Markov Decision Processes

Zhihua Jin (zjin80)
zjin80@gatech.edu

Forest management problem

The first MPD problem I chose is an example in PyMDPtoolbox (Pymdptoolbox.readthedocs.io, 2019). Concerning why it is interesting, it is not a grid-world problem that is normally used to study value iteration (VI) and policy iteration (PI) in tutorials. And it has finite states which makes the analysis more straightforward. Moreover, forest fire are causing huge problems around the world every year. This project may have practical use for forest management in the real life. It basically tries to manage a forest using given guidelines and make decisions: either cut or wait. There are 7 tree abundance classes and 13 fire classes representing the years since last fire, making the entire states 13* 7=91.

For a Markov decision process, there are five important things: S (states), A (actions), P (state transition probabilities), gamma (discount factor) and R (function reward). To use value iteration and policy iteration, P and R have to be known. As we know, strategy iteration actually means that each strategy is evaluated first, and the value function is obtained. Then, through the greedy algorithm, the optimal choice of each decision point is found through a series of choices to reach the improvement of strategy. However, value iteration does not carry out policy evaluation. It is directly obtained by iterating the value function to obtain the optimal value function and corresponding optimal strategy. In comparison, value iteration uses Bellman's optimal equation to update value, and the value obtained by convergence is the optimal value in that state.To some extent, value iteration algorithm is a simplification of policy iteration algorithm. Because in order to obtain accurate Vπ value, the policy evaluation is included in each policy iteration, and all states need to be swept several times. The huge amount of calculation directly affects the efficiency of the strategy iteration algorithm.

There are a few possible ways to store transition probabilities but in the tool box the Numpy array is chosen. The transition matrix P(A,S,S') is given. Besides, reward R(A,S) represents how each state is desired. When it comes to Q-learning, it does not know the model, rewards, policies, etc. , so the transition matrix is not provided.

Generally speaking, as q-learning explores the environment, it establishes a Q-table, containing different Q-value of all kinds of states and actions (in pair). For example, once a specified action is performed in a certain state, the feedback (reward) will be returned. Through several rounds of attempts according to past experience (Q-value), it constantly updates Q-table to achieve the optimal state. In episodes, when the agent reaches the targeted final state, that old episode ends while a new episode begins (IBM Developer, 2019).

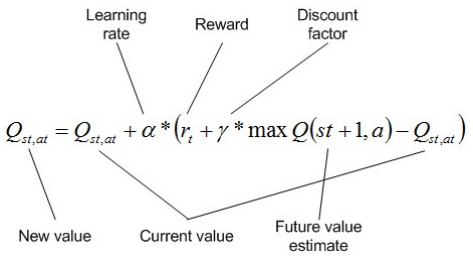
Its differences from other machine learning algorithms include:

1) There is no supervisor to tell the learner the best available action. The reward is given for each action and the learner has to sort out its own clue.

2) Reinforcement learning is different from other algorithms that focus on sequential data input.The next input in RL will always depend on the input of the previous state. The action taken right or wrong will affect the next input as well.

3) Another important characteristic of Q learning is its ability to select instant reward or delayed reward. Sometimes it is been rewarded for completing the entire task. Also, the action taken , if you choose the right one instead of leaving, the next input will be different.

According to the equation (IBM Developer, 2019), there are two hyper-parameters that matters a lot to reach the Q-value:



The first is learning rate (alpha), which defines the proportion of the new Q that an old Q will learn from the new Q. An alpha of 0 means that the agent will learn nothing new but just keep exploiting the old information, and a value of 1 means that the new information is the only information that is important. The next factor, known as the discount factor (gamma), defines the importance of future rewards. If the discount factor equals 0, then it means that only short-term rewards are considered, with a value of 1 valuing long-term rewards more.

In the actual implementation of q-learning iteration, epsilon (e) greedy strategy can be used to calculate the action. This is because if we choose the action with the highest utility value each time, the learner will be confined to past experience, leading to limited states. In this way, it at best is able to exploit past routes and pick out the current best option. If there are potential ways better than that, it cannot find it. However, if we keep exploring random state, the convergence speed will be too slow. So the solution is to keep a certain small probability to explore. Setting epsilon to a smaller value could achieve this goal.

