A Self-driving car simulation using Q-Learning

Tushar Agarwal (agarwal.270@osu.edu), **Wenxiao Zhan** (zhan.137@osu.edu)

Introduction

Creating intelligent machines has long been a human endeavor with a rich past as well as present. Reinforcement learning is one major component of this endeavor. As the name suggests, reinforcement learning refers to a reward based learning system designed for a machine to teach it the correct behavior of acting in a specified environment. It has its roots in behaviorist psychology and can be simply thought of as the reward based training routine usually followed while teaching pet animals like dogs. Such a training requires a time-dependent modeling of the processes.

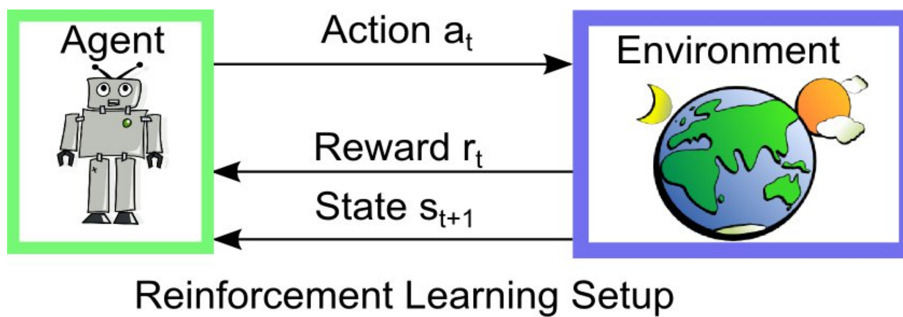


Figure 1: Reinforcement Learning Problem (Image from [1])

According to the classical and more theoretical approach , the environment is formulated as a Markov decision process (MDP) whose solution usually requires Dynamic Programming. In fact, many reinforcement learning algorithms also utilize dynamic programming techniques suitable to solve MDPs. But, the main difference between the classical methods and reinforcement learning algorithms is that the latter does not assume knowledge of an exact mathematical model of the MDP and they target large MDPs where exact methods become infeasible. Most practical applications are usually of this scale and thus, reinforcement learning has proven to be absolutely crucial given its high practical utility.
There are two key components that make reinforcement learning so powerful: the use of samples to optimize performance and the use of function approximation to deal with large environments. Some scenarios where it has been used are:

1. A model of the environment is known, but an analytic solution is not available.
2. Only a simulation model of the environment is available (the subject of simulation-based optimization).
3. The only way to collect information about the environment is to interact with it like the NASA's Mars Rover.

The first two of these problems could be considered planning problems since some form of model is available, but the last one is a genuine learning problem. However, reinforcement learning converts both planning problems to machine learning problems as well.

Theoretical Background

A reinforcement learning (RL) agent interacts with its environment in discrete time steps. At each time t, the agent receives an observation o_t, which typically includes the reward r_t. It then chooses an action a_t from the set of available actions, which is subsequently sent to the environment. The environment moves to a new state s_{t+1} and the reward r_{t+1} associated with the transition (s_t, a_t, s_{t+1}) is determined. Let the set of all possible states be referred as S and the set of all possible actions as A. The goal of a reinforcement learning agent usually is to maximize a cumulative reward.

Components

Important components of an RL agent are the policy, the value function and the model.

- The model predicts how the environment will behave next. It has 2 main sub-components:-
  The transition probability: $\mathcal{P}(s, a, s') = P(s_{t+1} = s'|s_t = s, a_t = a)$
  The Reward: $r(s, a) = E[r_{t+1}|s_t = s, a_t = a]$
- The policy π defines the agents behavior and is a mapping from state s_t to action a_t. It can be be deterministic or probabilistic.
  Deterministic policy: $a_t = \pi(s_t)$
  Probabilistic policy: $\pi(a|s) = P(a_t = a|s_t = s)$
- Value function is a prediction of the future reward. It tries to measure the goodness or badness of states $V(s_t)$ or state-action pairs $q(s_t, a_t)$ to help make a decision between available set of actions A at a given state.
  State value function for a given policy π: $V_\pi(s) = E_\pi[G_t|s_t = s]$
  State-action value function for a given policy π: $Q_\pi(s, a) = E_\pi[G_t|s_t = s, a_t = a]$

  where $G_t = r_t + \sum_{i>t} \gamma^{i-t} r_i$ is a measure of the cumulative reward. The term $\gamma \in [0, 1)$ is a discount factor used to represent the fact that the present reward is more valuable than the future reward as the future is always uncertain. This is a very practical assumption and has real life manifestations like time value of money in finances. It also allows the cumulative reward to be finite in case of infinite time horizon, i.e. when the agent takes actions forever.

