UDACITY MACHINE LEARNING NANODEGREE

PROJECT 4: Train a Smartcab to Drive

By Stephen Fox

Submitted September 2016

Overview

The objective of this project was to apply reinforcement learning techniques to train a self-driving smartcab to efficiently reach its destination within the allotted time. Udacity provided four python files for this purpose:

- environment.py: creates the smartcab environment, including dummy agents (i.e. traffic)
- planner.py: creates a high-level planner for the cab to follow to reach the destination
- simulator.py: creates the simulation and graphical user interface
- agent.py: creates the forum for smartcab reinforcement learning

In this project, only the agent.py file was modified. The initial modification was to get the smartcab to move around randomly within the environment.

After random movement was achieved, the next step was to implement the Q-Learning algorithm, enabling the smartcab to learn how to navigate its environment to avoid accidents, avoid breaking traffic laws and successfully reach its destination within the allotted time. To implement reinforcement learning, a set of possible states was defined for the smartcab. From each state, the smartcab could take four possible actions, namely go left, go right, go forward or do nothing. In order to learn, erroneous actions (e.g. those actions causing an accident or breaking a traffic rule) incurred a penalty (in the form of a negative reward) whereas correct actions earned a reward. Reaching the destination before running out of time earned a much larger reward. Once the state, action and reward space was defined for the smartcab, it was possible to apply the Q-Learning algorithm to let the smartcab learn. As part of this process, the learning variables (alpha, gamma, and epsilon) were tuned, to find values that gave a good performance.

Implement a Basic Driving Agent

The first task was to get the smartcab moving around the environment, without any regard to an optimal driving strategy. This was achieved by setting the ‘action’ variable to a random choice of the four possible actions (none, left, right or forward). After adding the code, I observed that with random actions and an infinite time horizon, the smartcab inevitably makes it to the destination.

However, in the process it often makes illogical moves that take it further from the destination and also sometimes break traffic rules, and it loses points because of these moves.

Inform the Driving Agent

The first major step in implementing reinforcement learning was to define a set of states that are appropriate for modeling the smartcab and environment. The key sources of state variables are the current inputs at the intersection (from environment.py), the next waypoint for the smartcab to take to reach the final destination (from planner.py) and the amount of time remaining before reaching the trip deadline (from environment.py). One challenge when implementing reinforcement learning is to balance the number of states appropriately. If you choose a relatively large number of possible states, learning will take a great deal of time, since the agent in theory needs to visit all states an infinite number of times. If you choose a relatively small number of states, you might not be properly modeling and capturing the complexities that exist in the environment.

I chose to ignore the trip deadline, for the purpose of defining states. My rationale for doing so is as follows: the optimal move for the smartcab should not be a function of the time remaining until the deadline is reached. I did not want to design a smartcab that would take major risks (e.g. running traffic lights or making illegal turns) simply to meet a deadline.

My states were therefore defined by the input and next waypoint variables. Each of the input and next waypoint variables can take on the following values (with the number in brackets showing the possible number of values a given variable can take):

- Light = red or green (2)
- Oncoming = None, right, left, or forward (4)
- Right = None, right, left, or forward (4)
- Left = None, right, left, or forward (4)
- Next waypoint = right, left, or forward (3)

If one considers every single possible combination of these five variables, there are 384 possible states (2 x 4 x 4 x 4 x 3). There are ways to reduce the number of states. For example, one could argue that whether there is a car to the right of the cab (i.e. the ‘Right’ input variable) is irrelevant in all instances, under United States traffic rules. For example, if the light is green, one can proceed regardless of whether a car is present or not to the right and if the light is red, one should do nothing or possibly turn right, neither of which would be impacted by the presence or absence of a car to the right. By eliminating the ‘Right’ input variable, the number of states would drop to a more manageable 96 possibilities. However, I opted to not drop the ‘Right’ input, since I wanted to maintain the agent’s flexibility. For example, if we wanted to utilize the same learning algorithm in a country where one drives on the left hand side of the road (e.g. England), then the ‘Right’ input would be critical, since it behaves much like the ‘Left’ input in the United States.

One could take this state reduction logic much further and use the rules of the road to reduce the number of possible states to only nine (see Figure 1 – the nine states are boxed). For example, one could argue that if the light is green and the next waypoint is forward, then the smartcab shouldn’t care about any other inputs, since it has the right of way and can legally proceed regardless of whether there are cars to the left, right or oncoming. However, taking this approach would go against the spirit of the project, since the smartcab should learn the logic and rules of the road, rather than having them explicitly built into the states.

