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 [ ]:

```
# 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: I can not see the animation, by reading the code, the learning agent won't move at all, others cars will move left, right ro foreward each turn on avoiding the violation. the world is spheric. the driving agent receive -5 or 0 average reward each turn (with a random difference ranging from -1 to 1). During the green light, stay on the same place will cause a -5 avg reward, during the red light, the agent get 0 avg reward.

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

learning: if not set, the agnet will take a random action at each step, otherwise it will choose the best possible action with a certain probability defined by epsilon

epsilon: if learning not set, there is no sense for this flag, otherwise, epsilon is the probability of which the agent choose the action randomly. To notice that the value of epsilon will decrease at each step and converge to 0.

alpha is the update rate of the Q value, when alpha grows, new Q value of the iteration influence more on the final value. Alpha should be kept small in order to converge.

Environment.py

act(self, agent, action)

Simulator.py

render_text() This is the non-GUI render display of the simulation. Simulated trial data will be rendered in the terminal/command prompt.

render() This is the GUI render display of the simulation. Supplementary trial data can be found from render_text.

planner.py

the planner consider the East-West direction first because it firstly begins with "dx". The goal of the planner is to adjust the heading of the car according to its destination. to notice that the agent always turn anticlockwisely when it needs to go back(exp: if heading north destination south then trun to west)

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.

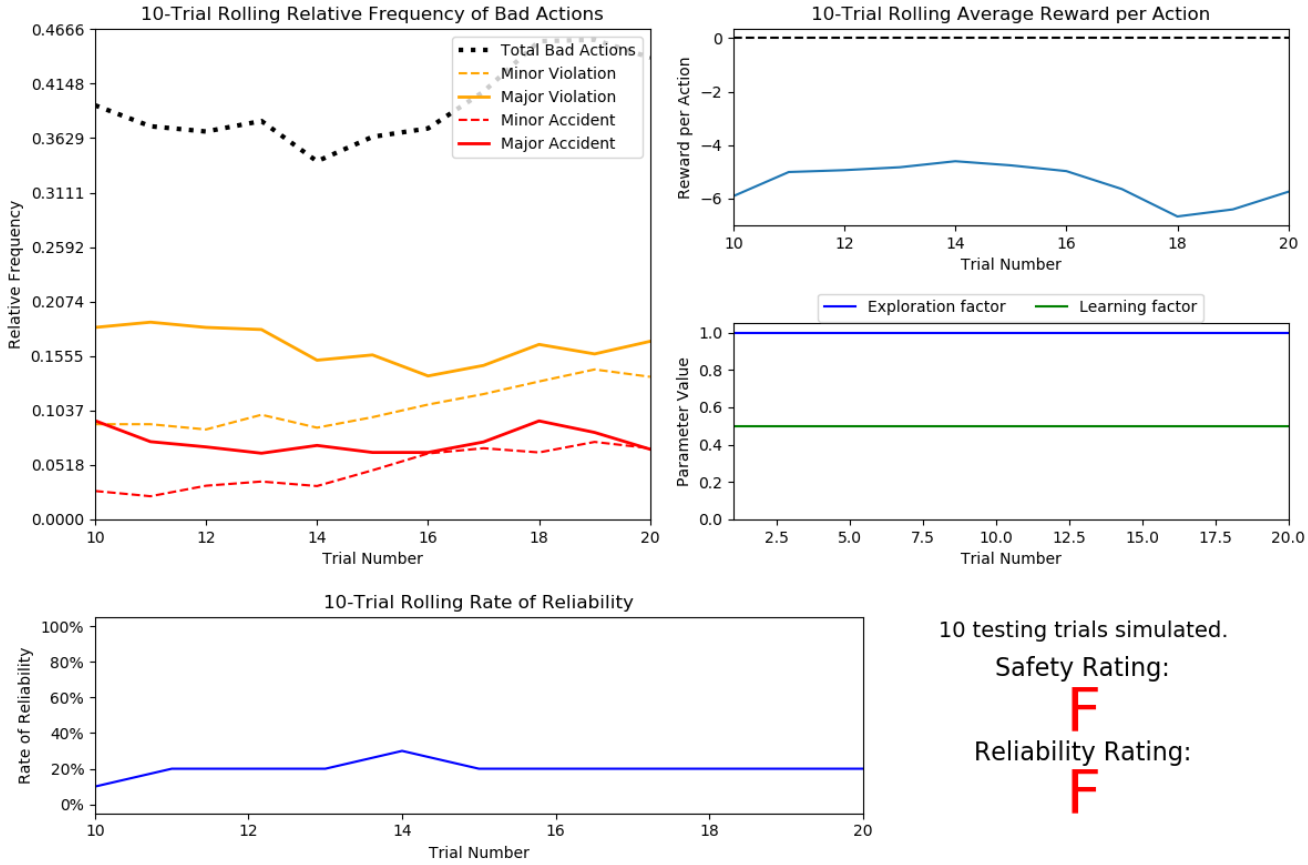
In [ ]:

```
# Load the 'sim_no-learning' log file from the initial simulation results

import os
import visuals as vs

abs_path = os.path.dirname(os.path.abspath(__file__))

vs.plot_trials(os.path.join(abs_path, '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:

Shown as the picture above, every panel shows the average value of 10 rolling trials. for about 40% to 45% of times the agent make bad decision. Among these bad decisions, the frequencies of each kind are:

major violation > minor violation > major accident > minor accident.

About 25% of bad decision causes accidents.

The rate of reliability is closed to 20%, this proves the difficulty of getting to the destination if the car chooses actions randomly.

Average reward per action is around -6, this is reasonable, if an agent acts randomly, it will get negative rewards in average.

As the number of trials increases, the outcome of results didn't change at all because the agent is not learning.

According to the testing result from the 10 testing trials, both reliability and safety get F. so the random agent is not reliable neither safe.

alpha=0.5 and epsilon=1, there is no sense for this 2 parameters see that the agent doesn't even learn.

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:

'waypoint' and 'deadline' are the most relevent factor to the efficiency.

'waypoint' can indicate the optimal direction for the current location to get to the destination. As the time passes the agent can get more pernalty for each action, the 'Deadline' is the only feature that indicates those pernlties so that the agent can learn to get to the destinaton as early as possible.

Light, left, right and oncoming are important features for the safety.

'light' can determine whether an action is a violation or not, for example, choose to stay is a good action when light is red, otherwise, it is a bad action. 'left', 'right', 'oncoming' may not influence the violation, but the agent needs to know the traffic around to check if its action will cause accident.

