Northeastern University

Artificial Intelligence Final Project Report

Self-driving Car – SmartCab

Presented by:
Parthasarathi Mitra (NUID: 001449581)
Karan Singh (NUID: 001425775)

Prof. Lawson L.S. Wong

Report Dated 12/14/2018

Foundations of Artificial Intelligence

SmartCab – Project Report

Introduction

We are in an age now when the relevance of autonomous vehicles cannot be overstated, and the marriage of the artificial intelligence fraternity and the automobile industry is giving birth to a new and exciting era of endless possibilities. Industry leaders like Waymo, Tesla, Mercedes-Benz, Audi and others are innovating in this space at a dizzying rate. With the steady growth of commercialization and the increasing affordability of these technologies, this is no longer a niche for the enthusiasts and the early adopters.

Although most of the current forays are being made into the personal vehicle market, this trend surely predicts the incorporation of autonomous vehicles into the commercial fleets the world around.

This gave us our inspiration. In the little time that we had, we have tried to create a simplistic demonstration of an automated public transportation system, specifically cabs.

Abstract

Our project deals with the creation and training of a “smart cab” that has been programmed with the knowledge to navigate the roads of a given environment to get its passenger to their destination, safely and on time. So, the agent (i.e. the “SmartCab”) starts at a certain point and drives to its destination in the allotted time, adhering to basic traffic rules like obeying the traffic signals and being cognizant of oncoming traffic.

We have tried to build it so that it does this more and more efficiently over time, as it trains itself with the help of reinforcement learning. Our goal with this project was to apply the concepts that we have been introduced to in class and try to make our agent navigate the environment with relative ease during its simulation and the long-term goal was to have a high success rate, success being defined as the SmartCab’s effective adherence to the rules and still reaching its destination within the time set for it.

Project Details:

Environment

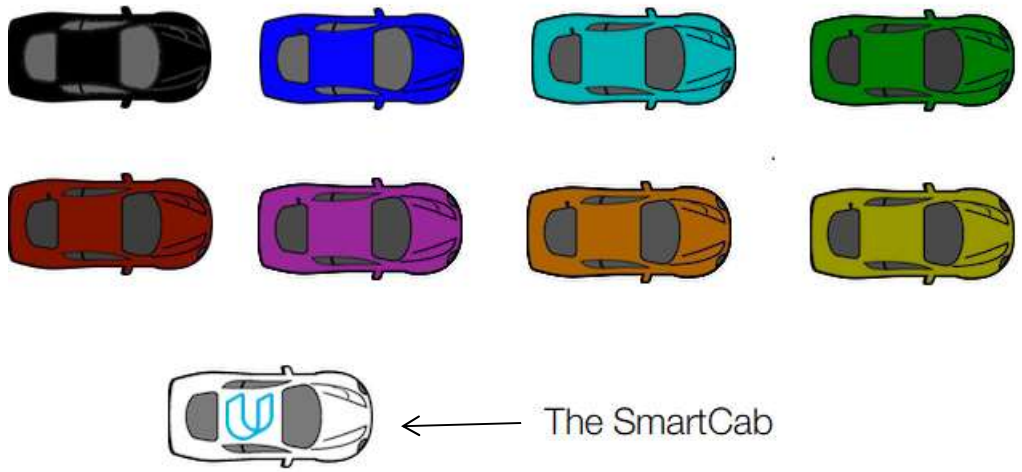
For our project, we have used the environment provided by Udemy as part of their training course. This environment portrays a grid like block of the city with uniform roads going from North to South and from East to West. This environment presents us with other vehicles on the road, which we will be taking into consideration in our model. However, there are no pedestrians to account for. Each intersection consists of a traffic light that can turn red or green allowing traffic to move vertically or horizontally through the grid.

The traffic rules enforced here are simple as well.

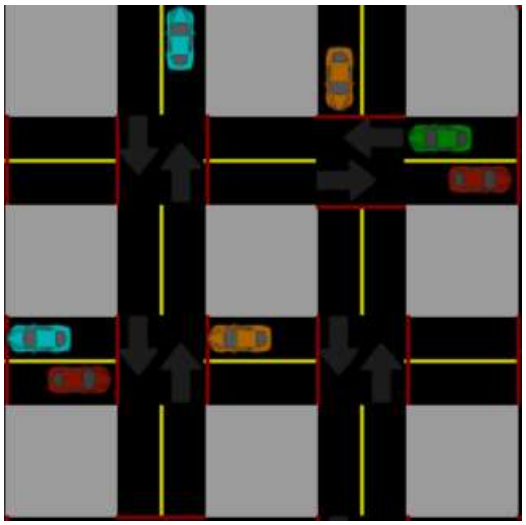
- On a red light, a right turn is permitted if there is no oncoming traffic from the left at the intersection.
- On a green light, a left turn is allowed if there is no oncoming traffic making a right turn or coming straight ahead.

