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.

Import the visualization code

```
import visuals
```

```
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:

- Here are few agent state observations : states coming Agent not enforced to meet deadline,Agent state not been updated! And Agent not enforced to meet deadline, no smart cab doesn't move at all the time during no learning in simulation it was still at few points also.
- Driving agent is getting rewarded with negative rewards between -4.5 to -6.5.
- For red light if agent is getting positive reward if idle and negative if agent is idle at green light and is moving forward in oncoming traffic.

Understand the Code

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

Question 2

In the agent.py Python file, choose three flags that can be set and explain how they change the simulation. In the environment.py Python file, what Environment class function is called when an agent performs an action? In the simulator.py Python file, what is the difference between the 'render_text()' function and the 'render()' function? In the planner.py Python file, will the 'next_waypoint() function consider the North-South or East-West direction first? Answer: (1) The

different flags are learning :- this will set the environment for Agent if he is in learning or in other mode, display :- if set to false will set the GUI to false, log_metrics :- to log values for simulation like for testing , training , default etc. (2) Act function is called when agent performs action which also returns the reward (3) render_text : this is non GUI display and print trial data only, where as render will GUI and also trial data (4) It will consider the East-West first and then North-South direction.

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!

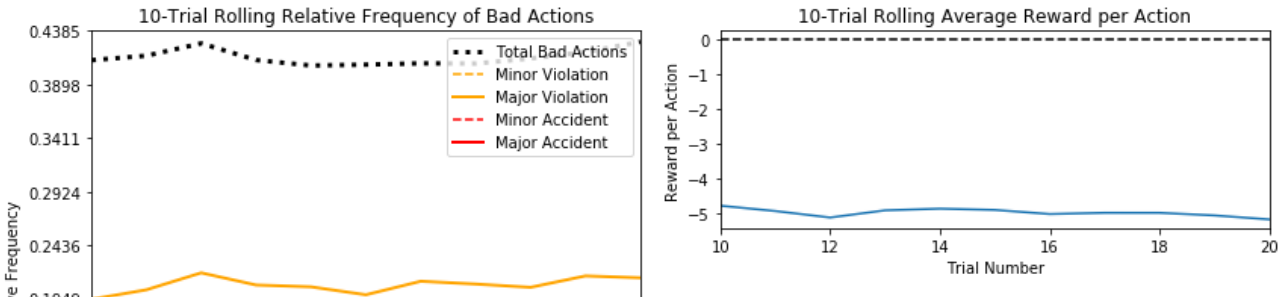
Once you have successfully completed the initial simulation (there should have been 20 training trials and 10 testing trials), run the code cell below to visualize the results. Note that log files are overwritten when identical simulations are run, so be careful with what log file is being loaded! Run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

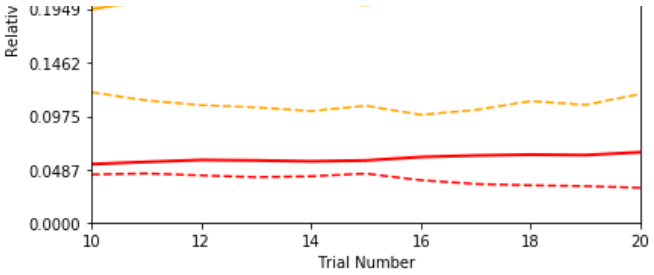
Load the 'sim_no-learning' log file from the initial simulation results

In [7]:

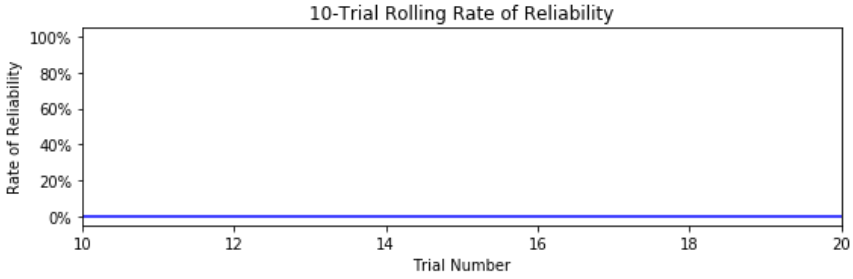
```
import visuals as vs
#Load the 'sim_no-learning' log file from the initial simulation results
vs.plot_trials('sim_no-learning.csv')
print("n_test=10")
```

```
pass
```





Simulation completed with learning disabled.



