

smartcab

August 31, 2017

1 Machine Learning Engineer Nanodegree

1.1 Reinforcement Learning

1.2 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.

1.3 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.

Grade	Safety	Reliability
D	Agent causes at least one minor accident, such as turning left on green with oncoming traffic.	Agent reaches the destination on time for at least 60% of trips.
F	Agent causes at least one major accident, such as driving through a red light with cross-traffic.	Agent fails to reach the destination on time for at least 60% of trips.

To assist evaluating these important metrics, you will need to load visualization code that will be used later on in the project. Run the code cell below to import this code which is required for your analysis.

```
In [1]: # Import the visualization code
import visuals as vs

# Pretty display for notebooks
%matplotlib inline
import seaborn
```

1.3.1 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*.

1.3.2 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'
```

The agent state is not updated during the simulation period. Agent experiences two states. Sometimes agent is standing idle at red light which is correct behavior and receives positive reward, and sometimes the agent stands idle at a green light with no incoming traffic and receives negative reward.

1.3.3 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.

1.3.4 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?**

Flags:

- **Flag 1: Grid size:** sets the number of intersection in environment. Larger environment will increase the run time of each episode.
- **Flag 2: epsilon:** is the rate of choosing random actions not selecting action greedily to based on current policy. Larger epsilon lead to an agent that explores the environment and decreasing it leads to a greedy agent that only chose actions based that maximize the result based on current policy
- **Flag 3: num_dummies:** Increase or decrease the number of dummy agent in the environment.

The class method `act` is executed when the agent performs an action.

The `render_text` method render the environment in text format and display information in the terminal. The `render` method display the graphical representation of the data in `render_text` method.

The `next_waypoint()` method first considers the `East_West` direction.

1.4 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.

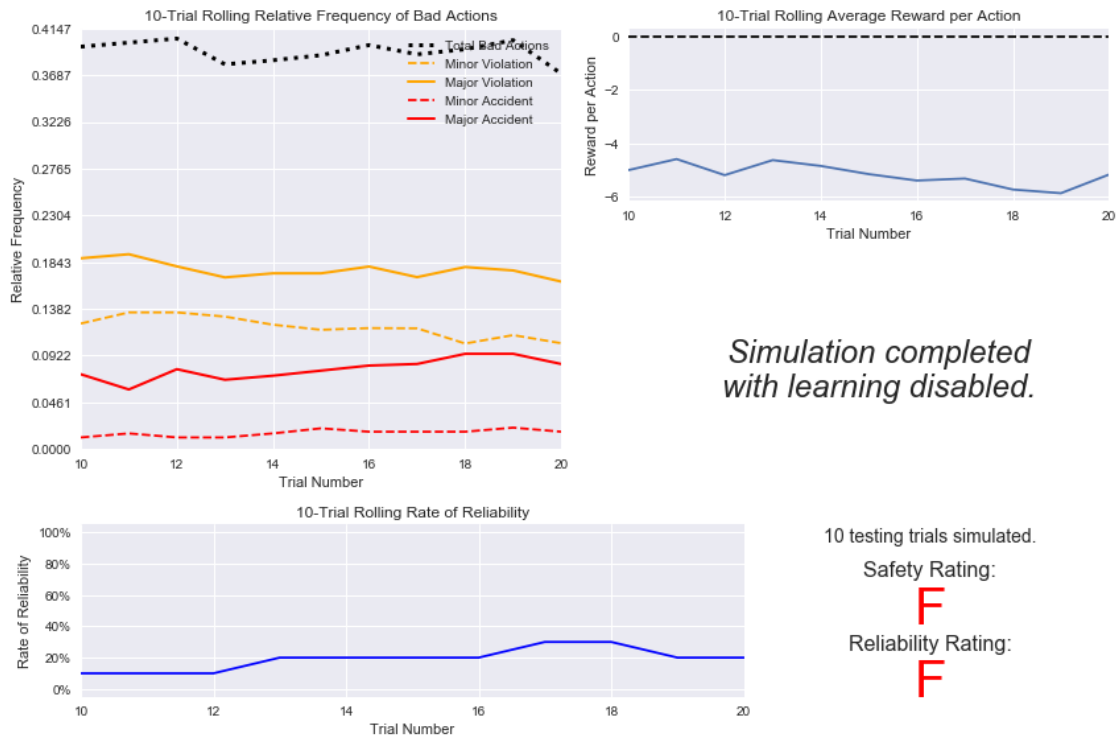
1.4.1 Basic Agent Simulation Results

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 simulation 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 [3]: # Load the 'sim_no-learning' log file from the initial simulation results
        vs.plot_trials('sim_no-learning.csv')
```



1.4.2 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?*

- Approximate close to 40% of time the agent chose the wrong action which approximate 18% of times led to major violation, 12% of minor violation, 9% major accidents and 2% minor accidents.
- Yes, the action space contains 4 possible options and it is expected that an agent acting randomly will choose the correct action 25% of times.
- The agent approximately received a reward close to -5 during the simulation period which is consistent with acting randomly and it was penalized accordingly for taking the wrong actions.
- No, the rate of reliability shows little fluctuations which indicates that the agent is not learning a better policy.

- The agent is not safe and/or reliable currently. The agent actions led to major and minor accidents and violations 50% of times and it was late 80% of the time.

1.5 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!

1.5.1 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.

1.5.2 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.

features appropriate for modeling:

way point: This variable is necessary for the agent to reach the intended destination,

inputs-light: The status of red/green light is necessary for the agent to avoid collisions and violations.

inputs-left: The intended direction of the vehicle on the left is needed to avoid collision.

inputs-oncoming: The intended travel direction of vehicle across intersection is necessary for the agent to safely turn at the intersection.

The variable deadline is not necessary since the reward negative reward for being idle force the agent to reach the destination as fast as it can. It may help agent to prefer safer action when it has enough time to reach the destination and riskier action when its closer to the deadline but the since it significantly enlarge the state space I prefer to not include this variable. Also since the turning right is legal it may not be necessary to know the direction which the car in the right want to move.

1.5.3 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!

1.5.4 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!

Considering the 5 features selected and the possible values for each feature, the state space size is 384. The state space is not to be large but we have to test whether vanilla Q-learning works in this case or not. There are few possible solutions and extensions that can solve large state space problems such as value function approximation or policy gradient methods.

1.5.5 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!

1.6 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.