A Comparison Between Reinforcement Learning Algorithms

Shiqi He

December 17, 2018

Abstract

Different reinforcement learning algorithm has its own pros and cons. A suitable algorithm can largely benefit the program. This report compares tabular methods and approximate methods. Codes are built to demonstrate the features of different algorithms.

1 Introduction

Reinforcement learning is about to maximize the reward and without knowing the environment model. Therefore, various methods are created to enable the agent to value its policies, adjust them, and finally, converge to the optimal solution. When facing a specific problem, there are several aspects we should consider, before we make our final decision. First of all, the state space and action space volume. Then the type of the environment. For instance, whether it is discrete or statistic. Last but not least, the convergence of each algorithm. A comprehensive decision could augment the efficiency of the program.

2 Tabular Methods

When the state space and action space is relatively small enough for the value functions to be represented as tables, then a tabular method is an economic choice. Tubular methods are opposite with Approximate methods, which would be discussed in the sequential part, they usually can find out the exact solutions, *i.e.* optimal policies [1].

The common tabular methods are the Monte Carlo method, Sarsa and Q-learning. For the Monte Carlo method, the agent learns in an episode-by-episode sense, but not step-by-step [1]. Therefore, the maximum steps of an episode would limit its usage. But for Sarsa and Q-learning, they are both temporal-difference learning algorithms, which means they bootstrap. That is one of the central and novel ideas of reinforcement learning [1]. The following part will focus on the comparison between Sarsa and Q-learning.

2.1 Sarsa: on-policy learning

A general form of Sarsa algorithm is shown in Fig.1.

The update rule indicates that Sarsa updates based on its next action. It learns whilst it explores. Therefore, it is clear that Sarsa is an on-policy learning. And due to this feature, when a large negative reward is near to an optimal path, it would take time for Sarsa to figure it out, since on-policy learning would take the risk into account. Therefore, we can draw the conclusion that Sarsa is a "conservative" algorithm.

2.2 Q-learning: off-policy learning

Q-learning format is similar to Sarsa, except that it updates according to the maximum estimated nest possible state, regardless of what exact action it would take. This feature contributes for Q-learning to become a more "aggressive" algorithm tan Sarsa, since the maximum policy would only tell the agent where is the discounted maximum reward lays. One of the classical demonstrations is called *Cliff walking* Then that proofs that Q-learning to be an off-policy learning. The algorithm is shown in Fig.2.

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Figure 1: Sarsa Algorithm

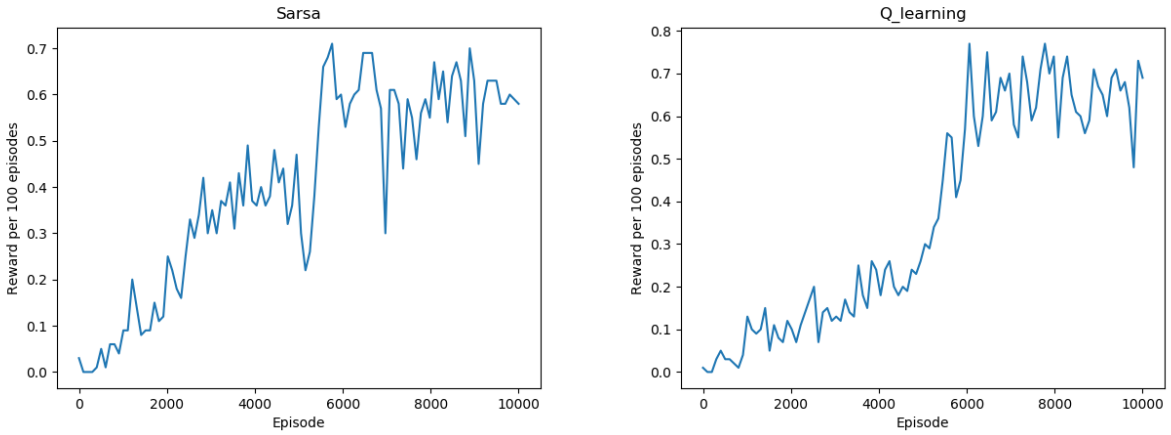
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 2: Q-learning Algorithm

2.3 Results Comparison

The two codes are implemented in the same environment called *FrozenLake-v0*, from *OpenAI GYM*. All the parameters are identical. However, the agent performances are not necessarily the same. As we can see (Fig.3), Saras converges to optimal policy faster than Q-learning, but the average reward it can gain is lower than Q-learning.