Classical Methods

The classical methods to solve this problem rely on an underlying model of the interactions of the agent with the environment. The formulation is typically based on an MDP model of the environment which yield's the famous Bellman Optimality Equation.

$$V_*(s_t) = \max_{a_t}[r(s_t, a_t) + \gamma \sum_{s_{t+1} \in S} \mathcal{P}(s_t, a_t, s_{t+1}) V_*(s_{t+1})] \quad (1)$$

$$\pi_*(s_t) = \underset{a_t}{argmax}[r(s_t, a_t) + \gamma \sum_{s_{t+1} \in S} \mathcal{P}(s_t, a_t, s_{t+1}) V_*(s_{t+1})] \quad (2)$$

where $V_*(s_t)$ and $\pi_*(s_t)$ refers to the optimal value functions and optimal policy of the state s_t at time t respectively. These equations are usually solved iteratively

using dynamic programming which reduces the complexity of the solution from $O(m^n)$ to $O(mn^2)$. where m, n denote the dimensions of the state space and total no. of time-steps respectively. These methods have their roots in optimal control theory but recently, people have started referring them also as Model-Based RL. There is a fundamental problem in using these methods for practical applications. We don't have transition probabilities available for most unknown or even known environments, making these methods of limited practical use. This is where model-free methods come to rescue.

Model-Free Methods

Model-Free methods are the ones with most wide practical applications. These methods do not require any underlying transition probabilities of agent states in the environment. But they are not exactly model free as we still need the reward information from the environment. Fortunately, some notion of reward is almost always available or can be artificially generated based on any prior knowledge of appropriate behavior of the agent in the specific environment.

The basic premise of Model-Free learning is instead of finding the estimates of transition probabilities, the value function itself is approximated by carrying out experiments. One major problem is with the form of value function $V_*(s_t)$. Even if it is possible to approximate it, we will still need $\mathcal{P}(s_t, a_t, s_{t+1})$ to evaluate optimal policy at every time-step which are not available. So instead, a new value function is defined known as the state-action value function $Q_*(s_t, a_t)$ which assigns value of to each state action pair instead of only the state itself.

$$Q(s_t, a_t) = r(s_t, a_t) + \gamma \sum_{s_{t+1} \in S} \mathcal{P}(s_t, a_t, s_{t+1}) \max_{a_{t+1}} [Q(s_{t+1}, a_{t+1})]$$

$$\Rightarrow Q(s_t, a_t) = \underset{s_{t+1} \in S}{E} [r(s_t, a_t) + \gamma \max_{a_{t+1}} (Q_(s_{t+1}, a_{t+1}))] \tag{3}$$

This helps simplify the original formulation considerably because now:-

$$V_*(s_t) = \max_{a_t \in A} Q(s_t, a_t)$$

$$\pi_*(s_t) = \underset{a_t \in A}{argmax}\, Q(s_t, a_t)$$

Looking at the definition of the new value functions, it is clear that the estimate would be just the mean of the random variable $G_t = r_t + \sum_{i>t} \gamma^{i-t} r_i$ obtained after experiments. So we approximate the true mean by the empirical mean. The change that occurred with respect to the original state value function formulation $V(s_t)$ is that the $E[G_t]$ is now conditioned on both s_t and a_t instead of just s_t. So, the empirical estimates of $Q(s_t, a_t)$ will form a 2-Dimensional array instead of 1-dimensional array in case of $V(s_t)$. Now, this 2-dimensional array will be hard to estimate accurately for all possible (s_t, a_t) pairs as the dimensionality of state-space and/or action-space may be large. But notice that it is not actually required to estimate all values. What is finally required are the optimal values $max_{a_t \in A} Q(s_t, a_t)$. So, all estimates are not actually required. But before finding all the Q values, it is not possible to decide which are the optimal ones. This results in the basic 'Chicken or Egg' problem. One of the

solutions to this Q-value estimation problem is ϵ-greedy search which will be discussed later in this section.

The task of estimating Q-values is broken down into 2 parts: Evaluation and strategic Improvement.

Evaluation

There are two primary methods to estimate value functions:-

1. Monte Carlo (MC) Methods and
2. Temporal Difference (TD) Methods

Learning through Monte Carlo methods is the same approach as estimating the bias of a coin by repeated tosses. Here, $Q(s,a) = E[G_t|s_t = s, a_t = a]$ where G_t is the random variable which is observed at the end of each episode. An episode is defined as the complete duration over which the agent starts from source and reaches its destination. Multiple complete episodes are run to find the required estimates.

An update rule is used to improve the estimates at the end of each episode. This is the basic incremental mean rule described as follows. Let $\{X_j\}$ be a sequence of random variables. Then the mean of any k such random variables can be incrementally updated as:-

$$\mu_k = \frac{1}{k}\sum_{j=1}^{k} X_j = \frac{1}{k}[X_k + \sum_{j=1}^{k-1} X_j] = \frac{1}{k}[X_k + (k-1)\mu_{k-1}] = \mu_{k-1} + \frac{1}{k}[X_k - \mu_{k-1}]$$

Applying this rule to the problem at hand, the incremental update rule for Q-value function becomes:-

$$Q_n(s_t, a_t) = Q_o(s_t, a_t) + \frac{1}{N(s_t, a_t)}(G_t - Q_o(s_t, a_t)) \tag{4}$$

where $Q_n(s_t, a_t)$ and $Q_o(s_t, a_t)$ denote new and old estimate of the Q-value at (s_t, a_t), $N(s_t, a_t)$ denotes total number of times agent passed through the (s_t, a_t) state-action pair and G_t denotes the value obtained at the end of current episode.
Notice that this is the exact evaluation of $Q(s_t, a_t)$. From here onward, several steps will be taken to modify the eq. (4) to make it more conducive for calculations in real-time.