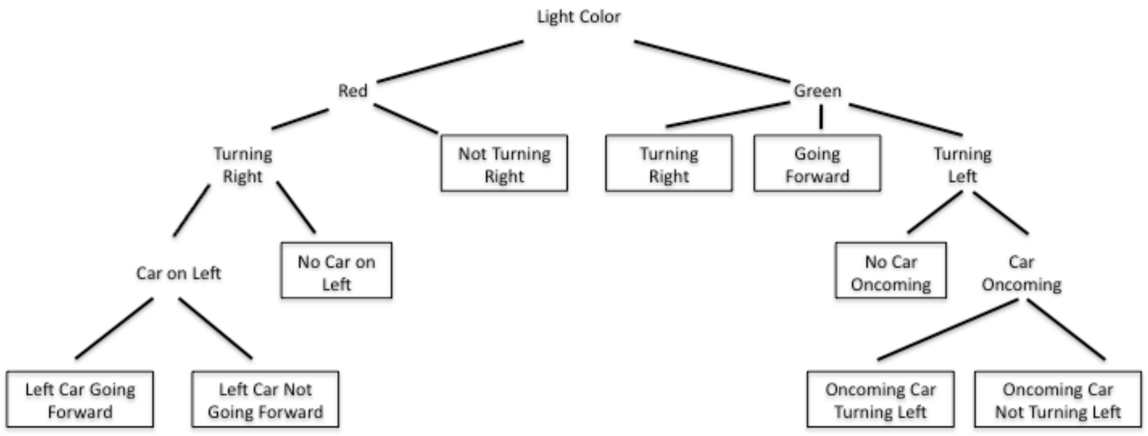

Figure 1: Decision Tree Demonstrating Nine Possible Smartcab States

Therefore, I opted to keep all 384 possible states. Although this number seems quite large relative to the number of simulation steps in which the agent has to learn (a few thousand), I believe a large majority of these possible states will never be experienced and hence the agent will get to experience the main states several times and learn appropriately.

Implement a Q-Learning Driving Agent

In order to implement a Q-learning driving agent, the following formula was encoded into the agent.py file:

$$Q(s,a) = (1 - alpha) * Q(s,a) + alpha * (reward + gamma * Q(s',a'))$$

where s, a, s’ and a’ are the current state, current action, future state and future action for the agent. The agent was encoded to choose the action with the maximum Q-value, for any given state. The Q-values were all initialized at zero, to ensure there were no initial biases built into the agent’s decision. In the case of a tie in Q-values, the agent would randomly choose an action and then update the Q-values accordingly.

For the baseline scenario, I used alpha and gamma values of 0.1. Under these conditions, the agent begins to learn and takes fewer illogical steps than when random actions were always taken. The agent also reaches the destination within the allotted time more frequently. This behavior is occurring because with each step, the agent updates the Q-values and therefore makes slightly more informed decisions on the next step, reducing the randomness and increasing the likelihood of success.

Performance metrics can be used to quantify the difference between the Q-Learning agent and the random agent. I generated the following performance metrics in this project:

- Number of successful trips
- Number of wrong moves
- Fraction of deadline remaining
- Cumulative reward earned
- Last failed trip
- Number of failed trips in the last 10 trips

The ‘number of successful trips’ measures how often the agent successfully reached the destination before running out of time for any given simulation of a certain number of trials (typically 100 trials were used). The ‘number of wrong moves’ measures how often the agent incurred a penalty (negative reward) for a given move. This could be the result of either an invalid move (e.g. breaking a traffic rule) or an incorrect move (e.g. not moving in the correct direction towards the destination). Note that remaining stationary (i.e. a move of ‘none’) does not count as a wrong move under this interpretation, even though remaining stationary might not be the correct strategy, since ‘none’ always generates a reward of zero. The ‘fraction of deadline remaining’ is a measure of how quickly the agent reaches the destination. For example, if a given trip has a deadline of 50 and the agent reaches the destination in 20 moves, then the ‘fraction of deadline remaining’ would be equal to 0.60 ((50 – 20) / 50). The ‘cumulative reward earned’ is the sum total of the reward over any given simulation of a certain number of trials. A higher cumulative reward means the agent is making fewer mistakes and reaching the destination more often, all else equal. The ‘last failed trip’ measures the last time in any given simulation that the agent failed to reach the destination. For example, on a simulation with 100 trials, if the agent failed to reach the destination on trials 1-5, 12 and 45, then the ‘last failed trip’ would be equal to 45. All else equal, an agent that is learning from its mistakes will stop failing sooner than an agent that is not learning. Finally, the ‘number of failed trips in the last 10 trips’ counts how many of the final 10 trips ended with the agent failing to reach the destination. For example, in a simulation of 100 trips, if the agent failed on trips 93 and 95 but passed on the balance of the last 10 trips, then this performance metric would have a value of 2. This performance metric is similar to the 'last failed trip' metric, but is not as prone to random unluckiness occurring on the very last trip and provides a good measure of how the agent would perform under a final assessment, once learning has run for many trips.