Using functions in the tool box and some revisions, VI, PI and q-learning are compared in following diagrams (Figure 1, Figure 2). For model-based learning, the epsilon is 0.01. If the value function change goes below this, then it is considered to converged to the optimal value function. Besides, the discount factor is set to 0.9. This means that future rewards have a discount rate of 10% .

	Value Iteration	Policy Iteration	Q-learning
Iteration	156	4	10000
Time (s)	0.03	0.012	0.22

Figure 1. Time and iteration for three algorithms

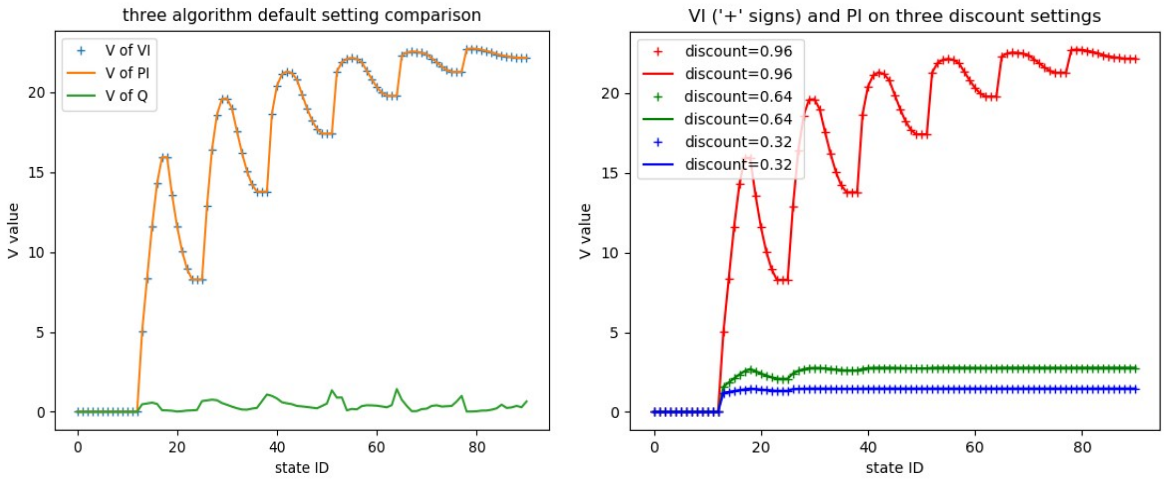


Figure 2. V-value and parameter tuning for three algorithms

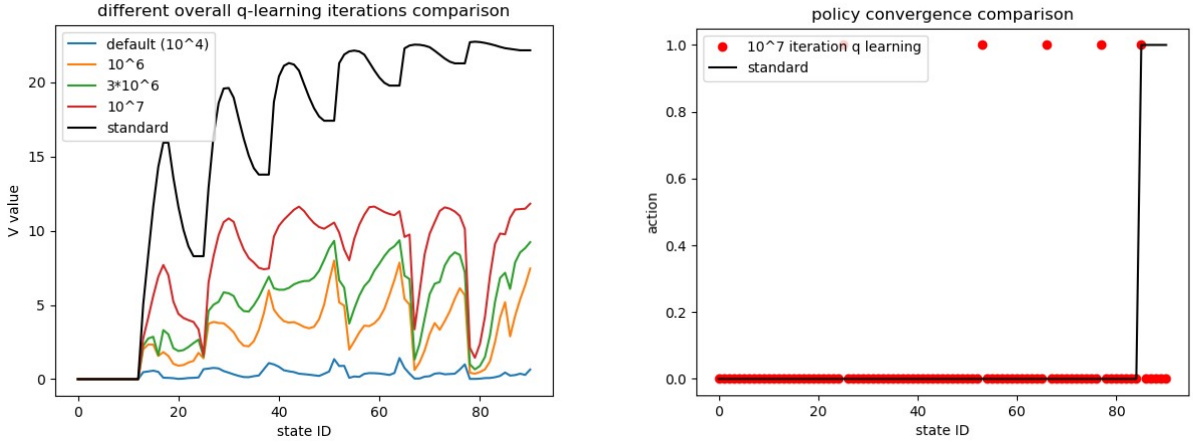
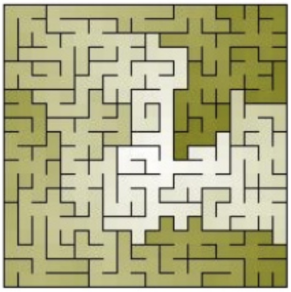


