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 [55]:

```
# 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: No, the Smartcab doesn't move at all. The message keeps showing "!! Agent state not been updated!" At the moment, there are three types of rewards, namely,

- Agent properly idled at a red light
- Agent idled at a green light with no oncoming traffic
- Agent idled at a green light with oncoming traffic.

If the light changes to green, the car should go but doesn't go. The rewards become negative. On the other hand, if the light changes to red, the car should not go. Since the car is idled all the time, the rewards are positive.

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:
As shown in agent.py, there are couple flags can be set up for different purposes. For instance,

- verbose can display additional output from the simulation
- learning can activate the Q-learning for the driving agent
- n_test can determine the discrete number of testing trials to perform

The function act(self, agent, action) in Environment class is called to perform the action. In simulator.py Python file, render_text() is the non-GUI render display of the simulation. The feedback of running code is shown in the terminal/command prompt. The render() is GUI the render display of the simulation. As shown in line #38 and #58 of planner.py Python file, the function next_waypoint() will consider East-West direction first, then North-South.

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 [ ]:

```
#import pandas as pd
#pd.__version__

# Load the 'sim_no-learning' log file from the initial simulation results
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: From 10-trail rolling relative frequency of bad actions, it indicates about more than 40% of actions are bad. Major violation happens about 20% of the time, while minor violation happens about 6% of the time. Major accident occurs about 6% of the time, and minor accident occurs 4% of the time. For a driver making decision randomly, I think the low rate (around 15%) of reliability is reasonable. The rating isn't improved from the random action after trials. From 10-trial rolling average reward per action plot, it shows the negative reward throughout the trials without any sign of improvement. There is no improvement of performance due to the same reason. As shown at the right hand corder of the plot, it gives F for both safety rating and reliability rating. The violation seems to be very likely to happen, so does accident. These observations are very intuitive that I won't consider to sit in a cab, where the car movement depends on rolling a dice.

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: The features of intersection light and traffic are most relevant for learning both safety and efficiency. The traffic light, green or red, determines whether agent should move or not. Likewise, the interaction conditions, say oncoming or left-hand-side traffic, determine whether your action makes accident. I would also argue that waypoint is related to efficiency. In order to reach our final destination in finite time step, the right action should be made among every possibility.

The feature "deadline" behaves like an indicator showing the remaining time. If one considers deadline in state space, one has to record the current state at each timestamp such that dimension increases dramatically without any gain. Furthermore, the agent might try too hard to reach the destination on time, even consider the bad driving behavior.

The traffic for a vehicle to the smartcab's right isn't relevant. If the traffic light is green, it means the agent can move while the car at the right hand side should wait. It traffic light is red, the movement of the vehicle at right hand side has no effect on the agent. So that the deadline and light['right'] are not appropriate to represent safety or efficiency.

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: The total number of states should be determined by all the relevant features. There are 3 possibilities, such as moving forward, turning right, and turning left in the feature waypoint. There are 2 options, green and red for traffic light. There are 4 possibilities, right, forward, left, and None associsted with the surrounding traffic. The traffic to your left and front should be taken into account here, but not the one at your right. Our reaction, moving forward, turning left, turning right, or none is also involved in the state. There is an arbitrary number indicating the deadline, which has nothing to do with state. I think the combination of 3(waypoint) * 2(traffice light) * 4(oncoming traffic) * 4(left traffic), 96 in total is the entire state space. I think agent should be able to learn a policy in these 96 possibilities with a reasonable number of trails.

In [56]:

```
from sets import Set
from random import choice

def chance_of_visiting_all_states(iterations, k, n=24):
    r = range(n)
    total = 0
    for i in range(iterations):
        s = Set()
        for j in range(k):
            s.add(choice(r))
            if len(s) == n:
                total +=1
                break
    return float(total)/iterations

steps_attempt = [100, 500, 750, 1000, 2000]
for steps in steps_attempt:
    print "Chance of visiting all states in {:4.0f} steps: {ch}"\
    .format(steps, ch = chance_of_visiting_all_states(2000, steps, 96))
```

```
/anaconda/lib/python2.7/site-packages/ipykernel/__main__.py:1: Dep
recationWarning: the sets module is deprecated
  if __name__ == '__main__':
```

```
Chance of visiting all states in  100 steps: 0.0
Chance of visiting all states in  500 steps: 0.593
Chance of visiting all states in  750 steps: 0.969
Chance of visiting all states in 1000 steps: 0.9975
Chance of visiting all states in 2000 steps: 1.0
```

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:

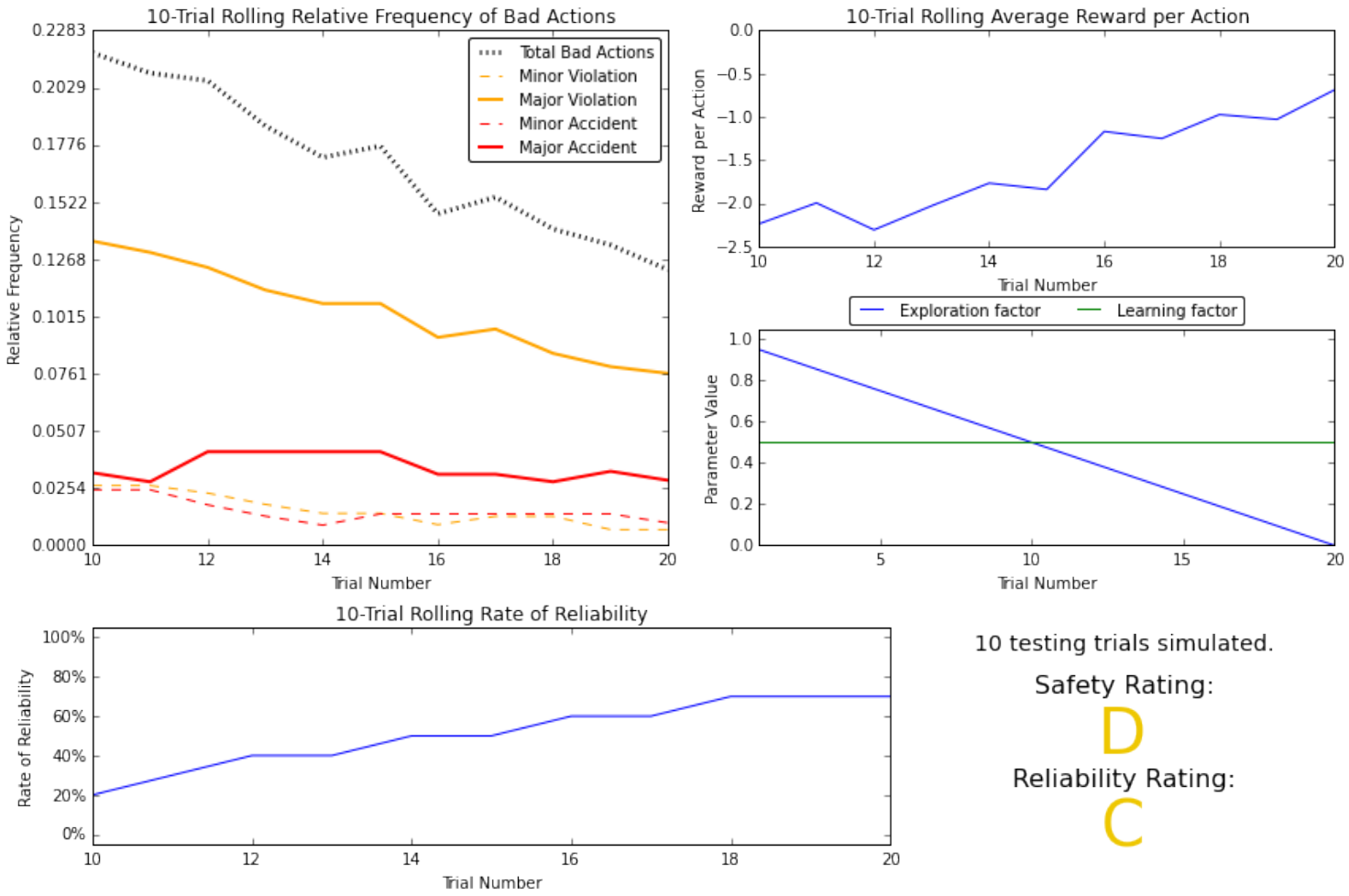
$$\epsilon_{t+1} = \epsilon_t - 0.05, \quad \text{for trial number } t$$

