The Jiangming's reading notes for the book *Reinforcement Learning: An introduction*, Richard S. Sutton and Andrew G. Barto (2015)

Contents

Background and History

The basic idea is simply to capture the most important aspects of the real problem facing a learning agent interacting with its environment to achieve a goal. The formulation of the problems that can be addressed by reinforcement learning must be able to **sense** the state of the environment, must be able to take **actions** that affect the state, and must have **a goal or goals** relating to the state of the environment.

RL vs supervised learning Supervised learning is learning from a training set of labeled examples provided by a knowledgable external supervisor. The object of this kind of learning is for the system to extrapolate, or generalize, its responses so that it acts correctly in situations not present in the training set. Different from supervised learning, reinforcement learning focuses on interative problem that is often impractical to obtain examples of desired behavior that are both correct and representative of all the situations in which the agent has to act.

RL vs unsupervised learning Unsupervised learning is about finding structure hidden in collections of unlabeled data. Different from it, reinforcement learning is trying to maximize a reward signal instead of trying to find hidden structure. But uncovering structure in an agent's experience indeed can be useful in reinforcement learning.

Therefore, reinforcement learning is considered to be a third machine learning paradigm, alongside of supervised learning, unsupervised learning, and perhaps other paradigms as well. There are several key point in reinforcement learning.

- The main challenge is to balance between exploration and exploitation. The agent has to exploit what it already knows in order to obtain reward, but it also has to explore in order to make better action selections in the future.
- It explicitly considers the whole problem of a goal-directed agent interacting with an uncertain environment. Facing the uncertainty about the environment, it can involve planning or supervised learning.
- Of all the forms of machine learning, reinforcement learning is the closest to the kind of learning that humans and other animals do, and many of the core algorithms of reinforcement learning were originally inspired by biological learning systems.

Reinforcement learning should have a *policy*, a *reward signal*, a *value function*, optionally, a *model* of the environment. A policy is a mapping from perceived states of the environment to actions to be taken when in those states. A reward signal defines the goal in a reinforcement learning problem. On each time step, the environment sends to the reinforcement learning agent a single number, a reward. The agent's sole objective is to maximize the total reward it receives over the long run. The value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state.

For the optional element, model. This is something that mimics the behavior of the environment, or more generally, that allows inferences to be made about how the environment will behave (e.g., transitions from current state to other states). Models are used for planning. Methods for solving reinforcement learning problems that use models and planning are called model-based methods, as opposed to simpler model- free methods that are explicitly trial-and-error learners. Modern reinforcement learning spans the spectrum from low-level, trial-and-error learning to high-level, deliberative planning.

The history of RL has two main thread. One thread concerns learning by trial and error that started in the psychology of animal learning. The other thread concerns the problem of optimal control and its solution using value functions and dynamic programming. A third, less distinct thread concerns temporal-difference methods.

Optimal control with value function and dynamic programming The term *optimal control* is firstly used in late 1950s to describe the problem of designing a controler to minimize a loss of a dynamical system's behavior over time. Bellman propose the concept of states and value function called ***Bellam Equaltion***. Bellman (1957) also introduced Markov Decision Processes (MDP). Howard (1960) devised policy iteration method for MDP. Dynamic programming is widely considered the only feasible way of solving general stochastic optimal control problems (the curse of dimensionality). Many works try to avoid exhaustive computing. Watkins (1989) is the first to use on-line learning for dynamic programming and formalize reinforcement learning with MDP. And then Bertsekas and Tsitsiklis (1996) combined dynamic programming with neural networks.

Trial-and-error learning The idea of trial-and-error learning can go back to the 1850s of Alexander Bain's discussion of learning by "groping and experiment" and more explicitly to the 1894 use of the term to describe Conway Lloyd Morgan's observation of animal behavior. The first success to express the essence of trial-and-error learning as a principle of leraning was Edward Thorndike and his ***Law of Effect***. In the Law of Effect, reinforcement is the strengthening of a pattern of behavior as a result of an animal receiving a stimulus — a reinforcer — in an appropriate temporal relationship with another stimulus or with a response.

When the interest of trial-and-error learning is shifted to generalization and pattern recognition (from RL to supervised learning), the distinction between them become confusing. So in 1960s and 1970a, research on trial-and-error learning becomes rare.

Temporal-difference learning Temporal-difference methods seem to be new and unique to reinforcement learning. The origins of temporal-difference learning are in part in animal learning psychology, in particular, in the notion of secondary reinforcers. Samuel (1959) was the first to propose and implement a learning method that included temporal-difference ideas. Instead from animal learning (Minsky's work), his inspiration comes from Shannon (1950)'s suggestion that a computer could be programmed to use an evaluation function.

In 1972, Klopf brought trial-and-error learning together with an important component of temporal-difference learning. By developing Klopf's work, Sutton (1985) extensively study the ***actor-critic architecture***. Then, The temporal-difference and optimal control threads were fully brought together in 1989 with Chris Watkins's development of Q-learning.

Multi-arm Bandits

The n-armed bandit problem is a nonassociative, evaluative feedback problem. You are faced repeatedly with a choice among n different options, or actions. After each choice you receive a numerical reward chosen from a stationary probability distribution that depends on the action you selected. Your objective is to maximize the expected total reward over some time period.

If you select a greedy action, we say that you are exploiting your current knowledge of the values of the actions. If instead you select one of the nongreedy actions, then we say you are exploring. Exploitation is the right thing to do to maximize the expected reward on the one step, but exploration may produce the greater total reward in the long run. **The key problem is how to balance them**.

Action-Value Methods we denote the true (actual) value of action a as $q(a)$, and the estimated value on the tth time step as $Q_t(a)$.

$$Q_t(a) = \frac{R_1 + R_2 + ... + R_{N_t(a)}}{N_t(a)}. \tag{1}$$

The *greedy* action selection method is $A_t = \arg\max_a Q_t(a)$, which always exploits current knowledge. We can use ϵ-greedy method for exploration, which take the $\arg\max$ with $1-\epsilon$ probability while random action with ϵ probability.

Incremental Implementation For one action, the estimated average rewards can be computed incrementally by

$$Q_{k+1} = \frac{1}{k}\sum_{i=1}^{k} R_i = Q_k + \frac{1}{k}[R_k - Q_k], \tag{2}$$

where Q_{k+1} is the average rewards on the first k samples, and Q_1 is the initialized value without any sample. The general form of updating is:

$$Q_{k+1}(a) = Q_k(a) + \alpha_k(a)[R_k(a) - Q_k(a)], \tag{3}$$

where $\alpha_k(a) \in (0, 1]$. If we expand the equation 3:

$$Q_{k+1}(a) = (1-\alpha_k(a))^k Q_1(a) + \sum_{i=1}^{k} \alpha_k(a)(1-\alpha_k(a))^{k-i} R_i, \tag{4}$$

where $(1-\alpha_k(a))^k + \sum_{i=1}^{k} \alpha_k(a)(1-\alpha_k(a))^{k-i} = 1$, so the value is the weighted average. The $\alpha_k(a)$ should satisfy $\sum_{k=1}^{\infty} \alpha_k(a) = \infty$ and $\sum_{k=1}^{\infty} \alpha_k(a)^2 < \infty$ to make. The first condition is required to guarantee that the steps are large enough to eventually overcome any initial conditions or random fluctuations. The second condition guarantees that eventually the steps become small enough to assure convergence. In the language of statistics, these methods are *biased* by their initial estimates.