1 testing trials simulated.

Safety Rating:

F

Reliability Rating:

F

```
n_test=10
```

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:

- In "10-Trial Rolling Relative Frequency of Bad Actions" visualization box Agent is making bad decisions having frequency between .3711 to .4175 every time its driving almost(50% of the time) and causing both major and minor accidents between frequency .0000 and .0942 i.e. around 10% of the time.
- In 10-Trial Rolling of Reliability percentage is lying on lower side below 40%.As we already decided for agent to be reliable it should have pass percentage should be more than 60%.Yes it make sense to have the reliability criteria because the agent aim is reach the destination with safety.
- In 10-Trial Rolling Average Reward per Action Agent is consistently receiving negative awards between -4.5 and -6.5 , as agent is getting more rewards less than -4.5 most of the time so these rewards suggests agent been penalized heavily
- In 10-Trial Rolling Rate of Reliability hasn't changed and mostly remain constant(10%) throughout and similar is case with major and minor accidents not much improvement.Minor violations have gone down over the time but to few extent only.
- If we see from 10-Trial Rolling Relative Frequency of Bad Acttions and 10-Trial Rolling of Reliability the major accidents have increased and so does major violation plus reliability percentage is on lower side so the Smartcab agent can't be considered safe and reliable at this time

Inform the Driving Agent

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:

Below are the features required for agent for learning both safety and efficiency:

- Waypoint - The Smartcab heading must be known by the agent so it should go in the correct direction to reach its intented destination at the end.
- (Inputs: light) - the Smartcab must know the color of the Traffic light so as to learn that it will be penalized for not doing anything when the light is green, or going through intersections with red traffic lights which might cause major or traffic violations and can result to accidents.
- (inputs: left) - This parameter inluded in the inputs is to indicate if there is a vehicle present on the left and its intented action or it can take any of the direction: None, forward, left or right. For no vehicle present and the light is red, then making a right turn maybe allowed and shouldn't go forward.For vehicle present and red light , then taking a right action will result in a major accident if there is also a vehicle on the left with the intent to go forward

a major accident if there is also a vehicle on the left with the intent to go forward.

- (inputs: oncoming) - This feature tell the Smartcab must know if there is a vehicle coming in the opposite direction when it is trying to make a left turn, so as to not go into the oncoming traffic when deciding to turn left. The agent decides to drive into oncoming traffic, then it will cause either a minor traffic violation and it will be given negative rewards or, can cause an accident if there is a vehicle present.

Below features not considered for the following reasons:

- (inputs: right) - Making a right turn on red, does not require any information on if there is a vehicle on the right, since the lane that the right vehicle is on is separate for the direction of travel. In case the light is green its only valid for the case of the smartcab going forward and the Smartcab agent has the right-of-way, fault will of other vehicle for causing any accident making a decision to come into traffic and is not abiding US triffic laws.
- deadline - not as relevent because agent can learn about it in the reward it receives as it builds it in Q table. Setting it this way may help improve on its reliability by causing minor safety violations to major accidents. We can avoid this and allow agent to learn it on its own.