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:

At the beginning, I tried to take all state information perceivable to the agent as the state dimension, but the result is not good. Not only because this take a huge state space and Q table size, but also because some information is in fact noise that could cause the agent to make bad decision.

The attrinutes I won't take:

1. Right, althought having or not a car from right side moving forward will differentiate the situation in which the agent try to move forward during the red light by minor violation without accident and major violation within accident, Our goal is to have a safety score of A or A+, so this kind of differentiation doesn't matter for us, as a result, if only the agent learns to respect the light color, the right side informlation is not needed any more.
2. Deadline, In the previous question, I said Deadline may infer on the efficiency, But here, after a deep analysis, I found that this is not true:

Firtly, this increas dramatically the state space. the average distance is 6/2/2+8/2/2 = 3.5, so the average duration is around 3.5*5 = 18 steps, this will enlarge the sate space by 18 times.

Secondly, in the code the remaining time(deadline) is related to the pernalty, this can give effect to the reward when violation is 0. Howerver this pernalty can not move the reward to negative when the deadline is closed, and it can not change the relative advantages among the 3 kinds of choices when violation is 0, So the pernalty itself has no crucial effet.

Further more, within the current rewarding system implemented in the file environment.py, If the agent knows the remaining time is still enough, instead of choosing the waypoint, it may choose another violation 0 action to gain more positive reward and it can catch back the reward of following the waypoint in the two next actions(see that the remaining time is enough, and the pernalty as reward decay is not enough to offset the advantage of this trick). the more it rests on the map, the more score may it gains. The worst is that the agent don't even care about if it can reach the destination by the lack of the reaching reward.

Finally, the agent can only perceive part of its environment, it can not know the current distance to the destination. So there is no sense to know the deadline without knowing the distance.

So finally I would choose the following:

'waypoint': left, right, forwards (for efficieny) 'light': red , green (for avoiding violation) 'left': left, right, forwards, none (for avoding accident within a violation 0 context) 'oncoming': left, right, forwards, none (for avoding accident within a violation 0 context)

final state space's size: 324*4 = 96

possible actions: none, forward, right, left

final state-action space's size: 96*4 = 384

Aroung 1000-5000 trials will be largely sufficient

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,
     ...
   },
   ...
}
```

Typesetting math: 100%

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!

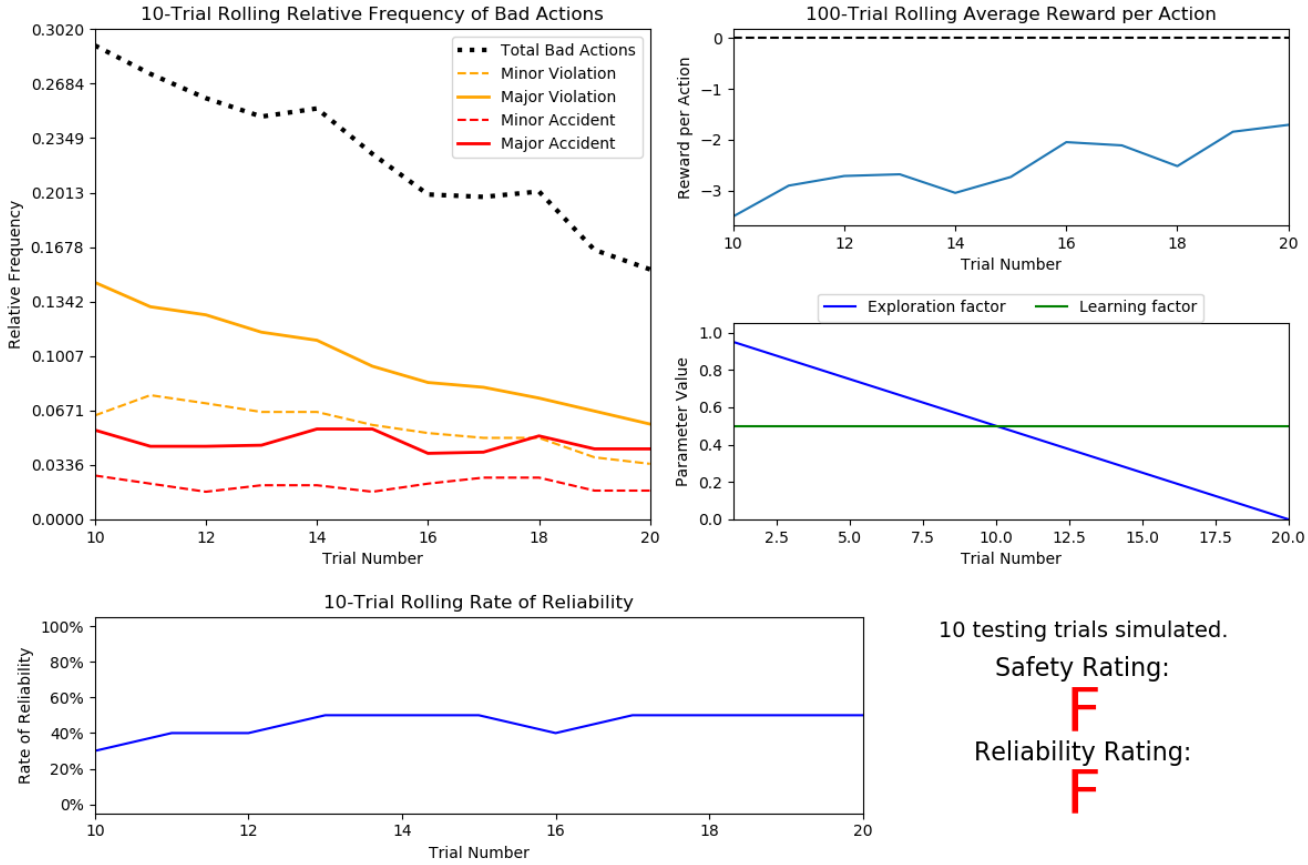
In [ ]:

```
# 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: Before interpreting the figure, I first explain a little bit my implementation in file "sim_learn_agent.py":

I suppose that all cars, lights and the primary agent act simutaneously for each step(similar to a cellular automata). So in the "update()" method, after acting and receiving reward, the primary agent should wait until all others car have acted in order to get its next state.

1. The reliabilities of basic agent have not been improved. Q-learning agent have higher reliabilitie up to 40% and slightly increasing.
2. As suggested by the statement, the number of trial is around 20 (1/0.05) but this is not enough according to my previous estimate.
3. The random decaying of epsilon has been acctually displayed in the panel as a linear decreasing function.
4. As the number of training trials increased, the number of all kind of bad actions decreased and the average reward increased.
5. Safety and reliability have been improved but not sufficient.