The visuals:

The vehicles in the environment:



As can be seen from the image of the environment above the directions of the roads open at a certain point of time are represented by the unidirectional arrows on the road. And the red signals are signified by the red bars at the intersections.



Inputs and Outputs

In this project, the smart cab is assigned a route plan based on the passenger's pickup and drop-off points. At each intersection, the route is split into waypoints which we have used for measuring the agent's performance. At any point of time, the agent considers the state of the traffic light and the oncoming traffic which it then uses to optimize its performance.

The agent encounters certain facets of the environment at each intersection.

These states could be the:

- The status of the traffic lights (red or green).
- The oncoming traffic.
- The waypoints to reach the destination.

These states have different variants to consider as well. So, the traffic lights could be red or green, the oncoming traffic may be approaching from Forward, Left or Right or there could be None as well and lastly, the waypoints could take either of three values of Forward, Left or Right.

This gives us a total of 2 x 4 x 3 = 24 states per position, which the Q learning algorithm uses to inform the agent about the merit of its actions.

Rewards and Goals

The smart cab agent will receive positive or negative rewards based on the action it has taken. The smart cab will receive small positive rewards when it chooses a “good” action and a varying amount of negative reward based on the severity of the traffic violation that it makes. Based on the nature of the rewards the Q learning algorithm will help train the agent in choosing the optimal policy of navigating the roads while obeying traffic rules, avoiding collisions and reaching the destinations on time.

Methods/Algorithms Used

For our project, we have used reinforcement learning to inform the agent in this environment. After getting acquainted with the environment, we created a basic driving agent, which took random actions at each intersection regardless of the penalties or rewards it was receiving. Although this was a naïve approach, it allowed us to ascertain that the agent was functioning within the environment and it helped us to better understand the penalties and the rewards set by the environment for each event.

With the different states identified, we implemented a Q-Learning algorithm for the SmartCab that helped guide the agent’s actions. Initially we started with randomly chosen values for alpha (leaning rate) and epsilon (exploration factor). We tried a few different sets of values. We tried with alpha as 0.5 and a linear degrading function for epsilon as the trials progressed. We also tried with alpha as 0.4 and epsilon as constant value of 1. With all these different values we were able to reach a maximum success rate of 87% for the agent during the simulations.

Now that the agent had started learning, our next goal was to improve the success rate of the agent with optimal values for the learning rate and the exploration. This was a long process as we had to go through many different functions to decay the exploration factor before we reached a model that we were satisfied with. The final values, that we settled on, were an alpha of 0.5 and an

epsilon which varied during the trials using a cosine function on the number of trials and an optimized factor of 0.05. We did this to provide a healthy mix in the trials (learning stage) between exploration and exploitation. Although our results with these factors were not extremely consistent throughout the multiple simulations that we ran, we were always able to get a success rate of 95% or higher to a few simulations even reaching the 100% success rate mark.

Results

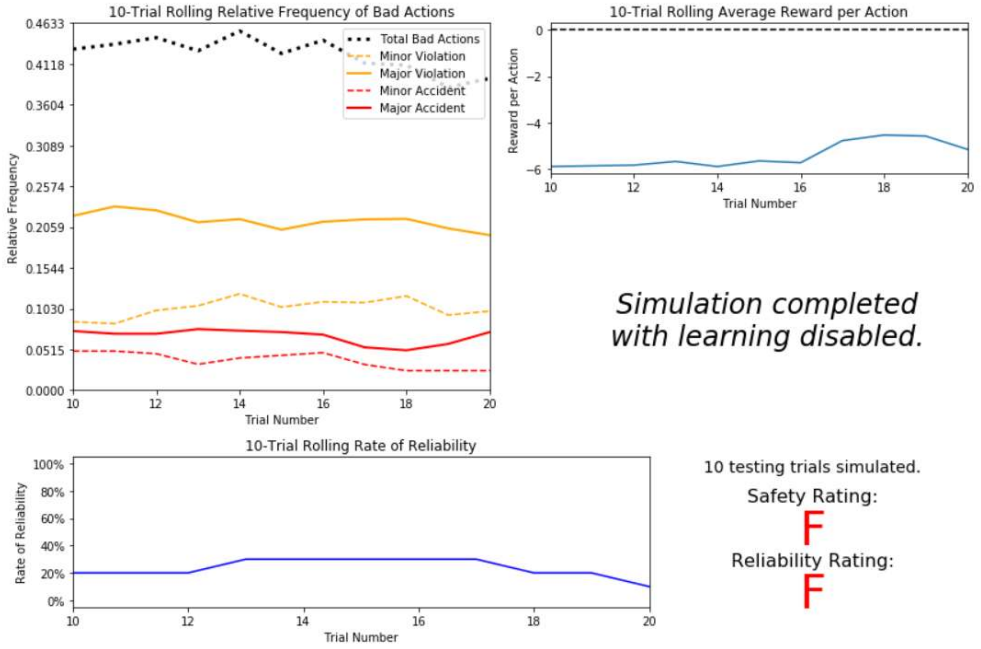
During the trials we chose to save logs for our trials and simulations which look something like this.