Now that the performance metrics used have been defined, lets compare the Q-Learning agent's performance versus a random agent (Table 1). In all cases henceforth, each set of parameters was simulated 10 times (with 100 trips per simulation) and the results shown are the average and the standard deviation (shown as ±) of those 10 simulations:

Table 1: Basic Q-Learning Agent vs. Random Agent (100 Trips; 10 simulations each)

Agent	# Successful Trips	# Wrong Moves	Avg. fraction of deadline remaining	Cumulative Reward	Last Failed Trip	# Failed Trips on last 10 trips
Random (alpha = 0; gamma = 0)	21.2 ± 5.0	1,587 ± 62	0.103 ± 0.038	7 ± 102	99.9 ± 0.3	7.2 ± 1.4
Q-Learning (alpha = 0.1; gamma = 0.1)	65.3 ± 23.9	84 ± 12	0.358 ± 0.160	1,665 ± 393	72.1 ± 43.6	3.4 ± 2.6

The basic Q-Learning agent behaves very differently than the random agent. Instead of moving in random, often illogical ways, the Q-Learning agent begins to exhibit reasonable behavior. This change is occurring because the Q-Table is being updated based on the outcome of each move, and thus is biasing the agent's movements towards moves that earn rewards.

The Q-Learning agent significantly outperforms the random agent in every performance metric, as shown in Table 1. It completes approximately 3 times as many trips, makes far fewer wrong moves and generates a much greater cumulative reward. However, there is clearly an opportunity for optimization, since the Q-Learning agent still failed on average to complete nearly 35% of the trips and wasn't very good even by the last 10 trips, with 3.4 of them still failing on average.

It is worth noting that the variance between simulations was significant (as seen by the relatively large standard deviation values) for the Q-Learning agent. For each of the 10 simulations, the Q-Learning agent either succeeded at or near 100% of trips or it failed on at least 40% of the trips and hence the average of 65% success has a large standard deviation (23.9%). This suggests that local minima are problematic for this learning environment and the agent often gets stuck in unproductive learning ruts, as will be addressed during the portion on epsilon greedy learning.

Improve the Q-Learning Driving Agent

In order to improve on the basic Q-Learning agent just presented, several approaches were utilized:

1. Tune the alpha and gamma values
2. Implement ‘epsilon greedy’ learning
3. Add a time decay to epsilon

Each of these three approaches will now be discussed.

Tune the Alpha and Gamma Parameters

The learning rate (alpha) and discount factor (gamma) are critical parameters in reinforcement learning. If alpha is too small, learning takes too long, whereas if alpha is too large, the agent will put most of the emphasis on the most recent action, which might not be desirable. If gamma is too small, too much weight is placed on only the current reward whereas if gamma is too large, too much weight is placed on the future and the present is deemphasized, which can lead to propagation of errors and instabilities in the simulation.

As a first step, various values of alpha were tested while keeping gamma fixed (at 0.1). The performance results of this tuning exercise are presented in Table 2:

Table 2: Alpha Parameter Tuning (Gamma fixed at 0.1; 100 Trips; 10 simulations each)

Alpha Value	# Successful Trips	# Wrong Moves	Avg. fraction of deadline remaining	Cumulative Reward	Last Failed Trip	# Failed Trips on last 10 trips
0*	17.8 ± 4.2	1,570 ± 72	0.0842 ± 0.0228	(8) ± 95	99.8 ± 0.4	7.2 ± 1.0
0.01	76.1 ± 23.5	82 ± 13	0.412 ± 0.149	1,856 ± 402	80.3 ± 30.6	2.7 ± 2.8
0.1	65.3 ± 23.9	84 ± 12	0.358 ± 0.160	1,665 ± 393	72.1 ± 43.6	3.4 ± 2.6
0.2	62.9 ± 19.1	89 ± 11	0.335 ± 0.130	1,663 ± 315	86.2 ± 28.9	3.0 ± 2.2
0.5	71.1 ± 23.0	82 ± 19	0.407 ± 0.142	1,777 ± 406	84.2 ± 29.5	2.4 ± 2.0
1.0	56.1 ± 16.4	96 ± 12	0.297 ± 0.093	1,489 ± 283	88.7 ± 31.2	3.1 ± 2.0

*Statistically different from the base case at the 0.05 significance level

Given the large variance in the results, a one-tailed T-Test was used to determine whether any of the results in Table 2 were statistically different from the base case (alpha = 0.1), based on the number of successful trips. It was determined that only the scenario where alpha was set to zero was statistically different (at the 0.05 significance level) from the base case. Thus, one can conclude that some value of alpha greater than zero is required for learning but that the actual choice of value does not appear critically important in this environment. For the purposes of moving forward, I selected the value with the most favorable mean for the fewest number of failures in the last 10 trips, which happened to be the alpha value of 0.5.