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):

$$ \epsilon = a^t, \textrm{for } 0 < a < 1 \hspace{50px}\epsilon = \frac{1}{t^2}\hspace{50px}\epsilon = e^{-at}, \textrm{for } 0 < a < 1 \hspace{50px} \epsilon = \cos(at), \textrm{for } 0 < a < 1$$ 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 [ ]:

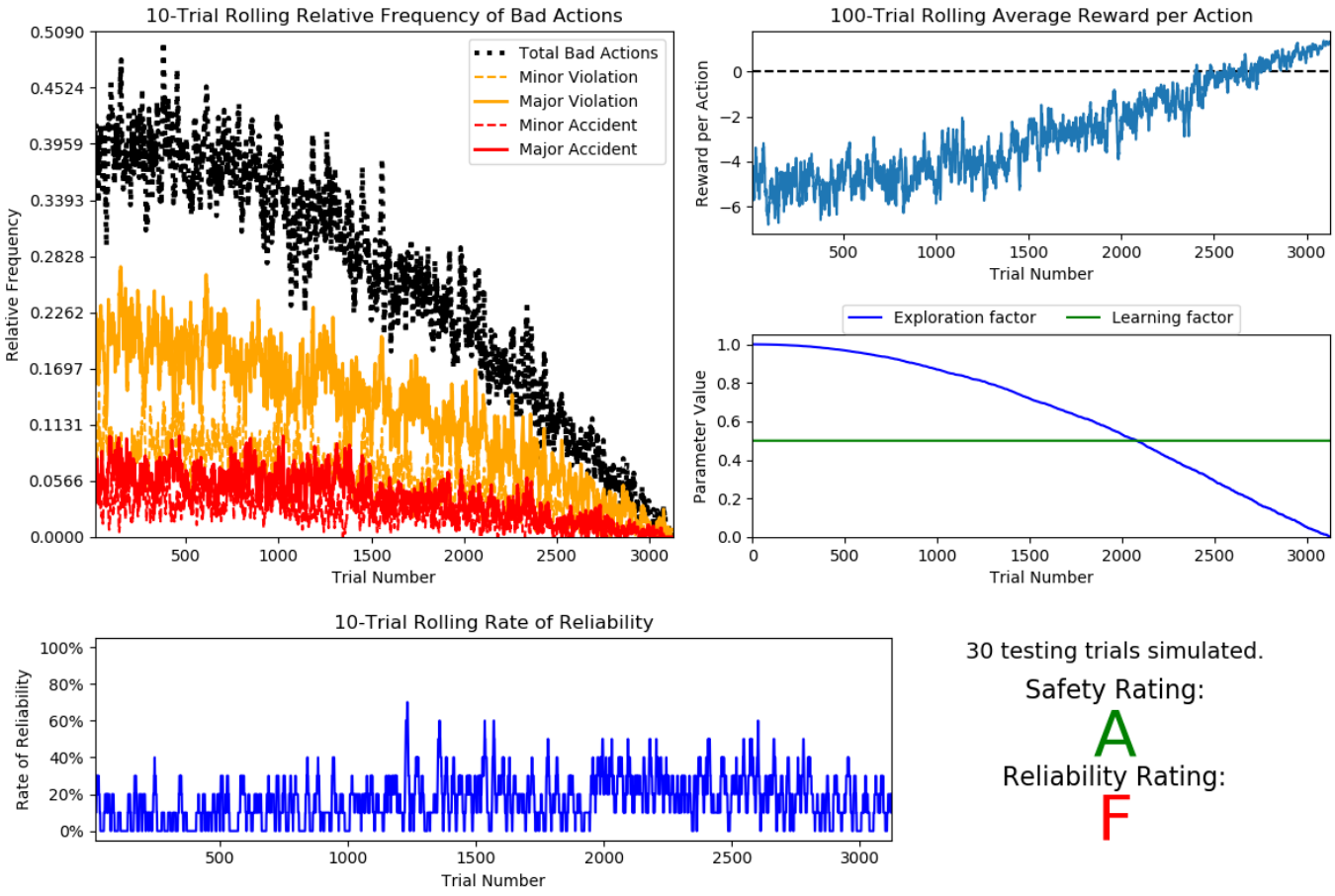
```
# Load the 'sim_improved-learning' file from the improved Q-Learning simulation
vs.plot_trials('sim_improved-learning.csv')
```

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: First of all, I tried with constant alpha = 0.5, with a enough amount of trials and good decay function for this works well for safety but too bad to reliability, even if I added a new feature to the state: "deadline // 4". The graph is like the following:



In order to find the cause of the failure of the reliability, I noticed that the Qtable is not even converged. I think the reason why alpha keeping constant doesn't work is that one state perceived by the agent is associated to multiple configurations of the real environment(see that agent can only perceive partial info of the env), but each time we take the best reward among all these configs for one agent's perceived state. This causes the Qtable to explose.

The reason why the non converging issue effects the reliability is that, for one state A, the agent doesn't know it's "freedom" = "deadline - disance". for a env config where A has a large "freedom", instead of following the waypoint to get to the destination ealierly without any additional rewards, it may prefer to take another "violation=0" way to collect more rewards, even the pernalty decay can not offset this effect. Thus, this state will be associated with the max rewards config, then with such an triky action that diverts the agent from the destination. This is the major reason why the reliability is even worth than simple-learning agent.

To solve the reliability problem, We have 2 ways:

1. decay the alpha, so that the initial one step direct reward for each state take more importance as we begin with a Q table filled by zeros.
2. set gamma to a small number