Figure 3. Different iteration setting for q-learning

Judging from figure 1 and 2, I would say VI converges faster than PI, and q-learning is the slowest. For iterations, VI needs more iterations to calculate the V value. Anyway, in this discounted finite MDP problem, VI and PI both can converge to optimal policy after a finite number of iterations. In addition, since higher V value is anticipated, the performance of VI and PI are good while q-learning is not really satisfactory. I changed discount factor to lower ones, then the V value of VI and PI decrease drastically. So in this case, discount factor around 0.96 is acceptable. Last but not least, I varied n_iter in q-learning to larger ones, but it still could not converge in 270 seconds (Figure 3). This means that it chooses non-optimal actions in many states. These can be deadly in real life events.

A grid-world maze problem

The second problem is a randomly generated maze world using Aldous-Broder algorithm. Every time it randomly selects a grid as a starting point, and randomly attempts to move to a neighbor grid. If the new grid has not been visited and there is a wall between the old and new cell, a passage will be carved. After all grids are visited, the maze will look like a random maze with walls (Gao, Rule and Hao, 2019). It is interesting because although it is a seemingly simple maze, it creates a dynamic environment for exploration. I wished to use the frozen lake grid- world, but the original version has too few states. So I turned to the wall maze, the maze could be adjusted on demand to very huge sizes so how different state sizes make a difference to convergence time and iterations, etc. can be researched on.



With a 5*5 maze (25 states), the maze agent uses value iteration and policy iteration, trying to find the way out on the upper left (grid in red). The gamma used here is 0.1. When the difference of $V_{\pi}(s)$ in two sequential iterations is smaller than 1e-5, then I define it has reached convergence.

As the iteration increases, it gradually finds its optimal policy (figure 4) and optimal value for each grid (figure 5). PI uses 17 iterations to converge while VI uses 89 iterations. The convergence time for PI is 0.031 while VI is 0.024 second. In a larger maze of 20 x 20 (400 states), the convergence time/iteration of PI is 0.24 seconds/16 while VI is 0.13/89. So for both small and large states, the PI is slower than VI, but its iterations are fewer.