Upper-Confidence-Bound Action Selection The ϵ-greedy action selection force the non-greedy actions to be tried. It had better consider the action potential for the optimization. Upper-Confidence-Bound Action Selection (UCB) is:

$$A_t = \arg\max_a [Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}}] \tag{5}$$

Gradient Bandits We define $H_t(a)$ for each action a, showing the preference of the action to be taken.

$$Pr\{A_t = a\} = \frac{e^{H_t(a)}}{\sum_{b=1}^{n} e^{H_t(b)}} = \pi(a) \tag{6}$$

. If we sample the action A_t with the reward R_t, the preference are updated by:

$$\begin{aligned} H_{t+1}(A_t) &= H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t)) \\ H_{t+1}(a) &= H_t(a) - \alpha(R_t - \bar{R}_t)\pi_t(a) \quad \forall a \neq A_t, \end{aligned} \tag{7}$$

where $\bar{R}_t$ is the average rewards to now (t). If the current reward is higher then the average reward, we increase the preference of action A_t and decrease others. The deep insight is:

$$H_{t+1}(a) = H_t(a) + \alpha \frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)}. \tag{8}$$

The prove is shown in pages 43–46 of the original article.

Finite Markov Decision Process

The setting reinforcement learning is that we have an *agent* and an *environment*, and we want to achieve an predefined goal in the environment by multiple interaction between the agent an the environment. The interactions are taken in a sequence of discrete time steps. At each time step t, the agent receives some representation of the environment's state, $S_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of possible states, and on that basis selects an action, $A_t \in \mathcal{A}(S_t)$, where $\mathcal{A}(S_t)$ is the set of actions available in state S_t. One time step later, in part as a consequence of its action, the agent receives a numerical reward, $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, and finds itself in a new state, S_{t+1}. For the ***boundary*** between agent and environment, anything that cannot be changed arbitrarily by the agent is considered to be outside of it and thus part of its environment. Reinforcement learning methods specify how the agent changes its policy $\pi_t(a|s)$ as a result of its experience, where $\pi_t(a|s)$ is the probability that $A_t = a$ if $S_t = s$.

The agent's goal is to maximize the cumulative reward it receives in the long run:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T. \tag{9}$$

A subsequence of an interaction is called an ***episode***, which has a ***terminal state***. On the other hand, in many cases the agent–environment interaction does not break naturally into identifiable episodes, but goes on continually without limit, i.e., $T = \infty$, which we call ***continuing tasks***. The additional concept that we need is that of ***discounting***:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{10}$$

where $0 \leq \gamma \leq 1$ called ***discount rate***

Certainly the state signal should include immediate sensations such as sensory measurements, but it can contain much more than that. Is it necessary to efficiently represent the whole environment, or does the whole environment including the whole history can significantly help the reinforcement learning?. On the other hand, the state signal should not be expected to inform the agent of everything about the environment, or even everything that would be useful to it in making decisions. What we would like, ideally, is a state signal that summarizes past sensations compactly, yet in such a way that all relevant information is retained, which is said ***Markov***. Move:

$$\Pr\{R_{t+1} = r, S_{t+1} = s' | S_0, A_0, R_1, ..., S_{t-1}, A_{t-1}, R_t, S_t, A_t\} \tag{11}$$

to

$$p(s', r|s, a) = \Pr\{R_{t+1} = r, S_{t+1} = s' | S_t, A_t\}. \tag{12}$$

This is the definition of finite **Markov Decision Process (MDP)**. More details, the expected rewards for state–action pairs are:

$$r(s, a) = \mathbb{E}[R_{t+1} | S_t = s, A_t = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a), \tag{13}$$

the state-transition probabilities are:

$$p(s'|s, a) = \sum_{r \in \mathcal{R}} p(s', r|s, a), \tag{14}$$

the expected rewards for state–action–next-state triples:

$$r(s, a, s') = \mathbb{E}[R_{t+1} | S_t = s, A_t = a, S_{t+1} = s'] = \sum_{r \in \mathcal{R}} r p(r|s, a, s') = \sum_{r \in \mathcal{R}} r \frac{p(s', r|s, a)}{p(s'|s, a)}. \tag{15}$$

Value Functions

In addition, If we have a policy π, we can define the value function on **the policy**. One is the value function on states that estimate how good it is for the agent to be in a given state:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s]. \tag{16}$$

Similarly, the other is the value function on state-action pairs that estimate how good it is for the agent to perform an action in a given state:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a]. \tag{17}$$

Using the dynamic programming method, the bellman equations of the state value function is:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]. \tag{18}$$

Intuitively, for the current state s, we can take all possible actions a under the policy π. For each possible action, we enumerate all possible reachable s' and its reward r with the probability $p(s', r|s, a)$. And then, we should sum up all possible values to get expectations from the reachable state s' plus r.

Optimal Value Functions

Our goal is to find a policy that achieves a lot of reward over the long run, i.e., for all $s \in \mathcal{S}$,

$$v_*(s) = \text{max}_\pi v_\pi(s) \tag{19}$$

and for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

$$q_*(s, a) = \text{max}_\pi q_\pi(s, a). \tag{20}$$

In particular, $q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a]$. Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems. Because the value function satify the bellman equations, so we can use DP to solve the optimization problem by:

$$\begin{aligned} v_*(s) &= \text{max}_{a \in \mathcal{A}(s)} q_*(s, a) \\ &= \text{max}_{a \in \mathcal{A}(s)} \mathbb{E}_{\pi^*}[Q_t|S_t = s, A_t = a] \\ &= \text{max}_{a \in \mathcal{A}(s)} \mathbb{E}_{\pi^*}[\textstyle\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \\ &= \text{max}_{a \in \mathcal{A}(s)} \mathbb{E}_{\pi^*}[R_{t+1} + \gamma \textstyle\sum_{k=0}^{\infty} \gamma^k R_{t+k+2}|S_t = s, A_t = a] \\ &= \text{max}_{a \in \mathcal{A}(s)} \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a] \\ &= \text{max}_{a \in \mathcal{A}(s)} \textstyle\sum_{s',r} p(s', r|s, a)[r + v_*(s')] \end{aligned} \tag{21}$$

If we use v_* to find the optimal policy, we have to do one-step search. The beauty of v_* is that if one use it to evaluate short-term consequences of actions, the greedy policy is actually optimal in long run.

$$q_*(s,a) \quad = \textstyle\sum_{s',r} p(s',r|s,a)[r + \gamma \mathrm{max}_{a'} q_*(s',a')] \tag{22}$$

If we use q_* to find the optimal policy, in state s, we can simply use the action a that maximizes $q_*(s,a)$. For the optimal value functions, if the dynamics of the environment are known ($p(s',r|s,a)$), then in principle one can solve this system of equations.