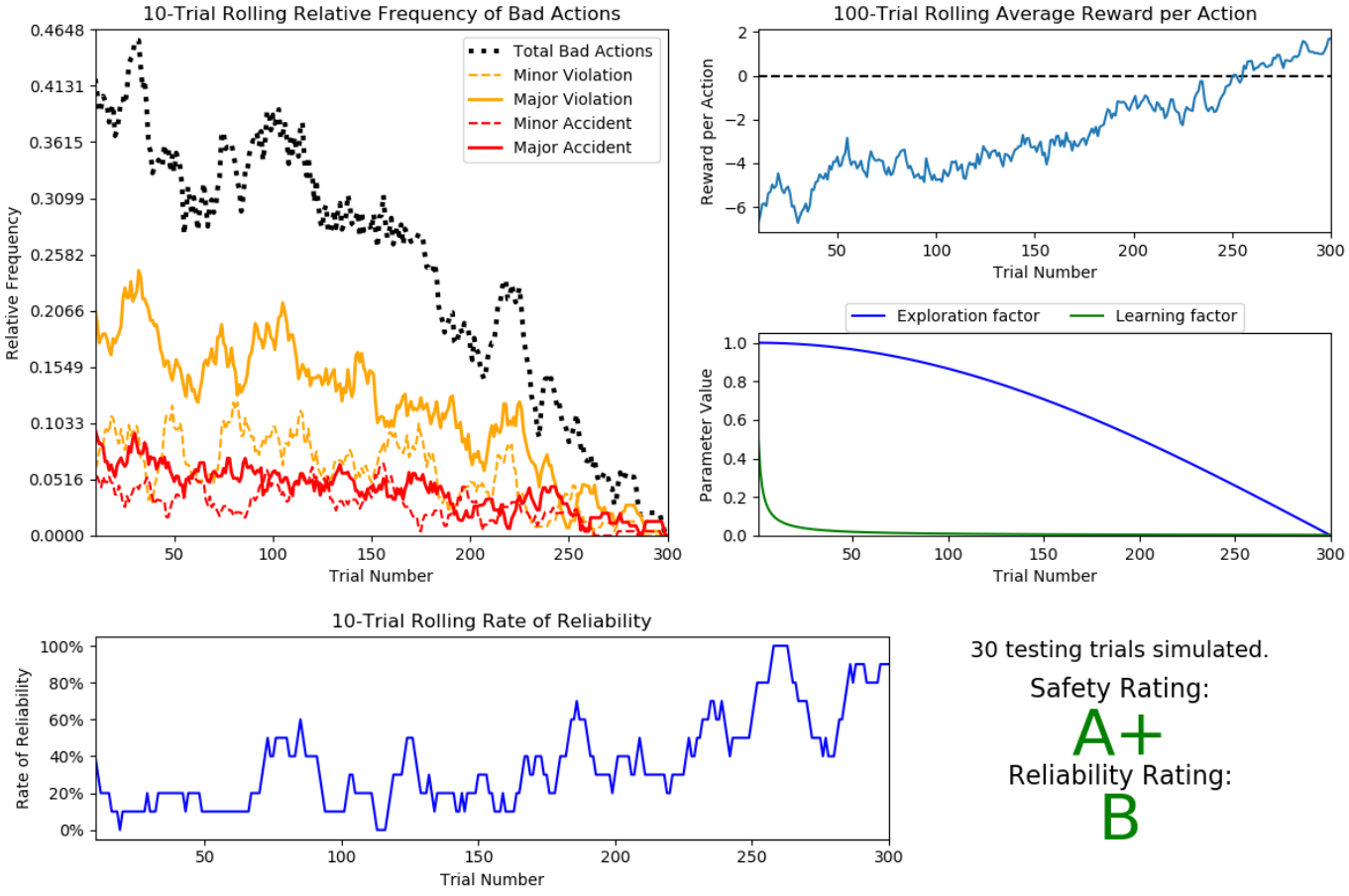
In fact, for this particular problem, we only need to maximize the expected direct reward for each action in order to maximize both the safety and the reliability.

Solution 1: alpha = 1/(t+1)

I noticed that the 3 first decay functions for epsilon are concave function while the last one is convex. Intuitively I would prefer convex function because it decreases less quickly at the beginning, this allows the agent to learn a larger variety of state-action pairs and it decreases more quickly at the end, this allows the agent converge faster to the more globaly optimal solution.

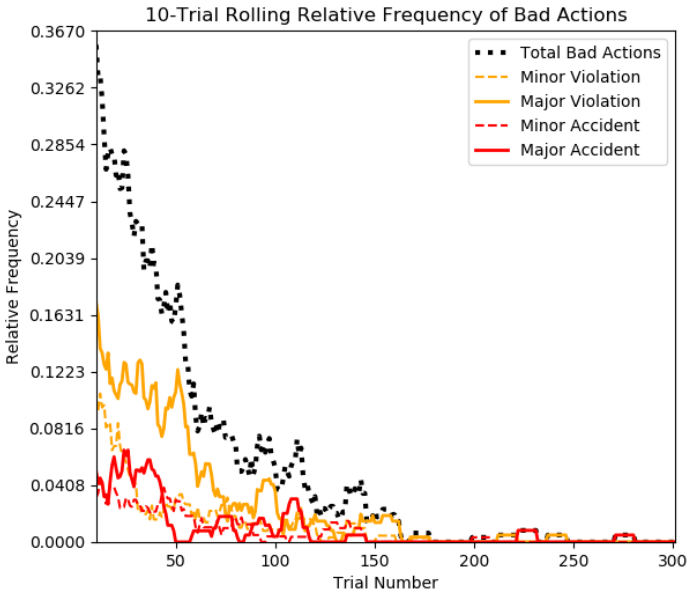
I tried these 3 decay functions(the fist and the third are equivalent) separately in order to compare them. Instead of fixing "a" and "tolenrance", I think fixing number of total training trials and the tolerance is more pragmatic:

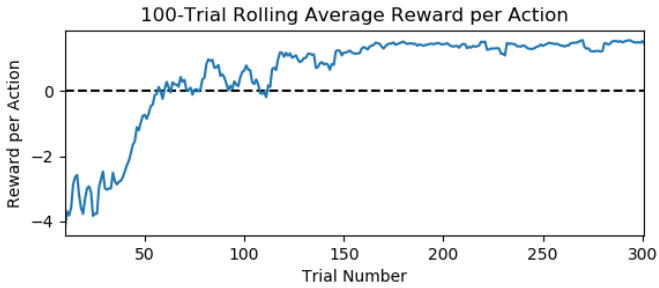
cos, num_trials = 300, tolerence = 0.001

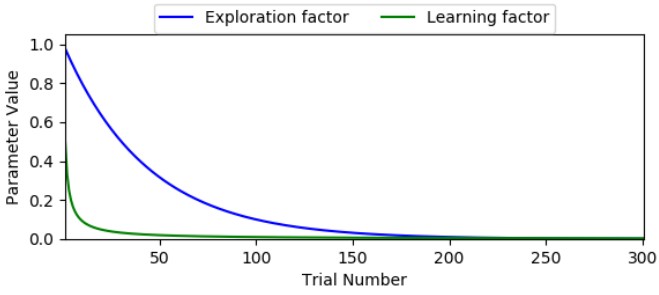


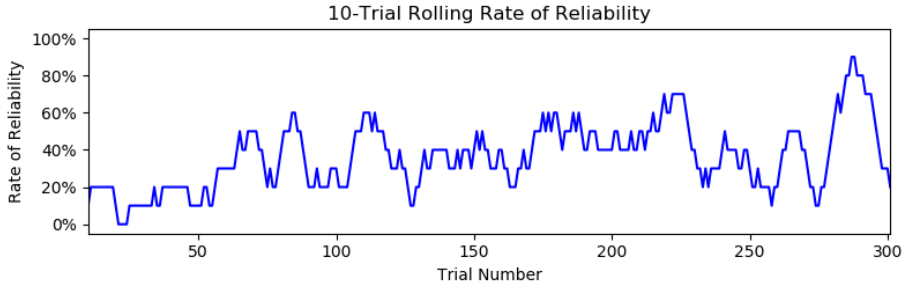
exponential, num_trials = 300, tolerence = 0.001

Typesetting math: 100%









30 testing trials simulated.