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:

With the above mentioned 4 features below is the description : -

Features	Number of states	States
1-Waypoint	3	forward, left, right
2-(inputs: light)	2	red, green
3-(inputs: left)	4	None, forward, left, right
4-(inputs: oncoming)	4	None, forward, left, right

The possible number of states are based on combination of 1x2x3x4 or 3x2x4x4 = 96, yes the agent should be able to learn with these number of states

Update the Driving Agent State

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

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

Implement a Q-Learning Driving Agent

The third step to creating an optimized Q-Learning agent is to begin implementing the functionality of Q-Learning itself. The concept of Q-Learning is fairly straightforward: For every state the agent visits, create an entry in the Q-table for all state-action pairs available. Then, when the agent encounters a state and performs an action, update the Q-value associated with that state-action pair based on the reward received and the iterative update rule implemented. Of course, additional benefits come from Q-Learning, such that we can have the agent choose the best action for each state based on the Q-values of each state-action pair possible. For this project, you will be implementing a decaying, ϵ-greedy Q-learning algorithm with no discount factor. Follow the implementation instructions under each TODO in the agent functions.

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

{ 'state-1': { 'action-1' : Qvalue-1, 'action-2' : Qvalue-2, ... }, 'state-2': { 'action-1' : Qvalue-1, ... }, ... } Furthermore, note that you are expected to use a decaying ϵ (exploration) factor. Hence, as the number of trials increases, ϵ should decrease towards 0. This is because the agent is expected to learn from its behavior and begin acting on its learned behavior. Additionally, The agent will be tested on what it has learned after ϵ has passed a certain threshold (the default threshold is 0.05). For the initial Q-Learning implementation, you will be implementing a linear decaying function for ϵ.

Q-Learning Simulation Results

To obtain results from the initial Q-Learning implementation, you will need to adjust the following flags and setup:

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

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

If you have difficulty getting your implementation to work, try setting the 'verbose' flag to True to help debug. Flags that have been set here should be returned to their default setting when debugging. It is

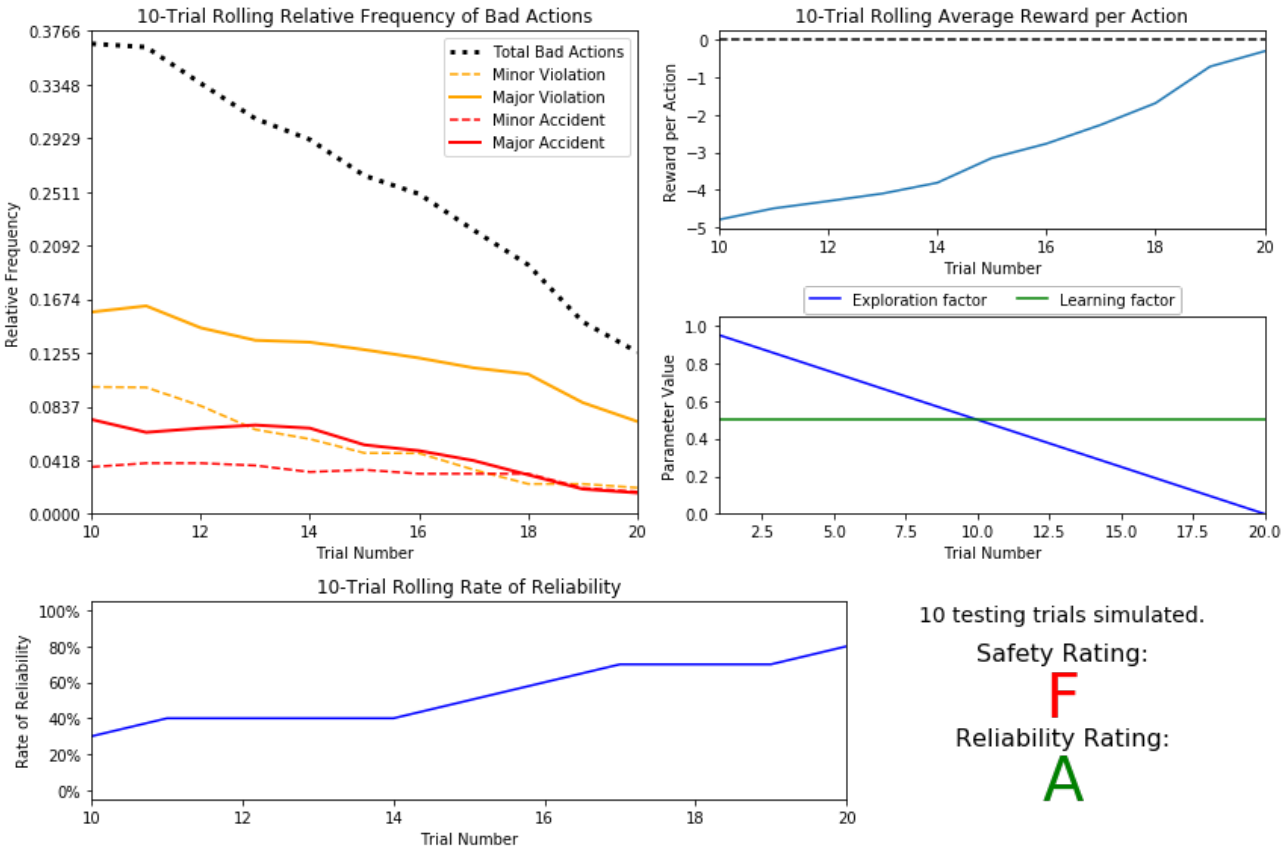
debug. Flags that have been set here should be returned to their default setting when debugging. It is important that you understand what each flag does and how it affects the simulation!

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 [8]:

```
#Load the 'sim_default-learning' file from the default Q-Learning simulatio
n
import visuals as vs
vs.plot_trials('sim_default-learning.csv')
print("decay:self.epsilon = self.epsilon - 0.05,tolerance: 0.05,alpha = 0.5
")
```

```
pass
```




```
decay:self.epsilon = self.epsilon - 0.05,tolerance: 0.05,alpha = 0.5
```

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?

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