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 [57]:

```
# 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: Here, we introduce default Q-Learning method. In 10-trails Rolling Relative Frequency of Bad Actions plot, total bad actions decrease while trial number increases. The number of bad actions goes from 22% to 13% in 10 trials, which is lower than the non-learning case (around 40%). The number of major violations decreases from 13% to 8% with increasing number of trials. Major accident, minor accident, and minor violation remain at low percentage around 3%, which are lower than non-learning cases (4-8%). The number of major violation is always larger than major accident, for both non-learning and learning cases. The rate of reliability is improved in 10 trials, from 20% to 70%. Similarly, reward per action is also increased with trials, even though it is still negative. Learning factor, α, remains the same as our intention. Exploration factor, ϵ, decreases from 1 to 0 evenly in 20 trials. I would say depending on the given decaying function, the number of training trials is determined (in our case that will be 20 trails). 80% as a rate of reliability is not satisfactory. At the end of 10 trials with learning method, we obtain D for safe rating and C for reliability rating. Furthermore, my other testing trials give me some slightly improved result. Even though the outcome with learning method is always better than non-learning case as shown in Question 3, I think there is still quite some room for improvement.

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):

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

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 [58]:

```
import math
# \epsilon = e^{-at}, \textrm{for } 0 < a < 1

print "It takes {:2.0f} training trials before begining testing. ".format(-ma
th.log(0.05)/0.1)
```

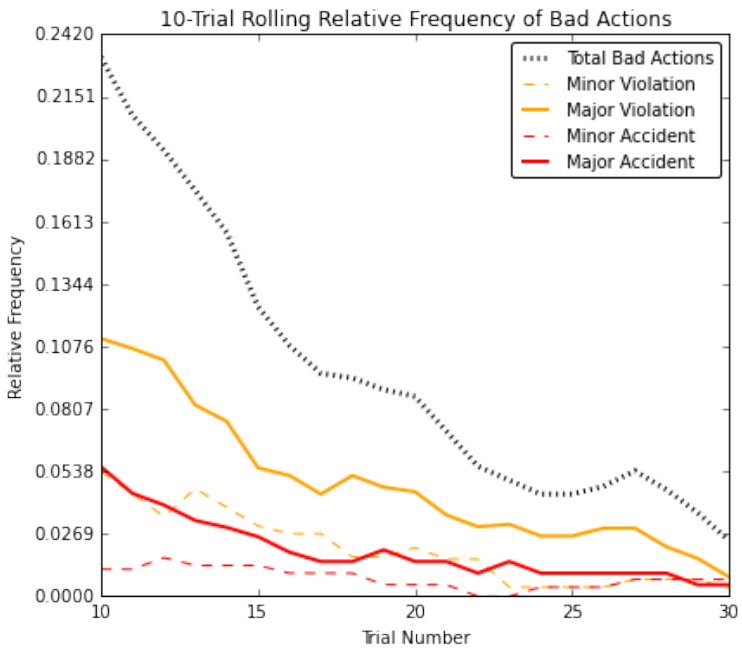
```
It takes 30 training trials before begining testing.
```

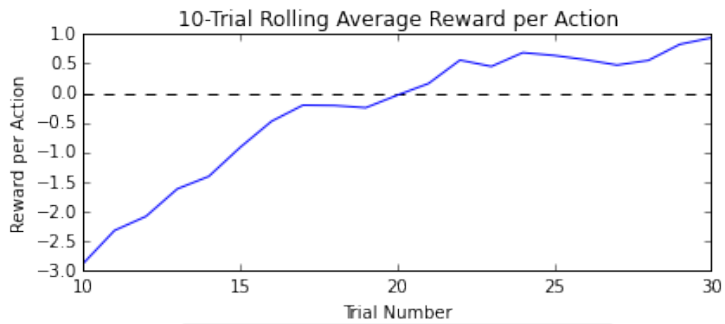
In [59]:

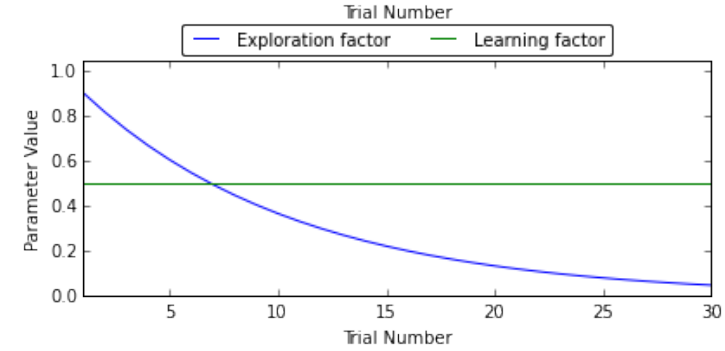
```
#
# https://williewong.wordpress.com/2012/07/24/using-ipython-notebook-for-manu
al-computations/
import my_mathjax as mj
m = mj.math_expr([r"\epsilon = e^{-at}, \textrm{for } 0 < a < 1 "])
m.show()

print "Using default epsilon-tolerance = 0.05 and alpha = 0.5"

# # Load the 'sim_improved-learning' file from the improved Q-Learning simula
tion
print "With a = 0.1 and build_state without waypoint"
vs.plot_trials('sim_improved-learning_exp_10_no_wp.csv')

print "With a = 0.1 and build_state with waypoint"
vs.plot_trials('sim_improved-learning_exp_10_with_wp.csv')
```

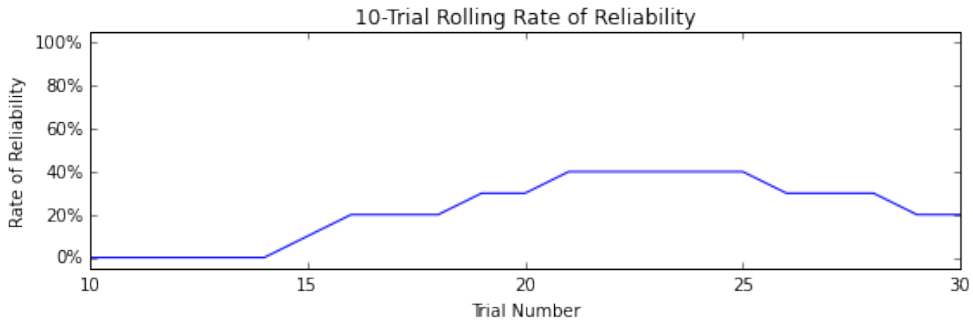
$\epsilon = e^{-at}$, for $0 < a < 1$

