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 didn't move, at all, during the simulation. The driving agent was receving positive and negative rewards regarding the actions it was taking, in its case as he was stood, he was only receiving rewards related with the light changing colors. The red light, as the driver was not moving, turned to be a positive reward (no traffic violation). The green light, as the driver was not moving, turned to be a negative reward (traffic violation).

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:

agent.py

- *num_dummies*: discrete number of dummy agents in the environment, default is 100.
- *grid_size*: discrete number of intersections (columns, rows), default is (8, 6).
- *update_delay*: continuous time (in seconds) between actions, default is 2.0 seconds.

environment.py

- The function called when the agent performs an action is the function **act(self, agent, action)**.

simulator.py

- The render_text() is the non-GUI output render, also called the command line simulation output.
- The render() is the output the uses GUI (Graphic User Interface) to show a visual output of the simulation.

planner.py

- The next_waypoint() sees the grid as set of Xs and Ys, and by saying that it's code was created to first consider North-South (X axis) than later East-West (Y axis).

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

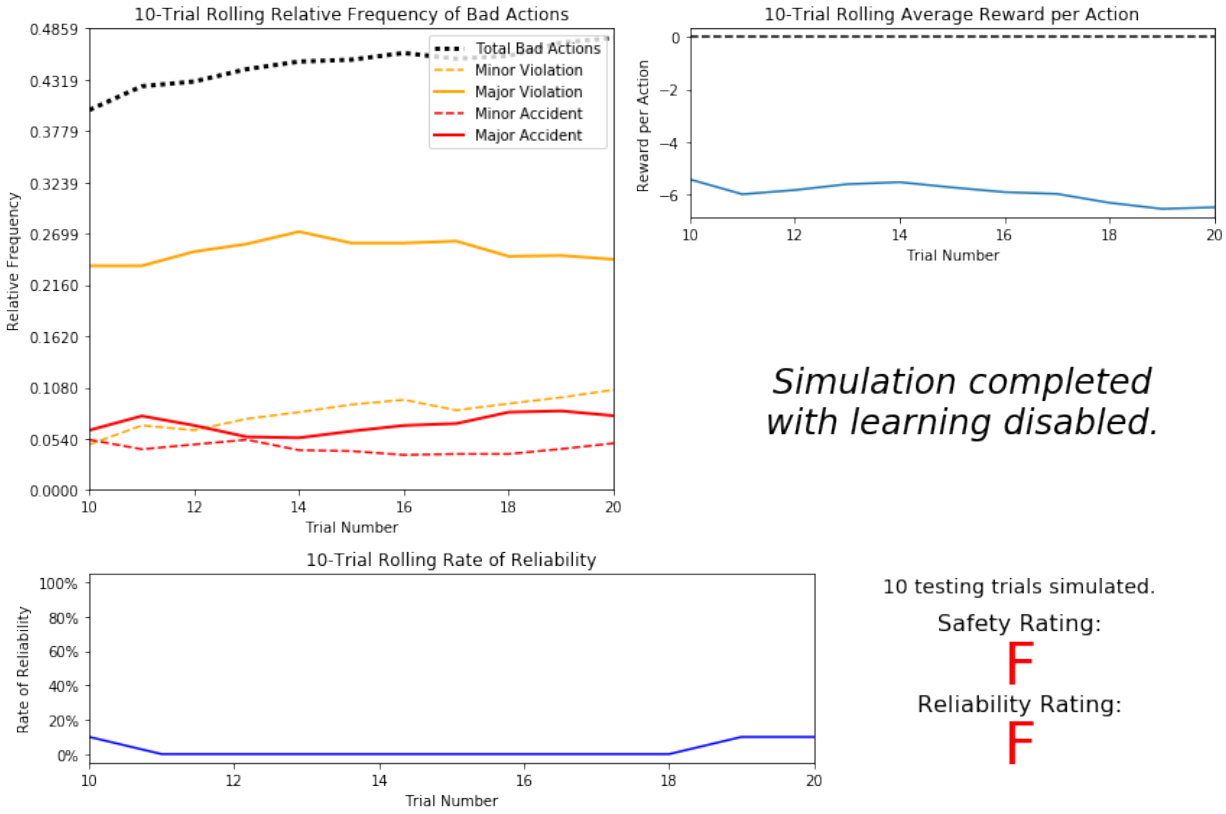
To obtain results from the initial simulation, you will need to adjust following flags:

- `'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!