First change that is made will be to use a variable $\alpha \in [0, 1)$ instead of $\frac{1}{N(s_t,a_t)}$ and in general $\alpha > \frac{1}{N(s_t,a_t)}$. This is done to reduce and eventually remove the contribution of very old estimates to $Q(s_t, a_t)$ evaluation, as time progresses. This is a very practical thing to do because very old values become meaningless due to the dynamic nature of the agent-environment interaction.

Next, a change will be made to overcome one of the limitations of the Monte-Carlo method. Notice that the way it is posed, an episode always needs to terminate to obtain the values of G_t and make the updates using eq. (4). This may be a very slow

process or even impossible to do in cases of infinite time horizon or processes that do not terminate. So, a modification needs to be to be done to make the updates online instead of waiting till the end. This is where the temporal-difference methods are used. The idea is simple, the random variable G_t is replaced by $r(s_t, a_t) + \gamma Q_o(s_{t+1}, a_{t+1})$, an estimate of G_t which is available at the very next time-step. This time-difference based update exploits the commonly found Markovian property of the environments. Thus, the update equation becomes:-

$$Q_n(s_t, a_t) = Q_o(s_t, a_t) + \alpha[r(s_t, a_t) + \gamma Q_o(s_{t+1}, a_{t+1} - Q_o(s_t, a_t)] \tag{5}$$

Notice that this makes TD methods faster and more capable in markovian environments but they have inferior performance compared to MC methods in non-markovian environments. In practice, it is possible to get the best of both worlds by using TD(λ) methods which basically making an update after some λ amount of time-steps. These are not discussed in detail here as they are irrelevant to the project but interested readers are encouraged to refer to [2].

Improvement using Exploration-Exploitation

Finding all the Q-values may not be possible and they are not actually required too. The most important values are the optimal ones. But due to the dilemma described above, it is not possible to get one without the other. Thus, to solve this problem the concept of Exploration-Exploitation is utilized via. the ϵ-greedy search method. This tries to achieve the best of both worlds:- Exploration: Search for new action paths through the state-space to reach the destination. Exploitation: Use the knowledge from the past to improve the estimates of the best possible paths.

The ϵ-greedy search method is similar to the greedy search where the next action a is decided based on the equation $a = argmax_{a_t} Q(s_t, a_t)$. But, to explore new actions which might be even better than the initially guessed best actions, sometimes (i.e. with probability ϵ) the next action is decided randomly instead of a greedy decision. Also, in cases when Q values at a given state are exactly same for multiple actions, a decision is made randomly.

Though it looks simple, this idea of Exploration-Exploitation has proven to be very effective in applications. In addition, it has been mathematically proven to eventually converge to finding the optimal value functions [3].

Experiment

The Q-learning algorithm is applied to make a self-driving car agent learn the decisions to be made. The environment and simulator were taken from the smartcab project in the Udacity's Github page [4] and have been used as is. An agent class has been implemented that utilizes the Q-learning algorithm and learns to make appropriate decisions in this simulated environment. The simulator and hence the entire project is built on python 2.7 and the simulator utilizes the pygame library. The simulator grid-world comprises of straight single lane roads with traffic signals at every intersection and several other cars (traffic).

Driving agent receives numeric (scalar) rewards depending on how 'acceptable' was the action. This reward logic is coded in the simulator and is based on U.S. right-of-way rules. The objective is to construct an optimized Q-Learning driving agent that will navigate a Smartcab through its environment towards a goal while following these rules. 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 is the only way to ensure that Smartcabs have a permanent place in the transportation industry. Safety and Reliability are measured using a letter-grade system as follows:

Grade	Safety	Reliability
A+	Agent commits no traffic violations, and always chooses the correct action.	Agent reaches the destination in time for 100% of trips.
A	Agent commits few minor traffic violations, such as failing to move on a green light.	Agent reaches the destination on time for at least 90% of trips.
B	Agent commits frequent minor traffic violations, such as failing to move on a green light.	Agent reaches the destination on time for at least 80% of trips.
C	Agent commits at least one major traffic violation, such as driving through a red light.	Agent reaches the destination on time for at least 70% of trips.
D	Agent causes at least one minor accident, such as turning left on green with oncoming traffic.	Agent reaches the destination on time for at least 60% of trips.
F	Agent causes at least one major accident, such as driving through a red light with cross-traffic.	Agent fails to reach the destination on time for at least 60% of trips.