Dynamic Programming

If we have a policy π, how to compute the value function for the policy. The value function computation is called ***policy evaluation***. Suppose we know the environment, i.e., the environment can provide the probability distribution $p(s',r|s,a)$. Recall that

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]. \tag{23}$$

Based on the Bellman equation, the value functions can be approximated by iterative updating from the random initialization v_0:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_k(s')]. \tag{24}$$

If the $k \to \infty$, the $\{v_k\}$ can converge to v_π. The algorithm is called ***iterative policy evaluation***.

How to get better policy

Let π and π' be any pair of deterministic policies (Noted that a deterministic policy should output only one action given a state instead of action distributions. Given a action distribution $p(a|s)$, we can obtain a deterministic policy by always choosing the action that has highest probability.) such that, for all $s \in \mathcal{S}$,

$$q_\pi(s,\pi'(s)) \geq v_\pi(s) = q_\pi(s,\pi(s)) \tag{25}$$

Then the policy π' must be as good as, or better than, π. That is for all $s \in \mathcal{S}$,

$$v_{\pi'}(s) \geq v_\pi(s). \tag{26}$$

There is a simple proof for any state s:

$$\begin{aligned} v_\pi(s) \quad &\leq q_\pi(s,\pi'(s)) \\ &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1},\pi'(S_{t+1}))|S_t = s] \\ &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma \mathbb{E}_{\pi'}[R_{t+2} + \gamma v_\pi(S_{t+2})]|S_t = s] \\ &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2})|S_t = s] \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ...|S_t = s] \\ &= v_{\pi'}(s). \end{aligned} \tag{27}$$

This inspire us that if we want to obtain a better policy π', we should always choose the action $a = \pi'(s) = \arg\max_a q_\pi(s,a)$. In this way, the policy can be interatively improved:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} v_{\pi_2} ... \xrightarrow{I} \pi_* \xrightarrow{E} v_* \tag{28}$$

However, $\xrightarrow{E}$ requires iterative approximation, which make the whole improvement really slow. In fact, policy iteration ($\xrightarrow{E}$) can be truncated without losing the convergence guarantees of policy iteration. We changes

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_k(s')] \tag{29}$$

to

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_k(s')]. \tag{30}$$

In this way, we can make the optimization (28) shorter.

****

The interaction between a agent with the environment can be regarded as a Markov Decision Processing. After the interaction, there will be a trajectory $\tau = [s_1, a_1, s_2, a_2, ..., s_T, a_T]$ and a reward $R(\tau)$. With the assumption that the reward is accumulated for each step reward r_t, $R(\tau) = \sum_{t=1}^{T} r_t$. Given a state s_t, the agent (actor) π will output an action a_t in each step, i.e., $a_t = \pi(s_t)$.

In general setting, the way to define the reward is given according to our goals. The question is how to obtain the optimal agent (actor) π_{θ^*} that can maximize the expectation of the rewards over all the possible trajectory:

$$\bar{R}_{\theta^*} = \sum_\tau p_{\theta^*}(\tau) R(\tau), \tag{31}$$

where $p_\theta(\tau)$ is the probability of a trajectory τ given by a model parameterized by θ over all the possible trajectories.

Policy Gradient

How to update the θ to maximize the reward $\bar{R}_\theta$. Policy Gradient directly optimize the policy π_θ that outputs a action a_t given a state s_t, by gradient ascent $\theta \leftarrow \theta + \lambda \nabla_\theta \bar{R}_\theta$. The policy gradient is

$$\begin{aligned}
\nabla_\theta \bar{R}_\theta &= \textstyle\sum_\tau R(\tau) \nabla_\theta p_\theta(\tau) \\
&= \textstyle\sum_\tau R(\tau) p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) \\
&= \mathbb{E}_{\tau \sim p_\theta(\tau)}[R(\tau) \nabla_\theta \log p_\theta(\tau)] \\
&\approx \tfrac{1}{N} \textstyle\sum_{n=1}^{N} R(\tau^n) \nabla_\theta \log p_\theta(\tau^n) \\
&= \tfrac{1}{N} \textstyle\sum_{n=1}^{N} \sum_{t=1}^{T_n} R(\tau^n) \nabla_\theta \log p_\theta(a_t^n | s_t^n)
\end{aligned} \tag{32}$$

There is an assumption that action is outputted according to the current state. The general training process is that we can use the current model θ to generate many samples (s_t, a_t) and rewards $R(\tau)$ to update the model.

Policy Gradient VS Classification The policy gradient is similar to classification that directly train a model output a action for the given state, $\frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n}\nabla_\theta \log p_\theta(a_t^n|s_t^n)$. The only difference is that we have global rewards $R(\tau^n)$ on the whole trajectory τ^n in policy gradient. The concerned thing is the rewards. There are some problems, by addressing which, we can make model robust.

- Problem 1 is that sometimes, the rewards are always positive, so we add a baseline to make the reward negative if it is below the baseline

$$\nabla_\theta \bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n}[R(\tau^n) - b]\nabla_\theta \log p_\theta(a_t^n|s_t^n) \tag{33}$$

- Problem 2 is that the reward $R(\tau)$ is the total reward of whole trajectory τ, and sometimes one action is bad but the total reward is positive or one action is good but the total reward is negative. So we change $R(\tau^n) = \sum_{t=1}^{T_n} r_t^n$ to be the reward from the current state to the end $\sum_{t'=t}^{T_n} r_{t'}^n$:

$$\nabla_\theta \bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n}[\sum_{t'=t}^{T_n} r_{t'}^n - b]\nabla_\theta \log p_\theta(a_t^n|s_t^n) \tag{34}$$

- Problem 3 is that the future rewards are not that important (empirically), so we add a decay to the future rewards, changing $\sum_{t'=t}^{T_n} r_{t'}^n$ to $\sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n$ where $\gamma < 1$:

$$\nabla_\theta \bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n}[\sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n - b]\nabla_\theta \log p_\theta(a_t^n|s_t^n) \tag{35}$$

  usually, the baseline b is predicted by a neural network, and $R(\tau^n)$ is estimated by $A^\theta(s_t, a_t)$ that compute the future rewards using the model θ.