In [2]:
```
# Load the 'sim_no-learning' log file from the initial simulation resu
lts
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 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:

About the Visualization Panels, the first one **Relative Frequency of Bad actions** is very important to see how often (statistically) are the bad actions happening during the testing cycle, and here we saw a considerable amount to bad actions around 0.43 of all actions. The second panel **Average Reward per action** is a good indicator if the driver agent is action according the expectation, and where we saw that my agent is doing very bad always presenting a **negative** reward (meaning bad actions). The third panel **Rate of Reliability** shows the rate of when our driving agent achieves its destination on time, and here we can see that it is very bad and below expectations. Both **Safety Rating** and **Reliability Rating** received grade **F** the Agent causes at least one major accident, such as driving through a red light with cross-traffic and the Agent fails to reach the destination on time for at least 60% of trips.

Some Analysis:

- **How frequently is the driving agent making bad decisions?** Around 43% of the time.
- **How many of those bad decisions cause accidents?** Around 5% of the them.
- **Given that the agent is driving randomly, does the rate of reliabilty make sense?** Yes, it is not considering the rewards for real.
- **What kind of rewards is the agent receiving for its actions? Do the rewards suggest it has been penalized heavily?** Mostly negatives, they suggest that the agent is most of the time randomly choosing bad actions.
- **As the number of trials increases, does the outcome of results change significantly?** The remain around the same, I wasn't able to see any major outcome change, I'm assuming that this is because of the randomness nature.
- **Would this Smartcab be considered safe and/or reliable for its passengers? Why or why not?** Not at all. Because it is acting based on bad decisions and around 80% of the time it ends without reaching the destination in time.

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?*

Answer:

It is important for the Agent to know its current location and the options to move forward. The agent have that consuming the features Waypoint and Inputs.

- **Important for learning Safety and Efficiency**: I considered the **inputs** (light, left and oncoming) very important, as they are used to detect possible accidents (collision route), they need to be part of the learning process of the Agent in order to minimize the collision routes as possible. Other important feture to be part of the learning process is the **waypoint** as the Agent needs to learn how to reach its destination based on where it is right now. So both **inputs** and **waypoint** need to be balanced so the Agent can learn how to reach the destination in the *efficient* and *safe way*.
