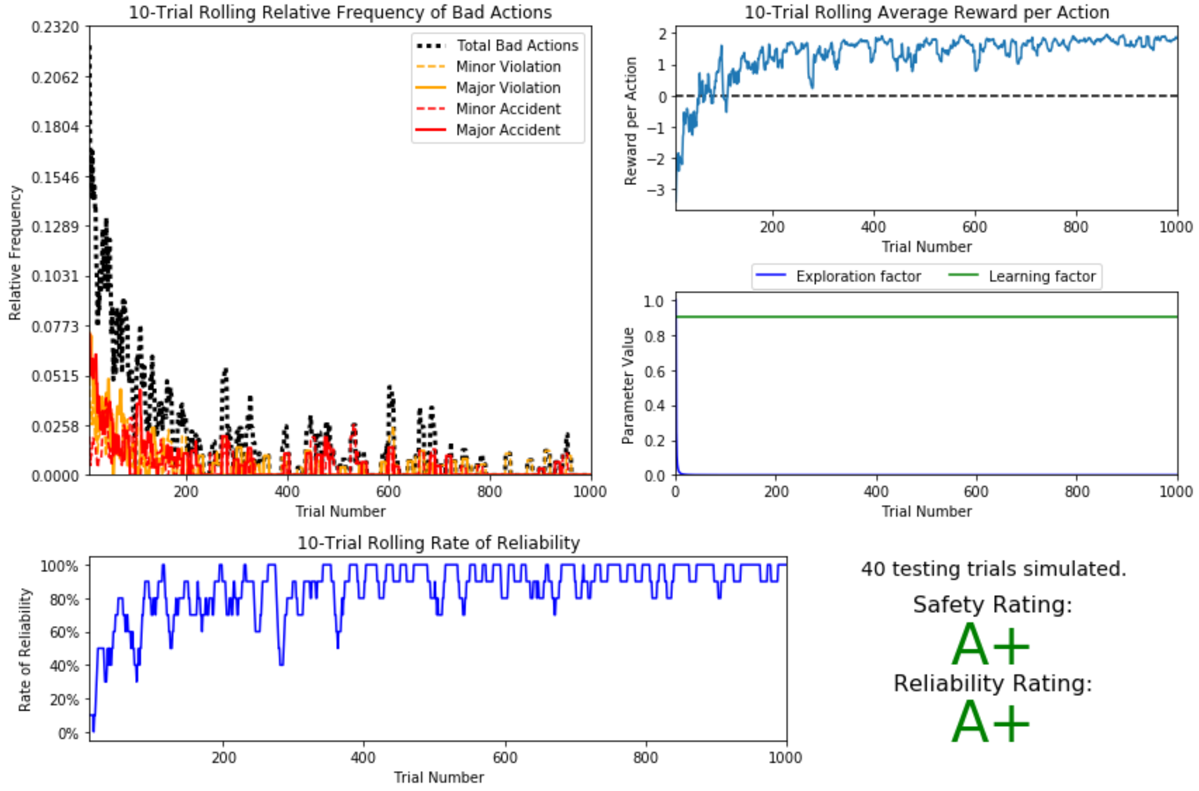
You may also use a decaying function for α (the learning rate) if you so choose, however this is typically less common. If you do so, be sure that it adheres to the inequality $0 \leq \alpha \leq 1$.

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 [2]:

```
# Load the 'sim_improved-learning' file from the improved Q-Learning simulation
vs.plot_trials('sim_improved-learning.csv')
```



10-Trial Rolling Relative Frequency of Bad Actions

10-Trial Rolling Average Reward per Action

10-Trial Rolling Rate of Reliability

40 testing trials simulated.

Safety Rating:

A+

Reliability Rating:

A+

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:

- The decaying function for epsilon is: epsilon = 1 / (t x t), where t is the number of tries.
- The agent needs to complete at least 20 training trials before beginning testing because this is enforced by the simulator. However, in order to get grade A for both safety and reliability, approximately 1000 training trials should be needed before testing as calculated below. The tolerance in use is 0.000001. Let 1 / (t x t) = 0.000001. The solution to this equation is t = 1000.
- The following values of epsilon, tolerance, and alpha were used:

```
- epsilon = 1.0
- tolerance = 0.000001
- alpha = 0.9
```

The epsilon is set to 1 at the beginning because the agent needs exploration to learn.

The tolerance is set to 0.000001 so that the agent will have enough (approximately 1000) tries before testing.

The alpha is set to a large value of 0.9 so that more weights will be put on current rewards in updating Q values.