Next, various values of gamma were tested while keeping alpha fixed (at 0.5). The performance results of this tuning exercise are presented in Table 3:

Table 3: Gamma Parameter Tuning (Alpha fixed at 0.5; 100 Trips; 10 simulations each)

Gamma Value	# Successful Trips	# Wrong Moves	Avg. fraction of deadline remaining	Cumulative Reward	Last Failed Trip	# Failed Trips on last 10 trips
0	61.8 ± 19.7	93 ± 11	0.345 ± 0.141	1,607 ± 328	85.4 ± 30.7	3.5 ± 2.4
0.01	70.5 ± 24.1	82 ± 13	0.399 ± 0.151	1,729 ± 412	77.1 ± 32.7	2.5 ± 2.4
0.05	85.2 ± 22.4	80 ± 16	0.481 ± 0.129	1,990 ± 399	60.8 ± 41.7	1.5 ± 2.3
0.1	71.1 ± 23.0	82 ± 19	0.407 ± 0.142	1,777 ± 406	84.2 ± 29.5	2.4 ± 2.0
0.2	66.7 ± 21.2	87 ± 7	0.364 ± 0.140	1,700 ± 349	89.3 ± 17.9	2.9 ± 2.1
0.5	65.4 ± 21.6	143 ± 35	0.342 ± 0.123	1,801 ± 350	97.0 ± 6.0	2.9 ± 2.0

Given the large variance in the results, the one-tailed T-Test was used to determine whether any of the results in Table 3 were statistically different from the base case (gamma = 0.1), based on the number of successful trips. It was determined that none of the results were statistically different from the base case. Therefore, for this particular system, the actual value of gamma does not appear to be very important. The averages appear to be better for gamma in the 0.01 to 0.10 range, but again, the results cannot be said to be statistically conclusive at the 0.05 significance level.

For the purposes of moving forward, I selected the value with the most favorable mean for the fewest number of failures in the last 10 trips, which happened to be the gamma value of 0.05. This value also had the most favorable mean value for the other five performance metrics presented in Table 3.

Implement Epsilon Greedy Learning

The previously discussed exercise of tuning alpha and gamma demonstrated a very important point about this particular learning environment: given the large variance in simulation results for the same values of alpha and gamma, this environment is prone to local minima. Depending on the initial random movements made by the smartcab, the learning agent is prone to getting stuck in unproductive regions, where learning no longer progresses. Epsilon greedy learning is an excellent technique for forcing the agent out of the local minima, by periodically making random moves, with a probability equal to epsilon. Those random moves force the agent to explore regions of the learning environment that it might otherwise never move to on its own.

In order to determine whether epsilon greedy learning would aid learning in this environment, I tested various epsilon values, with the results shown in Table 4:

Table 4: Epsilon Tuning

(Alpha & Gamma fixed at 0.5 & 0.05 respectively; 100 Trips; 10 simulations each)

Epsilon Value	# Successful Trips	# Wrong Moves	Avg. fraction of deadline remaining	Cumulative Reward	Last Failed Trip	# Failed Trips on last 10 trips
0	85.2 ± 22.4	80 ± 16	0.481 ± 0.129	1,990 ± 399	60.8 ± 41.7	1.5 ± 2.3
0.001	63.9 ± 20.4	84 ± 10	0.352 ± 0.121	1,630 ± 366	89.0 ± 17.5	2.4 ± 2.2
0.005	87.6 ± 19.3	80 ± 10	0.490 ± 0.124	2,045 ± 353	66.3 ± 29.3	1.1 ± 2.1
0.01*	94.3 ± 3.8	84 ± 10	0.526 ± 0.035	2,175 ± 60	65.3 ± 28.5	0.1 ± 0.3
0.02*	95.4 ± 2.8	93 ± 8	0.535 ± 0.019	2,190 ± 55	48.3 ± 29.5	0.1 ± 0.3
0.05*	95.6 ± 3.0	122 ± 11	0.532 ± 0.016	2,183 ± 45	59.9 ± 27.4	0.1 ± 0.3
0.1*	94.7 ± 2.4	171 ± 15	0.498 ± 0.032	2,218 ± 75	58.6 ± 30.4	0.1 ± 0.3
0.2	90.8 ± 2.9	276 ± 21	0.466 ± 0.026	2,149 ± 72	84.2 ± 15.1	0.6 ± 0.7
0.5	66.0 ± 4.1	720 ± 36	0.294 ± 0.033	1,709 ± 85	98.5 ± 1.6	2.9 ± 0.7

*Statistically different from the zero epsilon case at the 0.05 significance level

The results in Table 4 demonstrate that epsilon greedy learning had a strong effect on reducing the variance in the number of successful trips. With an epsilon value of zero, the mean is 85.2 trips and the standard deviation is 22.4, meaning many simulations result in a large number of failed trips, whereas with an epsilon of 0.05, the mean is 95.6 trips and the standard deviation is only 3.0, meaning that almost all simulations result in over 90% success rate and that the chance of a simulation resulting in many trips failing was close to zero.

The one-tailed T-Test was used to determine whether any of the results in Table 4 were statistically different from the base case (epsilon = 0), *based on the number of failed trips in the last ten trips*. The number of failed trips in the last 10 trips was identified as important for the purposes of judging the quality of the reinforcement learning, since if the agent has learnt how to navigate the streets, it should rarely fail during simulations 91-100. It was determined that modestly low (0.01, 0.02, 0.05, and 0.1) values of epsilon were statistically different from the base case, whereas very low (0.001, 0.005) and high (0.2 and 0.5) values were not. This intuitively makes sense: if the value is too low, there is not enough random movement to facilitate learning. If the value is too high, there are too many random failures still occurring at the end of the simulation to ensure a high chance of success on the final 10 trips.