Answer:

- Number of violation and actions are somewhat similar in Trial Run similar to no learning.
- 20 training trails are required before testing, yes it makes sense as the tolerance become less than epsilon the agent should learn about the states and take correct decisions.
- For decaying function I used

$$\epsilon = \frac{1}{t^2}$$

  functions which is correctly represented in the exploration factor as attached in the default learning visualizations as it starts from 1.0 then to 0.25 onwards.
- As can be seen from Rolling Relative Frequency of Bad Actiop value for Bad Action have decreased from near .44 to between .399 to .299 from this its concluded that bad actions are decreasing
- No of violations and accidents have gone down as compared to intial driving agent agent has got A+ and A ratings after testin trials

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}, \textr

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!

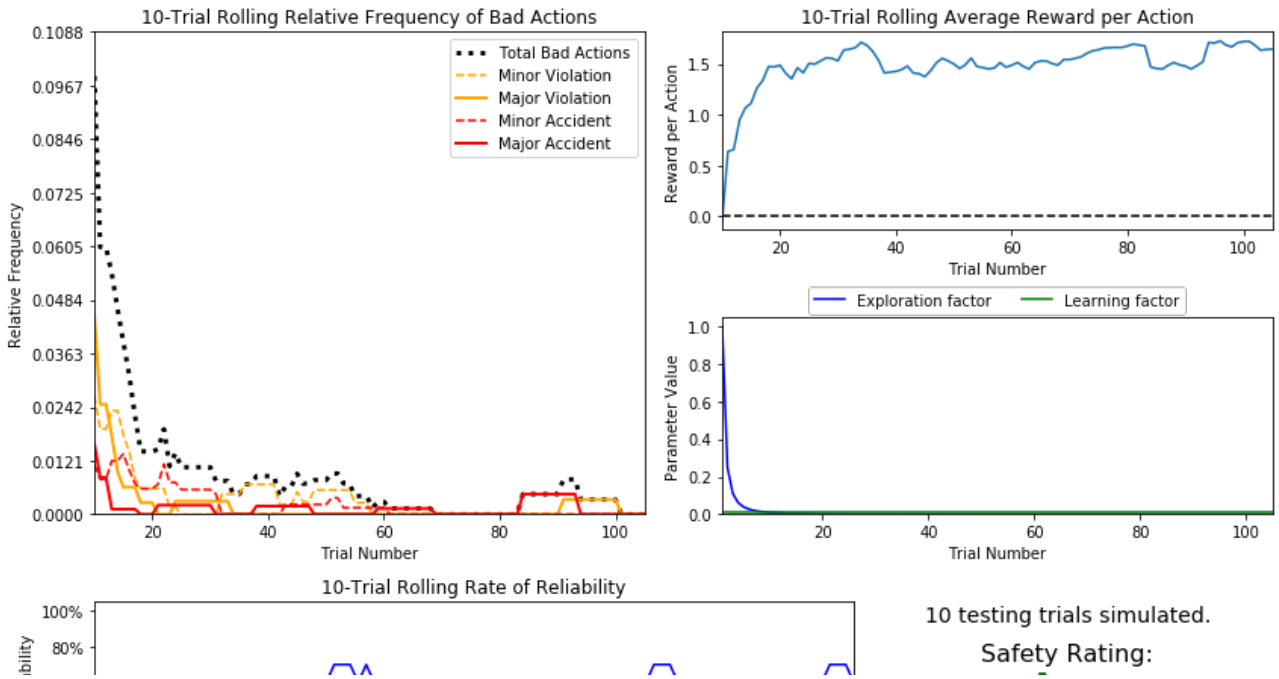
If you have difficulty getting your implementation to work, try setting the 'verbose' flag to True to help debug. Flags that have been set here should be returned to their default setting when debugging. It is important that you understand what each flag does and how it affects the simulation!

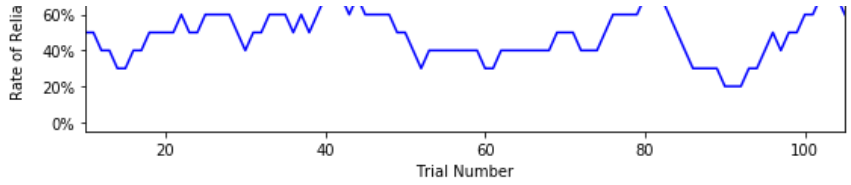
Once you have successfully completed the improved Q-Learning simulation, run the code cell below to visualize the results. Note that log files are overwritten when identical simulations are run, so be careful with what log file is being loaded!

In [3]:

```
# Load the 'sim_improved-learning' file from the improved Q-Learning
simulation
