**Tolerance : 0.05 to 0.01**

**Optimized : False to True**

**Eplison : 0.05 to 0.005**

**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 [88]:

# Import the visualization code

import visuals as vs

# Pretty display for notebooks

%matplotlib inline

**Understand the World**

Before starting to work on implementing your driving agent, it's necessary to first understand the world (environment) which the *Smartcab* and driving agent work in. One of the major components to building a self-learning agent is understanding the characteristics about the agent, which includes how the agent operates. To begin, simply run the agent.py agent code exactly how it is -- no need to make any additions whatsoever. Let the resulting simulation run for some time to see the various working components. Note that in the visual simulation (if enabled), the **white vehicle** is the *Smartcab*.

**Question 1**

In a few sentences, describe what you observe during the simulation when running the default agent.py agent code. Some things you could consider:

* *Does the Smartcab move at all during the simulation?*
* *What kind of rewards is the driving agent receiving?*
* *How does the light changing color affect the rewards?*

**Hint:** From the /smartcab/ top-level directory (where this notebook is located), run the command

'python smartcab/agent.py'

**Answer:**

The smartcab remains in the same spot during the entirety of the simulation since it has no function dictating movement. The rewards vary depending on the lights color.If the light is red, the car waits at the light and is positively rewarded. If the light is green the car waits through the light and negatively rewarded.

**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**
  + *Flag 1*: learning
    - When set to True the driving agent will use Q Learning implementation and make decisions based on the rewards it recieved for previoius actions in the same situation instead of random actions.
  + *Flag 2*: epsilon
    - The decaying exploration factor of the Q Learning algorithm set somewhere between 0 and 1. This number is compared to a random number also between 0 and 1 to determine whether the cab will use a random action or perform the action with the highest Q value. This number decays over trials determined by a reduction method. Once epsilon has been reduced to a certain threshold set in simulation.py the training trials will cease and the testing trials will begin if the testing flag is set to True.
  + *Flag 3*: enforce\_deadline
    - Determines whether or not the smartcab will have a pass/fail time limit to reach its destination. IF the smartcab doesn't reach the destination within the alotted moves the trial will be aborted.
* **environment.py**
  + act(self, agent, action)
* **simulator.py**
  + render\_text()
    - a text based explanation of what is happening in the simulation that is displayed in the terminal.
  + render()
    - the GUI where you can actually view what is happening in the simulation
* **planner.py**
  + East-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!

In [128]:

# 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:**

* The driving agent is making bad decisions around 40% of the time resulting in an accident (major or minor) around 10% of the time.
* Given the random nature of the driving agent the reliability being around a maximum of 30% is reasonable, and is reflected in the rewards. The rewards are around -5 suggesting most of the decisions that the driving agent is making are resulting in accidents or violations.
* The outcomes don't change that significantly throughout the tests.The violations and accidents are mostly consistent throughout. The rewards hover just above -5, and the reliability drops from 30% to under 20% as the trials progress.
* This is not a safe Smartcab, and I would not recommend this death machine to my worst enemy. It frequently gets into accidents or performs traffic violations that could endangering others.

**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:**

* inputs['light'],inputs['oncoming'], inputs['left], waypoint are the most important features for ensuring a safe and effiecient ride in the smartcab. The status of the light and the waypoint determine whether to stop or go and where to go next.
* The oncoming traffic feature is used to determine whether it's clear or not to make a left hand turn. Going forward or right can depend on the yielding of the other car in cases which they will always do in the simulation, which is why there is no purpose paying attention to the car coming from the right. However, special attention must be paid to the cars coming from the left since that will determine when the driving agent is able to make a right on red.

**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:**

* **2** (input, light = [red, green]) \*
* **4** (oncoming = [none, right, left, forward]) \*
* **4** (left = [none, right, left, forward]) \*
* **3** (waypoint = [right, left, forward]) =
* **96 states**
* Yes it could. In 200 trials each state could be executed approximately 15 times if distributed equally.

**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,* *-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* *(exploration) factor*. Hence, as the number of trials increases, 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 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 .

