report

October 4, 2018

1 Machine Learning Engineer Nanodegree

2 Project: Train a Smartcab to Drive

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.

Table of Contents

2.1 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 timefor 100% of trips.
A	Agent commits few minor traffic violations,such as failing to move on a green light.	Agent reaches the destination on timefor 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 timefor at least 80% of trips.

Grade	Safety	Reliability
C	Agent commits at least one major traffic violation, such as driving through a red light.	Agent reaches the destination on timefor 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 timefor at least 60% of trips.
F	Agent causes at least one major acci-dent,such as driving through a red light with cross-traffic.	Agent fails to reach the destination on timefor 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
```

2.1.1 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*.

2.1.2 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'
```

Answer: - So as I run this command `python smartcab/agent.py` to this directory on conda emulator, it executes/runs the `agent.py` file and starts to test the smartcab how it performs to different environments. Parallel to this, another window pops up called `'pygame window'` (simulator) to show the environment the smartcab is working in. It turns out that this window freezes and it shows complete blank screen, no environment seen whatsoever where the smartcab is working in. Also at the very beginning when the code starts executing, it shows this message `Simulator.__init__(): Error initializing GUI objects; display disabled`, but when I check the `__init__()` in `simulator.py` file, the `display` is set to `True` which makes me think that there's a bug in pygame. The conda emulator on the other hand which runs `agent.py` file, keeps executing the code.

- As the code is being exectued of `agent.py` file, it is testing the samrtcab performance in two conditions
  - Performance in Red Lights.
  - Performance in Green Lights.
- When smartcab is tested for red lights, it shows that it is in idle state and its been receiving postive rewards ranging from `2.98 to 2.25`, which makes sense since we should be in idle state on red ligths. When its tested under Green Lights with no oncoming traffic, it's still in idle state and its been receiving negative rewards ranging from `-4.9 to -4.25`, which makes sense since it shoudn't be idle on green lights even though there's no traffic ahead.
- Seeing such results its clear that the smartcab doesn't move at all. The red light gives the smartcab positive rewards while at green lights negative. The agent has not been yet taught the rules of traffic signals, or may be its been taught nothing. ;)
- And on neither performance tests the agent is forced to meet deadlines which is bad because without setting time limit we can't know how reliable the smartcab is to work in given environment.

2.1.3 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.

2.1.4 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?*

Answer: - In the `agent.py` file, these are the 3 flags that change the simulation. * **num_dummies**: This sets number of dummy agents in the environment we can set. The higher the dummy agents, the higher our chances of accidence and also slows our agent to let by go other dummies who has higher priority. * **alpha**: Sets the rate of learning, default is 0.5. At the rate 0, the agent will not learn anything and when set to 1 it would consider to take the recent information. * **display**: Default True, Set to False to disable the GUI if PyGame is enabled.

- In the `environment.py` file, `act()` function is called when an agent performs an action.
- In the `simulater.py` file, `render_text()` is the non-GUI render display of the simulation, simulated trail data is rendered in the command prompt, while `render()` is GUI render display of simulation.
- In the `planner.py` file, the `next_waypoint()` considers first whether the agent has reached the destination, if not then checks for whether the destination is East or West direction to its location, if not then checks for whether the destination is North or South direction to its location. Therefore it considers **East-West direction first.**

2.2 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.