trial	testing	parameters	initial_dea	final_dead	net_rewar	actions	success
1	FALSE	{'a': 0.5, 'e': 0.9987502603949663}	20	0	-80.0075	{0: 13, 1: 1, 2: 5, 3: 0, 4: 1}	0
2	FALSE	{'a': 0.5, 'e': 0.9950041652780258}	20	0	-86.5964	{0: 14, 1: 3, 2: 1, 3: 0, 4: 2}	0
3	FALSE	{'a': 0.5, 'e': 0.9887710779360422}	20	0	-90.5589	{0: 13, 1: 3, 2: 1, 3: 2, 4: 1}	0
4	FALSE	{'a': 0.5, 'e': 0.9800665778412416}	20	0	-94.7867	{0: 12, 1: 2, 2: 4, 3: 1, 4: 1}	0
5	FALSE	{'a': 0.5, 'e': 0.9689124217106447}	30	0	-173.521	{0: 16, 1: 3, 2: 8, 3: 1, 4: 2}	0
6	FALSE	{'a': 0.5, 'e': 0.955336489125606}	25	0	-76.033	{0: 17, 1: 2, 2: 5, 3: 0, 4: 1}	0
7	FALSE	{'a': 0.5, 'e': 0.9393727128473789}	30	0	-184.063	{0: 19, 1: 3, 2: 3, 3: 2, 4: 3}	0
8	FALSE	{'a': 0.5, 'e': 0.9210609940028851}	20	6	-47.8295	{0: 9, 1: 0, 2: 4, 3: 1, 4: 0}	1

These logs further helped us to use matplotlib and Jupyter notebook to plot out the results from the simulations to get a visual understanding of the data. Below we have provided three different results from our many tests which provide an insight into the three steps of our implementation. The first one is a no-learning model where the agent is just choosing random actions, the second one is the default-learning model where we used the arbitrary values to get our agent to start learning and the final one is the improved learning model where we used the optimized values to boost the performance of the agent.

The “No-Learning” Agent:

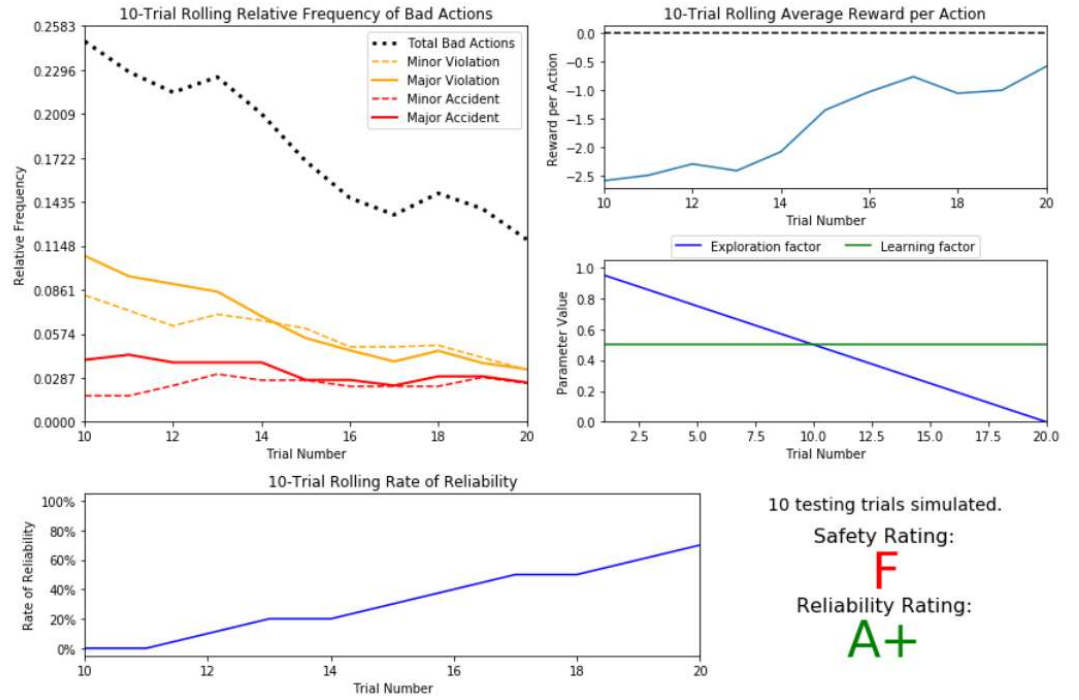
```
vs.plot_trials('sim_no-learning.csv')
```