```
Using default epsilon-tolerance = 0.05 and alpha = 0.5
With a = 0.1 and build_state without waypoint
```









10 testing trials simulated.

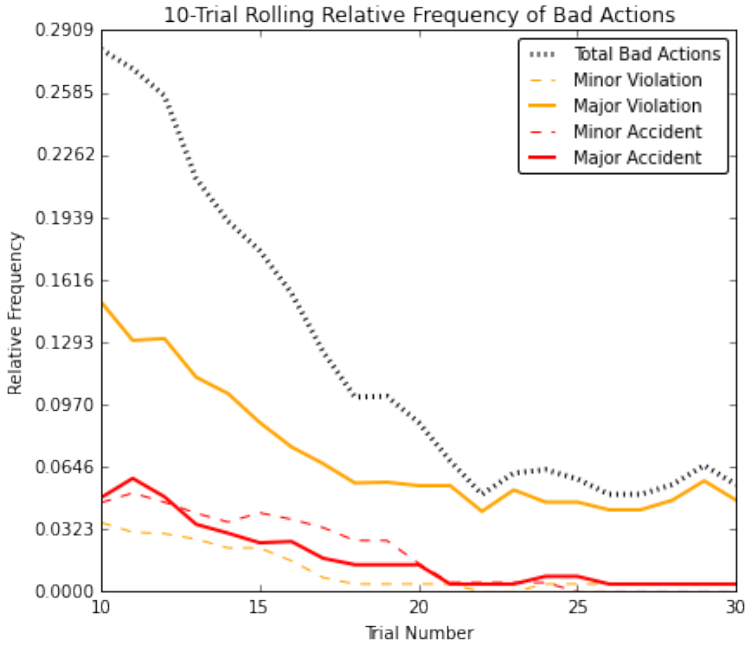
Safety Rating:

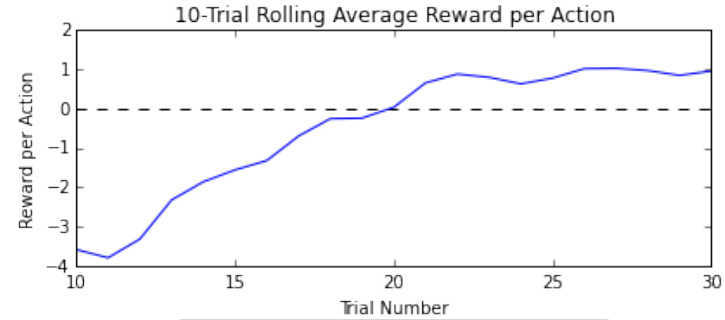
A+

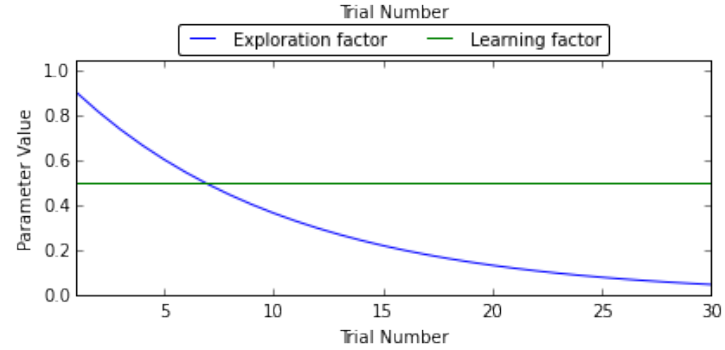
Reliability Rating:

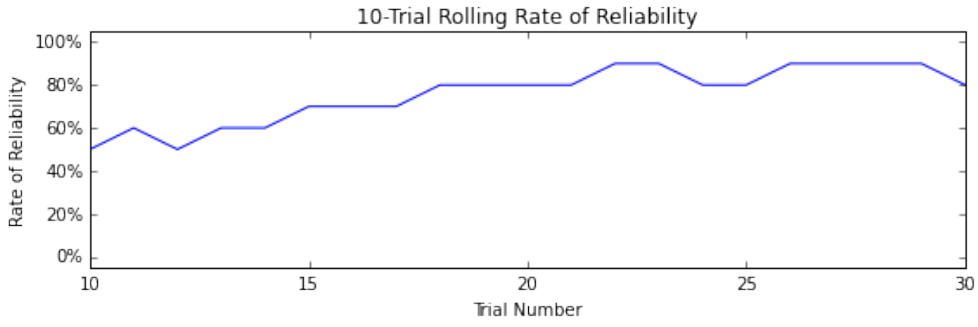
F

```
With a = 0.1 and build_state with waypoint
```









10 testing trials simulated.

Safety Rating:

F

Reliability Rating:

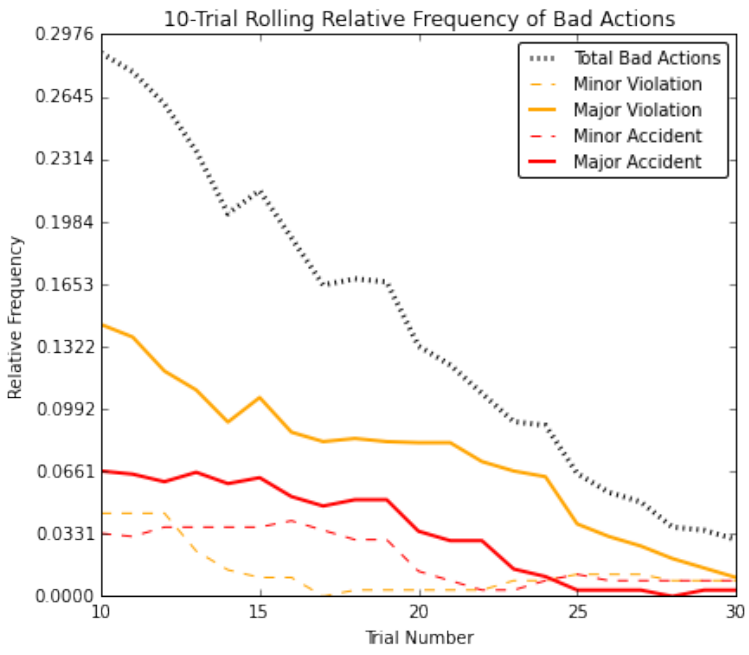
C

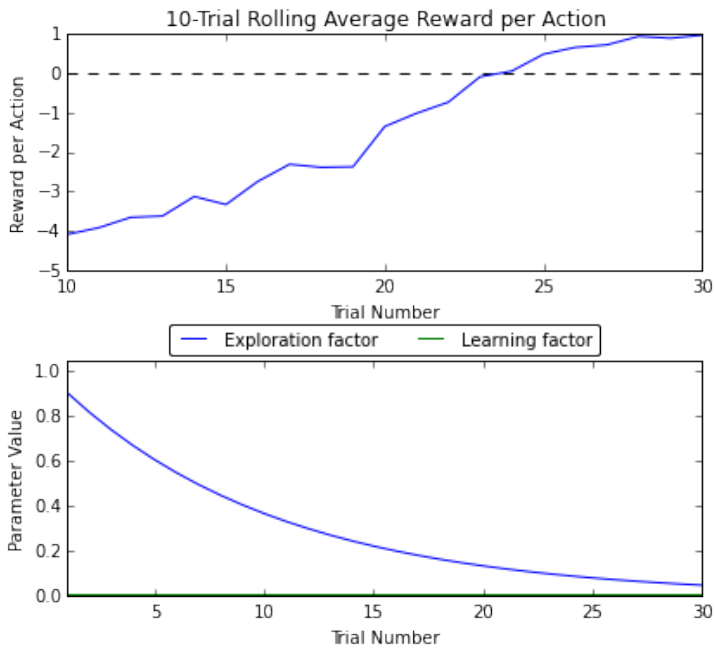
In [60]:

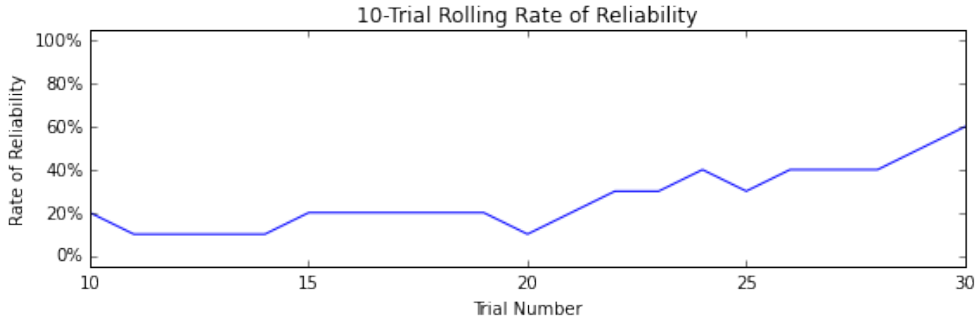
```
print "With a = 0.1 and build_state with waypoint using different learning fa
ctor alpha."