- **Not Important for learning Safety and Efficiency**: I consider the **deadline** a complex feature as it can vary a lot (for example a deadline equals to 100) and can have pass at the same position several times with different deadlines what can cause more confusion than real learning to our Agent (risk of create useless policies). From the existing inputs, the input **right** was discarded as per follows the Right of Way Rules, in which the driver of a vehicle is required to give way to vehicles approaching from the right at intersections.

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 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:

- **State**: (waypoint, light, left, oncoming)
  - Waypoint: [None, 'forward', 'left', 'right'] - 4 possible values
  - light: [None, 'forward', 'left', 'right'] - 4 possible values
  - left: ['red', 'green'] - 2 possible values
  - oncoming: [None, 'forward', 'left', 'right'] - 4 possible values

Combination of all states: (4x4x2x4) = 128 possible states.

I think it is possible for the Agent to learn a policy for each state using reasonable number of training trials.

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 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.

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.

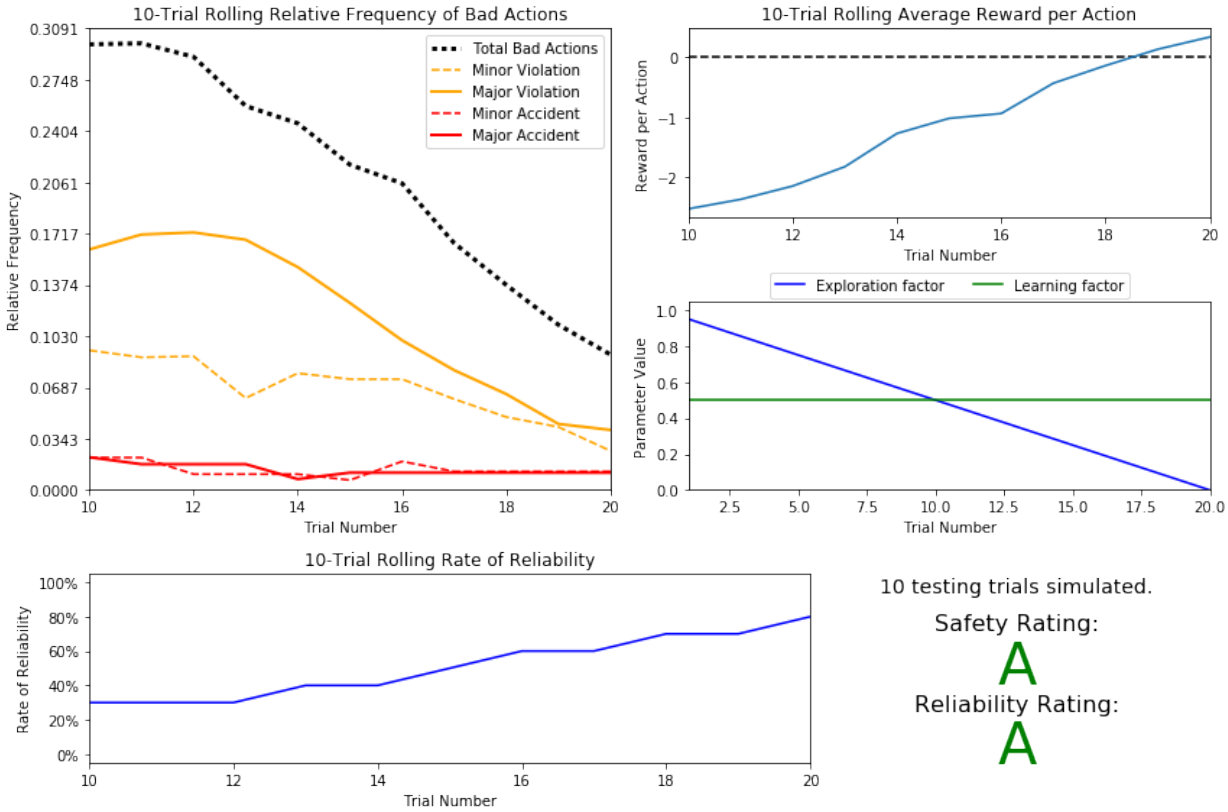
In addition, use the following decay function for ϵ:

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

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 simulation
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:

About the Visualization Panels, at the first one **Relative Frequency of Bad actions** we see that there was a learning effect as the number of bad actions was reducing by time. The second panel **Average Reward per action** also indicates the learning effect as the rewards are increasing by time, as well. The third panel **Rate of Reliability** on the other hand, remains bad and not showed any aparent effect of learning, probably due to a small number of training trials or my State definition is still too complex to be easily learnt.

Some Analysis:

- **Are there any observations that are similar between the basic driving agent and the default Q-Learning agent?** No, they all showed enhancements compared to the basic driving model. Bad Actions, Rewards and Reliability are tending to the expected directions (based on learning).
- **Approximately how many training trials did the driving agent require before testing? Does that number make sense given the epsilon-tolerance?** The agent did 20 trials before testing, as per following the Decay function that was to decrease 0.05 from each trial of a Epsilon starting in 1. So it took 20 trials and then 10 tests as expected.
- **Is the decaying function you implemented for ϵ (the exploration factor) accurately represented in the parameters panel?** Yes, it was represented as expected.