The above is the graphical representation of our agent in a no learning state taken from Jupyter notebook. As can be seen here, the agent is very unreliable and as a result the average rewards per action were exclusively negative for most of the trials that we did. Even the count of bad actions taken by the agent ranging from minor violations to major accidents were consistently high.

The “Default-Learning” Agent:

```
vs.plot_trials('sim_default-learning.csv')
```

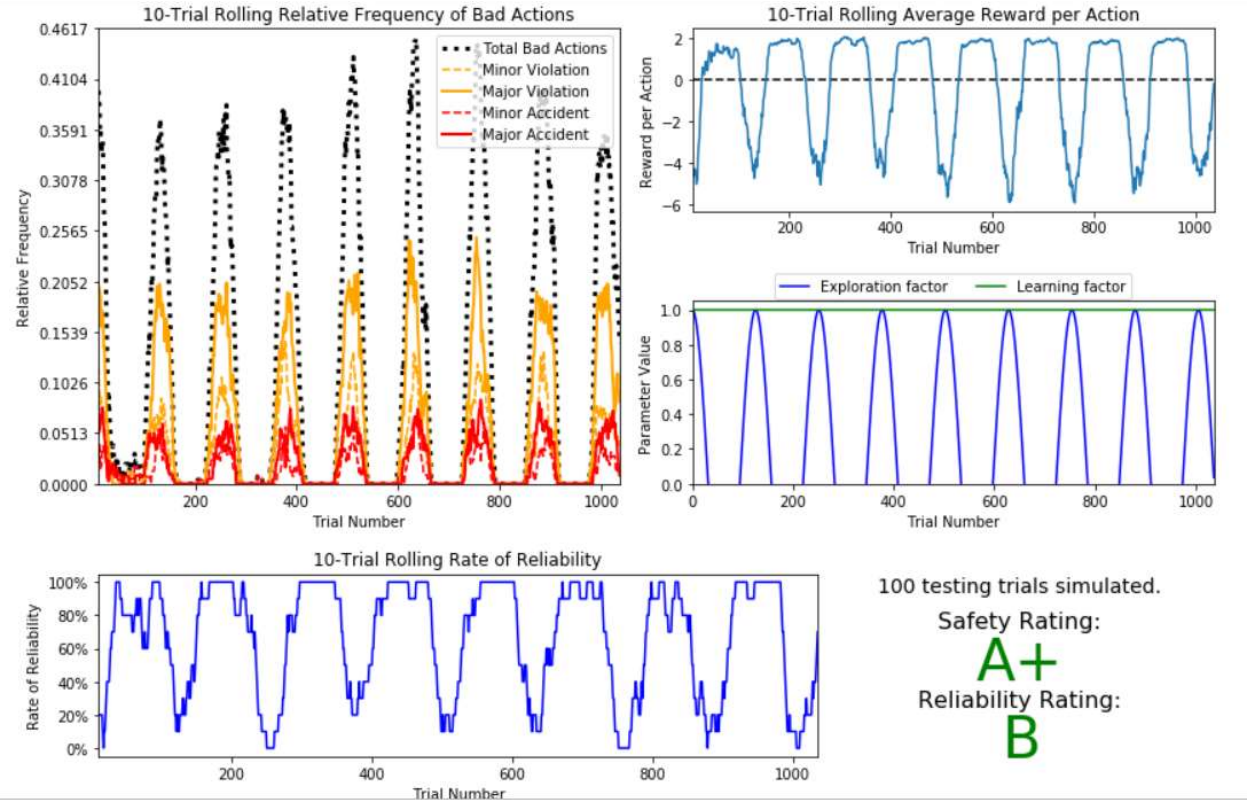


Here, as the agent starts learning, even though the learning is not opimized, we can clearly see from the results that the agent unequivocally improves over time. From the first graph we can see that over 20 trials the number of total bad actions comes down drastically. And even the average reward per action and the reliability of the agent improve as the trials run. This agent barely got the job done, but it was a decent starting point for the enhancements that we wanted to impose on it.