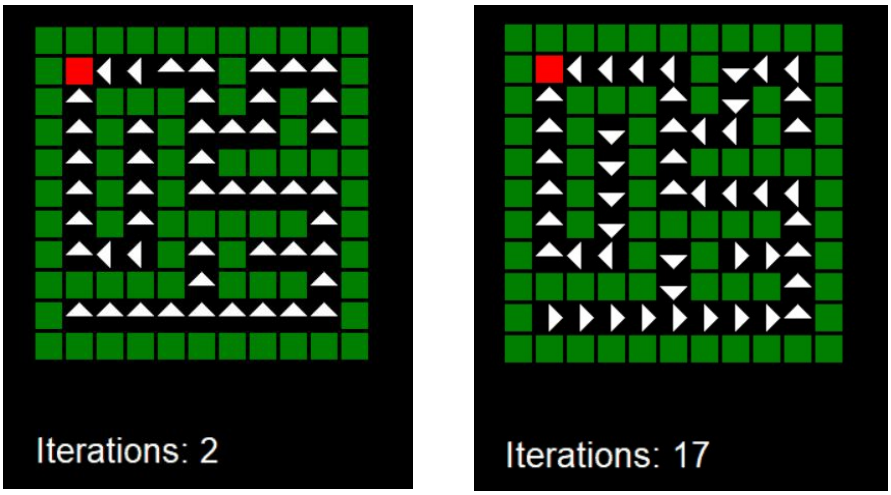


Figure 4. Policy iteration demo

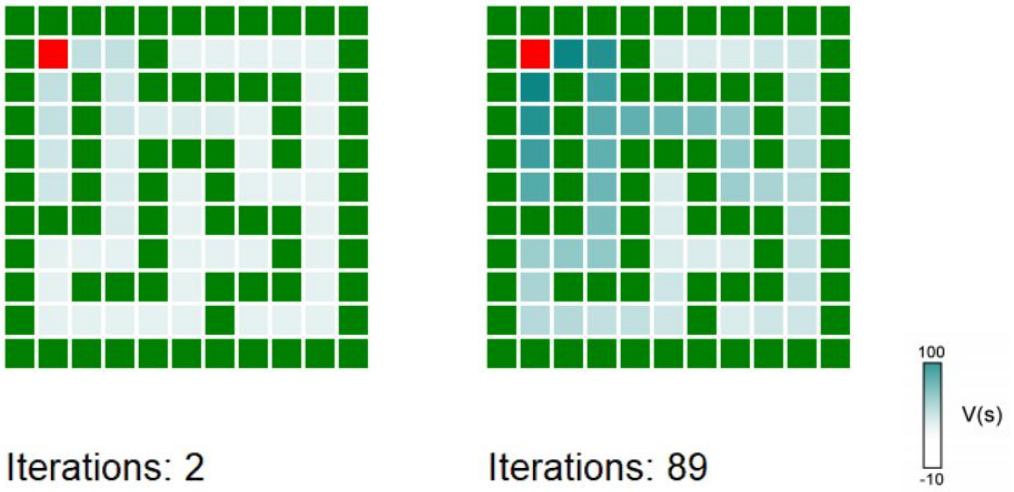


Figure 5. Value iteration demo

Since it is a random maze, it is not easy to determine if VI and PI has reached to the same answer. But as we increase the number of states, we could see intuitively (figure 6, left) that for the same maze, VI numbers of iteration is stable most of the time, while PI iteration starts small when the grids are very few, but then reaches a higher plateau as the maze map enlarges. Anyway, it is still smaller than the VI iterations. This is understandable because once the optimal policy is found, then the iteration ends. VI usually needs more iterations to allocate value with nuance. In terms of time, as the grid increases, the solving time for both increases. Meanwhile, VI outperforms PI as it needs less time (figure 6, right). However, this is not always the same. There remains uncertainty for different mazes generated. For certain maze structures, when the grid size reaches around 5000, the time PI used decreases to lower than VI. This is probably because of the characteristic of the problem.