- **As the number of training trials increased, did the number of bad actions decrease? Did the average reward increase?** Yes, both happened. Bad actions decreased by time and Reward increased by time showing that the algorithm was in fact presenting a kind of learning. Algorithm achieved rating **A** in Safety.
- **How does the safety and reliability rating compare to the initial driving agent?** They increased considerably. Algorithm achieved rating **A** in Reliability.

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.

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):

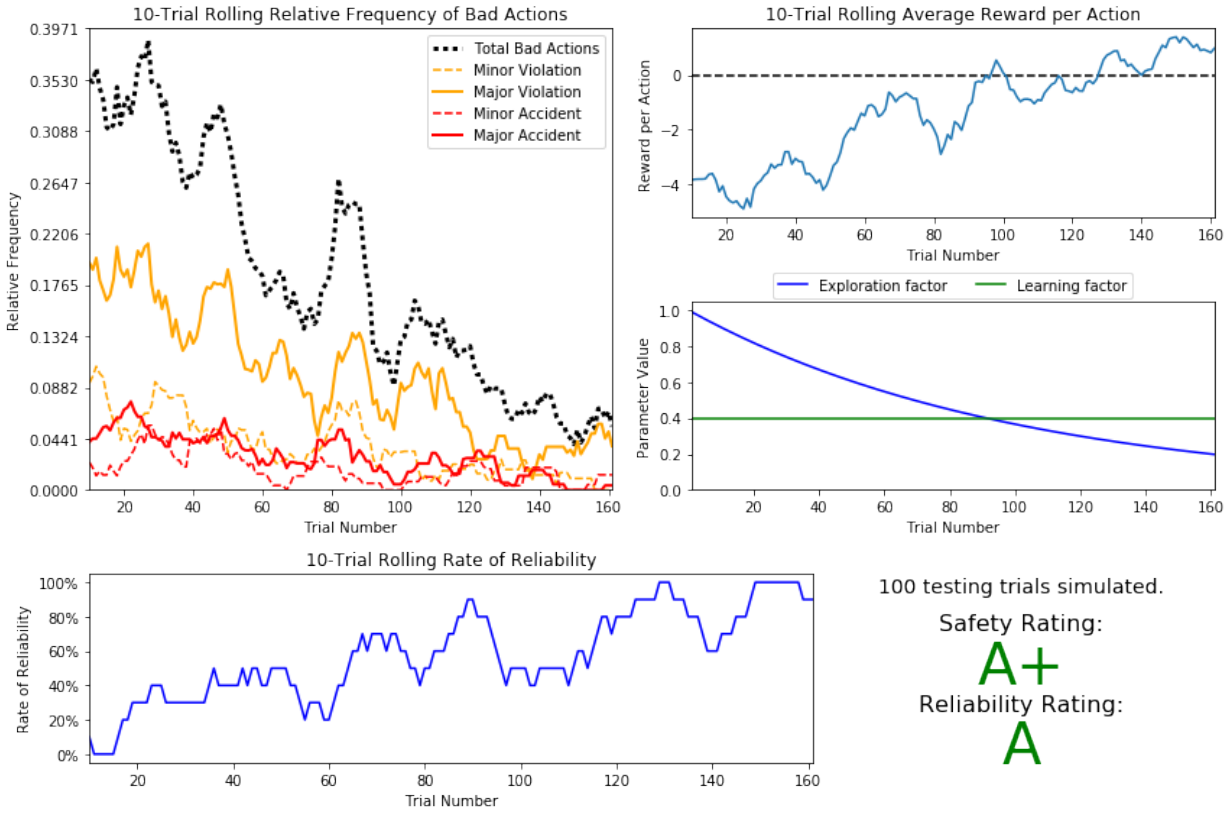
$$\epsilon = a^t, \text{ for } 0 < a < 1 \qquad \epsilon = \frac{1}{t^2} \qquad \epsilon = e^{-at}, \text{ for } 0 < a < 1 \qquad \epsilon = \cos(at), \text{ for } 0 < a < 1$$

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 $0 \leq \alpha \leq 1$.

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 [14]:
```
# Load the 'sim_improved-learning' file from the improved Q-Learning s
imulation
vs.plot_trials('sim_improved-learning.csv')
```



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:

About the Visualization Panels, at the first one **Relative Frequency of Bad actions** we see that there was a learning effect as the number of bad actions was considerably reduced by time. The second panel **Average Reward per action** also indicates the learning effect as the rewards are increasing by time, becoming positive. The third panel **Rate of Reliability** present considerable improvements and now reaches more that 80% most of time.

- **What decaying function was used for epsilon (the exploration factor)?** My decaying function had the intention to reduce Epsilon slowly compared to the one did on **Question 6**. It was:

$$\epsilon_{t+1} = (\epsilon_t * 0.05) / 10, \quad \text{for trial number } t$$

- **Approximately how many training trials were needed for your agent before begining testing?** The agent did 161 trials before testing, as per following the Decay function. So it took 161 trials and then 100 tests as expected.
- **What epsilon-tolerance and alpha (learning rate) did you use? Why did you use them?** For **epsilon-tolerance** I reduced it from 0.5 to 0.2 to only start testing and this minimum threshold was reached, which means, if necessary we could run at least one more learning trial to be sure. For **alpha** I did a slightly reduction on it from 0.5 to 0.4. Basically the as lower is the alpha more the algorithm will tend to not learn. I did a fine tuning here, trying the agent several times with alpha equals to 0.7, 0.5 and 0.3. For them the 0.5 looked better, but I did a last try with 0.4 and it got the **A+** in Safety, so I let it there, but I would probably use the default 0.5 (if wasn't the fine tuning analysis).
- **How much improvement was made with this Q-Learner when compared to the default Q-Learner from the previous section?** Safety was improved from **A** to **A+** and Reliability kept stable around **A**. I am really happy with the progress.