The “Improved-Learning” Agent:

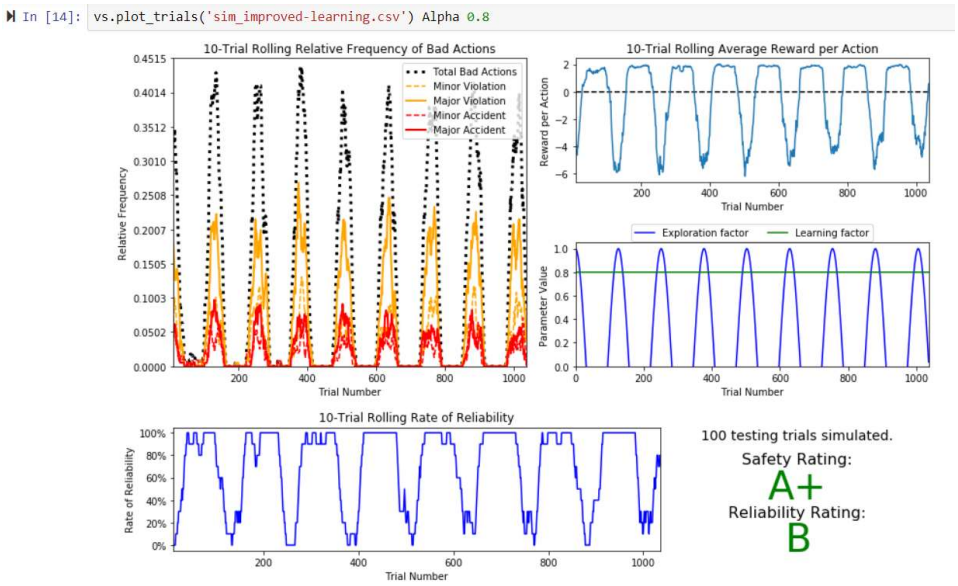
For training our agent in this phase we had to go through multiple stages of value selection for the parameters and the variation function and although some of them provide a reasonably similar result, there are others where the change is quite apparent. We have provided a few of them below.

- When alpha (learning rate) is set to 1:



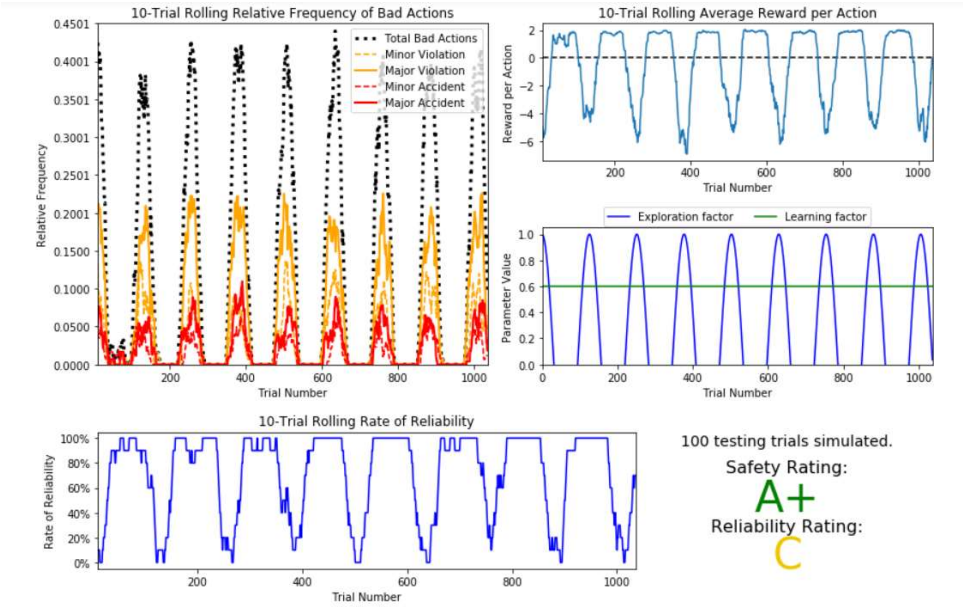
As we can see here, although the plots are much better than the no-learning and default-learning agents, the reliability of the agent is not ideal. Because as we know, when the learning rate is set at 1 the agent ignores previous knowledge and such an approach performs better in a deterministic environment than a non-deterministic one. Since we have a varying epsilon, the agent fails to use this approach to yield a consistently reliable result. Even the success rate of the agent is not ideal since out of 100 simulations the agent manages to only succeed an average of 85 times.