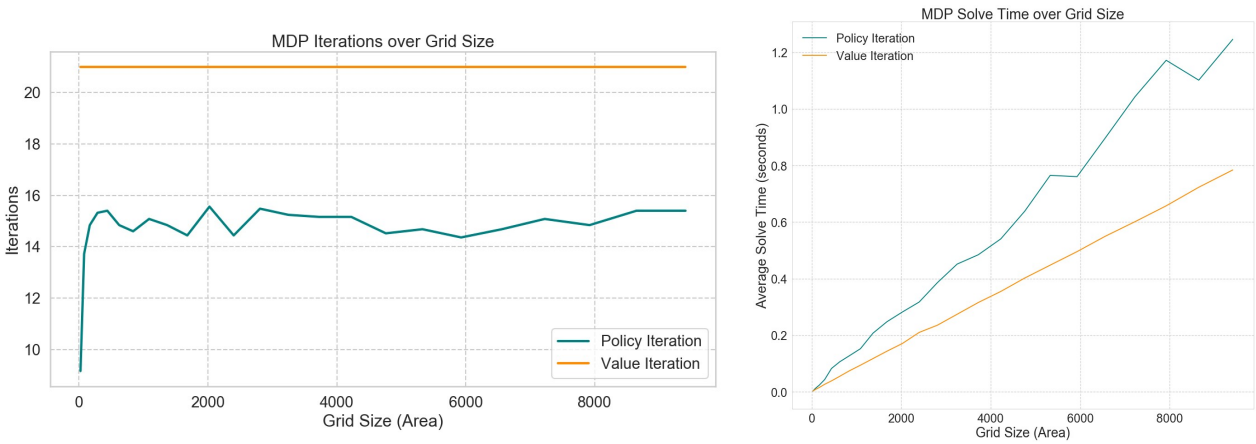


Figure 6. Iterations and time comparison for increasing states

Moreover, I checked the impact of discount factor gamma. From figure 7, we could see that for gamma below 0.9, both VI and PI are completed in fewer than 100 iterations. When the closer gamma is to 1, the iteration used is increasing. Especially for PI, when gamma is over 0.9, then the iteration numbers grow drastically. This is probably because the policy tries to optimize the gains even further. At the same time, value iteration becomes exponentially slower to converge. Actually, the best gamma depends on the domain. In my case, a small gamma of 0.1 is enough to find the optimal policy.

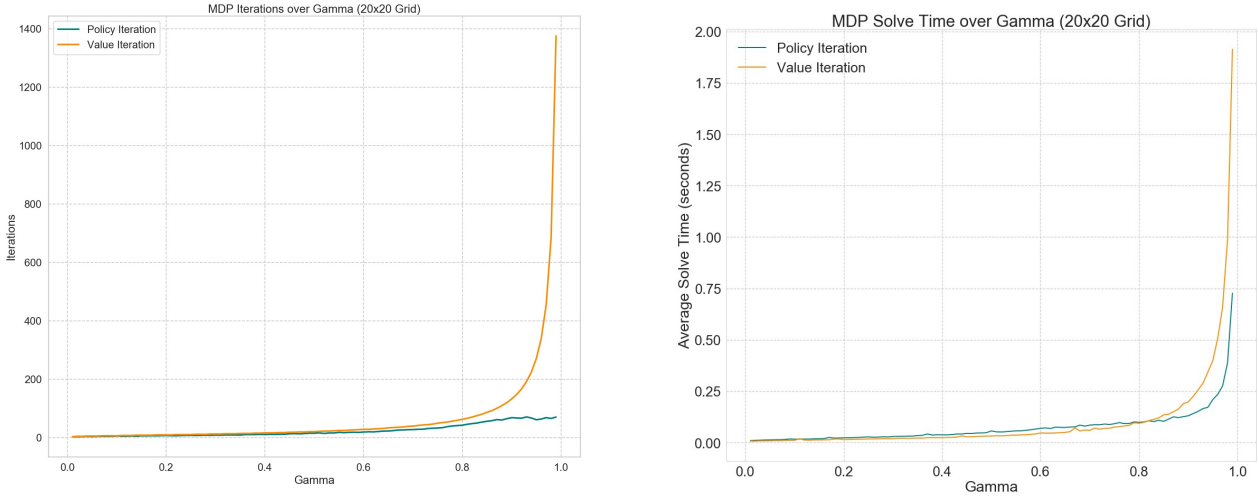
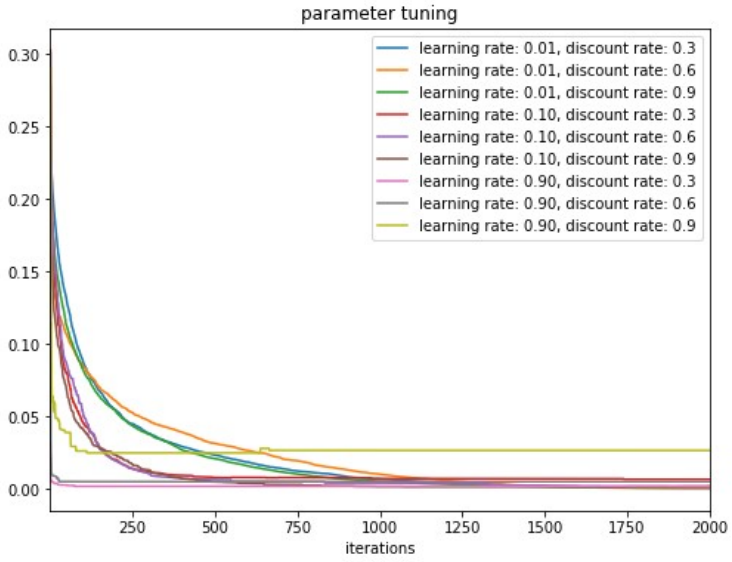