Q-learning

Different from policy gradient that aim to directly learn a policy (agent or actor) π, Q-learning is value-based method that aims to learn a critic to evaluate how good the current state is. *Firstly*, we define a state-value function $V^\pi(s_t)$ as the critic to evaluate how good the actor π in the current s_t, i.e., the rewards from current state to the end. For example, V^π is a neural network where the input is state s and the output is a scalar (score) to say how many scores we will obtained after visiting s. *Then*, we define action-value function $Q^\pi(s, a)$ that return the rewards to the end state after taking action a at state s.

$$V^\pi(s_t) = \mathbb{E}_{a_t \sim \pi(a|s_t)}[Q^\pi(s_t, a_t)] = \sum_a \pi(a_t|s_t) Q^\pi(s_t, a_t) \tag{36}$$

The reason why we use Q function is that sometimes the action space is too large, and we use sampling methods to solve the problem. If we have the learned $Q^\pi(s, a)$, we take the action $a^* = arg\ \max_a Q^\pi(s, a)$ for every states. There are two method to estimate $V^\pi(s)$ or $Q^\pi(s, a)$.

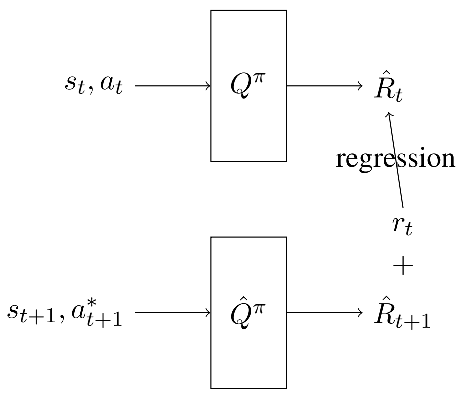

Figure 1: TD training on Q-learning.

- Exhaustive search: MC-based approaches. We compute the reward from the current state s_t to the end, i.e., $G_t(s_t) = \sum_t^T \sum_a r_t$, and then regress it to $V^\pi(s_t)$, or we compute the reward from the current state s_t with action a_t to the end, i.e., $G_t(s_t, a_t) = \sum_t r_t$, and then regress it to $Q^\pi(s_t, a_t)$.

- Dynamic programming: Temporal-difference (TD) approaches. We don't have to play to the end, and given a sample s_t, a_t, r_t, s_{t+1}, $V^\pi(s_t) - V^\pi(s_{t+1}) = \sum_a r_t$, and $Q^\pi(s_t, a_t) = r_t + Q^\pi(s_{t+1}, \max(\pi(s_{t+1})))$. For example, we can use neural network to obtain $V^\pi(s_t)$ and $V^\pi(s_{t+1})$, and then, we can update the value function by regression to the difference. MC methods have high variance while TD methods might be inaccurate.

Figure 1 shows the TD training on Q-learning, But there are some problems.

- If we use TD to train model, we have to do regression both on the $Q^\pi(s_t, a_t) = r_t + Q^\pi(s_{t+1}, \pi(s_{t+1}))$ and $Q^\pi(s_{t+1}, \pi(s_{t+1})) = Q^\pi(s_t, a_t) - r_t$. The both side of the equation are changed that makes the training hard and not converged. We fix one value as target and only update Q function by regression on one, i.e., $Q^\pi(s_t, a_t) = r_t + Q^\pi(s_{t+1}, \pi(s_{t+1}))$ or $Q^\pi(s_{t+1}, \pi(s_{t+1})) = Q^\pi(s_t, a_t) - r_t$ but not both. So at the beginning of the training, we will copy Q to $\hat{Q}$ that is fixed to produce target, and Q is updated. After several round updating, we will copy the updated Q to $\hat{Q}$ and continue training.

- The Q-learning heavily depends on sampling. If states and actions are not sampled (not seen), we will always not make the unseen action. So if we do sampling, we have to take more explorations. 1) Epsilon Greedy. $a = arg \max_a Q(s, a)$ with a probability $1 - \epsilon$, otherwise a is random picked. 2) Boltzmann Exploration. $p(a|s) = \frac{exp(Q(s,a))}{\sum_a exp(Q(s,a))}$.

- For one actor π, the samples are not much different. So we build a buffer to store the samples from different π, and replay (train) Q function by picking a batch from the buffer (This trick can make training samples diverse that similar to off-policy).

Algorithm 1 Q-learning algorithm

```
1: We initialize $Q$
2: repeat
3:     Sample ($s_t$, $a_t$, $r_t$, $s_{t+1}$) based on $Q$
4:     $y = r_t + \max_a Q(s_{t+1}, a)$
5:     update $Q$ to make $Q(s_t, a_t)$ close to $y$
6: until end
```

The original training of Q is shown in Algorithm **??**. If we consider the above 3 problems, the training is shown in Algorithm 2.

Algorithm 2 Q-learning algorithm with considering the three problems

```
1: We initialize $Q = \hat{Q}$
2: repeat
3:     Sample ($s_t$, $a_t$, $r_t$, $s_{t+1}$) based on $Q$ (epsilon greedy trick)
4:     Store ($s_t$, $a_t$, $r_t$, $s_{t+1}$) to buffer
5:     Sample ($s_i$, $a_i$, $r_i$, $s_{i+1}$) from buffer (replay trick)
6:     $y = r_i + \max_a \hat{Q}(s_{i+1}, a)$
7:     update $Q$ to make $Q(s_i, a_i)$ close to $y$ (target network trick)
8:     every $C$ steps reset $\hat{Q} = Q$
9: until end
```

Actor-Critic

We have the policy gradient method and Q-learning method. We can combine the two method together. The gradient of the rewards in policy gradient is:

$$\nabla_\theta \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T_n} [\sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n - b] \nabla_\theta \log p_\theta(a_t^n | s_t^n) \tag{37}$$

where $\sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n$ is the output of th Q function according to the definitions, i.e.,

$$Q^{\pi_\theta}(s_t, a_t) = \sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n, \tag{38}$$

and the baseline b aims to make the step reward can be positive and negative. So b should be the means over all possible values of $Q(s_t, a_t)$, i.e,

$$V^{\pi_\theta}(s_t) = b \tag{39}$$

So the gradient of the rewards in actor-critic is:

$$\nabla_\theta \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T_n} [Q^{\pi_\theta}(s_t^n, a_t^n) - V^{\pi_\theta}(s_t^n)] \nabla_\theta \log p_\theta(a_t^n | s_t^n). \tag{40}$$

Because $Q^{\pi_\theta}(s_t, a_t) = E[r_t + V^{\pi_\theta}(s_{t+1})]$, the above equation can be rewritten as:

$$\nabla_\theta \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T_n} [r_t^n + V^{\pi_\theta}(s_{t+1}^n) - V^{\pi_\theta}(s_t^n)] \nabla_\theta \log p_\theta(a_t^n | s_t^n). \tag{41}$$

The useful tricks are 1) V function network and π network can share the parts of parameters, 2) exploration on smooth π. Asynchronous A2C (A3C) can calculate gradients on multiple CPU/GPU.