**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 :

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 [99]:

# 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 (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:**

* They both have charts in their output. While the basic driving agent maintains poor rating consistently, the Q Learning agent has the possibility of performing well amongst the failures.
* Approximately 20 trials before testing. This is the correct amount of trials given the epsilon-tolerance, which can be seen decaying by 0.05 in the trial number graph.
* The number of bad actions decreased constistenly as the rewards consistenly increased during the trials progression. The safety and realiabilty ratings definitely have the potential to be higher than the intial driving agent. These are the expected results from implementing the Q Learning agent.

**Improve the Q-Learning Driving Agent**

The third step to creating an optimized Q-Learning agent is to perform the optimization! Now that the Q-Learning algorithm is implemented and the driving agent is successfully learning, it's necessary to tune settings and adjust learning paramaters so the driving agent learns both **safety** and **efficiency**. Typically this step will require a lot of trial and error, as some settings will invariably make the learning worse. One thing to keep in mind is the act of learning itself and the time that this takes: In theory, we could allow the agent to learn for an incredibly long amount of time; however, another goal of Q-Learning is to *transition from experimenting with unlearned behavior to acting on learned behavior*. For example, always allowing the agent to perform a random action during training (if 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 (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 , the number of trials):

You may also use a decaying function for (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 .

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 [109]:

# Load the 'sim\_improved-learning' file from the improved Q-Learning simulation

vs.plot\_trials('sim\_improved-learning.csv')

**Question 7**

Using the visualizatio n 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 kept epsilon = 1.0 and changed the decay function to epsilon - 0.005 and the tolerance = 0.001, which resulted in approximately 200 trials before testing. The learning rate, alpha = 0.25, and was decided upon after experimenting with different values.
* The improved learner commonly maxes out its safety and reliability rating compared to the high amount of failures produced by the default learner. It has a lower decay function, whcih results in 10x the amount of trials compared to the default learner, its average reward per action was above 0 for the last 25 trials and capped near 2, and reliability maxed out at 100% while all accidents and violations zeroed out.

**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:**

Using the states I defined optimal policies would be:

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

* Optimal 1:
  + ('forward', 'green', 'left', None)
    - forward : yes
    - right : no
    - None : no
    - left : no
* Optimal 2:
  + ('left', 'green', 'forward', 'left')
    - forward : yes
    - right : no
    - None : no
    - left : no
* Optimal 3:
  + ('left', 'green', None, 'right')
    - forward : no
    - right : no
    - None : no
    - left : yes
* Incorrect policy from sim\_improved-learning.txt:
  + ('forward', 'red', 'right', 'right')
    - forward : 0.00
    - right : 0.22
    - None : 0.00
    - left : 0.00

This policy is incorrect due to the waypoint directing forward, but the driving agent choosing to go right instead of waiting for the light change and progressing forward. It seems during the trials this state maybe only came up one time and the out come randomly chose to go right and was scantily rewarded since it doesn't violate any traffic laws but didn't follow the waypoint.

* Correct policy from sim\_improved-learning.txt:
  + ('right', 'red', 'None', 'left)
    - forward : -7.27
    - right : 2.14
    - None : 1.03
    - left : -26.30

This is the correct policy for the state above with an interesting caveat: On a red light it is legal to turn right if there is no cross traffic coming from the left. There must have been a situation where the smartcab turned right on red and was successful since there is a positive number in the right action. However, since inputs['left'] is not included in our states, the optimal policy for this state will always be to stop at the red light since the driving agent is not checking the traffic coming from the left.

**Optional: Future Rewards - Discount Factor, 'gamma'**

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

**Optional Question 9**

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

**Answer:**

* *Issue 1*:
  + There is no point using future rewards since the the destinations the cab will be driving towards will change.
  + The future rewards will favor routes of earlier trials and could lead the driving agent further away from its current goal.
* *Issue 2*:
  + Since waypoint has already decided the driving agent's next move there is no sense in rewarding the smartcab with future rewards that it's suppose to be making if it can anyways.
  + When assuming future rewards for future states wouldn't red lights, in the context of predicting future states, become a problem on anything besides a legal right turn since that'd be the only state where the driving agent continues moving? I guess the same can be said for random traffic as well.

**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.