An Empirical Evaluation of the Basic Algorithms in Reinforcement Learning

Ke Bai

Abstract

In order to have a better understanding of reinforcement learning and lay the foundation for my future research. I implement several representative algorithms and try to figure out how they work. Here is my code https://github.com/beckybai/reinforcement_learning_pytorch

I. Introduction

I implement several basic algorithms in deep reinforcement learning, including SARSA, Q-learning[6], REINFORCE[7], Actor-Critic[1], DDPG[3], and TRPO[4].

There are two reasons I start doing this. One is that I want to have a better and deeper understanding of reinforcement learning. The theory and the implementation usually has a hug gap. It is hard for me to reproduce other's advanced work if I do not know how to deal with the simple one.

The other reason is that there lacks a well-written reinforcement learning code under the framework of Pytorch. Pytorch is easier to read and debug for a beginner than Tensorflow. In future, I will continue to work on this project, adding more algorithms I haven't done yet and adding more notes to make it easier to read.

In the following part, firstly, I will introduce each algorithm I implemented, show the pseudo code and the performance of my implementation. Then I will discuss the result I got, the problem I meet, talk about my own understanding of these algorithms.

II. Algorithms and Results

Reinforcement learning (RL) is based on the reward hypothesis. The reward comes from the interaction between the environment and the agent. There are various ways to categorize RL. We can divide the agent into three classes: value-based, policy-based and actor-critic; we can also categorize the agent into the model-based and model-free.

Table I: Algorithm Category

NAME	AGENT	Policy	Policy	Model
SARSA	Value function approximation	on	Implicit policy(ϵ-greedy)	model-free
Q-Learning	Value function approximation	off	Implicit policy(ϵ-greedy)	
REINFORCE	Policy	on	Learnt Policy $\pi(a\|s)$ (gradient descent)	
Actor-Critic	Policy +Value function approximation	on	Learnt Policy $\pi(a\|s)$ (gradient descent)	
Actor-Critic	Policy +Value function approximation	on	Learnt Policy $\pi(a\|s)$ (gradient descent)	
DDPG	Deterministic Policy + Value function approximation	off	Learnt Policy $f(a\|s)$ (gradient descent)	
TRPO	Policy optimization + Value function approximation	on	Learnt Policy $\pi(a\|s)$ (trust region policy optimization method)	

Table II: Specific implement

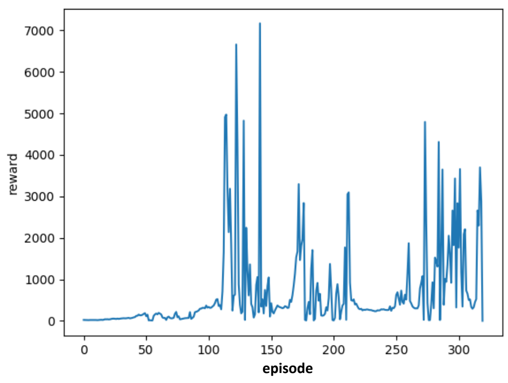
	TD or MC	Policy	policy evaluation	task	converage	function approximator
SARSA	TD	ϵ-greedy	Q(s,a)	Cart-Pole	yes	neural network
Q-Learning	TD	ϵ-greedy	Q(s,a)	Cart-Pole	yes	
REINFORCE	MC	P(a|s)	$\Sigma_{i=t}^{T}(\eta^{i-t}r_t)$	Cart-Pole	yes	
Actor-Critic	TD	P(a|s)	V(s)	Cart-Pole	yes	
Actor-Critic	TD	P(a|s)	Q(s,a),A(s,a),V(s)	Pendulum	not yet	
DDPG	TD	P(a|s)	Q(s,a)	Pendulum	yes	
TRPO	TD	P(a|s)	A(s,a)	Pendulum	not yet	

I select several representative tasks, trying to cover different categories. Table. I shows the attributes of the algorithm I implemented.

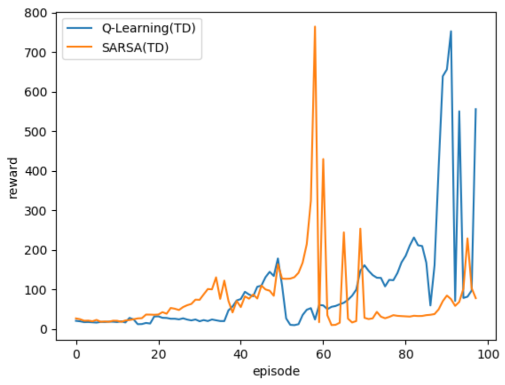
An algorithm can have different variants and can be adapted to different tasks. Table. II summarizes more details of each implementation. In the following, I will introduce how to understand and implement each algorithm and show the performance of the algorithm on a specific task.