Figure 2: Description of performance grades (Image from 'smartcab.ipynb' in [4])

There is a visualization of the simulation available which has been disabled by default. This is done to make the simulation run faster and to ensure the simulation runs on any machine having Python (2.7) without the need of installing pygame library and other dependencies of the simulation. Optionally, the visualizations can be enabled by setting the 'display' flag of the 'Simulator' object to True. The white vehicle in the visual, is the Smartcab.

The first step is to define the valid actions the agent can take. In this case, a valid action is one of None $\Rightarrow$ (do nothing), 'left' $\Rightarrow$ (turn left), 'right' $\Rightarrow$ (turn right) or 'forward' $\Rightarrow$ (go forward). The set of valid actions is referred to as A.

The next step is to decide which inputs are the most essential to include in the state of the agent. This is a critical step and is similar to feature-engineering in supervised learning. If all the inputs are naively included in the state, the state-space will be so large that the learnt Q-values will always be sparse and it would be practically impossible as well as inefficient to make the agent learn. The driving agent receives the following data from the environment:

1. 'waypoint' $\Rightarrow$ The direction the Smartcab should drive leading to the destination, relative to the Smartcab's current position in the grid-world.

2. 'inputs' ⇒ The sensor data from the Smartcab. It includes:-
   - 'light' ⇒ The color of the traffic signal.
   - '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.
3. 'deadline' ⇒ The number of time-steps remaining for the Smartcab to reach the destination.

Assuming traffic is following regulations, there isn't a need of information about the vehicles on the left and right because the traffic signals take care of it. Hence, out of these inputs, the inputs chosen to be the part of the state are 'waypoint', 'light' and 'oncoming'. 'deadline' is right now neglected because in authors' opinion, safety is more of a concern primarily. If the agent receives decent reliability grade ('A' or higher) without including 'deadline' in the state, then this input will also be ignored in favor of smaller state-space, less storage requirements and faster performance.

The Q-table is formed as a python dictionary. Each state is a key of the dictionary, and each value will then be another python dictionary that holds the action and Q-value. Here is an example:-
{ 'state-1': 'action-1' : Qvalue-1,'action-2' : Qvalue-2, ...,
'state-2': 'action-1' : Qvalue-1,'action-2' : Qvalue-2, ...,
... }

Varying parameters α and ϵ

It is necessary to adjust learning parameters so that the driving agent learns both safety and efficiency and converges close to true optimal values. Typically this step requires a lot of trial and error, as some settings will invariably make the learning worse. Another thing to keep in mind is the act of learning itself and the time that this takes. In theory, the agent is allowed to learn for an incredibly long amount of time. However, another goal of Q-Learning is to transition from experimenting with unlearned behavior to acting on learned behavior. For example, always allowing the agent to perform a random action during training (if $\epsilon = 1$ and never decays) will certainly make it learn, but never let it act. Hence, as the number of trials increases, ϵ should decrease towards 0. Also, it is desired to slowly decay the learning rate α towards 0 to converge Q-values to optimum values, otherwise these will keep oscillating. The agent will be tested on what it has learned after ϵ has gone below a certain 'threshold' (specified as 0.05), i.e. the end of training phase is decided by value of ϵ.

Designing these decay functions took some time and in the end, the authors came up with the following functions:-

$$\epsilon_t = f_\epsilon(t) = \begin{cases} (0.5/(1.2^t)) + 0.5 & t <= expr_{iter} \\ \min\{(0.449|cos((90t/n_{iter})^o)|) + 0.051), k/(t+1)^2\} & t > expr_{iter} \end{cases}$$

$$\alpha_t = f_\alpha(t) = \min\{\alpha, \sqrt{k}/(t+1)\}$$

Where $\alpha = 0.5$ is a constant, n_{iter} is the no. of training iterations desired ($n_{iter} = 300$), $expr_{iter} = \lfloor n_{iter}/10 \rfloor$ and $k = tolerance \times (n_{iter}^2)$ are derived constants.

Hence, a total of 300 training trials and 50 testing trials are performed.

Q-Learning Algorithm

Building upon the theoretical background in the previous section, the final Q-learning algorithm used is described below as Algorithm 1.