Add Time Decay to Epsilon

One way to address the problem posed by high values of epsilon causing too many random movements towards the end of a simulation is to let epsilon decay with time. In the first few simulations, a high epsilon will promote lots of random movement and learning whereas in the last few simulations, a low epsilon will discourage random movement and therefore curtail the resulting occasional failures.

Four different types of epsilon decay were tested here, where 't' is the cumulative number of moves:

1. Epsilon = 0.9 / (1 + t / 10)
2. Epsilon = 0.05 / log (t + 2)
3. Epsilon = 0.05 for t < 1,500 and 0 for t ≥ 1,500
4. Epsilon = 0.2 for t < 500 and 0 for t ≥ 500

On a standard simulation with 100 trips, the value of t by the end of the simulation will run into the low thousands, since each trip has a deadline of 20 – 50 moves, of which many are often required to complete the trip. The first type of epsilon decay tested is a constant decay. The second type is a logarithmic decay, with a sharper drop off occurring initially, followed by a leveling out. The third and fourth decay use a constant epsilon for an initial number of simulation points and then let epsilon go to zero, thereby turning off epsilon greedy learning in a step change manner. The results for these four types of epsilon decay are compared to the best constant epsilon values (from Table 4) in Table 5.

Table 5: Epsilon Time Decay

(Alpha & Gamma fixed at 0.5 & 0.05 respectively; 100 Trips; 10 simulations each)

Epsilon Value	# Successful Trips	# Wrong Moves	Avg. fraction of deadline remaining	Cumulative Reward	Last Failed Trip	# Failed Trips on last 10 trips
0.01	94.3 ± 3.8	84 ± 10	0.526 ± 0.035	2,175 ± 60	65.3 ± 28.5	0.1 ± 0.3
0.02	95.4 ± 2.8	93 ± 8	0.535 ± 0.019	2,190 ± 55	48.3 ± 29.5	0.1 ± 0.3
0.05	95.6 ± 3.0	122 ± 11	0.532 ± 0.016	2,183 ± 45	59.9 ± 27.4	0.1 ± 0.3
0.1	94.7 ± 2.4	171 ± 15	0.498 ± 0.032	2,218 ± 75	58.6 ± 30.4	0.1 ± 0.3
0.9 / (1+t/10)	96.1 ± 2.0	91 ± 8	0.544 ± 0.023	2,196 ± 47	61.0 ± 31.1	0.1 ± 0.3
0.05 / log(t+2)	93.3 ± 6.5	79 ± 7	0.528 ± 0.048	2,136 ± 102	58.9 ± 34.9	0.1 ± 0.3
0.05 (t < 1,500)	93.8 ± 2.7	126 ± 9	0.510 ± 0.029	2,192 ± 80	63.9 ± 27.4	0.2 ± 0.4
0.2 (t < 500)	95.3 ± 1.5	130 ± 7	0.535 ± 0.017	2,196 ± 59	63.7 ± 24.8	0.0 ± 0.0

Both the constant and decaying epsilon scenarios all gave very good results in terms of successful trips (>93%). The number of failed trips in the last 10 trips was consistently low for all cases too, with a mean value less than 0.2 for all cases.

To differentiate between the different scenarios, the one-tailed T-Test was used to determine whether any of the results in Table 5 were statistically different from the base case (constant epsilon of 0.05), based on several measurements (successful trips, last failed trip, failed trips in the last 10 trips). After applying this test, it was determined that none of the scenarios presented in Table 5 are statistically different from each other at the 0.05 significance level. In other words, they are all equally effective learning models for epsilon, in this particular environment.

Given this result, I have chosen to use the model where epsilon is 0.2 for the first 500 steps and then zero thereafter. I like this model for two reasons. First, I find the step change intuitively appealing. The idea is initially to let the learner take lots of random steps to learn (i.e. an 'explore' phase), but to then turn this behavior off after a certain number of steps, to eliminate random behavior later in the simulation, when good results are more important (i.e. an 'exploit' phase).

Second, this model did not fail at all on the last 10 trips for the simulations I ran. Although this result was not statistically significant, it served as a tiebreaker.

Therefore, my final agent has the following key parameters:

- Alpha = 0.5
- Gamma = 0.05
- Epsilon = 0.2 for first 500 moves and 0 thereafter

As a final assessment of my agent, the Q-Table values were outputted for the last simulation run for the optimal settings shown above. The resulting Q-Table for this simulation is presented in Appendix A. The Q-Table has been marked up as follows: the correct action (from a rules of the road perspective) for any given state is underlined and presented in a different colored font. If the correct action has the maximum Q-Value, then it is colored in blue and underlined with a single, solid line. If the correct action has one of the maximum Q-Values (e.g. it is tied with an incorrect action), then it is colored in blue and underlined with a single, dashed line. If the correct action does not have the maximum Q-Value (i.e. there is an incorrect action that has a higher Q-Value), then it is colored in red and underlined with a double, solid line.