- When alpha is set to 0.8:



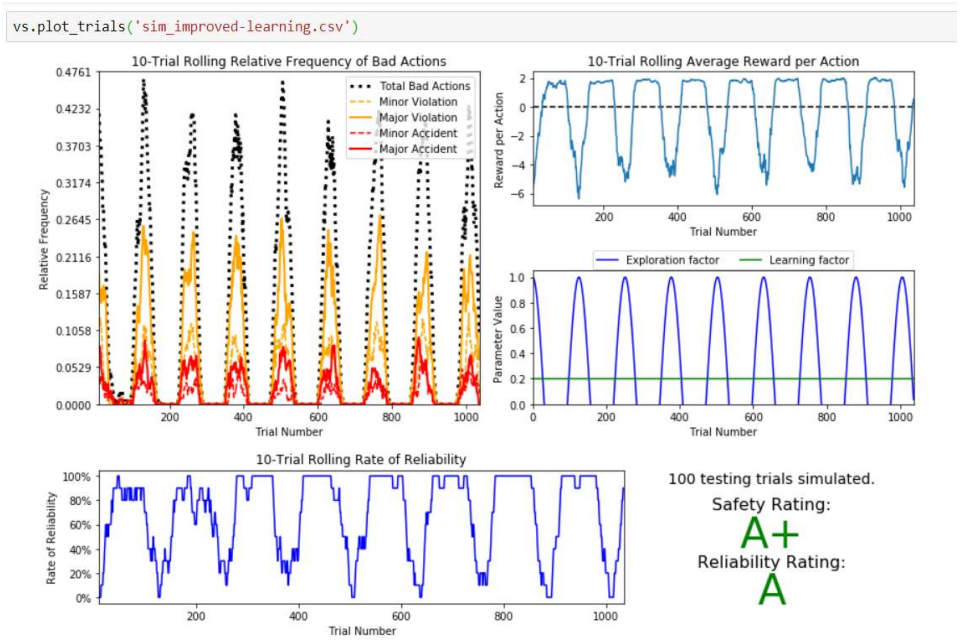
Here, although the agent manages to get the same rating holistically, the average success rate creeps up to 87%.

- When alpha is set to 0.6:



Surprisingly, with alpha set to 0.6, the agent not only slides down in the reliability rating (how often the cab reaches within the deadline to its destination) but even the success rate falls to 79%. We have not been able to clearly identify why this behavior crops up.

- When alpha is set to 0.4:



With alpha set to 0.4, we can see that the ratings for both safety and reliability are optimal. However, the average time remaining at the end of a trip is around 13 time steps, which we will see is possible to reduce even further with a more optimal value of alpha.

For our epsilon, we chose a variation function which will generate different values of epsilon for each run which will randomize the outcome to some extent while the trials are running. Obviously, during the final simulations we are setting the epsilon to 0.

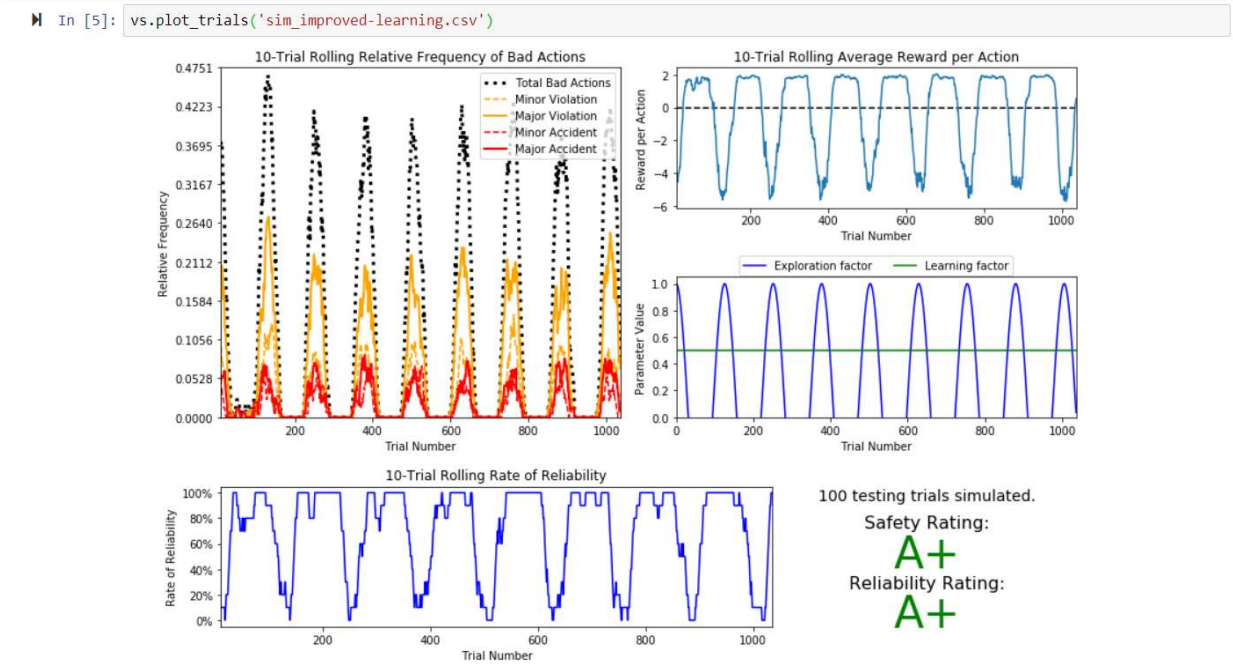
For this variation function, we went through different iterations of what that would look like. We found that a sinusoidal graph works best with our application. Before arriving at this point, we tried functions like:

- $\varepsilon = 1$ / the number of trials. We also tried multiple factors of both the numerator and denominator to varying degrees of success.
- $\varepsilon = \alpha$ * number of trials. We tried this equation as well with multiple degrees of variation.

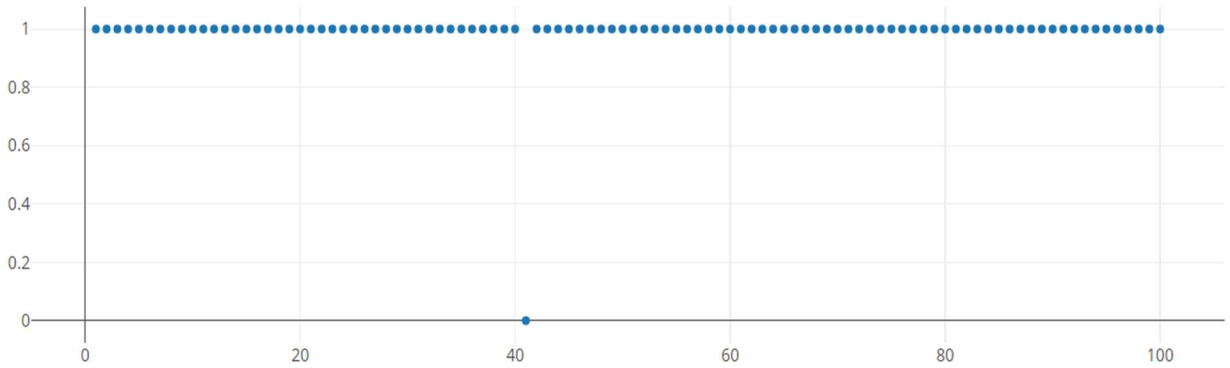
Through all these different variations of epsilon, we did not get a huge bump in performance. There were however small changes in the success rate, which reached a peak when we tried the sinusoidal function for the variation of epsilon.

The function we ended up using derives the epsilon value from the cosine of the number of trials. This varies it enough over the different trials and hence randomizes the approach providing the agent with a good experience set to draw from.

- When alpha is set to 0.5:



This is the final agent that we implemented. Here, we can clearly see that the agent has significantly reduced on the total number of bad actions and over the trials we managed to get bouts of excellent performance and sometimes flawless performance during the simulations after the trials. The exploration factor curves introduce some uncertainty into the mix with the cosine variation function and hence the nature of the trial results appear periodic. Even so at its best we can see the agent have major peaks both in reliability and reward per action(which finally have managed to reach the positive zone) and at the same time dips in the number of bad actions taken by the agent. After these trials in a few of our simulations the agent even completed the 100 simulations we ran on it flawlessly i.e. with a 100% success rate, although most of the times the success rate hovered around a 97-99% range.



- In this link we have shared a snapshot of what training the agent in the environment looks like: https://bit.ly/2CdeDdL.