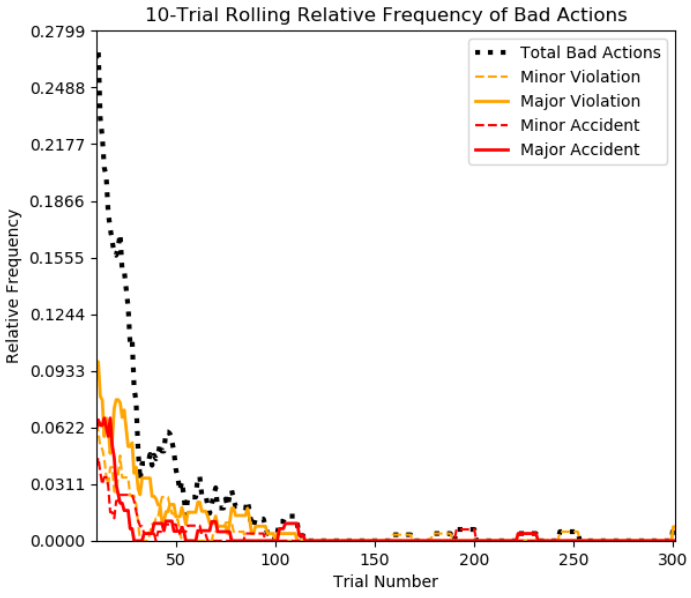
Safety Rating:

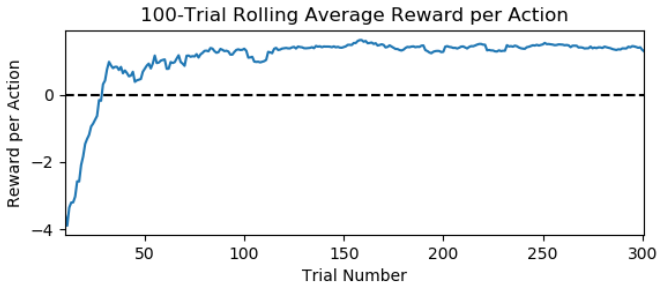
F

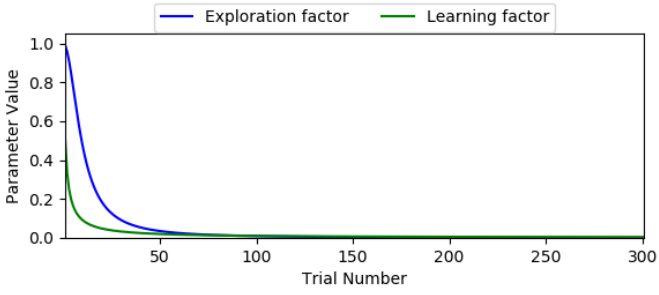
Reliability Rating:

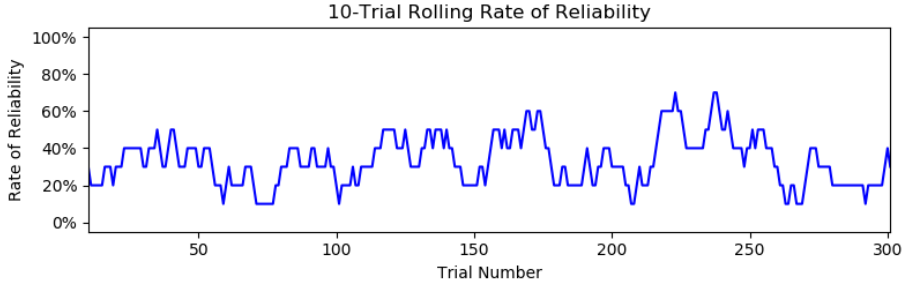
F

trial_square, num_trials = 300, tolerence = 0.001









30 testing trials simulated.

Safety Rating:

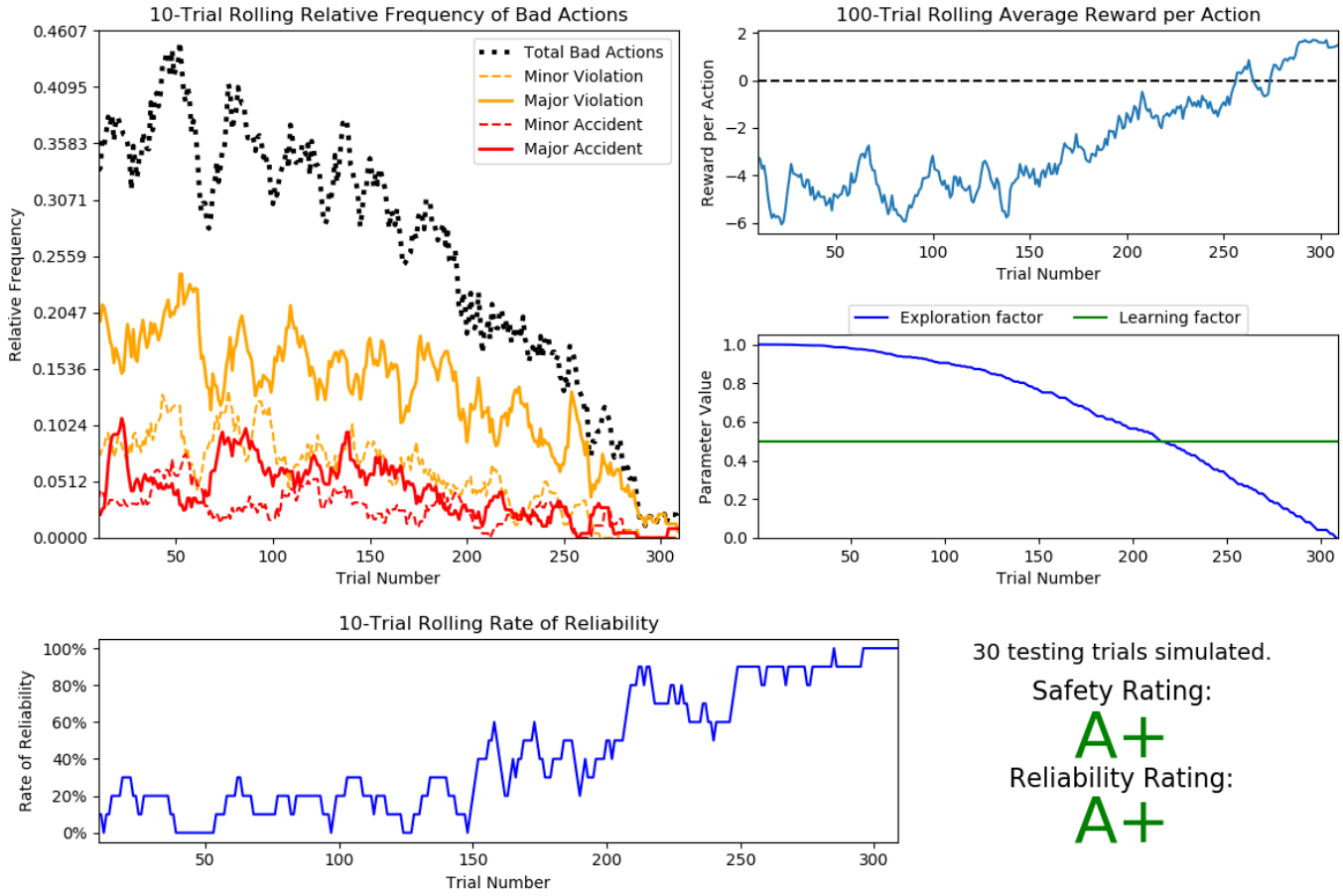
A+

Reliability Rating:

F

As what I supposed, "cos" gives better result.

Solution 2: gamma = 0, alpha=0.5, only 100-200 trials:



As what I speculated, if only the gamma = 0, even a constant alpha can lead to best result easily

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**.

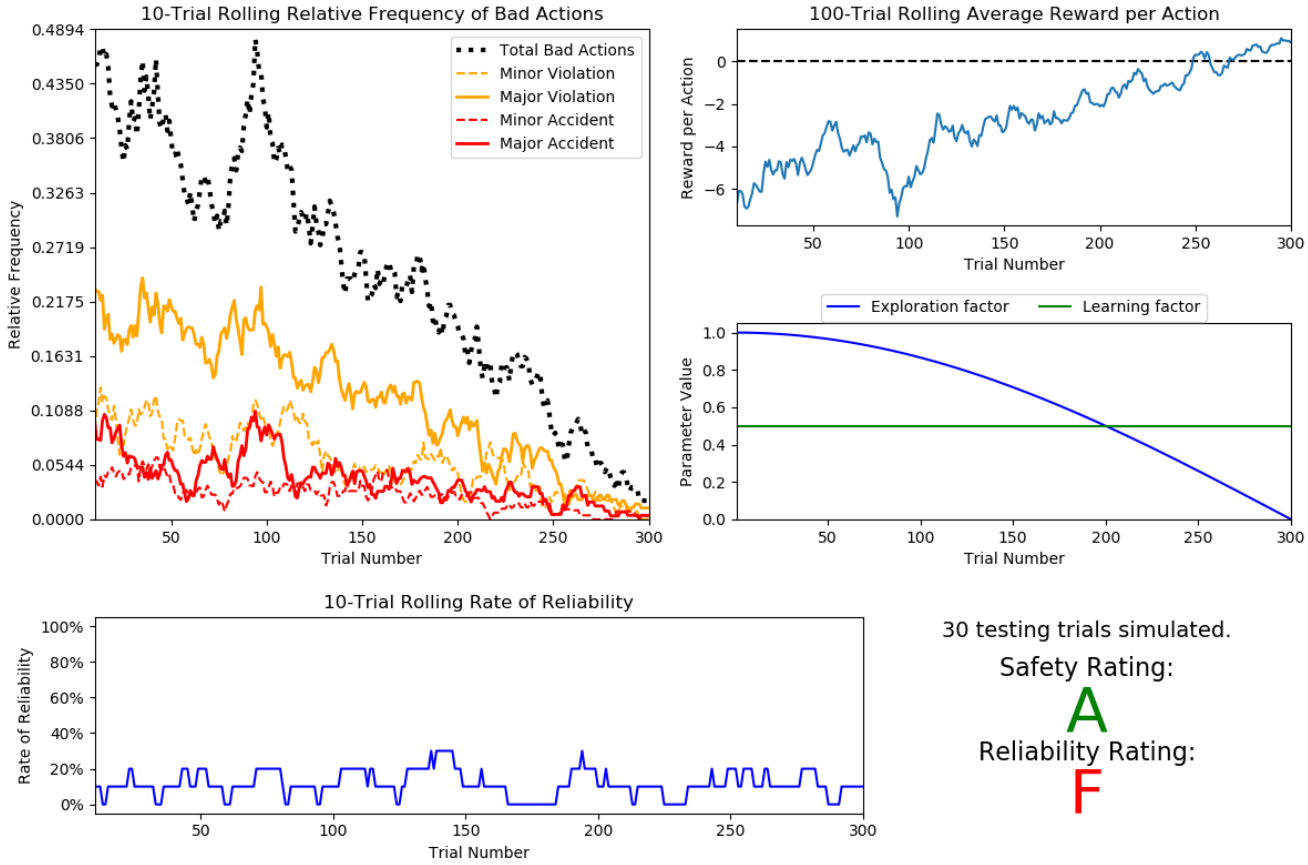
Question 8

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:

1. To summerize, if the agent always avoid any kind of violation > 0, and does not choose the waypoint only if waypoint has violation > 0, we can say that the agent act optimally. Howerver, it's possible that by avoiding the sophisticated light and car, the agent eventually can not get to the destination, here it will rationnaly sacrifice reliability to ensure the safety.
2. Take example of "cos" decay function where alpha = 0:



By investigating the 'sim_improved-learning.txt' file, I found a curious Q item:

('right', 'green', None, 'right') -- forward : 32.77 -- none : 11.58 -- right : 18.27 -- left : 12.05 The optimal way is to go right but the agent think going forward is optimal. The reason I have already explained is that, It trys to stay longer in the map in order to get more long term rewards although this is not actually the reality.

3. Take example of "cos" decay function where alpha alos decays:

('left', 'red', 'forward', 'right') -- forward : -0.27 -- none : 0.15 -- right : 0.02 -- left : -4.17

('forward', 'red', None, 'right') -- forward : -0.97 -- none : 0.29 -- right : 0.03 -- left : -0.49

The agents knows to avoid violation despite of its waypoint indication.