Figure 7. Iterations and time comparison for different gamma

For q-learning, I changed hyper-parameters alpha and epsilon as well. There is a trade-off of exploitation and exploration. According to figure 8, the time used for every episode in the first 2000 episodes are plotted. The reward is also calculated.



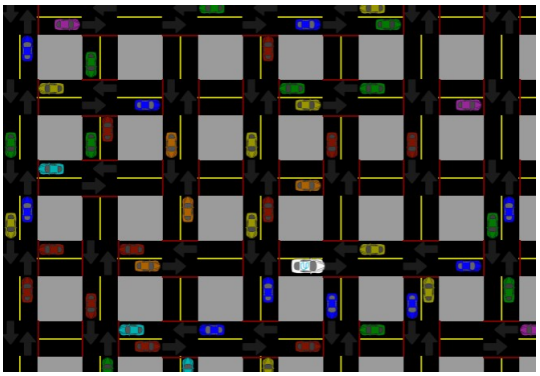
Learning rate	Discount rate	Q-value
0.01	0.3	170
0.01	0.6	90
0.01	0.9	115
0.10	0.3	104
0.10	0.6	95
0.10	0.9	92
0.90	0.3	52
0.90	0.6	51
0.90	0.9	44

Figure 8. Iteration time and reward comparison for different learning rate and discount rate

Taking these together, I think (learning 0.1, discount 0.3) is probably the best when the time is tight, since the reward is quite high and the time used is not too much. In other cases with higher rewards, the time consumed is twice as much. No wonder, a small learning rate means the learner is more focused on exploiting its known domain knowledge than exploring new environment. This problem certainly needs a lot experience. Meanwhile, future rewards are also taken into consideration rationally to reach the exit.

A more complicated grid-world problem

I took Udacity's Machine Learning Nanoprogram before and there is one more interesting problem (Medium, 2019), which is much more complicated than the previous one I mentioned.



The goal is to build an optimized Q-learning driving agent that can manipulate a smart taxi to move from the departure point and arrive at the destination. Its surrounding environment has been constructed with different rewards given to different actions. The departure and landing places are randomly generated. For correct actions, a reward with positive number will be given; for wrong actions, a penalty will be returned. And the two evaluation metrics are: safety and reliability. For example, if the driving agent still lets the taxi to move when the red light is on or it almost runs into accidents, then the agent is considered unsafe. Similarly, if the car can't arrive timely all the times, then the agent would be considered unreliable. The grade of learner is given accordingly to following rules:

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 9. Grading metrics

A series of visualization functions have been provided as help function to figures out how good the agent really performs. When the agent is not learning, it has fail grades for both metrics for sure. But as we can see from figure below, once it starts learning from past actions (with epsilon set to $\epsilon_{t+1} = \epsilon_t - 0.05$, for trial number set), even though the grade is still F, the total bad actions decrease while the average reward increases.