Pairwise Derivative Policy Gradient When we want to learn Q function using TD method, we have to regress the value on the best action given by $Q(s, a)$, $y = r_i + \max_a \hat{Q}(s_{i+1}, a)$. The argmax problem $a^* = arg\max Q(s, a)$ during gradient update, particular in continuous action space. We can use GAN similar method to build a actor as generator to generate action, and make Q function as discriminator to ensure the action is the best action. The Q-learning is modified to the Algorithm 3. Figure 2 shows the Actor-Critic model. The bottom structure of AC model is similar to a GAN. If we fix π, i.e., say π can always outputs the optimal action a, we can use Q-learning to update Q function. If we fix Q function, i.e., say Q can always give the optimal action a with the highest score, we can use the score to update π.

Algorithm 3 Actor-Critic algorithm

```
1:  We initialize $Q = \hat{Q}$, $\pi = \hat{\pi}$
2:  repeat
3:      Sample ($s_t$, $a_t$, $r_t$, $s_{t+1}$) based on $\pi$ (epsilon greedy trick)
4:      Store ($s_t$, $a_t$, $r_t$, $s_{t+1}$) to buffer
5:      Sample ($s_i$, $a_i$, $r_i$, $s_{i+1}$) from buffer (replay trick)
6:      $y = r_i +$ ~~$\max_a \hat{Q}(s_{i+1}, a)$~~ $\hat{Q}(s_{i+1}, \hat{\pi}(s_{i+1}))$
7:      update $Q$ to make $Q(s_i, a_i)$ close to $y$ (target network trick)
8:      update $\pi$ to maximize $Q(s_i, \pi(s_i))$
9:      every $C$ steps reset $\hat{Q} = Q$, $\hat{\pi} = \pi$
10: until end
```

Off-policy

In previous sections, we actually use the similar off-policy idea that we will fixed actor to interact with the environment (sampling). The general difference between on-policy and off-policy is that on-policy use the learned θ to interact with environment to obtain samples to train θ, while off-policy uses the another fixed θ' to obtain samples to train θ. In order to address the off-policy, we use the importance sampling strategy.

$$\mathbb{E}_{x\sim p(x)}[f(x)] = \mathbb{E}_{x\sim q(x)}[\frac{p(x)}{q(x)} f(x)] \tag{42}$$

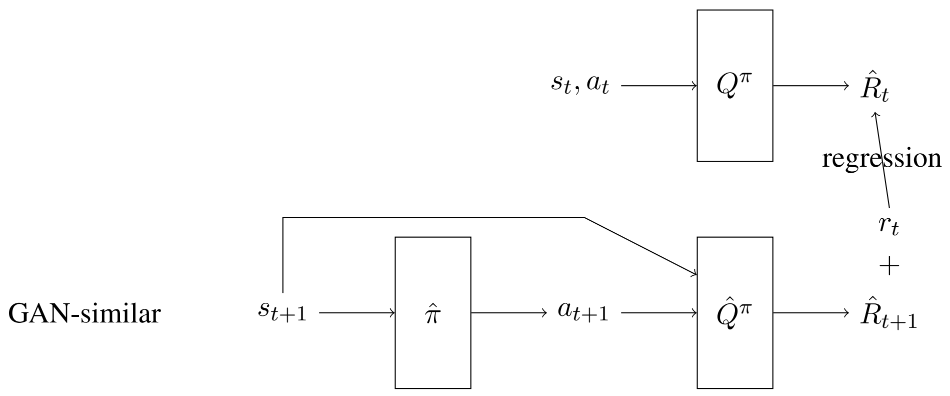

Figure 2: TD training on Actor-Critic.

Noted that the expectation is the same but the variances are different, i.e. $Var[f(x)] \neq Var[\frac{p(x)}{q(x)}f(x)]$. So the off-policy gradient is

$$\begin{aligned}\nabla_\theta \bar{R}_\theta &= \mathbb{E}_{\tau \sim p_\theta(\tau)}[R(\tau)\nabla_\theta \log p_\theta(\tau)] \\ &= \mathbb{E}_{\tau \sim p_{\theta'}(\tau)}[\frac{p_\theta(\tau)}{p_{\theta'}(\tau)}R(\tau)\nabla_\theta \log p_\theta(\tau)]\end{aligned} \tag{43}$$

If the action only depends on currents state, the gradient can be rewritten as:

$$\begin{aligned}\nabla_\theta \bar{R}_\theta &= \mathbb{E}_{(s_t,a_t)\sim\pi_\theta}[A^\theta(s_t,a_t)\nabla_\theta \log p_\theta(a_t|s_t)] \\ &= \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta'}}[\frac{p_\theta(s_t,a_t)}{p_{\theta'}(s_t,a_t)}A^{\theta'}(s_t,a_t)\nabla_\theta \log p_\theta(a_t|s_t)] \\ &= \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta'}}[\frac{p_\theta(a_t|s_t)p_\theta(s_t)}{p_{\theta'}(a_t|s_t)p_{\theta'}(s_t)}A^{\theta'}(s_t,a_t)\nabla_\theta \log p_\theta(a_t|s_t)] \\ &= \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta'}}[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t)\nabla_\theta \log p_\theta(a_t|s_t)] \quad p_\theta(s_t)\approx p_{\theta'}(s_t)\end{aligned} \tag{44}$$

So the objective function is total reward:

$$J(\theta) = \bar{R}_\theta = \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta'}}[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t)] \tag{45}$$

The off-policy is based on importance sampling, so we have to make θ and θ' close addressing variance issue. So the objective function should add regularization term to make the θ and θ' close:

$$J_{PPO}(\theta) = \bar{R}_\theta = \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta'}}[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t)] - \beta KL(\theta,\theta') \tag{46}$$

where $KL(\theta,\theta')$ makes the constraints on behaviors (actions a_t) not on parameters, which is called Proximal Policy Optimization (PPO) or Trust Region Policy Optimization (TRPO). The PPO/TRPO algorithm is shown in Algorithm 4. There is a improved PPO algorithm called PPO2 by removing KL term with cliping strategy:

$$J_{PPO2}(\theta) = \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta'}}[\min(\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t), clip(\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}, 1-\epsilon, 1+\epsilon)A^{\theta'}(s_t,a_t))] \tag{47}$$

where $clip(\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}, 1-\epsilon, 1+\epsilon)$ means if $\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)} < 1-\epsilon$, return $1-\epsilon$ and if $\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)} > 1+\epsilon$, return $1+\epsilon$. Intuitively