A. Q-Learning and SARSA(Algo. 1)

We update the Q function of SARSA and Q-Learning using the temporal-difference learning.



(a)



(b)

Figure 1: (a) The reward of SARSA. There are one hundred trajectories in each episode. The y axis is the summation of the reward for a trajectory on average in an episode. (b) The comparison of the reward of SARSA and Q-Learning

If we update the Q function every step, the result will not converge to the expected range. This problem is solved when we update the Q function every n steps ($n = 6$ here, it also means the batch size of training is 6). (Fig. 4a) and (Fig. 4b) show the result. The y-axis is the reward we get for a trajectory on average in an episode. Each episode contains 100 trajectories.

The result we get in both algorithms is very unstable. Changing the batch size is useless. In theory, TD-learning has a low variance but high bias. However, the variance of each point is also very large when the reward is large(the variance doesn't show in this figure).

As far as I am concerned, the stability of the network not only depend on how we estimate the value function, like using the temporal difference or Monte Carol but also has the relationship to how we train the neural network. For example, using a memory replay always tends to stabilize the model.

One thing should be noticed. Normally, when the total reward in a trajectory reaches a value (200 in the Cartpole-v0 task), the environment will stop the agent forcefully. Here, we remove this constraint in order to study the performance of these two model. The maximum reward of these two figure (Fig. 4a), (Fig. 4b) can be infinite.

Algorithm 1 SARSA(Q-Learning)

```
function SARSA
Initialize $w$,$\theta$
Initialize the environment.
    for each episode do
        Initialize $s$
        for each step do
            Choice $a$ from $Q$                                          ▷ $\epsilon$-greedy
            [$s_{next}$, $r$, $done$] ← env.step($a$)
            Choose $a_{next}$ from $Q$
            $\delta = r + Q_\theta(s_{next}, a_{next}) - Q_\theta(s, a)$
                    ▷ For Q-learning, $\delta = r + argmax_{a_{next}} Q_\theta(s_{next}, a_{next}) - Q_\theta(s, a)$
            $\theta \leftarrow \theta + \alpha\delta\nabla_\theta Q_\theta(s, a)$                    ▷ Assume we use L-2 loss
            $s \leftarrow s'$
        end for
    end for
end function
```

B. REINFORCE (Algo. 2) and Actor-Critic (Algo. 3)

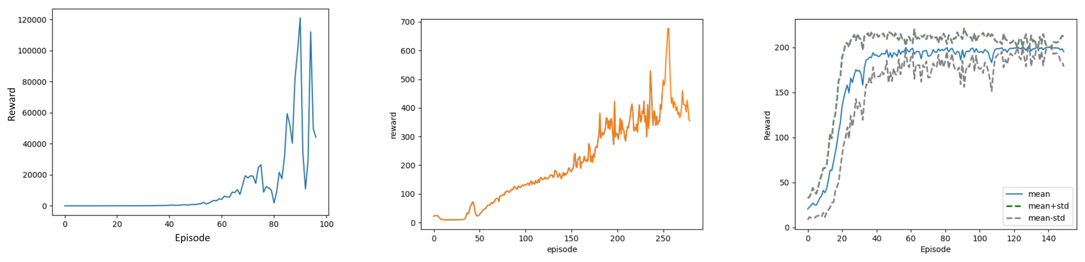
Under the same environment setting as the latest section, REINFORCE(Fig. 2a) and Actor-Critic(Fig. 2b), these two policy-based algorithms are stabler and better than the value-based one.

The main difference between these two algorithms is their critic functions. REINFORCE uses $\Sigma_{i=t}^{T}\eta^{i-t} * r_t$, a Monte-Carlo estimation. Actor-Critic uses the value-based ($V(s)$) temporal difference learning. REINFORCE algorithm is better since it can receive more information from the future than the second one.

We also run an experiment(Fig. 2c) when environment restricts the summation of the reward of each trajectory (when the total reward reaches this limitation, the game over). After reaching the limited reward, the variance of the reward tends to decrease as the training episode increases.

C. Actor-Critic and DDPG

The former algorithms mainly deal with problem whose action space is discrete. The policy gradient methods mainly solve the problems with continuous action space.



(a) (b) (c)

Figure 2: (a) REINFORCE algorithm. The reward is unlimited. (b) Actor-Critic algorithm in the discrete action space. The environment setting is the same as (a), the reward is increase but slower than REINFORCE. (c) REINFORCE algorithm. The reward is limited. The dashed line shows the standard deviation error.