('right', 'green', 'forward', 'left') -- forward : 0.09 -- none : -0.71 -- right : 0.37 -- left : -3.65

The agent follow the waypoint when this is violation 0. However, I see an irrational action, because of the inefficiency of the alpha converge: ('forward', 'green', 'forward', 'right') -- forward : 0.07 -- none : -1.06 -- right : 0.17 -- left : -0.30

Instead of moving forward, the agent goes right.

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!

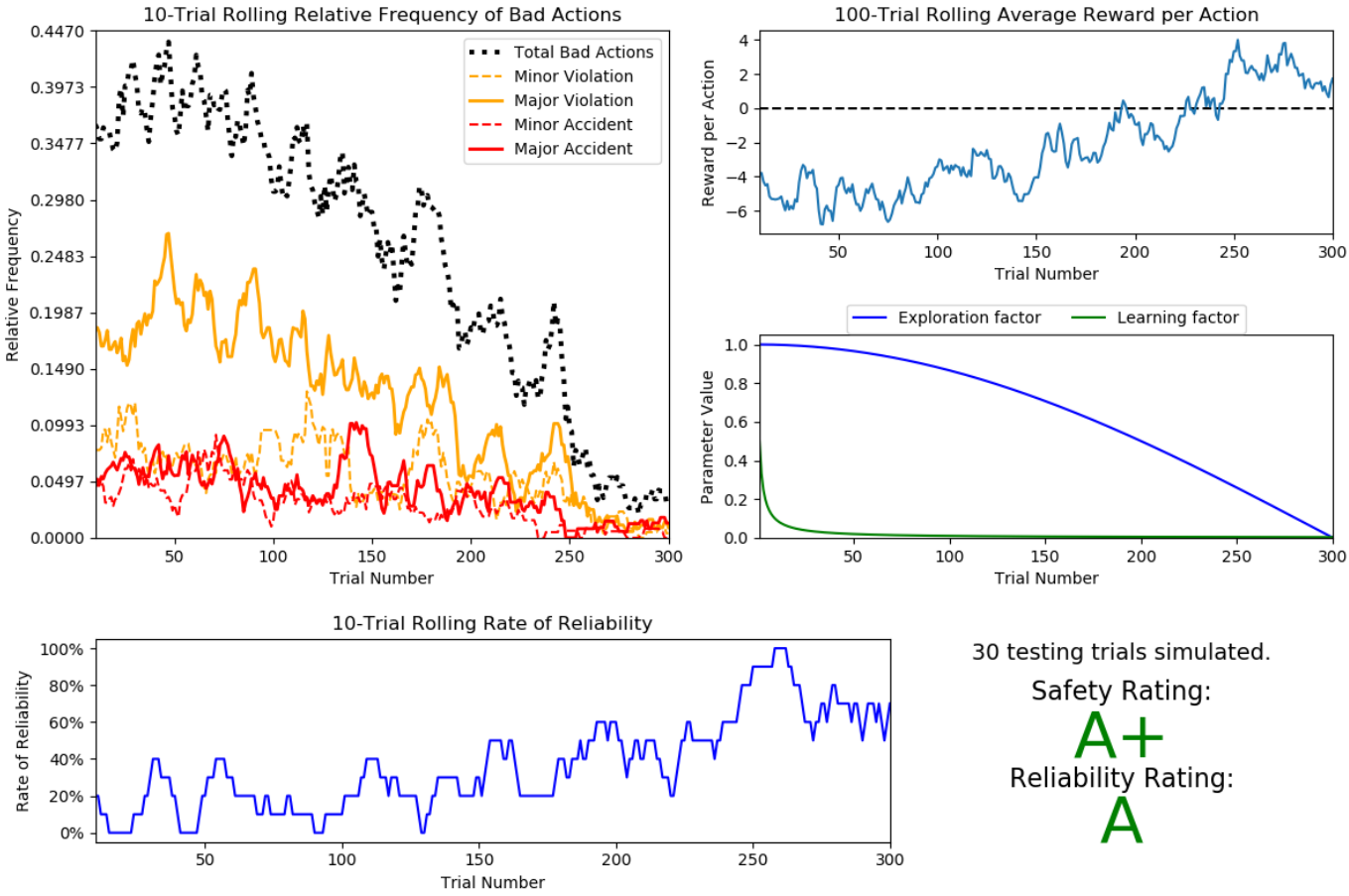
Optional Question 9

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?*

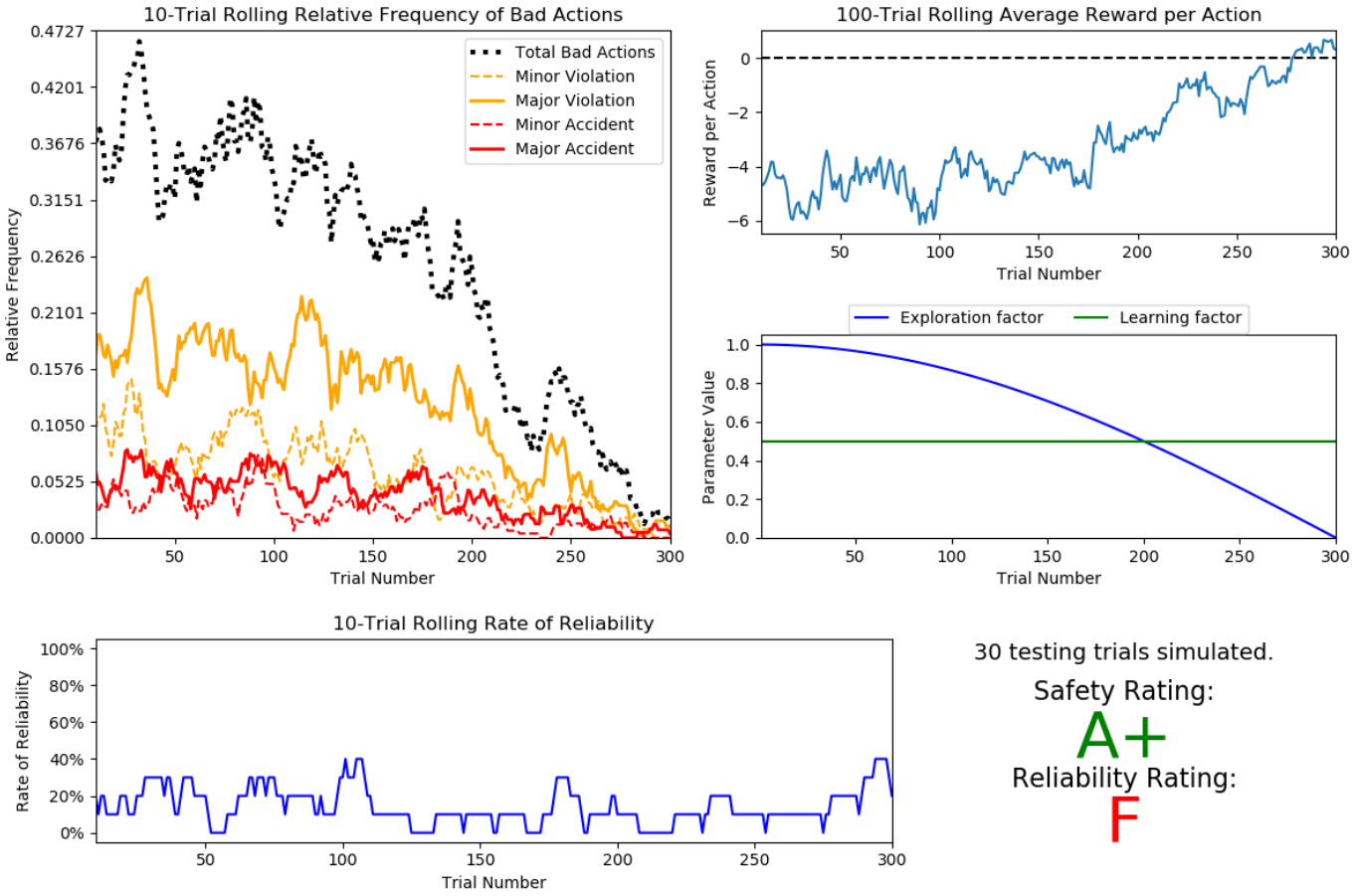
Answer:

1. One is that maximizing the agent's direct reward have already resulted in optimal policy, the agent doesn't need additional noise to modify it. greedy maximize partial observated stated's reward will probagate overestimated rewards.
2. The environment does not have a reward for reaching the goal.

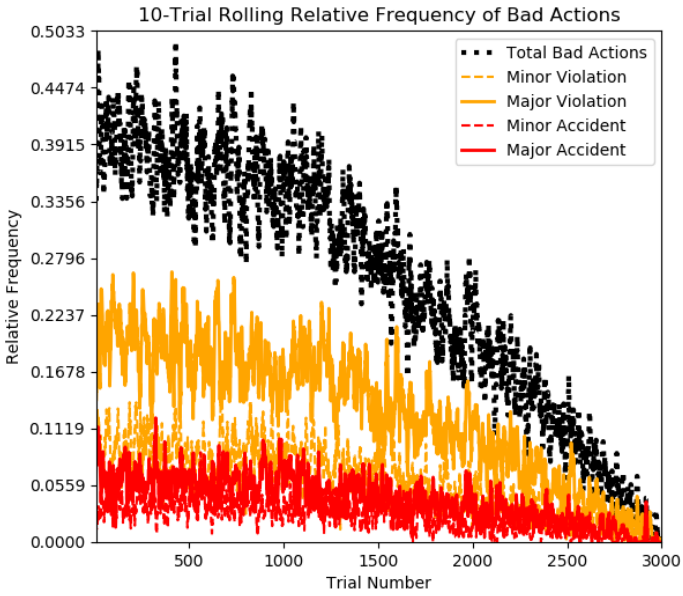
I tried adding additional reward when the agent reach the goal, at the same time, I deleted all rewards from violation 0(not necessary anymore, no optimal policy from direct reward meaning no need to hard code it). epsilon decay function: "cos", reaching destination reward: 50, no rewards for any violation 0 action, alpha decay from 1 to 0, gamma = 0.8, 300 training trials:

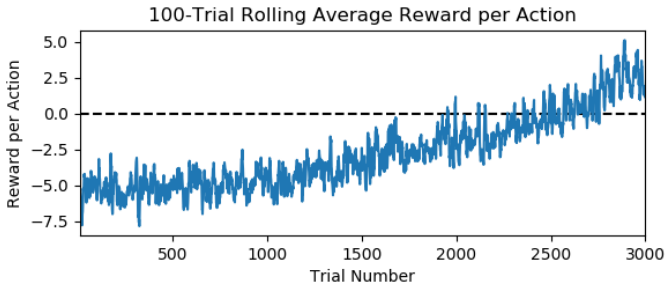


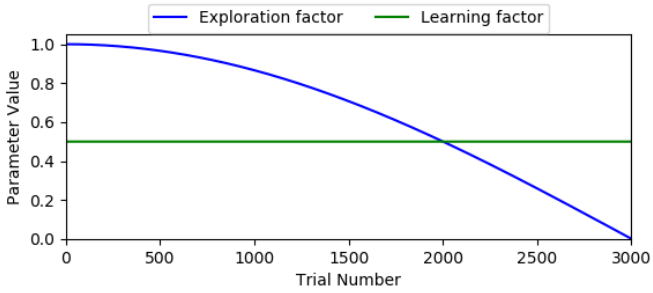
but if I set the alpha to 0.5 as a constant, the reliability will fail:

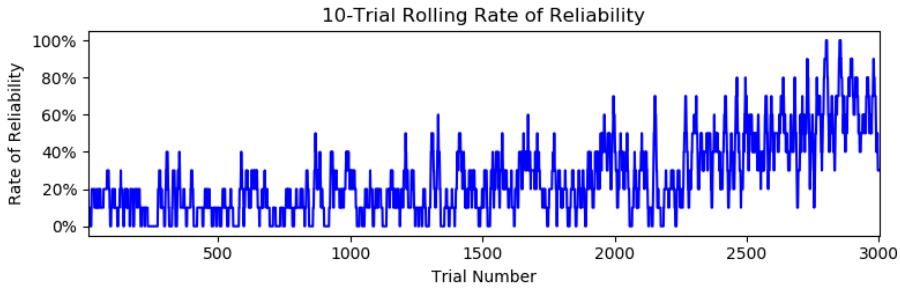


Finally, I added another feature to the agent's perceivable state, which is the difference between deadline and the distance. Thus, the agent will have relatively complete observation to its environment so that one state can only be associated to fewer more relevent real environment configs. See that the state-action space grows up to 612, I will run 3000 trials with alpha constant = 0.5:









30 testing trials simulated.

Safety Rating:

A

Reliability Rating:

C

As what I suggest before, if the agent perceives the full environment, even with a constant alpha, the Q table is more likely to be converged so that the final result is not bad.

> **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.

Typesetting math: 100%