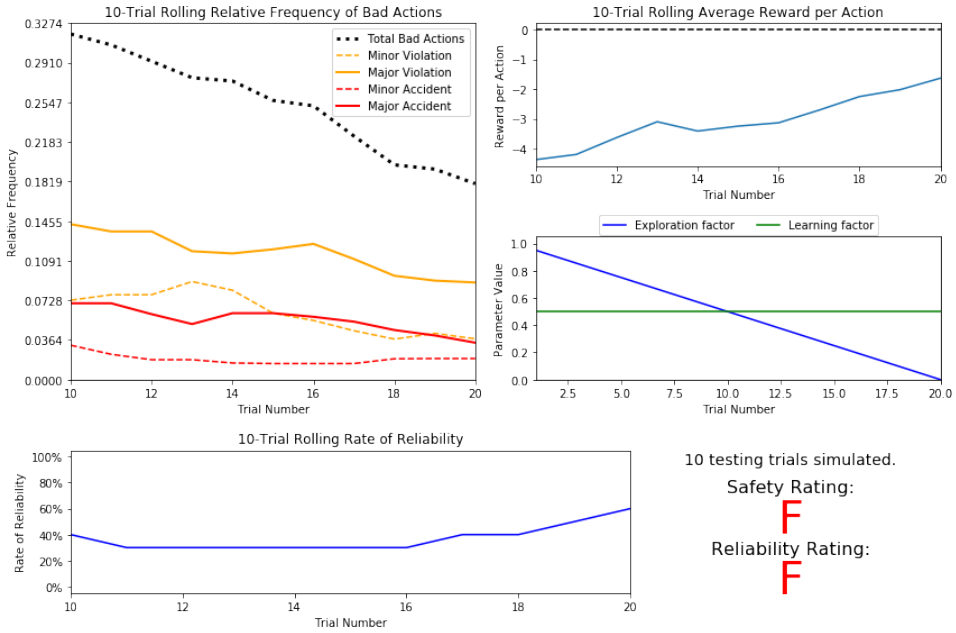


Figure 10. Non-learner performance

The next step is to optimize the agent even further. To certain extent, q-learning tries to transfer exploration and experiment into a learning and exploiting decision maker. Learning rate and epsilon are changed several runs to observe the performance of driving agent. The focus is mostly put on tuning epsilon.

$$\epsilon = a^t, \text{for } 0 < a < 1 \qquad \epsilon = \frac{1}{t^2} \qquad \epsilon = e^{-at}, \text{for } 0 < a < 1 \qquad \epsilon = \cos(at), \text{for } 0 < a < 1$$

For example, in the following figure, the epsilon equals exp (-0.01 * t), and learning rate equals 0.5 (which is a mediocre choice). The total bad actions decrease to a rather small number after 250 trials.

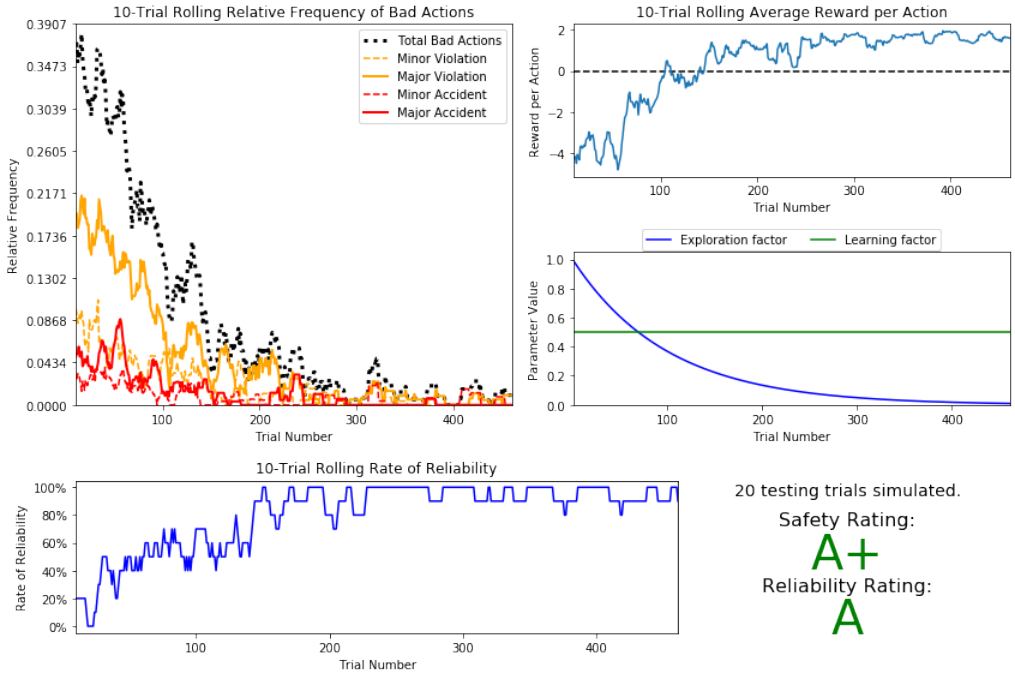


Figure 11. Q-learner Performance (e= exp (-0.01 * t), apha= 0.5)