Algorithm 2 Monte-Carlo Policy Gradient

```
function REINFORCE
Initialize $\theta : \pi_\theta(a|x)$
    for each episode do
        Run the model to get one trajectory $\{s_1, a_1, r_2, \cdots, s_{T-1}, a_{T-1}, r_T\}$
        $V_t = \Sigma_{i=t}^{i=T} \eta^{i-t} r_i$
        for $t = 1, \cdots, T-1$ do
            $\theta \leftarrow \theta + \alpha \nabla_\theta \log(\pi_\theta(s_t, a_t)) v_t$
        end for
    end for
end function
```

However, no matter which critic function I choose in the vanilla Actor-Critic algorithm, the algorithm cannot converge to a reasonable value. Then I use a more complex model, deep deterministic policy gradient algorithm. The result converges to a good value in finite episodes efficiently.

D. TRPO(Algo. 6)

All of the above algorithms use the gradient descent to optimize the objective function. They are easy to implement but have no guarantee of monotonic converge, which is exactly the unstable fluctuation we have observed.

Algorithm 3 Actor Critic algorithm for discrete action space

```
function ACTOR_CRITIC_DISCRETE Initialize $w$,$\theta$
Initialize the environment.
    for each episode do
        Initialize $s$
        for each step do
            Sample $a \sim \pi_\theta(s, a)$
            $[s_{next},\ r,\ done] \leftarrow$ env.step($a$)
            $\delta = r + V_w(s_{next}) - V_w(s)$
            $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) V_w(s)$
            $w \leftarrow w + \beta\delta\nabla_w(V_w(s))$          ▷ Assume we use L2-norm distance here
            $s \leftarrow s'$
        end for
    end for
end function
```

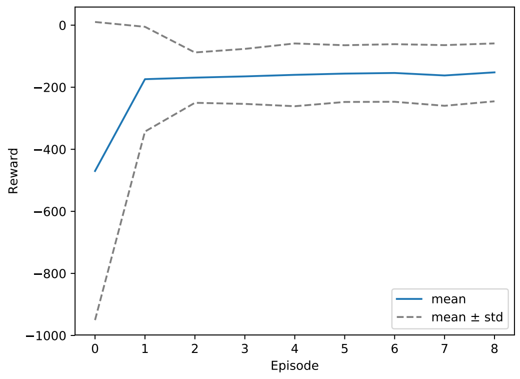


Figure 3: Deep deterministic policy gradient. The solid line is the mean and the dotted line is the mean plus or minus the standard deviation

The trust region policy optimization method provides a monotonic improvement guarantee for general stochastic policies. The hardest part of the implementation is to calculate the conjugate gradient with Hessian-vector products or Fisher-vector product. The Hessian-vector products are easier to implement but slower than Fisher-vector product. I am still working on it.

Algorithm 4 Actor Critic algorithm for continuous action space

```
function ACTOR_CRITIC_CONTINUE
Initialize $w$,$\theta$
Initialize the environment.
    for each episode do
        Initialize $s$
        for each step do
            $\mu, \sigma = f(s)$
            Sample $a \sim N(\mu, \sigma^2)$
            [$s_{next}$, $r$, $done$] $\leftarrow$ env.step($a$)
            $\delta = r + Q_w(s_{next}, a_{next}) - Q_w(s, a)$
            $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$
            $w \leftarrow w + \beta \delta \nabla_w (Q_w(s, a))$        ▷ Assume we use L2-norm distance here
            $s \leftarrow s_{next}$
        end for
    end for
end function
```

III. Thoughts

Reinforcement learning is hard but interesting, sometimes even magical. Even if two algorithms are nearly the same, like SARSA and Q-learning. One is on-policy, the other is off-policy. A very tiny change can improve the performance greatly, for example, comparing to the natural policy gradient methods [2], trust region policy optimization adds a line search at the end of the optimization, the performance improved a lot.

Implementation of these well-developed algorithms helps me fully understand many concepts. I write down some of my thoughts and my understandings to these basic models in this section.