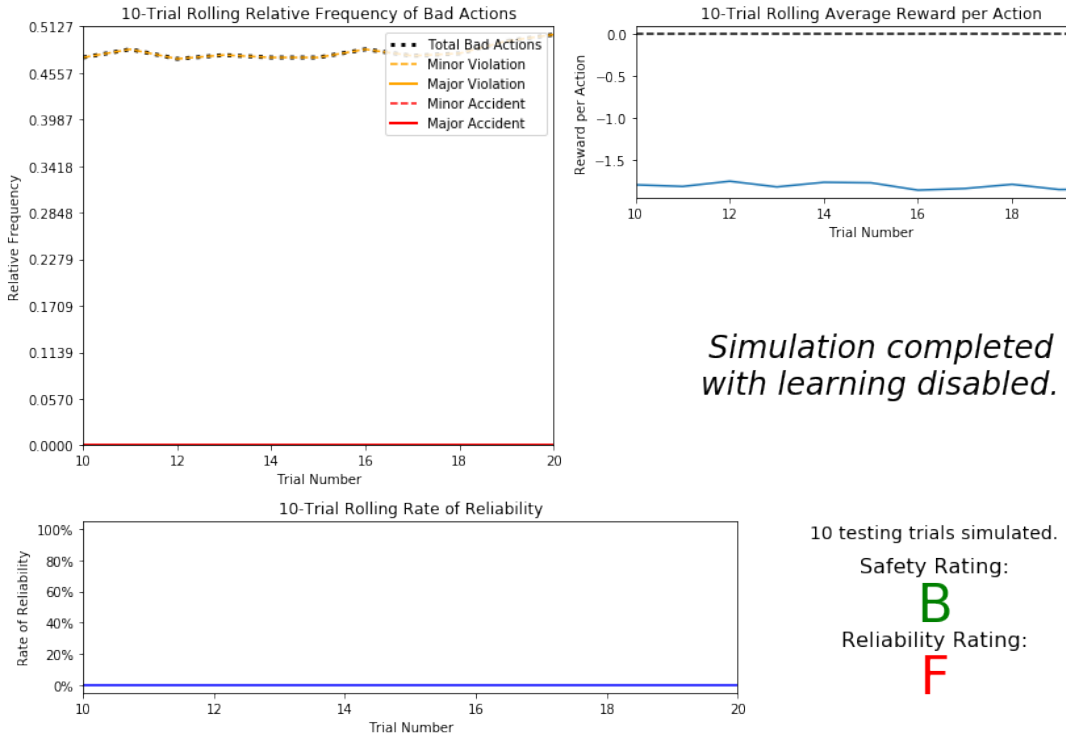
2.2.1 Basic Agent Simulation Results

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



2.2.2 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?*

Answer: - When drving agent makes bad decisions in other words takes wrong actions, it gets negative rewards. The worst the decision the higher negative rewards to it. As we can see in the above visualization of our initial simulation, out of 20 trails that were conducted, in all last 10 trails it got negative rewards for every trail, all `averaging to -1.8`. - Even though its receiving negative rewards, we don't see it making any major or minor accidents. But yes we do see that it has commited to some minor violations at the frequency rate ranging from `0.47 to 0.50 (approx)`. Commiting to Minor violations means failing to move on green light. Since we have already discussed that the cab isn't been taught yet to drive, meaning its not the smartcab yet, so commiting to minor violation makes sense. - If the agent is driving randomly then this means that it has no destination to go to. There's no target for the agent to achieve. Reliability means your agent is being tested whenther it reaches its destination in time or not. Here destination being nothing, testing for reliability woudn't make sense. And as we can see it has 0% reliability score in last 10 trials which falls in our`F` grade. - As we can see, as the number of trials increases, we dont see much of significant change in terms of bad actions, it ranges between `0.47 to 0.50 (approx)`, reliability on the other hand is being constant to 0. And the average reward for bad action they take is `-1.8` for all trails. So increasing the trail number here woudn't help. - After seeing all this I woudn't consider this safe or reliable, in fact as I said earlier this cab is not been taught to drive, its just at the idle state, being in idle state gives you good safety scores (Relative Freq.) when the signal is red but the score drops when the signal is at green since it need to move hence giving us B grade in terms of safety.

2.3 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!

2.3.1 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.

2.3.2 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.

Answer:

Let's see which features are relevant to us that will make our smartcab safety and efficient -

- The most important feature here to consider would be the `inputs that we get from the sensors of the smartcab`. It takes 4 inputs as we can see above, `Light` sensor is useful to determine the color of the traffic lights which is helpful because we want the smartcab to hault and go when it should. In other words, the color of the `light` which is our state should trigger right action in smartcab in order to avoid traffic violation which would avoid accidents which would in turn determine our safety.

- Now once our smarcab is trained for signal regulation, we should then program it to take safe actions when other cars are present nearby. For example avoiding crashing with other cars and hault when other cars are passing by. All this will be taken care of by other 2 sensors `left and oncoming`. This would help us in getting better safety grades.

  - `input[left]` = If 'left' has a car that intends to go straight, then the car should not take a right turn on a red light, as this would cause an accident. However, if the car on the 'left' intends to go right or left, the agent is also safe to take a right.
  - `input[oncoming]` = 'Oncoming' is important when our agent wants to turn left, as when the light is green it still needs to wait.

- `waypoint` sensor will help us to reach our destination by giving direction depending on where we are. This sensor will help us in getting better at efficiency since this would help us ensure that the smartcab reaches the destination in the most efficient manner possible.

- `deadline` will not be included in our features because it does not help our agent to learn how to operate safely in the environment. Adding deadline to our features would add another dimension in our Q-matrix, causing it take a lot of time in training and it would find hard time learning proper actions.

- `input[right]` doesn't seem to have any importance here unless the car would be able to take a U-turn within an intersection.

2.3.3 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!

2.3.4 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!

Answer: Let's create a table showing types & number of states each feature has:

Features	States	No. of States
inputs-light	Red, Green	2
inputs-left	None, left, right, forward	4
inputs-oncoming	None, left, right, forward	4
waypoint	left, right, forward	3

Now in order to get the size of the state space, we multiply all the number of possible states. `2 x 4 x 4 x 3 = 96`. I woudn't need much training trials here as it doesn't have much states.

2.3.5 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!

2.4 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.