print "alpha = 0.01"
vs.plot_trials('sim_improved-learning_exp_10_with_wp_alpha_01.csv')

print "alpha = 0.99"
vs.plot_trials('sim_improved-learning_exp_10_with_wp_alpha_99.csv')
```

With a = 0.1 and build_state with waypoint using different learning factor alpha.

alpha = 0.01







10 testing trials simulated.

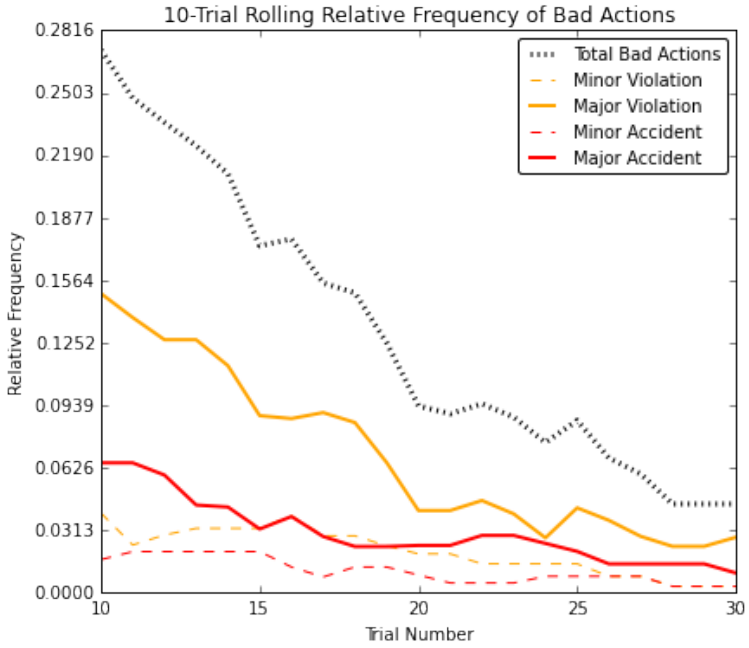
Safety Rating:

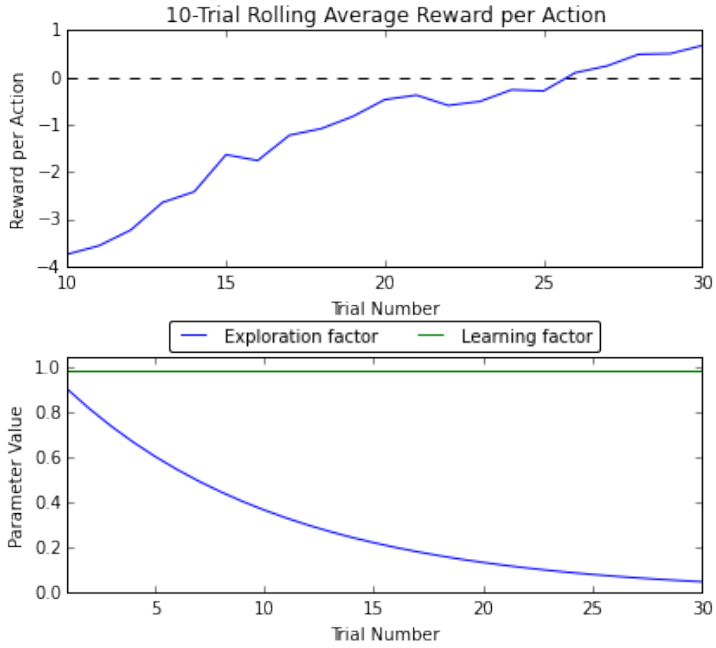
F

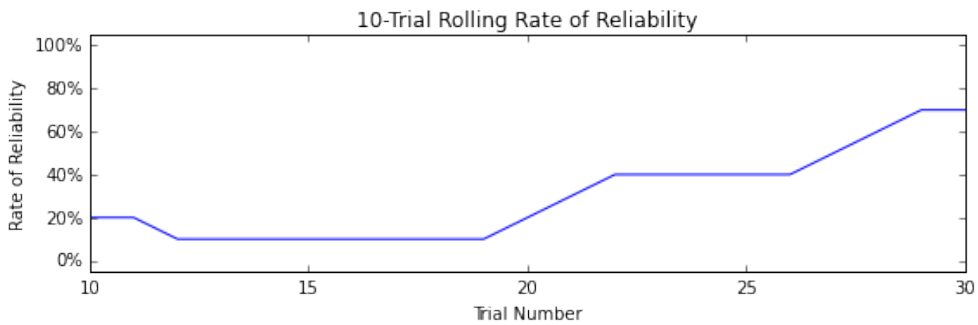
Reliability Rating:

F

alpha = 0.99







10 testing trials simulated.

Safety Rating:

C

Reliability Rating:

C

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: I am using $\epsilon = e^{-at}$ as decaying function with a = 0.1 for the first two cases. ϵ-tolerance and α remain as default, 0.05 and 0.5, respectively. It takes 30 training trials before testing. In first case, I didn't include waypint in our state space but include input['light'], input['ongoing'], and input['left']. In other word, the agent drives by following the traffic rule without knowing where to go. The safety is improved after trials and safety rating reachs A+ score by the end. However, reliability rating is F at the end. It is logical since the agent just drives around but not moves toward the destination. In second case, the feature waypoint is included. Reliability rating is improved (C) but safety rating is poor (F). The agent tries to drive to destination but is still not very efficient.

Intuitively, the waypoint should be considered since one tries to reach the final destination. Therefore I perform other two tests with waypoint included, but with different learning factor α. Here we consider most extreme scenario, $\alpha = 0.01$ and $\alpha = 0.99$. From Q-learning formalism (http://www.cse.unsw.edu.au/~cs9417ml/RL1/algorithms.html), learning rates determine how Q-value is updated. While learning rate is low, the safety and reliability rating are poor due to the lack of experience. In our third case, we obtain F for both safety and reliability rating. While learning rate is high, the new information become more important. In our fourth case, we obtain C for both safety and reliability. I feel one should find the balance to find the optimal learning rate α.

Among all the tests above, I am not satisfied with the safety and reliability ratings. I feel the number of trials is not sufficient enough. We should modify the exploration factor, making sure that the agent considers both exploration and exploitation.

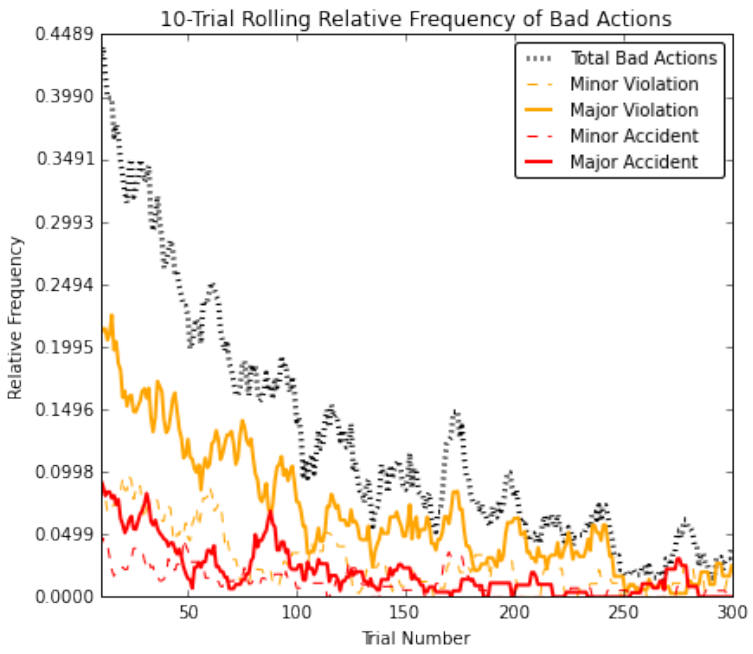
In [104]:

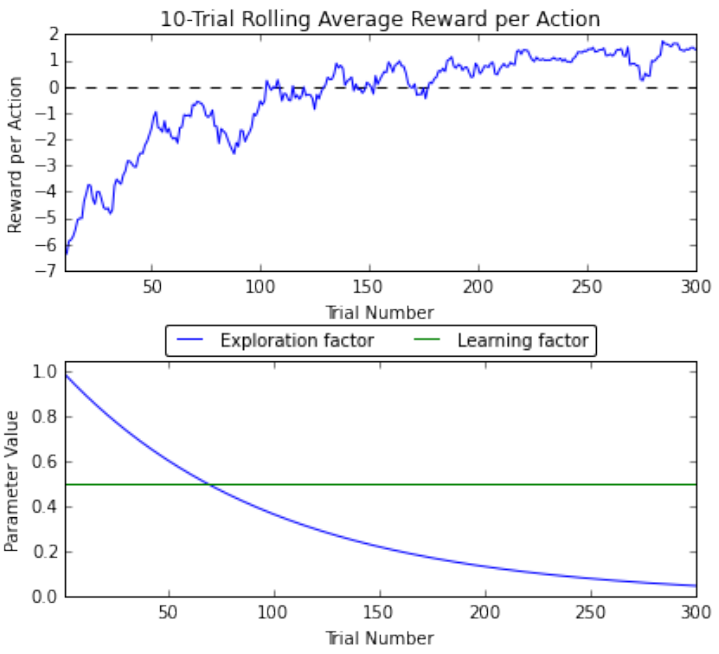
```
print "My optimal case"
#import math
# \epsilon = e^{-at}, \textrm{for } 0 < a < 1
m = mj.math_expr([r"\epsilon = e^{-at}, \textrm{for} 0 < a < 1 "])
m.show()
a = 0.01
print "a = {}, alpha = 0.5, and epsilon-tolerance = 0.05".format(a)
print "It takes {:2.0f} training trials in my optimal case. ".format(-math.lo
g(0.05)/a)