- **Would you say that the Q-Learner results show that your driving agent successfully learned an appropriate policy?** For sure, the Agent with the improved Q-Learner is now a trustworthy Agent, I would have it as my Cab Driver, at any time.
- **Are you satisfied with the safety and reliability ratings of the *Smartcab*?** I am very satisfied with both, but I think the Reliabiliy could reach **A+** too.

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

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:

The Agent State is defined by:

state = (waypoint, inputs['light'], inputs['left'], inputs['oncoming'])

Not all states followed the **optimal policy** but in their majority they followed it right, down below you can check some examples of success and fails regarding the **policy learning**.

1) In this example you can see from the agent state perpective that the waypoint was to be moving forward, but the light was Red, so the agent respected the **Optimal Policy** by deciding its action to **None**:

- ('forward', 'red', 'forward', None)
- None : 2.01
- forward : -38.20

- right : -18.75
- left : -34.56

- ('right', 'red', 'left', 'right')
- None : 1.62
- forward : -4.00
- right : 0.00
- left : -16.34

- ('left', 'red', 'forward', 'right')
- None : 1.64
- forward : 0.00
- right : -7.84
- left : -16.07

2) In this example you can see that waypoint was to be turning right and that the light was Red, and our agent decided to turn right on red, what in some states of US is a **valid policy** but in some states, such as New York, North Carolina, and California, a right turn on red is prohibited when a red arrow is displayed which means that we had a **policy violation**:

- ('right', 'red', 'right', 'forward')
- None : 0.58
- forward : -3.98
- right : 1.19
- left : 0.00

- ('right', 'red', None, 'right')
- None : 1.19
- forward : -6.39
- right : 2.05
- left : -25.40

3) In this two scenarios we can see that our agent learnt his **own policy**, deciding to move forward.

- ('left', 'green', 'forward', 'right')
- None : -2.36
- forward : 0.31
- right : 0.00
- left : 0.00

- ('left', 'green', None, 'right')
- None : -3.23
- forward : 0.06
- right : 1.27
- left : -8.16

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!

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:

There is an important variable **deadline** that is not part of the states, due to its nature, and it not aware of the correct position of the **final destination** as it only have the waypoint by state. So our Smartcab is not aware of the remaining moves until it reaches the destination. By adding the **gamma** and propagating the rewards we could lead our Agent to become a reward seeker that would not be focused in arrive early, only in win more and more rewards (pretty much like bad cab drivers that do the longest ride with tourists).

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.