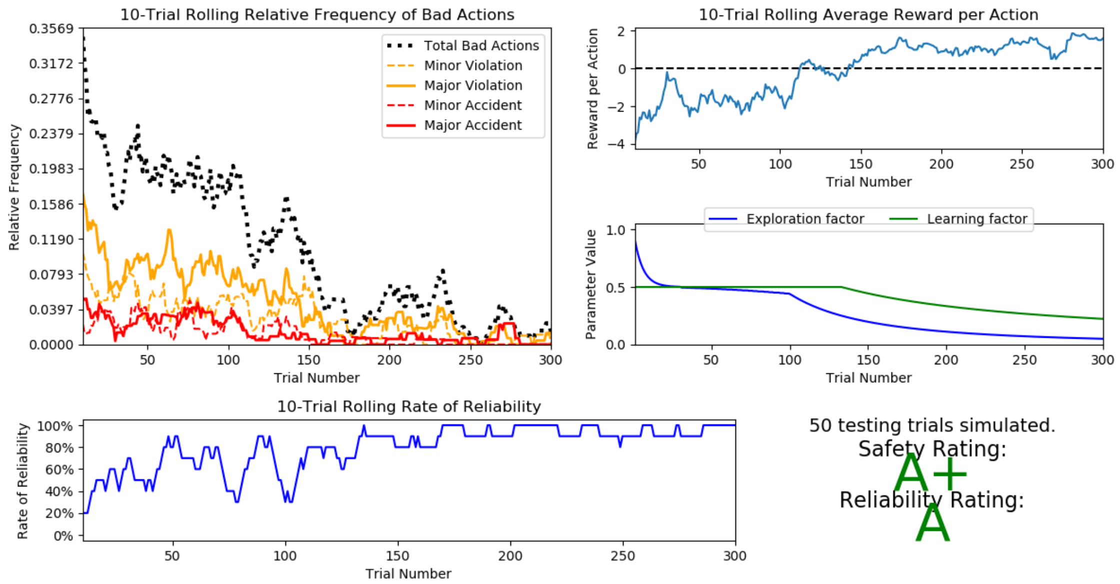
$t = 0$;
while $t <=$ *maximum no. of time-steps* **do**
  $\alpha_t = f_\alpha(t)$;
  $\epsilon_t = f_\epsilon(t)$;
  Get all the {available inputs} from environment.;
  Build the state $s_t \subset$ {available inputs}.;
  if $s_t \notin$ *Q-table* **then**
    create an entry $Q(s_t, a) = 0 \; \forall \, a \in A$;
  end
  if $\exists \, s_{t-1}$ **then**

$$Q_n(s_{t-1}, a_{t-1}) = Q_o(s_{t-1}, a_{t-1}) + \alpha_t[r(s_{t-1}, a_{t-1}) + \gamma \max_{a \in A}(Q_o(s_t, a)) - Q_o(s_{t-1}, a_{t-1})];$$

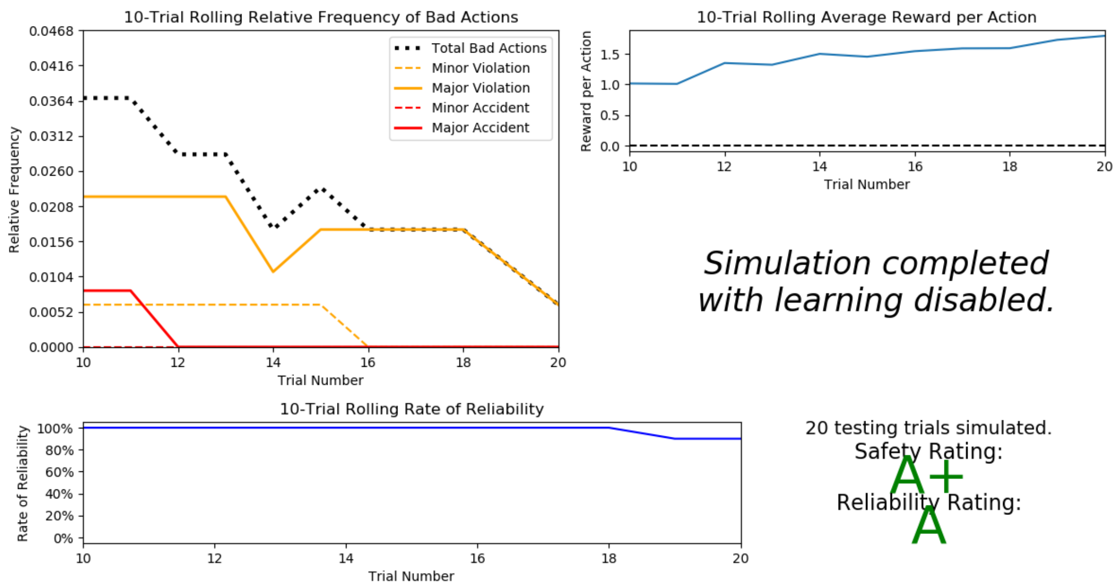
  end
  $p = uniform\,randomly\,from\,[0, 1)$;
  if $p < \epsilon_t$ **then**
    $a_t = uniform\,randomly\,from\,\{x : x = Q_o(s_t, a) \; \forall \, a \in A\}$;
  else
    $a_t = uniform\,randomly\,from\,\{x : x = argmax_{a \in A}\, Q_o(s_t, a)\}$;
  end
  Receive reward $r(s_t, a_t)$ from the environment simulator.;
  $s_{t-1} = s_t$;
  $a_{t-1} = a_t$;
  $r(s_{t-1}, a_{t-1}) = r(s_t, a_t)$;
  $t = t + 1$;
end