Algorithm 4 PPO/TRPO algorithm

```
1: using $\theta'$ to interact with the environment to collect $s_t$, $a_t$ and compute $A^{\theta'}(s_t, a_t)$
2: optimizing $J_{PPO}(\theta) = \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta'}}[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)} A^{\theta'}(s_t, a_t)] - \beta KL(\theta, \theta')$
3: if $KL(\theta, \theta') > KL_{max}$, increase $\beta$
4: if $KL(\theta, \theta') < KL_{min}$, decrease $\beta$
```

- if $A^{\theta'}(s_t, a_t)$ positive and a_t is good, so we have to make $p_\theta(a_t|s_t)$ larger. If $p_\theta(a_t|s_t)$ is too small and $\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}$ is smaller than $1 - \epsilon$, we use $1 - \epsilon$ instead, but it cannot be larger than $1 + \epsilon$
- $A^{\theta'}(s_t, a_t)$ negative and a_t is bad, so we have to make $p_\theta(a_t|s_t)$ smaller. If If $p_\theta(a_t|s_t)$ is too larger and $\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}$ is larger than $1 + \epsilon$, we use $1 + \epsilon$ instead, but it cannot be smaller than $1 - \epsilon$

Related Works

Robot control tasks (Levine et al., 2016; Watter et al., 2015), image/video recognition (Mnih et al., 2014; Ba et al., 2015), games (Mnih et al., 2015; Maddison et al., 2015; Silver et al., 2016).

Double Q-learning

Hasselt (2010) proved that a double estimator can soft the biased estimation by single estimator, where the single estimator will overestimate the rewards, and the double estimator will underestimate the rewards. The training is shown in Algorithm 5. The Double Q-learning

Algorithm 5 Double Q-learning

```
1: We initialize $Q^A, Q^B$
2: repeat
3:     Sample ($s_t$, $a_t$, $r_t$, $s_{t+1}$) based on $Q^A$ and $Q^B$
4:     Sample $c$ from {A, B}
5:     if $c = A$ then
6:         $a^* = \arg\max_a Q^A(s_{t+1}, a)$
7:         $y = r_i + Q^B(s_{i+1}, a^*)$
8:         update $Q^A$ to make $Q^A(s_i, a_i)$ close to $y$
9:     else
10:        $a^* = \arg\max_a Q^B(s_{t+1}, a)$
11:        $y = r_i + Q^A(s_{i+1}, a^*)$
12:        update $Q^B$ to make $Q^B(s_i, a_i)$ close to $y$
13:    end if
14: until end
```

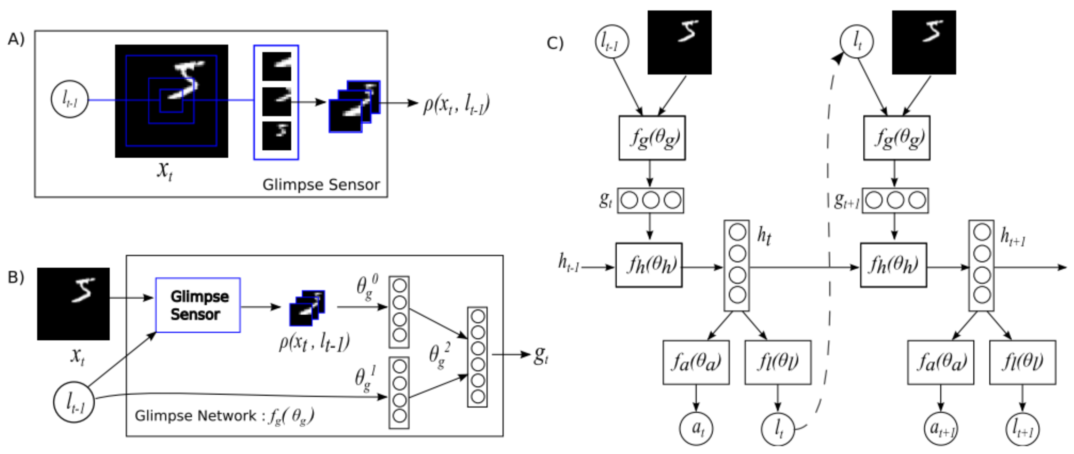

Figure 3: RAM from the original paper (Mnih et al., 2014). **A) Glimpse Sensor:** Given the coordinates of the glimpse and an input image, the sensor extracts a retina-like representation $\rho(x_t, l_{t-1})$ centered at l_{t-1} that contains multiple resolution patches. **B) Glimpse Network:** Given the location (l_{t-1}) and input image x_t, uses the glimpse sensor to extract retina representation $\rho(x_t, l_{t-1})$. The retina representation and glimpse location is then mapped into a hidden space using independent linear layers parameterized by θ_g^0 and θ_g^1 respectively using rectified units followed by another linear layer θ_g^2 to combine the information from both components. The glimpse network $f_g(.; \theta_g^0, \theta_g^1, \theta_g^2)$ defines a trainable bandwidth limited sensor for the attention network producing the glimpse representation g_t. **C) Model Architecture:** Overall, the model is an RNN. The core network of the model $f_h(.; \theta_h)$ takes the glimpse representation g_t as input and combining with the internal representation at previous time step h_{t-1}, produces the new internal state of the model ht. The location network $f_l(.; \theta_l)$ and the action network $f_a(.; \theta_a)$ use the internal state h_t of the model to produce the next location to attend to l_t and the action/classification at respectively. This basic RNN iteration is repeated for a variable number of steps.

can be interpreted as a kind of off-policy strategy. Hasselt et al. (2016) show that the overestimation is caused by the error estimation. In the original double Q-learning algorithm, two value functions are learned by assigning each experience randomly to update one of the two value functions. Hasselt et al. (2016) modify deep Q-network (DQN; Mnih et al. 2015) with double Q-learning. The method is similar to the target network trick mentioned above. The training is shown in Algorithm 6.

Algorithm 6 Double DQN

```
1: We initialize $Q = \hat{Q}$
2: **repeat**
3:     Sample $(s_t, a_t, r_t, s_{t+1})$ based on $Q$
4:     $y = r_t + \max_a \hat{Q}(s_{t+1}, a)$
5:     update $Q$ to make $Q(s_t, a_t)$ close to $y$
6:     every $C$ steps reset $\hat{Q} = Q$
7: **until** end
```