(a) Sarsa rewards

(b) Q rewards

Figure 3: Results Comparison

2.4 Evaluation

Two algorithms both have its own advantages and disadvantages. A comparison between them is shown as the table below.

Table 1: Features of Sarsa and Q-learning

	Sarsa	Q-learning
Learns from	Sub-optimal policies	Optimal-policies
Sample variance	Lower	Higher
Converge rate	Faster	Slower

According to the characters of on-policy learning and off-policy learning, how to choose between them depends on the requirement of the case. If mistakes are costly in a case, for instance, training robot in practical, a mistake could contribute to its damage, then using Sarsa could reduce the risk of high cost. On the contrary, if we are stimulating on a computer, meaning that a mistake is not so serious, then Q-learning would outrun Sarsa.

3 Approximate solution methods

When the state space or action space become enormous, it is not effective or even possible to apply tabular methods. For instance, a continuous environment would have infinite states. Then the methods mentioned above would become invalid. This is when we should apply approximate methods. The most significant feature of approximates method is that it can generalize from its previous encounters, and applies to the current one. One way to tackle huge state space problem is by importing Neural Networks.

3.1 Deep Q-learning

If we combine Q-learning with Neural Network, then we can call it Deep Q-learning. The algorithm works in this way: once the states are feed into the Neural Network, they would run through all the hidden layers, which contain weights and biases. In the end, the network will output all the actions with their scores. We treat the highest scored action as an optimal policy. In the meantime, the Neural Network itself would update by using backpropagation and loss function to update the weights. A well-trained Neural Network means that the weights will bring out the correct result or optimal policy. This algorithm's validation is proved by Volodymyr Mnith, *et al* [2].

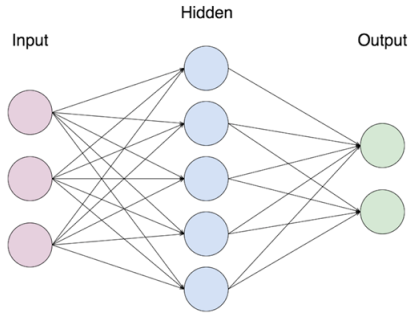


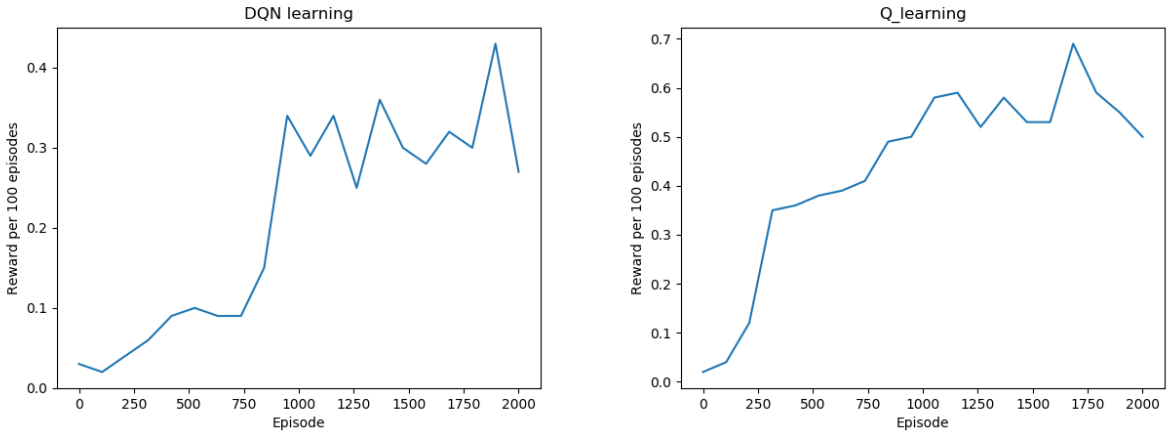
Figure 4: Neural Network

Q-learning updates by using the Bellman Equation. For Deep Q-learning, instead of directly updating the Q table, we use a loss function to estimate outputs. In the code, we choose the loss function as:

$$Loss = \sum (Q_{target} - Q)^2$$

3.2 Compare DQL and Q-learning

The performance of Deep Q-learning and Q-learning under the same environment is compared. The environment does not change(*FrozenLake-v0*). A one layer Deep Q Network is built. Surprisingly, in this case, Deep Q-learning is actually less effective than Q-learning (Fig.5). One possible explanation is that the environment is relatively simple, so it cannot bring out the superiority of Deep Q Network. And it is true that once a neural network is implemented, it might encounter some problem with stability. There are several methods to fix this problem, or at least to improve the network's performances. For instance, experience replay and freezing target network.



(a) DQL

(b) Q-learning

Figure 5: Results Comparison

3.3 Policy Gradient

So far the algorithms mentioned above select an action based on their value functions. However, for policy gradient methods the agents learn without consulting to a value function. Since the policy gradient method gets rid of value functions, it has a number of advantages. Firstly, it has a better convergence than value-based methods. It updates smoothly without oscillation. Secondly, it can be applied to tackle infinite action space problem. Last but not least, it can learn from stochastic policy, meaning that there is no concern about the trade-off between exploration and exploitation.

In summary, policy gradient method would overtake Deep Q-learning when the action state is enormous or infinite.

The frame of the algorithm comprises two steps. Step one, after the agent takes a policy (π), a score function $J(\theta)$ is applied to measure the quality of the policy π. Then in step two, we apply policy gradient ascent to find the best parameter θ that improves the former policy π. After that parameter Theta would be updated and this algorithm iterates.

Gradient ascent is logarithm function of policy:

$$\mathbb{E}_\pi[\nabla_\theta(log\pi(s,a,\theta))G_t]$$

Update rule:

$$\theta \leftarrow \theta + \alpha\nabla_\theta log\pi(s,a,\theta))G_t$$

G_t is a score function, which can be treated as a scaler. A high G_t means that the corresponding action would return a good reward. So the possibility of taking this action would be increased. And vise verses.

3.4 Results of Policy Gradient

There is a cluster of policy gradient methods. Here *Monte Carlo Policy Gradient method* is applied on the environment called *CartPole-v0*. The goal is by controlling the cart and stabilize the pole vertically. We plot the evolution of G_t (Fig.6).

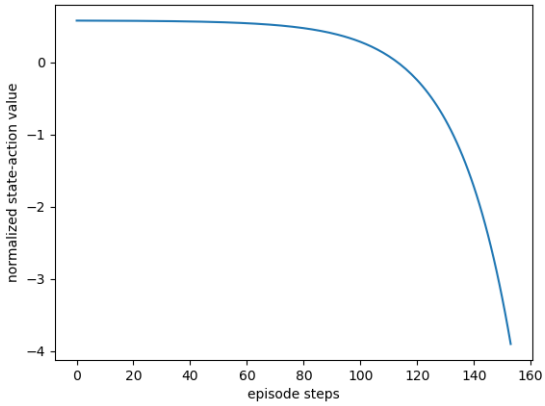


Figure 6: MC Policy gradient

In the beginning, a positive G_t feedback is gained, which illustrates that the agent is making good improvements to stabilize that pole. And in the end, G_t deceases, that means the pole is already stabilized then G_t does not need to make big changes.

4 Summary

Although there is no strict regulation between problems and algorithms, we still can evaluate those algorithms by their features. A suitable algorithm can be time-saving and cost-saving. In summary, tabular methods fit relatively simple environments, whereas approximate methods should be applied to huge state space or action space problems. And then we can use different sub-scheme to refine the agent's performance.

References

[1] Rechard S. Sutton, and Andrew G. Barto *REINFORCEMENT LEARNING: an introduction*. 2018.

[2] V. Mnih, D. Silver, and M. Riedmiller, *Playing Atari with Deep Reinforcement Learning*, pp. 1–9.