- The following improvement was made with this Q-Learner when compared to the default Q-Learner from the previous section:

```
- The 10-trier rolling relative frequency of total bad actions was dec
reased from about 20% to 5%.
- The 10-trier rolling relative frequency of major violation was decre
ased from about 6% to 1%.
- The 10-trier rolling average rewards per action was increased from -
2.0 to 1.8.
- The 10-trier rolling rate of reliability was increased from about 2
0% to 90+%.
```

- I would say that the Q-Learner results show that my driving agent successfully learned an appropriate policy.

As an example, the following results show that by taking the action with highest Q value, the agent learned to stop at red light appropriately.

```
('left', {'light': 'red', 'oncoming': 'right', 'right': 'right', 'left': 'le
ft'})
 -- forward : 0.00
 -- None : 1.63
 -- right : 0.00
 -- left : 0.00
```

- Yes, I am satisfied with A ratings for both safety and reliability of the Smartcab.

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**.

Question 8

1. Please summarize what the optimal policy is for the smartcab in the given environment. What would be the best set of instructions possible given what we know about the environment? *You can explain with words or a table, but you should thoroughly discuss the optimal policy.*
2. Next, 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?*
3. Provide a few examples from your recorded Q-table which demonstrate that your smartcab learned the optimal policy. Explain why these entries demonstrate the optimal policy.
4. Try to find at least one entry where the smartcab did *not* learn the optimal policy. Discuss why your cab may have not learned the correct policy for the given state.

Be sure to document your `state` dictionary below, it should be easy for the reader to understand what each state represents.

Answer:

- the optimal policy can be summarized as follows:

```
  (1) If waypoint is forward, then:
      if the light is green, then the action is forward. Otherwize the actio
n is None (idle).
  (2) If waypoint is right, then:
      if the light is green, then the action is right.
      if the light is red and the input['left'] is not forward, then the act
ion is right.
      if the light is red and the input['left'] is forward, then the action
 is None (idle)
  (3) If waypoint is left, then:
      if the light is green and the input['oncoming'] is not forward, then t
he action is left.
      Otherwise, the action is None (idle).
```

- There are states where the policy is different than what would be expected from the above optimal policy.

For example, the following record in sim_improved-learning.txt shows that when the waypoint is left and the light is red, the action with the highest Q value is right, while according to the optimal ploicy, the action should be None.

```
('left', {'light': 'red', 'oncoming': 'forward', 'right': 'forward', 'left':
 'left'})
 -- forward : 0.00
 -- None : 0.00
 -- right : 0.10
 -- left : 0.00
```

- The following are three examples from the recorded Q-table, which demonstrated that the smartcab learned the optimal policy.

The first example followed the Optimal Policy (1). The second example followed the Optimal Policy (2). The third example followed the Optimal Policy (3).

```
('forward', {'light': 'green', 'oncoming': 'left', 'right': None, 'left': No
ne})
 -- forward : 2.53
 -- None : -4.95
 -- right : 0.00
 -- left : 0.00

('right', {'light': 'red', 'oncoming': 'forward', 'right': None, 'left': 'le
ft'})
 -- forward : -9.14
 -- None : 0.00
 -- right : 1.47
 -- left : -8.44

('left', {'light': 'red', 'oncoming': None, 'right': 'left', 'left': 'forwar
d'})
 -- forward : 0.00
 -- None : 2.51
 -- right : -17.87
 -- left : -36.05
```

- Try to find at least one entry where the smartcab did not learn the optimal policy. Discuss why your cab may have not learned the correct policy for the given state.

For example, the following record in sim_improved-learning.txt shows that when the waypoint is left and the light is green, the action with the highest Q value is forward, while according to the optimal ploicy, the action should be None.

```
('left', {'light': 'green', 'oncoming': 'forward', 'right': 'right', 'left':
 None})
 -- forward : 1.84
 -- None : -4.75
 -- right : 0.00
 -- left : -18.47
```

One of the possible reasons that the Smartcab may have not learned the correct policy is that the rewards policy is not designed for following the correct policy. For examle, the rewards policy might have put too much penalty on idle at green light.

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 propagating 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!

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:

- The characteristic with the Smartcab (simulator) itself is that the agent does not have a full view of the entire grid. At the time of updating Q value, given a pair of state and action, the agent only knows the previous Q value of the pair and the new reward. The agent does not have the information for estimating future rewards.
- The other characteristic with the environment is that the environment resets the rewards and related status at the beginning of each trial. Because of this, it is impossible for the Smartcab to accumulate rewards across different trials.

> **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.