import visuals as vs
#Improved Learning with 1/t^2 decay method
vs.plot_trials('sim_improved-learning-t2.csv')
print("epsilon = 1.0/(t**2), alpha=0.01, tolerance=0.001, n_test=10 , train
ing_trials = 105")
```

```
pass
```





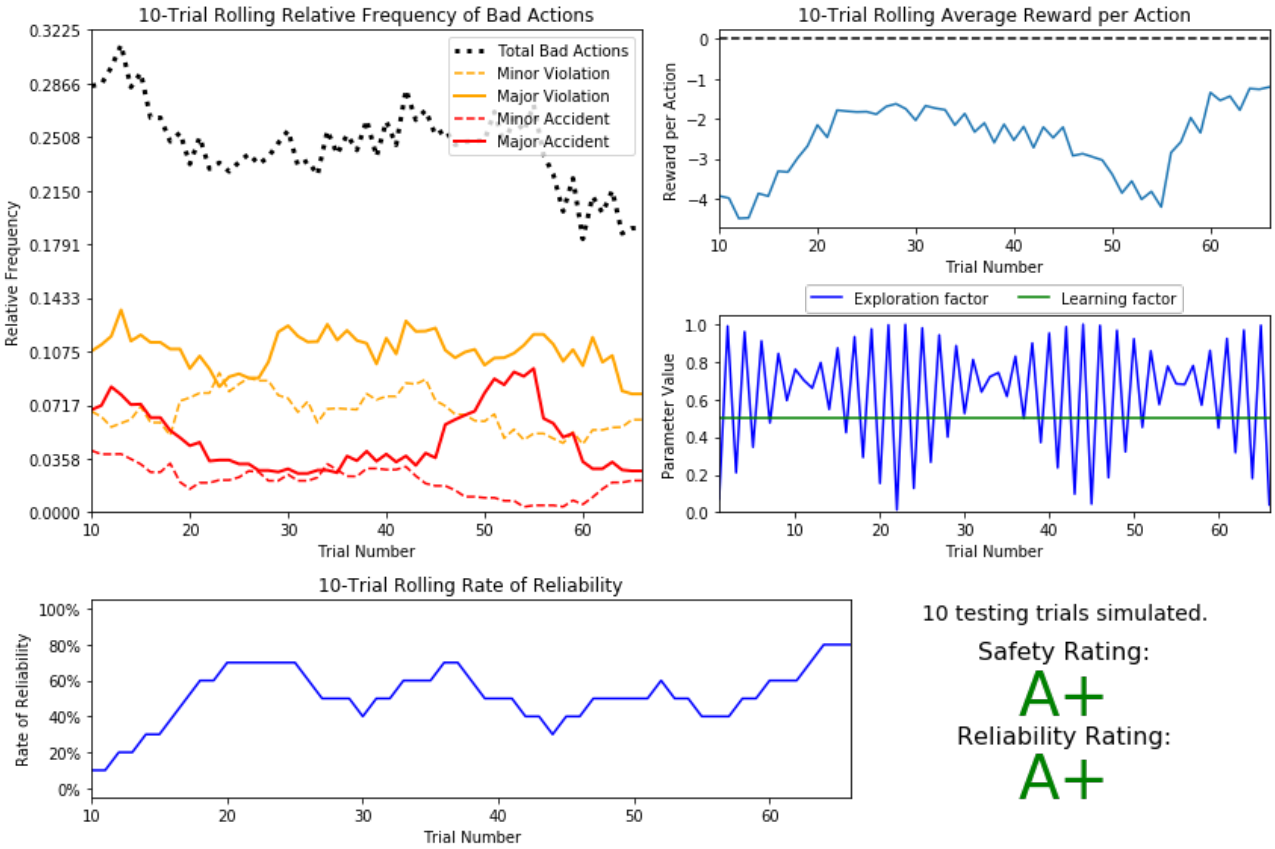
A+
Reliability Rating:
D

epsilon = 1.0/(t**2), alpha=0.01, tolerance=0.001

In [5]:

```
# Load the 'sim_improved-learning' file from the improved Q-Learning simulation
import visuals as vs
#Improved Learning with cos decay method
vs.plot_trials('sim_improved-learning_cos.csv')
print("epsilon = math.fabs(math.cos(self.alpha*self.decayval)), alpha=0.5, tolerance=0.1, n_test=10, training trials=60")
```

pass



10 testing trials simulated.
Safety Rating:
A+
Reliability Rating:
A+

epsilon = math.fabs(math.cos(self.alpha*self.decayval)), alpha=0.5, tolerance=0.1, n_test=10, training trials=60

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:

- Below mentione Decaying functions are being used :-

S.No.	Exploration Factor	Alpha	Tolerance	Safety	Reliability	No of Tests
1	$\epsilon = \frac{1}{t^2}$	0.01	0.001	A+	D	10
2	$\epsilon = ABS(COS(at))$	0.5	0.1	A+	A+	10

- It took me around 100 training trials before the final optimized agent testing was done
- For

$$\epsilon = ABS(COS(at))$$

  decay function: epsilon 1.0 and tolerance 0.1 because values were becoming zero at this point and again shifting towards 1.0 again so to break it tolerance requires was 0.1 but with 1/t^2 dacy method it was 0.001 as values for epsilon dips quickly as compared to cos decay.I have used this tolerance values so that agent is able to learn all the possible 96 combinations I have mentioned in question no 5 above.