Algorithm 1: Q-learning algorithm

Figure 3: Simulator visualization using pygame



(a) Training and Testing Results during learning phase



(b) Testing Results after learning

Figure 4: Results of the simulation

Results

The experiment is run on a windows 10 PC with Intel Xeon CPU E5-2623 (@ 3 GHz) and RAM of 32 GB. It takes less than 3 minutes to complete. The results are produced using a separate 'visuals.py' module. They are shown in figure 4a. All the data shown is the 10-trial rolling average. It can be clearly seen that, during the training phase, all bad actions including violations and accidents gradually drop down towards zero. Although there are some violations in the last trial, there are no accidents whatsoever. The relative frequency here refers to the frequency relative to total number of actions taken during that trial. The 10-trial rolling average reward increases from -4 to 2. Similarly, even the reliability goes up to 100%. During the testing in learning phase, the agent receives the best possible grade of 'A+' for safety which was the primary target of the project. Also, the reliability grade is 'A' which is satisfactory. In fact, in only 2 out of 50 testing trials, the agent wasn't able to reach the destination within allotted time giving 96% test reliability. The decay of parameters α and ϵ according to the designed decay functions, can also be seen.

After learning phase is finished, the learning of the agent is disabled and the simulation is ran again to test the performance. These testing results are shown in figure 4b. Given the simplicity of the algorithm and the model itself, these results are pretty convincing for justifying practical application of Q-learning in general RL problems. The rules learnt by the agent can be seen in the Q-table python dictionary shown in Appendix listing (1) and the code for this experiment is in listing (2).

Conclusion

This project was an attempt to make a basic self-driving car agent using Q-learning which is a Temporal Difference based learning method. The algorithm was understood and implemented successfully in python (2.7). The important features that form the state of the agent at any time 't', were carefully chosen from all the inputs received. The model was learnt using only 3 features as state , viz., the next way-point, the traffic signal and the direction of travel of oncoming traffic. As a result, the learning was very fast and took around 3 minutes with 300 iterations. The performance of the agent, obtained from testing trials was given grade 'A+' for Safety and 'A' for reliability. In fact, the reliability was 96% with agent failing to reach destination on time in only 1 trial out of total 50. Thus, it can be concluded that Q-learning is a very simple, elegant and effective way of learning about the optimum behavior of an agent in an unknown environment.

Future Work

This project introduced a very important RL algorithm, Q-learning. There is a lot of active research going on in this domain, especially in regard to storing state-action pair values as function approximations. An application based effort is proposed as follows.

MLP training agent

The authors made an innovative effort to use the Q-learning algorithm to train an RL agent for tweaking hyper-parameters of a fully connected Multilayer Perceptron. The rewards were taken as the negative of validation loss. It was assumed that the perceptron network would have a typical trapezoidal shape with the state-space defined as $S = \{$ no. of neurons in first hidden layer (length_in), no. of layers (depth), no. of neurons in last hidden layer (length_out) $\}$. The action space was defined as $A = \{$ increase length_in by 20, decrease length_in by 20, increase depth by 1, decrease depth by 1, increase length_out by 2, decrease length_out by 2 $\}$. Although there can be more elegant ways of defining the action space and even the state-space, the authors tried to keep it simple in the beginning efforts. The agent was trained for 300 iterations with similar parameters as the smartcab agent described in this project. A part of the results was good where the agent seemed to learn that the action of increasing depth was preferred in many scenarios, which we know to be true in the field. But, the results are below satisfactory at this point. There is a need of deeper and better understanding of other RL techniques and modifications to the initial problem definition. Authors will work in this direction in future.

References

[1] Daniel, C. et al. ”Reinforcement Learning for Movement Skills,” *Intelligent Autonomous Systems*, Technische Universität Darmstadt, http://www.ausy.tu-darmstadt.de/uploads/Research/Research/ReinforcementLearning.pdf

[2] Silver, D., *UCL Course on RL*, http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching

[3] Sutton, R.S. and Barto, A.G., “Reinforcement Learning: An Introduction,” *The MIT Press*, Second Edition, April 2018.

[4] Udacity.com, *Project: Train a Smartcab How to Drive*, https://github.com/udacity/machine-learning/tree/master/projects/smartcab , January 2018.