A. On-policy and Off-policy

Difference between SARSA(TD) and Q-learning(TD): The update processes of SARSA-TD is $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s', a') - Q(s,a)]$, the latter one is $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_a' Q(s', a') - Q(s,a)]$. The difference is on how to deal with the Q-function of the next state(s') and the next action(a'). Since SARSA is an on-policy

5 Deep Deterministic Policy Gradient algorithm for continuous action space

```
function DDPG_CONTINUOUS
Initialize critic network $Q(s, a|w)$, actor network$\mu(s|\theta)$
Initialize target critic and target actor network with the same parameters.
Initialize the environment and a random process $N$ for action exploration.
Initialize memory replay buffer R
    for each episode do
        Initialize $s$
        for each step do
            $a = \mu(s_t|\theta) + N_t$
            $[s_{next}, r, done] \leftarrow$ env.step($a$)
            Store transition$(s_t, a_t, r_t, s_{t+1})$ in R
            Sample a random batch of N transitions$(s_i, a_i, r_i, s_{i+1})$ from R
            $\delta = r + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta')|w') - Q(s_i, a_i|w)$
            $w \leftarrow w + \beta \nabla_w (\frac{1}{n} \Sigma \delta_i^2)$        ▷ Assume we use L2-norm distance here
            $\theta \leftarrow \theta + \alpha \nabla_\theta Q(s, \mu(s|\theta))$
            $s \leftarrow s'$
            $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$
            $w \leftarrow \tau w + (1 - \tau)w'$
        end for
    end for
end function
```

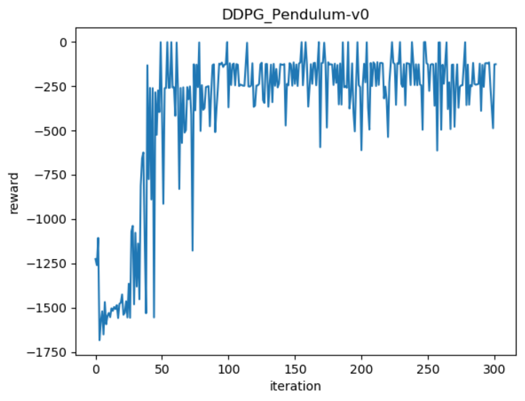
algorithm, we get a' from s' using the same policy as we do to get a from s. Q-learning is an off-policy algorithm, there is no constraints on the policy we use for the next state. The optimal choice is to maximize the $Q(s', a')$ over a'.

From the perspective of implementation, these two algorithms are equivalent when we use the greedy policy to choose the action.

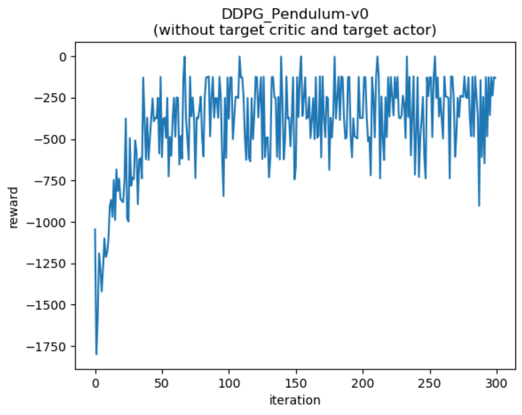
It is a good example to understand on-policy and off-policy model. At the beginning, I didn't understand why SARSA(λ) can also be called as an on-policy algorithm. Now, I know that on-policy doesn't mean we have to update the parameters each step, it just means we estimate the future value based on the policy we use now.

6 Trust Region Policy Gradient Methods

```
function TRPO
Initialize $\theta : \pi_\theta(a|x)$
    for each episode do
        Run the model using $\pi_\theta(a|x)$ to sample several trajectories$\{s_i, a_i, r, s_{i+1}, a_{i+1}\}$
                                                        ▷ Sampling Trajectories
        $\delta = r + \gamma Q(s_{i+1}, a_{i+1}|\theta_{old}) - Q(s_i, a_i|\theta)$
        Use L-BFGS to minimize $\frac{1}{n}\Sigma\sigma_i^2$ over the parameter $\theta$
                                  ▷ Use Q(s,a) to approximate the advantage function
        $g \leftarrow \nabla_\theta \frac{1}{n}\Sigma_i[\frac{\pi_\theta(a_i|s_i)}{\pi_{\theta_{old}}(a_i|s_i)} Q_{\theta_{old}(s_i,a_i)}]$
        Use conjugate gradient method (with Hessian-vector products) to compute
$\theta - \theta_{old} = F^{-1}g$
                  ▷ $D_{KL}(\pi_{\theta_{old}}(\cdot|s)||\pi_\theta(\cdot|s)) \approx \frac{1}{2}(\theta - \theta_{old})^T F(\theta - \theta_{old})$
        Compute rescaled step $s = \alpha(\theta - \theta_{old})$ with line search and line search.
                                        ▷ $D_{KL}(\pi_{\theta_{old}}(\cdot|s)||\pi_\theta(\cdot|s)) \le \delta$
        $\theta \leftarrow \theta + s$
    end for
end function
```



(a)



(b)

Figure 4: (a) DDPG with the target network (b) DDPG without the target critic and the target actor

B. Policy gradient methods on continuous and discrete action space

I begin to pay attention to this point when I use the basic actor-critic algorithm to deal with Cart Pole problem and Pendulum problem. The discrete one is easy to train. For the continuous one, I tried three critic functions; REINFORCE, Q function and TD value function. None of the three can converge and increase the reward consistently. I also tried to change the structure of the neural network, but turn to be effortless.