vs.plot_trials('sim_improved-learning_exp_01_with_wp.csv')
```

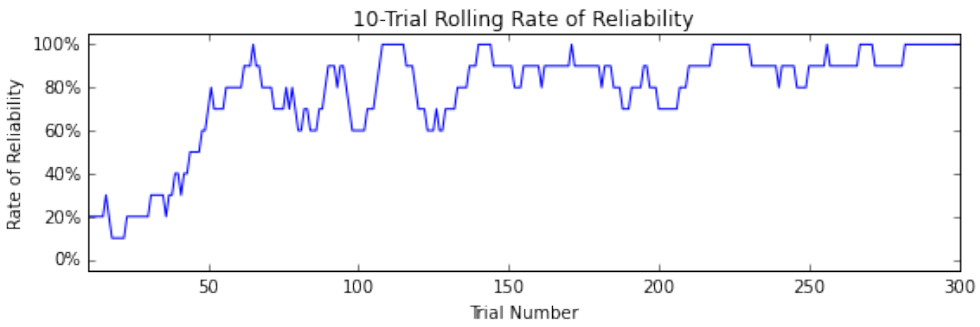
My optimal case

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

a = 0.01, alpha = 0.5, and epsilon-tolerance = 0.05
It takes 300 training trials in my optimal case.







10 testing trials simulated.

Safety Rating:

A+

Reliability Rating:

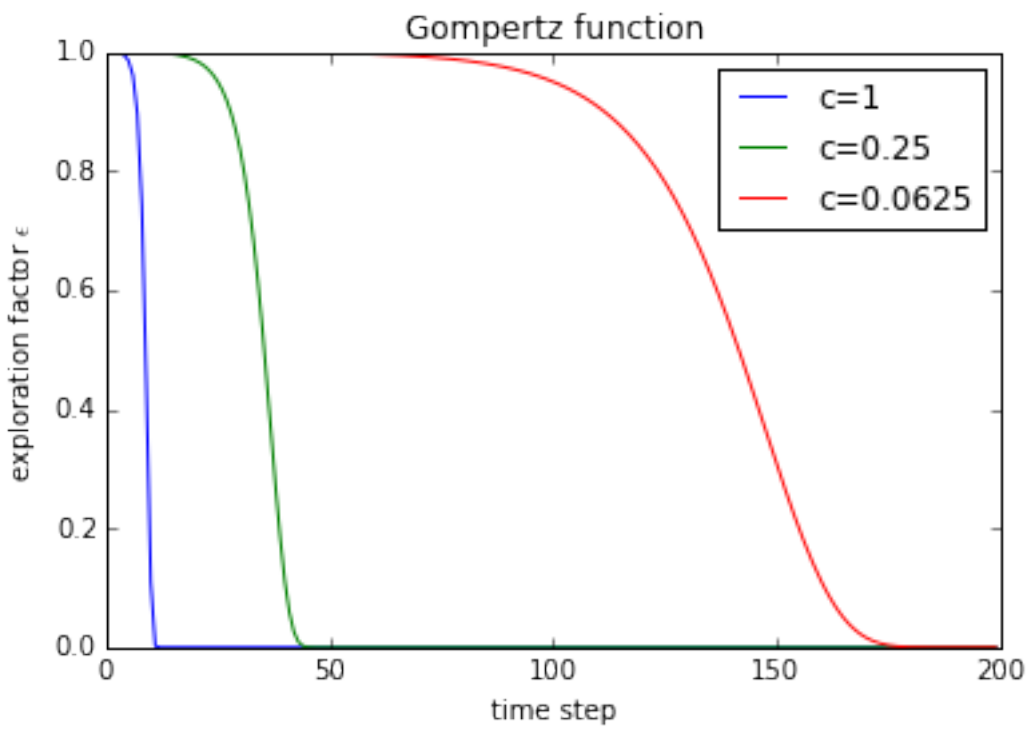
A+

Gompertz function

In [93]:

```
### Gompertz function

import matplotlib.pyplot as plt
import numpy as np
% matplotlib inline

a = 1.0
b = 0.0001

x = np.arange(0,200)
y1 = a*np.exp(-b*np.exp(x))
y2 = a*np.exp(-b*np.exp(x*0.25))
y3 = a*np.exp(-b*np.exp(x*0.0625))

l1 = plt.plot(x, y1)
l2 = plt.plot(x, y2)
l3 = plt.plot(x, y3)

plot_lines = [l1, l2, l3]

legend1 = plt.legend(["c=1", "c=0.25", "c=0.0625"], loc=1)

plt.gca().add_artist(legend1)

m = mj.math_expr([r"\epsilon = ae^{-be^{ct}} \textrm{for }a=1, b=0.0001 "])
m.show()

plt.title('Gompertz function')
plt.xlabel(r'time step')
plt.ylabel('exploration factor $\epsilon$')
plt.show()
```

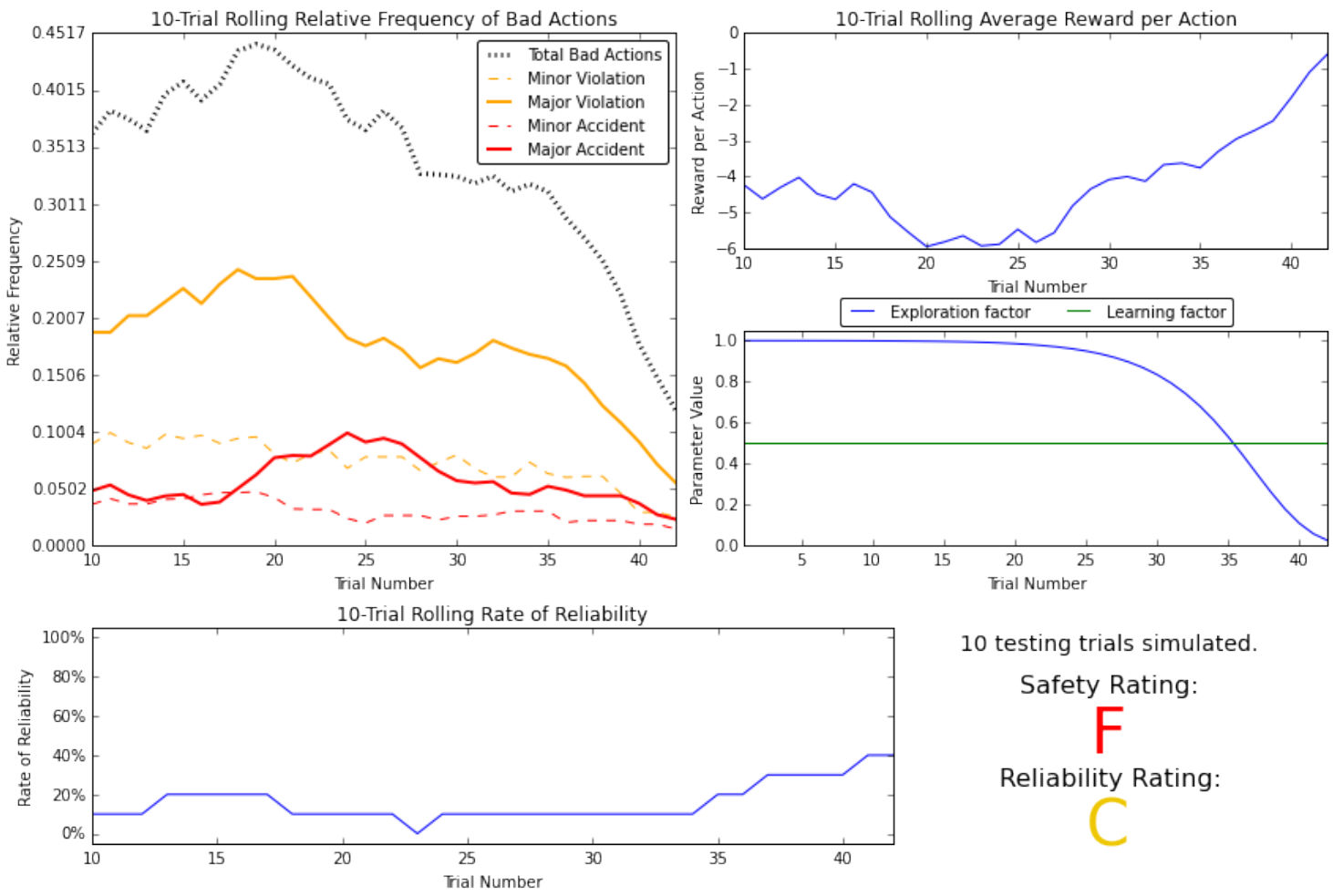
$\epsilon = ae^{-be^{ct}}$ for $a = 1, b = 0.0001$



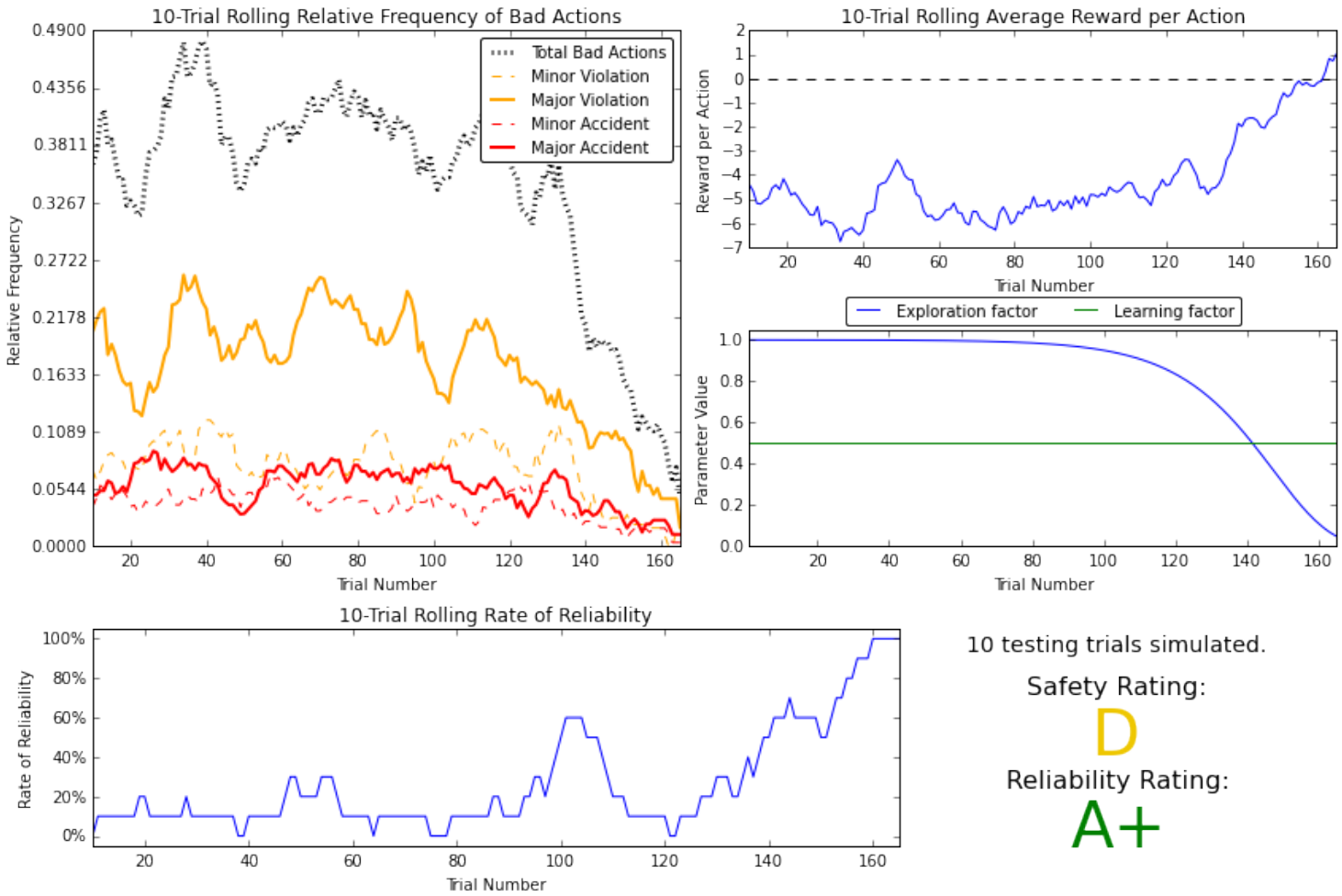
In [107]:

```
#import math
a = 1.0
b = 0.0001
e_tol = 0.05

print "Gompertz function"
c_check = [1, 0.25, 0.0625, 0.03125]
for c in c_check:
    print "For a = {}, b = {}, c = {}, and epsilon tolerance = {}".format(a,b
,c,e_tol)
    print "It takes {:2.0f} training trials before begining testing. "\
    .format(1/c * math.log(math.log(e_tol/a)/-b))

print "\nGompertz function with c = 0.25"
vs.plot_trials('sim_improved-learning_Gompertz_c_point25.csv')
print "Gompertz function with c = 0.0625"
vs.plot_trials('sim_improved-learning_Gompertz_c_point0625.csv')
```

```
Gompertz function
For a = 1.0, b = 0.0001, c = 1, and epsilon tolerance = 0.05
It takes 10 training trials before begining testing.
For a = 1.0, b = 0.0001, c = 0.25, and epsilon tolerance = 0.05
It takes 41 training trials before begining testing.
For a = 1.0, b = 0.0001, c = 0.0625, and epsilon tolerance = 0.05
It takes 165 training trials before begining testing.
For a = 1.0, b = 0.0001, c = 0.03125, and epsilon tolerance = 0.05
It takes 330 training trials before begining testing.

Gompertz function with c = 0.25
```




```
Gompertz function with c = 0.0625
```



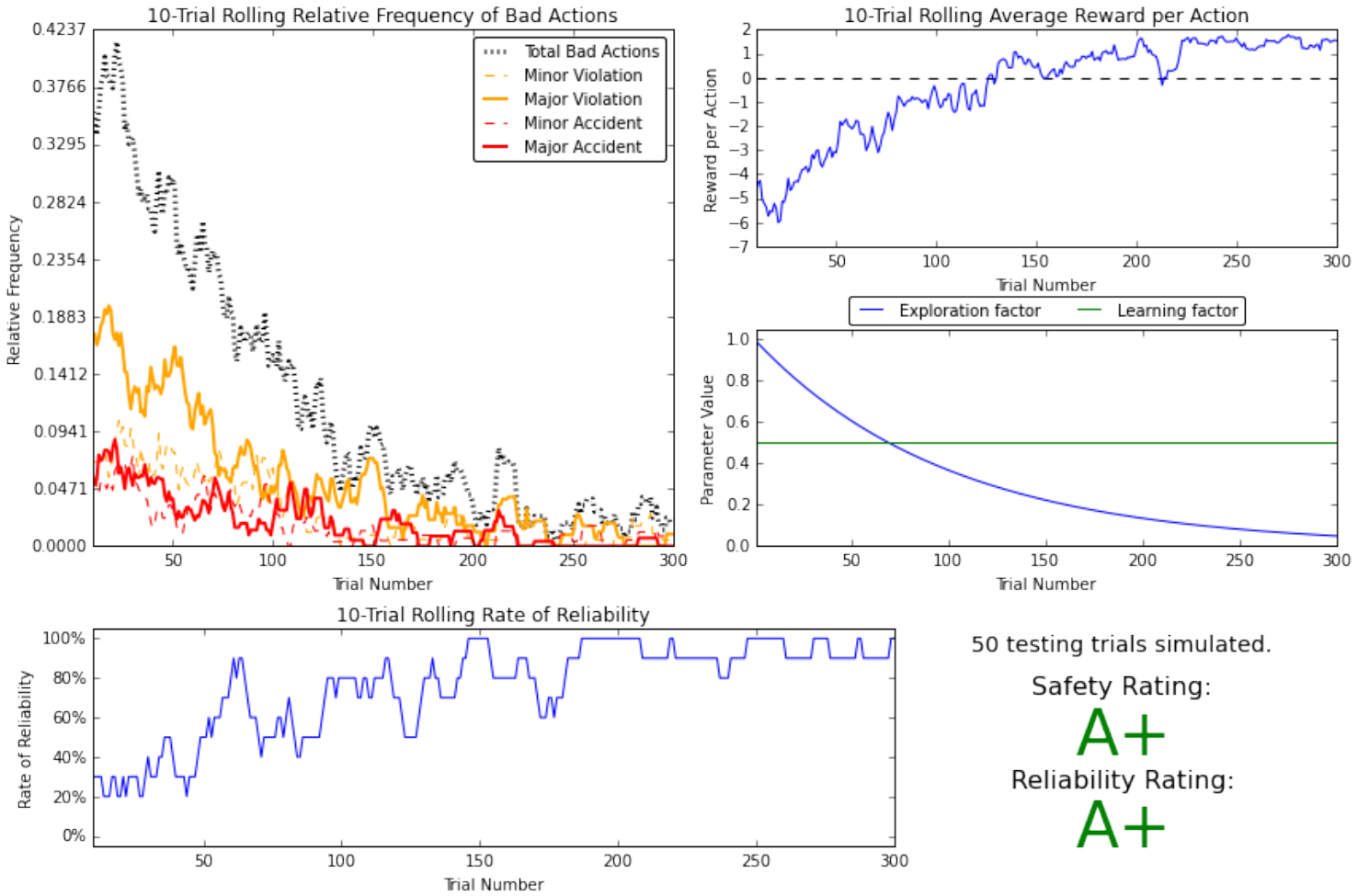
As suggested by the reviewer, 10 testing trials are not sufficient to make solid conclusion regarding its safety and reliability. The following cases run for 50 testing trials.

In [113]:

```
print "case (a)"
m = mj.math_expr([r"\epsilon = e^{-at}, \textrm{where } a = 0.01 "])
m.show()
vs.plot_trials('sim_improved-learning_exp_01_with_wp_test50.csv')

print "case (b) Gompertz function"
m = mj.math_expr([r"\epsilon = ae^{-be^{ct}} \textrm{where }a=1, b=0.0001, c
= 0.03125 "])
m.show()
vs.plot_trials('sim_improved-learning_Gompertz_c_point03125_test50.csv')
```

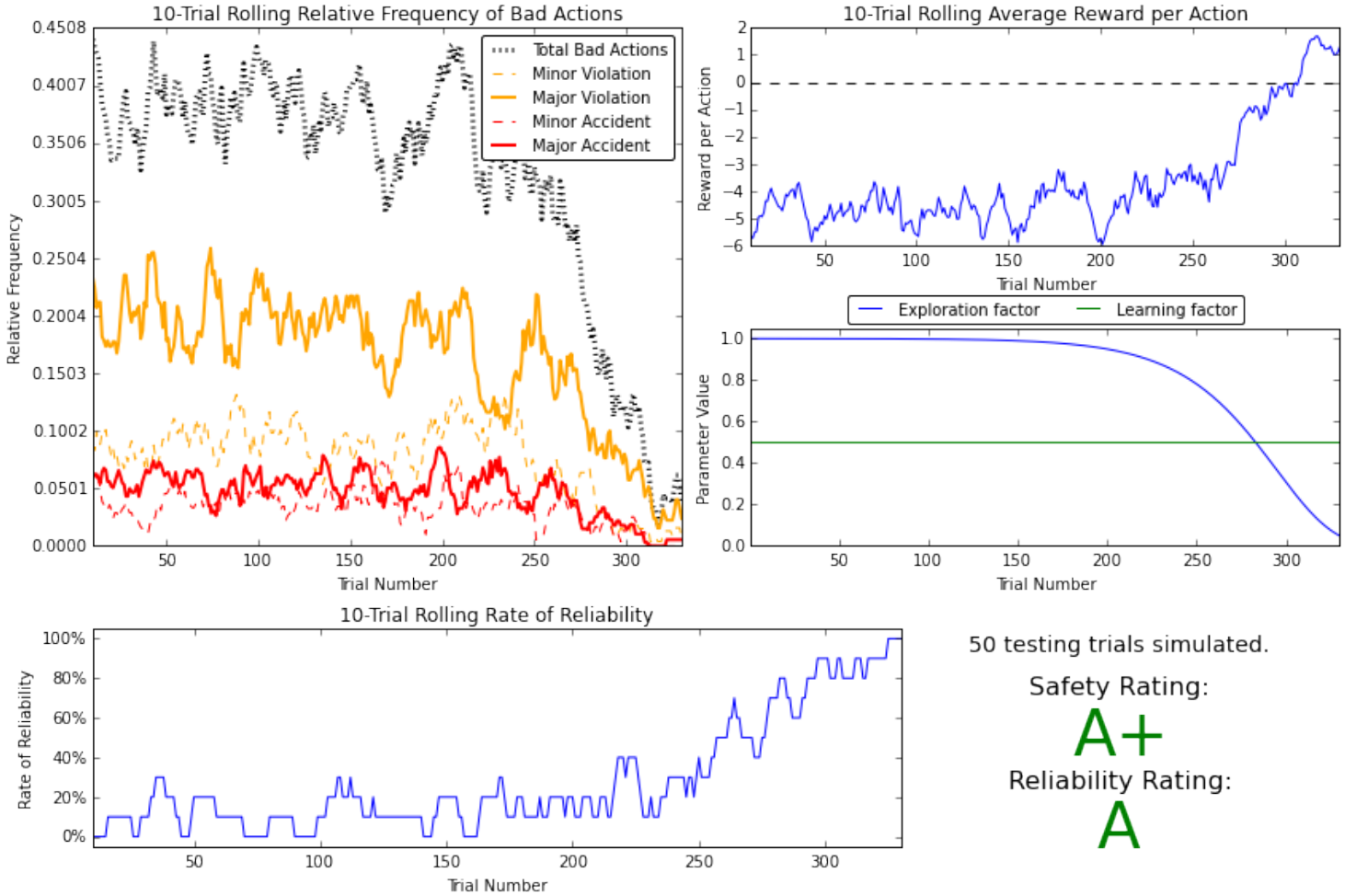
```
case (a)
```

$\epsilon = e^{-at}$, where $a = 0.01$



`case (b) Gompertz function`

$\epsilon = ae^{-be^{ct}}$ where $a = 1, b = 0.0001, c = 0.03125$



First case we use decaying function (a) $\epsilon = e^{-at}$ with a = 0.01, it takes about 300 training trials before begining testing. In order to make comparison with similar number of trials, the parameters of Gompertz function (b) $\epsilon = ae^{-be^{ct}}$ are a = 1.0, b = 0.0001, and c = 0.03125, where 330 trails before testing is performed.

The behaviors of (a) and (b) are very different. For (a), exploration factor gradually decreases such that safety and reliability improve noticeably with increasing trials. For the trial more than 150, total bad actions drop below 10%. The reward per action becomes positive beyond that point. Eventually safety and reliability rating reach A+.

On the other hand for case (b), the total bad action remains above 30% before 250 trials. The rate of reliability is low, around 30%. The bad actions go down while exploration factor starts to decrease. Above 300 trials, the agent seems to be at its exploitation stage such that safety and reliability rating are improved.

In conclusion, the exploration factor ϵ plays a very important role that the agent behaves very different at its training stage. The bottom line is that the number of training trails should be sufficient large for the agent to explore and exploit the policy.

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: I have defined waypoint, input['light'], input['ongoing'], and input['left'] in my state space. For example,

waypoint	light	ongoing	left	my action

forward	green	None	right	forward
forward	red	None	right	None

The next_waypoint function in planner suggests the best action at next intersection. If it is a green light and our waypoint suggests us to move forward, the Q-value of moving forward should be maximum. If it is a red light and our waypoint suggests us to move forward, the Q-value of stop (None) should be maximum. If it is a green light and our waypoint suggests us to turn left, our action is determined by the onging traffic.

Few examples from my sim_improved-learning.txt, where states are defined as (waypoint, inputs['light'], inputs['left'], inputs['oncoming']), are listed below.

('left', 'red', None, 'forward')

- forward : -11.70
- right : 1.12
- None : 2.28
- left : -10.35

('right', 'red', 'left', 'forward')

- forward : 0.00
- right : 1.88
- None : 0.00
- left : 0.00

('right', 'red', 'forward', None)

- forward : -40.18
- right : -19.67
- None : 1.18
- left : -37.58

('forward', 'green', 'left', None)

- forward : 1.58
- right : 0.58
- None : -4.42
- left : 0.68

The action of maximum Q-value is always taken. For example in first case, the Q-table suggested to do nothing (maximum) because the traffic light was red. Turning right was also not an good choice since next_waypoint suggested to go left. The agent went right at red light in second case, and the next_waypoint suggested to turn right. In third example, the next_waypoint suggested to turn right. The agent was allowed to turn right at red traffic light but he/she didn't, due to the car to agent's left was moving forward. In last example, the agent moved forward because the traffic light was green and the next_waypoint suggested the same. (Note: I noticed that the traffic to agent's left was turning left when his lighit was red. Does it mean the other car might break the role?) Overall, I didn't find any state where the policy is different than what would be expected from an optimal policy.

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-

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:
For Smartcab itself, the agent only has the local knowledge at current state. The decision is made according to present information. The decision we have made one step before or will make one step after should not make an influence on our current decision. In other word, our current action is not correlated to our next turning decision. Therefore, γ should be zero such that future rewards don't play a role on changing current action.

For the environment, the traffic and the traffic light are randomly determined. There is no specific rule of the traffic condition. Therefore, we should not consider future rewards for this project.