Appendix

Listing 1: Q-Table

```
/-----------------------------------------
| State-action rewards from Q-Learning
\-----------------------------------------

('left', 'red', 'right')
 -- None : 3.54
 -- forward : -10.51
 -- right : 0.44
 -- left : -36.43

('forward', 'red', 'forward')
 -- None : 3.72
 -- forward : -8.35
 -- right : -3.52
 -- left : -24.48

('forward', 'green', 'right')
 -- None : -3.35
 -- forward : 3.46
 -- right : 2.19
 -- left : -17.70

('right', 'red', 'forward')
 -- None : 2.45
 -- forward : -24.24
 -- right : -5.55
 -- left : -15.72

('forward', 'red', None)
 -- None : 3.66
 -- forward : -8.53
 -- right : 2.27
 -- left : -17.76

('right', 'green', None)
 -- None : -3.53
 -- forward : 2.29
 -- right : 3.46
 -- left : 2.81

('right', 'red', 'right')
 -- None : 2.26
 -- forward : -8.22
 -- right : -2.27
```

```
 -- left : -38.65

('left', 'red', None)
 -- None : 3.68
 -- forward : -16.12
 -- right : 2.17
 -- left : -10.35

('right', 'green', 'left')
 -- None : -3.19
 -- forward : 2.02
 -- right : 3.21
 -- left : 1.50

('forward', 'red', 'left')
 -- None : 3.68
 -- forward : -9.33
 -- right : -1.45
 -- left : -15.50

('left', 'red', 'left')
 -- None : 3.45
 -- forward : -11.19
 -- right : -1.17
 -- left : -12.12

('forward', 'green', 'forward')
 -- None : -3.40
 -- forward : 3.85
 -- right : 2.43
 -- left : -17.92

('left', 'green', 'left')
 -- None : -3.69
 -- forward : 2.54
 -- right : 2.87
 -- left : 3.89

('left', 'green', 'forward')
 -- None : -3.52
 -- forward : 2.97
 -- right : 1.81
 -- left : -18.38

('left', 'green', 'right')
 -- None : -3.94
 -- forward : 2.69
 -- right : 1.72
```

```
 -- left : -17.31

('right', 'green', 'forward')
 -- None : -3.02
 -- forward : 2.19
 -- right : 3.31
 -- left : -18.56

('right', 'red', None)
 -- None : 2.00
 -- forward : -11.41
 -- right : 0.67
 -- left : -12.23

('forward', 'green', 'left')
 -- None : -3.40
 -- forward : 3.50
 -- right : 2.20
 -- left : 2.61

('right', 'green', 'right')
 -- None : -3.14
 -- forward : 1.88
 -- right : 3.94
 -- left : -13.60

('right', 'red', 'left')
 -- None : 2.66
 -- forward : -18.65
 -- right : 0.38
 -- left : -22.52

('left', 'green', None)
 -- None : -3.64
 -- forward : 2.06
 -- right : 2.48
 -- left : 3.70

('left', 'red', 'forward')
 -- None : 3.05
 -- forward : -8.44
 -- right : -1.13
 -- left : -20.02

('forward', 'red', 'right')
 -- None : 3.83
 -- forward : -23.31
 -- right : -8.19
```

```
 -- left : -37.67

('forward', 'green', None)
 -- None : -2.67
 -- forward : 3.15
 -- right : 2.68
 -- left : 2.36
```