When we deal with the value function based methods, we can hardly deal with the task with continuous space. When we use policy gradient methods, we can solve both the discrete problem and the continuous problem. What is the difference when we use this algorithm on the continuous action space task and the discrete one?

For the discrete problem, we generate a discrete probability function and sample on these state points, the probability of taking this action is the output of our policy function. For the continuous one, we generate the parameter of the distribution(usually Gaussian) and sampling from it. The probability of taking this action is the density function at this point given the parameter of this distribution. The differences raised from how these two probabilities are generated. But how this difference influence the training process? Can DDPG solve this problem? It worth me to explore and have a deeper understanding in future.

Finally, I give up using the vanilla actor-critic method to deal with the Pendulum-v0 problem and turn to DDPG.

C. Further understanding of DDPG

When people talk about the actor-critic method, they always mention deep deterministic policy gradient method [3][5]. In my former proposal, I think DDPG is just an improvement on the basic actor-critic algorithm. However, when I read the paper and reproduce the experiment, I began to realize that there are many differences between these two algorithms.

The pseudo code of advantage actor-critic(A2C) algorithm and DDPG show many differences between these two. Which part is the most crucial one to the good performance of this model? The deterministic policy? or the target actor and critic network?

Firstly I did two experiments to figure out the function of the target networks. The only difference between these two experiments is whether the target network exists. From (Fig. 4), we can clearly see that the DDPG with the target critic and actor is more stable. But without the target net, the result can converge as well.

Is DDPG really a deterministic model? The authors believe they use a deterministic policy $\mu_\theta(s)$. The action $\mu_\theta(s)$ is indeed deterministic. We still need to add a noisy term, $a \sim \mu_\theta(s) + N_t$. The authors take the N_t as a term for the exploration. In the primal Actor-Critic algorithm, we use the policy network to generate the mean and the variance of the action space $(mean, \sigma) \sim \mu_\theta(s)$. Using a normal distribution, we get $a \sim \sigma^2 N(0,1) + mean$. If $N_t \sim N(0, \sigma^2)$, these two policies are the same. Although the author uses Ornstein–Uhlenbeck process to generate the noise, I can change it to the Gaussian noise, which will not have a huge influence the performance of the whole task. This is verified in my experiments. So I think we shouldn't call this method "deterministic" just because of the policy is deterministic.

What really matters is how to train the "actor" when we get a deterministic policy. The policy gradient of actor-critic is

$$\nabla_\theta J(\pi_\theta) = E_{s\sim\rho^\pi, a\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s,a)] \tag{1}$$

. The policy gradient of DDPG is

$$\nabla_\theta J = E_{s_t\sim\rho^\beta}[\nabla_a Q(s,a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta_\mu} \mu(s|\theta^\mu)|_{s=s_t}] \tag{2}$$

This is the key part that make DDPG perform much better than the origin actor critic method. This also worths me to spend time on in future.

IV. Summary

I list a lot of algorithms and an application in the proposal. I am sorry that I only implement several representative ones. Reinforcement learning is not an easy as I thought before. Since there is so many symbols, details, and variants in an algorithm, only if I truly understand an algorithm, can I implement it. Even if I implement the right code, there still need time to guarantee the converge. Because it is related to probability sampling, it is harder than a common deep learning to tune its hyper-parameters. But the good thing is that many algorithms have their own theoretical supports.

In the implementing process, I indeed learned a lot. Apart from verifying some conclusion in classes, I also know learn some new methods and familiar with some tricks of tuning the parameters and organizing the network structure.

V. Instructor's Effort

My instructor is Siyang Yuan.

We discuss some details of the implementation together. When I have some doubts about my code, I will ask her opinion.

[1] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.

[2] S. M. Kakade. A natural policy gradient. In *Advances in neural information processing systems*, pages 1531–1538, 2002.

[3] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[4] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 1889–1897, 2015.

[5] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 387–395, 2014.

[6] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[7] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.