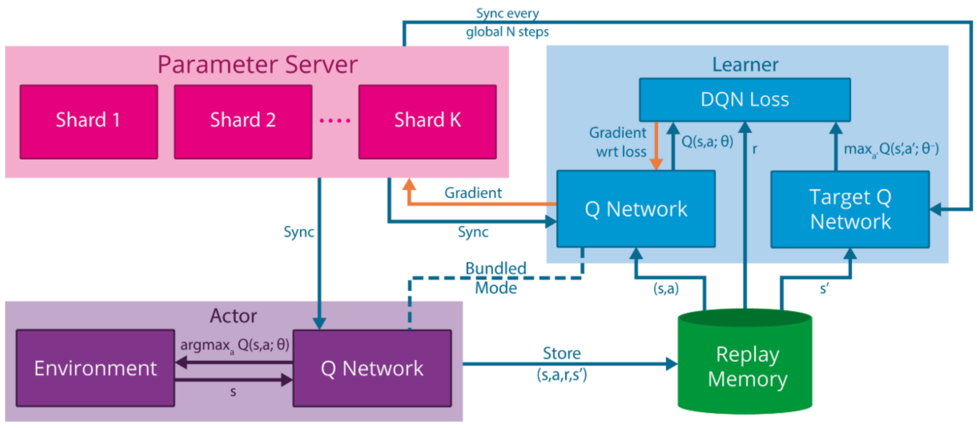

Figure 4: Distributed DQN from the original paper (Nair et al., 2015).The Gorila agent parallelises the training procedure by separating out learners, actors and parameter server. In a single experiment, several learner processes exist and they continuously send the gradients to parameter server and receive updated parameters. At the same time, independent actors can also in parallel accumulate experience and update their Q-networks from the parameter server.

Recurrent Attention Model (RAM)

Mnih et al. (2014) propose a novel recurrent neural network model that is capable of extracting information from an image or video by adaptively selecting a sequence of regions or locations and only processing the selected regions at high resolution, where glimpse networks are learned to select informative parts of images or videos, and recurrent models are used to generate several actions, each action will interact with the environment, and it outputs the next state and current reward.

The architecture is shown in Figure 3. The number of glimpses is the length of the recurrent networks. Without the gold actions provided, we have to reinforce it by maximizing the accumulated rewards. In the paper or most cases, the goal is to predict the label. We can regard the ground-true label as the gold actions, where f_a is optimized in standard classification problem, while f_l is optimized with reinforcement learning.

Ba et al. (2015) extend the RAM to multiple object recognition, i.e., we have to assign separate labels for all the objects in single image or videos. For each object, there is a sequence of actions. The total number of actions are $N * (S + 1)$, where N is the number of glimpses and S is the number of objects.

Distributed Deep Q-Network (DQN)

Nair et al. (2015) follow the idea of the distributed system (DistBelief; Dean et al. 2012) and make it possible to Distributively train DQN. The model architecture is shown in Figure 4. In parameter server, they shard the parameters and each shard of parameters are updated in different machine at parallel.

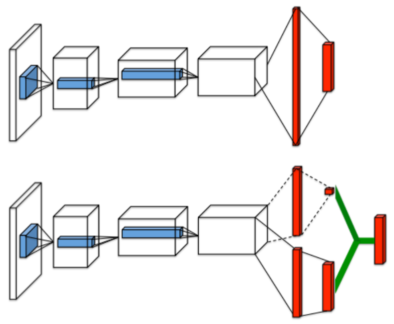

Figure 5: From the original paper (Wang et al., 2016). A popular single stream Q-network (**top**) and the dueling Q-network (**bottom**). The dueling network has two streams to separately estimate (scalar) state-value, $V(s)$ and the advantages for each action, $A(s, a)$; the green output module combines them. Both networks output Q-values for each action

The proposed duel DQN is shown in Figure 5

Dueling DQN

The insight behind Dueling DQN is that it is unnecessary to estimate the value of each action.

> *In some states, it is of paramount importance to know which action to take, but in many other states the choice of action has no repercussion on what happens.*

Wang et al. (2016) propose to use two streams to construct the Q-function, i.e.,

$$Q(s, a) = V(s) + A(s, a). \tag{48}$$

Because $V^\pi(s) = \mathbb{E}_{a\sim\pi(s)}[Q^\pi(s,a)]$, the advantage function $A^\pi(s,a)$ must satisfy $\mathbb{E}_{a\sim\pi(s)}[A^\pi(s,a)] = 0$. If we obtain the deterministic policy by $a^* = \arg\max_a Q(s,a)$, the value function $V(s) = Q(s,a^*)$ because we cannot obtain other actions given the deterministic policy. So following the equation 48, the advantage $A(s,a^*) = 0$.

The problem is that the $V(s)$ and $A(s,a)$ are just paramterized estimation. If we directly use the outputs of these function, we should add some constraints to make them satisfy the definition.

This lack of identifiability is mirrored by poor practical performance when this equation is used directly.

The equation 48 should be rewritten as:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a,\theta,\alpha) - \max_{a'\in\mathcal{A}} A(s,a';\theta,\alpha)). \tag{49}$$

So that for $a^* = \arg\max_{a'\in\mathcal{A}} Q(s,a;\theta,\alpha,\beta) = \arg\max_{a'\in\mathcal{A}} A(s,a;\theta,\alpha)$, we can get $Q(s,a^*;\theta,\alpha,\beta) = V(s;\theta,\beta)$. Furthermore, in order to make the stability of optimization, the equation 49 is rewritten as:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a,\theta,\alpha) - \text{avg}_{a'\in\mathcal{A}} A(s,a';\theta,\alpha)). \tag{50}$$

It deliver the similar result to equation 48 though it losses the definition of V and A.

Exploitation and Exploration

In a real environment, we don't know how often the state will happen or what is the probability to reach one state from another state. If we don't care about the transition probability, we can directly optimize the model-free policy (we hardly reach the infrequent state). If not, we should optimize the model-based policy (state space is too large). Brafman and Tennenholtz (2002) propose a R-max algorithm that is simplified from E^3 (Kearns and Koller, 1999). [1] However, these work become intractable when the size of states increases (Smart and Kaelbling, 2000; Kearns and Singh, 2002). Some works propose Bayesian Reinforcement Learning (BRL) avoiding Alternatively, some works move the focus on **incentivize** exploration, i.e., methods of adding bonus to the rewards aiming to balance **exploration** and **exploitation**. A lots of early works formatted the problem as Partially Observed MDP and used Bayesian Reinforcement Learning (BRL) to solve the problem (Poupart et al., 2006; Asmuth, 2009; Kolter and Ng, 2009; Sorg et al., 2010; Araya-López et al., 2012).

Dueling DQN Dueling DQN decompose $Q(s,a) = V(s) + A(s,a)$ with the constraint $\sum_a A(s,a) = 0$. If we want to update $Q(s,a)$ according to a sample s_t, a_t, we are more likely to update the value of $V(s)$, resulting all other actions in $Q(s,a)$ is updated, because rewards on all actions are added by $V(s)$.

[1] `http://nanjiang.cs.illinois.edu/files/cs598/note7.pdf`

Prioritized Replay The original replay take a buffer to train the Q function with used samples. However, these samples in buffer have different importances. Prioritized Replay regards samples having high TD errors should be used in training more times (high priority).