- Rewards have gone towards slightly positive side. Minor and major violations and accidents have gone down a little more.Now the agent has got A+ rating for both reliability and safety.
- The agent has got A+ rating more than the basic agent for safety and reliability it means agent has learned an optimal policy
- I am satisfied with A+ rating for safey also A+ ratings

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

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.

Next, investigate the 'sim_improved-learning.txt' text file to see the results of your improved Q-

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?

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.

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. Below are few optimal policy examples taking into consideration state discionary and condition in environment:

State Dictionary	(self.inputs['light'] + "-" + CreateStateFromDictionary(self.inputs['left']) + "-" + CreateStateFromDictionary(self.inputs['oncoming']) + "-" + CreateStateFromDictionary(waypoint))
Conditions for optimal policy	if action == 'forward': if light != 'green': # Running red light violation = 2 # Major violation if inputs['left'] == 'forward' or inputs['right'] == 'forward': # Cross traffic violation = 4 # Accident # Agent wants to drive left: elif action == 'left': if light != 'green': # Running a red light violation = 2 # Major violation if inputs['left'] == 'forward' or inputs['right'] == 'forward': # Cross traffic violation = 4 # Accident elif inputs['oncoming'] == 'right': # Oncoming car turning right violation = 4 # Accident else:# Green light if inputs['oncoming'] == 'right' or inputs['oncoming'] == 'forward': # Incoming traffic violation = 3 # Accident else: # Valid move! heading = (heading[1], -heading[0]) # Agent wants to drive right: elif action == 'right': if light != 'green' and inputs['left'] == 'forward': # Cross traffic violation = 3 # Accident else: # Valid move! heading = (-heading[1], heading[0]) # Agent wants to perform no action: elif action == None: if light == 'green': violation = 1 # Minor violation

No.	State	Action	policy
1	green-left-left-forward	forward	Correct
2	green-None-left-forward	forward	Correct
3	green-left-forward-forward	forward(optimal),right(suboptimal)	Correct
4	green-left-left-left	left	Correct
5	green-left-right-right	right	Correct
6	green-forward-left-forward	forward	Correct

2,3. As seen seen sim_improved-learning.txt file below are the few results and explanations for agent learns the optimal policy

- Policy Explanations As oncoming traffic in neither right or forward so forward action is optimal

State	Q-Table Data
green-left-left-forward	-- forward : 1.78 -- right : 1.50 -- None : -2.48 -- left : 0.55
green-None-left-forward	-- forward : 2.56 -- right : 1.26 -- None : -5.23 -- left : 1.20

- Policy Explanations If action is forward the policy is optimal as input left is not forward

State	Q-Table Data
green-left-forward-forward	-- forward : 1.36 -- right : 1.13 -- None : -2.26 -- left : -10.39

- Policy Explanations As oncoming is left instead of right or forward so left will be the optimal policy

State	Q-Table Data
green-left-left-left	-- forward : 0.75 -- right : 0.93 -- None : -3.82 -- left : 1.85

- Policy Explanations As the light is green and action is right and got positive rewards which is correct for below table:

State	Q-Table Data
green-left-right-right	-- forward : 0.96 -- right : 1.68 -- None : -2.58 -- left : -9.68

- Policy Explanations If action is forward it will be optmial policy else right or left is sub optimal as oncoming is not right or forward :

State	Q-Table Data
green-forward-left-forward	-- forward : 1.74 -- right : 0.59 -- None : -3.20 -- left : 0.58

Optional: Future Rewards - Discount Factor, 'gamma'

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

Optional Question 9

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

Answer:

The first thing about this environment is that is not exactly like the real world scenarios like how the cabs are using GPS for tracking thier destinations and second thing destination goals are randomly selected without any coordinates to make it more realiable for future rewards ready.

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.