Future Enhancements

Deep Q-Learning:

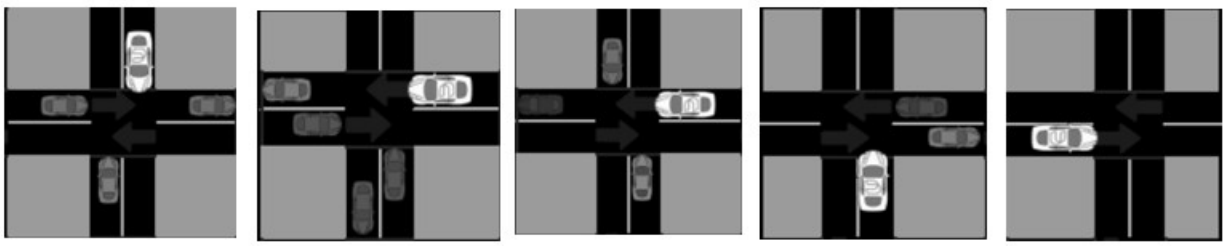
Although due to the brevity of the span of this project and a few steps which took up all our time and efforts we have not been able to fully implement and train our SmartCab in this environment with Deep Q-Learning, that was our vision for the future enhancements in this project. It is unclear whether applying Deep Q-Learning to this facile problem would yield any significant improvements (since we are already at an extremely high success rate with our model), this new direction would have definitely given us a great understanding of how Deep Q-Learning compares to the traditional Q-Learning algorithm. However, had we been able to apply this algorithm to our problem, we have an idea of how we would have initially approached it.

Deep Q-Learning is a significant step-up when it comes to larger state-space environments since it becomes quite cumbersome to construct a Q-table using the conventional approach in such a setting.

In this implementation, we would be using a neural network to estimate Q values for different actions at each step and decide the appropriate action.

The input to this neural network would be frames of the states. Here we can cut down the processing with a bit of pre-processing. The states can be simplified before feeding them to the neural network. We can use grayscaling for this since color is not a valuable information in our problem. We can also divide the entire environment into blocks for each intersection the agent is at. As long as the directions and the bars at the intersection are indentifiable, the neural should, in theory, be able to recognize the state in play.

The inputs might look something like this.



Note: There might be need to process these images a bit more before feeding them to the neural network to highlight some features like the arrows and the red bar (traffic signal).

Now once these images are fed to the network, they will be processed by the convolution layers for spatial relationships. We should not need to use more than 2 or 3 convolution layers, since it has been seen that the gain in information beyond that number of convolution layers, for most problems, is not that significant. These layers can use any of the established activation functions like ELU, ReLU (although we have yet to identify any of the different activation functions will have any major advantages over the others in our problem set).

Each covolution layer with an activation function and an output layer produces a Q value estimation for a given state.

So we go through the following steps to inform our agent:

- We sample the environment and perform actions on it and store the derived rewards and states in memory.
- We select random entries from the results obtained above and learn from it using a gradient descent update step.

For Q-Learning we used the Bellman equation to derive the Q values for the states.

$$NewQ(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma max\, Q'(s',a') - Q(s,a)]$$

New Q value for that state and that action

Current Q value

Learning Rate

Reward for taking that action at that state

Discount rate

Maximum expected future reward **given the new s' and all possible actions at that new state**

In contrast, with Deep Q-Learning, we want to update the weights associated with our neural network to reduce the error.

$$\Delta w = \alpha[(R + \gamma\, max_a\, \hat{Q}(s',a,w)) - \hat{Q}(s,a,w)]\, \nabla_w \hat{Q}(s,a,w)$$

Change in weights

learning rate

Maximum possible Qvalue for the next_state (= Q_target)

Current predicted Q-val

TD Error

Gradient of our current predicted Q-value

Our understanding of this process is still largely from a theoretical standpoint. However, we are confident that some variation of this approach should yield us a reasonable implementation of Deep Q-Learning for our particular application.

Difficulties and Conclusion

The difficulties we encountered during this project were mostly in choosing the value of the decay function for the exploration factor and the learning factor, although there was a significant amount of time spent in setting up the tools and environment and getting ourselves acquainted with them. Although we are satisfied with our results, there are a few improvements that we would have wanted to make that we could not get to due to certain steps along the way taking up too much of our time. We had decided that once our Q-learning algorithm gave us a good success rate, we would move on to implementing a Deep Q-learning algorithm. However, getting the Q-learning algorithm to an acceptable stage took unexpectedly long and we were unable to spend time developing the Deep Q-learning approach.

If we had more time, we would have liked to compare the results of Q-learning and Deep Q-learning algorithms for this agent which would have given us a deeper understanding of the inner workings of each of the metrics in both approaches. Also, if we had a more robust environment to work with, which more closely resembles real life scenarios like dealing with pedestrians, parking lots, ongoing road repair and different types of roads, we could had a lot of freedom to experiment with the reliability of the agent in a practical setting.