A portion of attempts are listed here:

Series	Q-Learning episilon	Alpha	Tolerance	n_test	Safety	Reliability
1	epsilon = math.exp(-0.01 * t)	0.5	0.1	20	F	B
2	epsilon = math.cos(t * 0.1)	0.5	0.1	20	A	D
3	epsilon = math.cos(t * 0.0015)	0.5	0.1	20	A+	A
4	epsilon = math.cos(t * 0.0015)	math.e**(-0.001 * t)	0.1	20	A+	A+

Figure 12. Q-learner Performance with different hyper-parameters

We can see that the fourth attempt reaches A+ in both metrics.When epsilon = cos(t * 0.0015), the performance is the best as it ensures the learner to explore and learn as much as possible in the early stage. Later, it learns less and exploits more, which optimizes the q-learning process. The total iterations are 1023.

Another thing worth mentioning about this q-learner is that q value uses no future rewards but short-term rewards. This is because the cab does not know how far the destination is. As a very practical problem, the actual vehicles and traffic situations are not predictable, Neither could they be stored as effective geographic information in the Q table, so the smart taxi can only refer to its current state to determine the next action. We know that in the case of the maze, where the starting point and end point are fixed, every state is valid information and has more robust dependencies, future rewards can play the role.

Conclusion and Further work

We all know that dynamic programming requires a completely known environment model, which is definitely hard to do in reality. This makes on-policy value iteration and policy iteration somehow useless. Besides, continuous states may also increase the difficulty. It can be partially conquered if input is discretized into bins. To solve this more thoroughly, I checked a lecture note and found an algorithm called fitted value iteration algorithm. For infinite continuous state, linear regression could be used to try to make V^{π} value approximately close to the true value. But the fitted value iteration does not always converge, which would make the comparison of efficiency difficult (Cs229.stanford.edu, 2019).

In comparison, the advantage of off-policy q-learning is apparent. Especially for those more complicated problems, a more powerful algorithm is deep Q learning network. A derivative called Greedy-GQ extends to non-linear realm using neural network is assured convergence even when it uses approximated action values (En.wikipedia.org, 2019). If more time is given, these aspects could be explored in more depth.

References

1. Gao, S., Rule, D. and Hao, Y. (2019). *mazemdp*. [online] Available at: https://github.com/sally-gao/mazemdp [Accessed 23 Nov. 2019].
2. Pymdptoolbox.readthedocs.io. (2019). *Markov Decision Process (MDP) Toolbox: example module — Python Markov Decision Process Toolbox 4.0-b4 documentation*. [online] Available at: https://pymdptoolbox.readthedocs.io/en/latest/api/example.html [Accessed 23 Nov. 2019].
3. En.wikipedia.org. (2019). *Q-learning*. [online] Available at: https://en.wikipedia.org/wiki/Q-learning [Accessed 23 Nov. 2019].
4. Cs229.stanford.edu. (2019). *Reinforcement Learning and Control*. [online] Available at: http://cs229.stanford.edu/summer2019/cs229-notes12.pdf [Accessed 23 Nov. 2019].
5. IBM Developer. (2019). *Train a software agent to behave rationally with reinforcement learning*. [online] Available at: https://developer.ibm.com/articles/cc-reinforcement-learning-train-software-agent/ [Accessed 23 Nov. 2019].
6. Medium. (2019). *Udacity-Machine Learning(3) — Reinforcement Learning*. [online] Available at: https://medium.com/@holfyuen/academics-machine-learning-nanodegree-programme-at-udacity-3-35b837b5e234 [Accessed 23 Nov. 2019].