The first interesting observation from the Q-Table is that only 60 of the possible 384 states were visited over the 100 simulated trips. This is consistent with the expectations stated earlier in this report when defining the states for the agent, namely that only a small subset of the possible states would be visited over 100 trips.

The second interesting observation is that by and large, the agent learned the correct action for a given state. Of the 60 states visited, it learnt the correct action for 37 of them (62%). It was on the right track for 17 (28%) of them (i.e. the correct action was one of the actions with the maximum Q-Value) and only failed to learn the correct policy on 6 of the states visited (10%). It should be noted that in all but one of these six failures, the erroneous action was a fairly harmless one (i.e. the agent opted to do nothing when it should have taken an action, such as turning right on a red light). The only exception was that the agent learnt to turn right on a red light even when the oncoming car was turning left. Had the simulation been run longer and had the agent been able to experience this state more often, I am confident it would have converged on the right action for this state.

I believe my agent comes close to finding an optimal policy. I would describe an optimal policy as follows: if the agent can proceed without breaking a traffic rule or causing an accident, then the next move should be equal to ‘next_waypoint’. If the agent cannot proceed without breaking a traffic rule or causing an accident, then the next move should be ‘none’. As shown in the sample Q-Table in Appendix A, the maximum Q-values for each state correspond to the correct (from a traffic law perspective) move for 90% of the states sampled on that simulation. I would find the results more satisfying if the agent was correct for 100% of the states sampled, but given the large number of possible states (384), I believe a more correct Q-Table would only be achieved by running well more than 100 trips.

Appendix A: Sample Q-Table Values

Key Simulation Settings:

- Alpha = 0.5
- Gamma = 0.05
- Epsilon = 0.2 for first 500 moves and 0 thereafter
- 100 Trips

```
Q-TABLE:
{
(('light', 'green'), ('left', None), ('right', 'forward'), ('oncoming',
None), ('next', 'forward')): {'forward': 2.0300547854392073,
'right': 0, None: 0, 'left': -0.19828153639797524},

(('light', 'red'), ('left', 'left'), ('right', None), ('oncoming',
None), ('next', 'right')): {'forward': 0, 'right':
1.0500637194695872, None: 0, 'left': 0},

(('light', 'red'), ('left', 'right'), ('right', None), ('oncoming',
None), ('next', 'right')): {'forward': 0, 'right': 0, None: 0, 'left':
-0.5},

(('light', 'red'), ('left', None), ('right', 'left'), ('oncoming',
None), ('next', 'forward')): {'forward': -0.4737400770642993, 'right':
-0.19939349490479358, None: 0.0073456809890395216, 'left': -
0.49826077368769034},

(('light', 'green'), ('left', 'forward'), ('right', 'left'),
('oncoming', None), ('next', 'left')): {'forward': -
0.20560633399783473, 'right': -0.19861934557801741, None: 0, 'left':
0},

(('light', 'green'), ('left', None), ('right', 'right'), ('oncoming',
None), ('next', 'right')): {'forward': 0, 'right': 0, None: 0, 'left':
-0.19984765625},

(('light', 'red'), ('left', None), ('right', None), ('oncoming',
'right'), ('next', 'forward')): {'forward': 0, 'right': 0, None:
0.0025331435636226814, 'left': 0},

(('light', 'green'), ('left', None), ('right', None), ('oncoming',
'right'), ('next', 'right')): {'forward': 0, 'right': 0, None:
0.04936141969714919, 'left': 0},

(('light', 'green'), ('left', None), ('right', None), ('oncoming',
'right'), ('next', 'left')): {'forward': -0.375, 'right': -
```

```
0.19844032817258497, None: 0, 'left': 0},

(('light', 'red'), ('left', 'forward'), ('right', None), ('oncoming',
None), ('next', 'left')): {'forward': 0, 'right': -0.50796875, None: 0,
'left': 0},

(('light', 'green'), ('left', None), ('right', None), ('oncoming',
'left'), ('next', 'left')): {'forward': 0, 'right': -
0.21127606985335534, None: 0, 'left': 0},

(('light', 'green'), ('left', 'right'), ('right', None), ('oncoming',
None), ('next', 'right')): {'forward': -0.2709062023925781, 'right': 0,
None: 0, 'left': 0},

(('light', 'red'), ('left', None), ('right', None), ('oncoming',
'right'), ('next', 'right')): {'forward': 0, 'right':
1.001322390735922, None: 0, 'left': 0},

(('light', 'green'), ('left', None), ('right', 'right'), ('oncoming',
None), ('next', 'left')): {'forward': 0, 'right': 0, None:
2.020197049301996e-05, 'left': 0},

(('light', 'green'), ('left', None), ('right', None), ('oncoming',
'left'), ('next', 'right')): {'forward': -0.25, 'right':
1.5375373110988722, None: 0, 'left': 0},

(('light', 'red'), ('left', 'left'), ('right', None), ('oncoming',
None), ('next', 'forward')): {'forward': -0.49903218448837783, 'right':
0, None: 0.019654560052955104, 'left': -0.5},

(('light', 'red'), ('left', None), ('right', None), ('oncoming',
'left'), ('next', 'left')): {'forward': -0.6980121033519424, 'right':
0, None: 0, 'left': -0.755625},

(('light', 'green'), ('left', None), ('right', None), ('oncoming',
'left'), ('next', 'forward')): {'forward': 2.0633165931453927,
'right': -0.20005438979738419, None: 0, 'left': 0},

(('light', 'red'), ('left', None), ('right', 'left'), ('oncoming',
None), ('next', 'right')): {'forward': 0, 'right': 0, None:
0.05012492625034797, 'left': 0},

(('light', 'green'), ('left', None), ('right', 'right'), ('oncoming',
'left'), ('next', 'forward')): {'forward': 0, 'right': 0, None: 0.0,
'left': -0.19839064076557641},

(('light', 'red'), ('left', None), ('right', None), ('oncoming',
'left'), ('next', 'right')): {'forward': -0.75, 'right':
```

```
1.953062331882089, None: 0.0, 'left': 0},

(('light', 'green'), ('left', 'forward'), ('right', None), ('oncoming',
'forward'), ('next', 'forward')): {'forward': 0, 'right': -
0.2013917482516212, None: 0, 'left': 0},

(('light', 'red'), ('left', None), ('right', None), ('oncoming', None),
('next', 'left')): {'forward': -0.8251087795947684, 'right': -0.409375,
None: 0.06029498958081197, 'left': -0.969353515625},

(('light', 'red'), ('left', None), ('right', None), ('oncoming', None),
('next', 'right')): {'forward': -0.8317041023788165, 'right':
1.7760702723925599, None: 0, 'left': -0.8805947526091955},

(('light', 'green'), ('left', None), ('right', 'forward'), ('oncoming',
None), ('next', 'right')): {'forward': -0.25, 'right':
2.044927847853585, None: 0, 'left': 0},

(('light', 'green'), ('left', None), ('right', 'left'), ('oncoming',
None), ('next', 'right')): {'forward': 0, 'right':
1.8754517097177954, None: 0, 'left': 0},

(('light', 'red'), ('left', 'forward'), ('right', None), ('oncoming',
None), ('next', 'forward')): {'forward': -0.4991039119810112, 'right':
0, None: 0.0, 'left': -0.4486987942236463},

(('light', 'green'), ('left', 'right'), ('right', None), ('oncoming',
'left'), ('next', 'forward')): {'forward': 1.025, 'right': 0, None:
0, 'left': 0},

(('light', 'red'), ('left', None), ('right', None), ('oncoming', None),
('next', 'forward')): {'forward': -0.977088922828053, 'right': -
0.4023536015144413, None: 0.054905767749678584, 'left': -
0.8628248554507847},

(('light', 'green'), ('left', 'left'), ('right', 'left'), ('oncoming',
None), ('next', 'right')): {'forward': -0.223828125, 'right': 0, None:
0, 'left': 0},

(('light', 'green'), ('left', None), ('right', None), ('oncoming',
'forward'), ('next', 'forward')): {'forward': 1.7897553221340525,