Listing 2: Python Code

```
import random
from environment import Agent, Environment
from planner import RoutePlanner
from simulator import Simulator
import numpy as np
import visuals as vs

class LearningAgent(Agent):
    """ An agent that learns to drive in the Smartcab world.
        This is the object you will be modifying. """

    def __init__(self, env, learning=False, epsilon=1.0, alpha=0.5):
        super(LearningAgent, self).__init__(env) # Set the agent in the
            ↪  environment
        self.planner = RoutePlanner(self.env, self) # Create a route
            ↪ planner
        self.valid_actions = self.env.valid_actions # The set of valid
            ↪ actions

        # Set parameters of the learning agent
        self.learning = learning # Whether the agent is expected to
            ↪ learn
        self.Q = dict() # Create a Q-table which will be a dictionary
            ↪ of tuples
        self.epsilon = epsilon # Random exploration factor
        self.alpha = alpha # Learning factor
        self.t=0 #counter for Q-table


    def reset(self, destination=None, testing=False):
        """ The reset function is called at the beginning of each
            ↪ trial.
            'testing' is set to True if testing trials are being used
            once training trials have completed. """

        # Select the destination as the new location to route to
        self.planner.route_to(destination)

        # Update epsilon using a decay function
```

```
    # Update additional class parameters as needed
    # If 'testing' is True, set epsilon and alpha to 0
    n_iter=300 #>=20
    expr_iter=n_iter//10
    k=0.05*(n_iter**2)
    self.t=self.t+1;

    self.epsilon=int(self.t<=expr_iter)*((0.5/(1.2**self.t))+0.5)+\
    int(self.t>expr_iter)*np.min([(0.449*np.abs(np.cos(np.deg2rad(
        ↪ self.t*90/n_iter)))+0.051),k/(self.t+1)**2]) # update
        ↪ epsilon

    self.alpha=np.min([self.alpha,(k**0.5)/(self.t+1)])

    if testing:
        self.epsilon=0;self.alpha=0
    return None

def build_state(self):
    """ The build_state function is called when the agent requests
        ↪  data from the
        environment. The next waypoint, the intersection inputs,
            ↪ and the deadline
        are all features available to the agent. """

    # Collect data about the environment
    waypoint = self.planner.next_waypoint() # The next waypoint
    inputs = self.env.sense(self) # Visual input - intersection
        ↪ light and traffic
    deadline = self.env.get_deadline(self) # Remaining deadline

    # Because the aim of this project is to demonstrate
        ↪ Reinforcement Learning, we have not done
    # aggressive feature engineering. Instead, the aim was to
        ↪ adjust epsilon and alpha,
    # and thus learn about the balance between exploration and
        ↪ exploitation.
    # With the hand-engineered features, this learning process may
        ↪  get entirely negated.

    # Set 'state' as a tuple of relevant data for the agent

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


def get_maxQ(self, state):
    """ The get_maxQ function is called when the agent is asked to
```

```
        ↪  find the
        maximum Q-value of all actions based on the 'state' the
            ↪ smartcab is in. """

    # Calculate the maximum Q-value of all actions for a given
        ↪ state
    maxQ = np.max(self.Q[str(state)].values())

    return maxQ


def createQ(self, state):
    """ The createQ function is called when a state is generated
        ↪ by the agent. """

    # In learning phase, check if the 'state' is not in the Q-
        ↪ table
    # If it is not, create a new dictionary for that state
    # Then, for each action available, set the initial Q-value to
        ↪ 0.0

    if str(state) not in self.Q:
        self.Q[str(state)]={k:v for (k,v) in zip(self.valid_actions,
            ↪ np.zeros(len(self.valid_actions)))}

    return

def choose_action(self, state):
    """ epsilon greedy: The choose_action function is called when
        ↪ the agent is asked to choose
        which action to take, based on the 'state' the smartcab is
            ↪ in. """

    # Set the agent state and default action
    self.state = state
    self.next_waypoint = self.planner.next_waypoint()

    # When learning, choose a random action with 'epsilon'
        ↪ probability
    # Otherwise, choose an action with the highest Q-value for the
        ↪  current state
    # When choosing an action with highest Q-value selection is
        ↪ made randomly between actions that "tie".
    all_QVs=np.array(self.Q[str(state)].values()) #extract all QVs
    all_keys=np.array(self.Q[str(state)].keys())
    rand_no=random.uniform(0,1) #get a random no. between [0,1)

    if rand_no<self.epsilon: #event with epsilon probability
```

```
            apt_acts=all_keys
        else:
            apt_acts=[all_keys[i] for i in range(len(all_QVs))\
                      if all_QVs[i]==self.get_maxQ(state)] #form list of
                          ↪ apt actions

        act_idx=random.randint(0,len(apt_acts)-1) #choose random no.
        action = apt_acts[act_idx]
        return action


    def learn(self, state, action, reward,state_new):
        """ The learn function is called after the agent completes an
            ↪ action and
            receives a reward. This function does not consider future
                ↪ rewards
            when conducting learning. """


        # When learning, implement the value iteration update rule
        gamma=0.5
        Qnew_opt=self.get_maxQ(state_new)
        Q_0=self.Q[str(state)][action]
        Q_1=Q_0+self.alpha*(reward + gamma*Qnew_opt-Q_0) # update rule
        self.Q[str(state)][action]=Q_1
        return


    def update(self):
        """ The update function is called when a time step is
            ↪ completed in the
            environment for a given trial. This function will build the
                ↪ agent
            state, choose an action, receive a reward, and learn if
                ↪ enabled. """

        state = self.build_state() # Get current state
        self.createQ(state) # Create 'state' in Q-table
        if hasattr(self, 'state_old'):
            self.learn(self.state_old,self.action_old,self.reward_old,
                ↪ state) # Q-learn
        action = self.choose_action(state) # Choose an action using eps
            ↪ -greedy
        reward = self.env.act(self, action) # Receive a reward

        # save values to Q learn in next time-step
        self.state_old=state
        self.reward_old=reward
```

```
        self.action_old=action

        return


# In[Training phase]

""" Driving module for running the simulation.
    Press ESC to close the simulation, or [SPACE] to pause the
        ↪ simulation. """

# Create the environment
# Flags:
# verbose - set to True to display additional output from the
    ↪ simulation
# num_dummies - discrete number of dummy agents in the environment,
    ↪ default is 100
# grid_size - discrete number of intersections (columns, rows),
    ↪ default is (8, 6)
env = Environment(verbose=False)

# Create the driving agent object
agent = env.create_agent(LearningAgent,learning=True,alpha=0.5,epsilon
    ↪ =1)

# Follow the driving agent
# Flags:
# enforce_deadline - set to True to enforce a deadline metric
env.set_primary_agent(agent)
env.enforce_deadline=True

# Create the simulation
# Flags:
# update_delay - continuous time (in seconds) between actions,
    ↪ default is 2.0 seconds
# display - set to False to disable the GUI if PyGame is enabled
# log_metrics - set to True to log trial and simulation results to /
    ↪ logs
# optimized - set to True to change the default log file name
sim = Simulator(env,update_delay=0.01, log_metrics=True,display=False,
    ↪ optimized=True)


# Run the simulator
# Flags:
# tolerance - epsilon tolerance before beginning testing, default is
    ↪ 0.05
# n_test - discrete number of testing trials to perform, default is 0
```

```
sim.run(n_test=50,tolerance=0.05)
vs.plot_trials('sim_improved-learning.csv') #plot_results


# In[Testing phase]


#Extra testing
agent.learning=False #stop learning
sim_test = Simulator(env,update_delay=0.01, log_metrics=True,display=
    ↪ False,optimized=True)
sim_test.run(n_test=20)
vs.plot_trials('sim_no-learning.csv')
```