'right': 0, None: 0, 'left': 0},

(('light', 'red'), ('left', None), ('right', 'right'), ('oncoming',
None), ('next', 'left')): {'forward': -0.4558909828743867, 'right': -
0.21122039695898703, None: 0.02588731113516951, 'left': 0},

(('light', 'red'), ('left', 'left'), ('right', None), ('oncoming',
None), ('next', 'left')): {'forward': -0.5, 'right': -
0.20591128282344204, None: -0.0026339565944155, 'left': -
0.499908592219829},

(('light', 'red'), ('left', None), ('right', None), ('oncoming',
'left'), ('next', 'forward')): {'forward': -0.5, 'right': -
0.3154436405582129, None: 0.05846933956613639, 'left': 0},

(('light', 'green'), ('left', 'forward'), ('right', None), ('oncoming',
None), ('next', 'forward')): {'forward': 2.0728193319800674,
'right': 0, None: 0, 'left': 0},

(('light', 'green'), ('left', None), ('right', None), ('oncoming',
None), ('next', 'right')): {'forward': -0.375, 'right':
2.0371937774092186, None: 0.038787129948949395, 'left': -
0.3637307922843931},

(('light', 'green'), ('left', 'left'), ('right', None), ('oncoming',
None), ('next', 'forward')): {'forward': 2.034104779458594, 'right':
-0.22378905078692388, None: 0, 'left': 0},

(('light', 'green'), ('left', 'right'), ('right', None), ('oncoming',
None), ('next', 'left')): {'forward': 0, 'right': 0, None:
0.047026368651634516, 'left': 0},

(('light', 'green'), ('left', None), ('right', 'left'), ('oncoming',
None), ('next', 'forward')): {'forward': 2.053299145775461, 'right':
-0.21189452539346193, None: 0, 'left': 0},

(('light', 'red'), ('left', 'forward'), ('right', None), ('oncoming',
None), ('next', 'right')): {'forward': 0, 'right': 0, None: 0, 'left':
-0.4613078735680167},

(('light', 'green'), ('left', None), ('right', 'right'), ('oncoming',
None), ('next', 'forward')): {'forward': 1.5262528609961614,
'right': -0.19818961839282667, None: 0.0, 'left': 0},

(('light', 'red'), ('left', None), ('right', None), ('oncoming',
'forward'), ('next', 'forward')): {'forward': 0, 'right': -
0.2499314551024O114, None: 0.08130944764424419, 'left': 0},

(('light', 'red'), ('left', None), ('right', 'left'), ('oncoming',
None), ('next', 'left')): {'forward': -0.5, 'right': -
0.21123505800467873, None: 0.0011736365876752171, 'left': -0.5125},

(('light', 'green'), ('left', None), ('right', 'forward'), ('oncoming',
'right'), ('next', 'forward')): {'forward': 0, 'right': -
0.20620375145518938, None: 0, 'left': 0},
(('light', 'red'), ('left', None), ('right', 'forward'), ('oncoming',
None), ('next', 'forward')): {'forward': -0.4984729450009738, 'right':
-0.19906251556850516, None: 0.0015404704716269627, 'left': 0},

(('light', 'green'), ('left', None), ('right', None), ('oncoming',
None), ('next', 'forward')): {'forward': 2.032632653984976, 'right':
-0.4794250584986425, None: 0.06052840956274891, 'left': -
0.41132232082366205},

(('light', 'green'), ('left', 'forward'), ('right', None), ('oncoming',
None), ('next', 'right')): {'forward': -0.2484353583579318, 'right':
1.5387873320043306, None: 0.0, 'left': -0.2012771606057016},

(('light', 'green'), ('left', 'left'), ('right', None), ('oncoming',
None), ('next', 'right')): {'forward': -0.20465076999802864, 'right':
1.5762237545422206, None: 0, 'left': -0.3238089199349856},

(('light', 'green'), ('left', 'forward'), ('right', None), ('oncoming',
None), ('next', 'left')): {'forward': -0.20594726269673097, 'right': -
0.24870518197617453, None: 0.0, 'left': 0},

(('light', 'green'), ('left', 'right'), ('right', None), ('oncoming',
None), ('next', 'forward')): {'forward': 1.0020425176236023,
'right': -0.1998867357139332, None: 0, 'left': 0},

(('light', 'green'), ('left', None), ('right', None), ('oncoming',
'forward'), ('next', 'left')): {'forward': -0.21123644463875738,
'right': -0.24766016969884722, None: 0.04721961482604997, 'left':
0},

(('light', 'green'), ('left', 'left'), ('right', 'forward'),
('oncoming', None), ('next', 'right')): {'forward': -
0.20288267333984375, 'right': 0, None: 0, 'left': -0.206234375},

(('light', 'red'), ('left', None), ('right', 'right'), ('oncoming',
None), ('next', 'forward')): {'forward': -0.4624554086698101, 'right':
-0.37382434078370913, None: 0.0020833959495554 08, 'left': -
0.7493530635766972},

(('light', 'green'), ('left', None), ('right', 'left'), ('oncoming',
None), ('next', 'left')): {'forward': -0.24947646967900874, 'right': -
0.26875, None: 0.00018585917884444438, 'left': 0},

(('light', 'red'), ('left', None), ('right', None), ('oncoming',
'forward'), ('next', 'right')): {'forward': 0, 'right':
1.889791410124018, None: 0, 'left': 0},
(('light', 'green'), ('left', None), ('right', None), ('oncoming',
'right'), ('next', 'forward')): {'forward': 1.9904762234727396,
'right': 0, None: 0, 'left': 0},

(('light', 'red'), ('left', None), ('right', None), ('oncoming',
'forward'), ('next', 'left')): {'forward': -0.45021755918953676,
'right': -0.19811974584022418, None: 0.059732728502302296, 'left':
0},

(('light', 'red'), ('left', None), ('right', 'right'), ('oncoming',
None), ('next', 'right')): {'forward': 0, 'right':
1.5384682516466175, None: 0, 'left': 0},

(('light', 'red'), ('left', 'right'), ('right', None), ('oncoming',
'left'), ('next', 'forward')): {'forward': 0, 'right': 0, None: 0,
'left': -0.44883527230585424},

(('light', 'green'), ('left', None), ('right', None), ('oncoming',
None), ('next', 'left')): {'forward': -0.375, 'right': -
0.31021308119265145, None: 0.05174041612022531, 'left':
1.9750339009876887}
}
```