Multi-step In TD training, we sample (s_t, a_t, r_t, s_{t+1}), while in MC training, we sample from current state to the end. We are looking for a balance between TD training and MC training. We can look multiple steps when sampling, i.e., (s_t, a_t, r_t, ..., s_{t+N}), and $a_{t+N} = arg\ \max_a Q(s_{t+N}, a)$. We do temporal difference on multiple steps like $Q(s_t, a_t) = \sum_{t'=t}^{t+N} r_{t'} + Q(s_{t+N}, a_{t+N})$.

Noisy Net The noisy net is trick for action exploration. Different from the original epsilon greedy and others that sample actions for each times step, result sampled actions come from different policy, Noisy Net adds noisy (Gaussian or others) to the parameter of Q function, resulting sampled actions come from the sample policy (because Q function with noisy is not changed during episode).

- Epsilon Greedy: $a = arg\ \max_a\ Q(s, a)$ with a probability $1 - \epsilon$
- Noisy Net: $\tilde{Q}(s, a) = Q(s, a) + noisy$, and then $a = arg\ \max_a\ \tilde{Q}(s, a)$

Distributional Q-function $Q(s, a)$ is the accumulated reward expects to be obtained after seeing state s and a. The different distribution could have the same expectation (mean). Distributional Q-function aims to build the distributions for each action a for state s. The original method use bars tricks.

Ensemble The rainbow system applies all these tricks together to obtain strong Q-functions.

0.0.1 Continuous actions

In the Algorithm 2, we have to obtain the $\max_a\ \hat{Q}(s_{i+1}, a)$ and $a_{i+1} = arg\ \max_a\ \hat{Q}(s_{i+1}, a)$. But if the action space is continuous, it is hard to solve. We can do sampling from continues space or using gradient ascent to find the optimal a. However, these methods are inefficient. So we can let Q network outputs a vector $\mu(s)$, a matrix $\Sigma(s)$ and a scalar $v(s)$. Given a continuous a:

$$Q(s, a) = -(a - \mu(s))^T \Sigma(s)(a - \mu(s)) + v(s) \tag{51}$$

and $a = \mu(s) = arg\ \max_a\ \hat{Q}(s, a)$

0.1 On-Policy to Off-Policy

0.2 Actor-Critic

We have the policy gradient method and Q-learning method. We can combine the two method together. The gradient of the rewards in policy gradient is:

$$\nabla_\theta \bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n}[\sum_{t'=t}^{T_n}\gamma^{t'-t}r_{t'}^n - b]\nabla_\theta \log p_\theta(a_t^n|s_t^n) \tag{52}$$

where $\sum_{t'=t}^{T_n}\gamma^{t'-t}r_{t'}^n$ is the output of th Q function according to the definitions, i.e.,

$$Q^{\pi_\theta}(s_t, a_t) = \sum_{t'=t}^{T_n}\gamma^{t'-t}r_{t'}^n \tag{53}$$

, and the baseline b aims to make the step reward can be positive and negative. So b should be the means over all possible values of $Q(s_t, a_t)$, i.e,

$$V^{\pi_\theta}(s_t) = b \tag{54}$$

So the gradient of the rewards in actor-critic is:

$$\nabla_\theta \bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n}[Q^{\pi_\theta}(s_t^n, a_t^n) - V^{\pi_\theta}(s_t^n)]\nabla_\theta \log p_\theta(a_t^n|s_t^n). \tag{55}$$

Because $Q^{\pi_\theta}(s_t, a_t) = E[r_t + V(s_{t+1})]$, the above equation can be rewritten as:

$$\nabla_\theta \bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n}[r_t^n + V^{\pi_\theta}(s_{t+1}^n) - V^{\pi_\theta}(s_t^n)]\nabla_\theta \log p_\theta(a_t^n|s_t^n). \tag{56}$$

The training is shown in Algorithm 7. The useful tricks are 1) V function network and π

Algorithm 7 Actor-Critic algorithm

```
1: We initialize $V$, $\pi$
2: Sample data using $\pi$ to environments
3: Update $V$ function
4: Update $\pi$ based on $V$ function
```

network can share the parts of parameters, 2) exploration on smooth π. Asynchronous A2C (A3C) can calculate gradients on multiple CPU/GPU.

0.2.1 Pairwise Derivative Policy Gradient

When we want to learn Q function using TD method, we have to regress the value on the best action given by $Q(s, a)$, $y = r_i + \max_a \hat{Q}(s_{i+1}, a)$. The argmax problem $a^* = arg\max Q(s, a)$ during gradient update, particular in continuous action space.

We can use GAN similar method to build a actor as generator to generate action, and make Q function as discriminator to ensure the action is the best action. The Q-learning is modified to the Algorithm 8.

Algorithm 8 Q-learning algorithm with actor-critic

```
1: We initialize $Q = \hat{Q}$, $\pi = \hat{\pi}$
2: Sample ($s_t$, $a_t$, $r_t$, $s_{t+1}$) based on $\pi$ (epsilon greedy trick)
3: Store ($s_t$, $a_t$, $r_t$, $s_{t+1}$) to buffer
4: Sample ($s_i$, $a_i$, $r_i$, $s_{i+1}$) from buffer (replay trick)
5: $y = r_i +$ ~~$\max_a \hat{Q}(s_{i+1}, a)$~~ $\hat{Q}(s_{i+1}, \hat{\pi}(s_{i+1}))$
6: update $Q$ to make $Q(s_i, a_i)$ close to $y$ (target network trick)
7: update $\pi$ to maximize $Q(s_i, \pi(s_i))$
8: every $C$ steps reset $\hat{Q} = Q$, $\hat{\pi} = \pi$
```

0.3 Sparse Reward

If we want to complete a task that is very far from the current state, the reward is sparse. For example, if we want to have a good job, we have to learn some skills and take the right actions. The problem is that sometimes learning one skill or take a action could be assigned with a small positive reward, leading a large negative reward in the final goal (get a good job). The question is how to get the right reward for these actions that are far from the final objectives.

0.3.1 Reward Shaping

One solution is reward shaping, we manually design some reward function assigning reward to all the possible action that we could take. For example, if we take "study" action, we will be not happy (reward is negative), but it can lead to high positive reward in the future. So we need to shape the reward of taking "study" action to be positive. There are many research works on reward shaping.

Curiosity We can add a curiosity rewards to each action, $R(\tau) = \sum_t^T r_t + r_t^i$ where r_t^i is curiosity reward in the t step. For example, $r_t^i = \text{diff}(\hat{s}_{t+1}, s_{t+1})$ where $\hat{s}_{t+1} = f(a_t, s_t)$, showing that we use the network f to predict the next state $\hat{s}_{t+1}$, and the curiosity reward is the difference between the predicted $\hat{s}_{t+1}$ and the s_{t+1} in our current policy π. If the difference is high that means next state is hard to predict (the future is hard to determine), the reward is high that encourages curiosity. However, the problem is that if we make high rewards to the uncertainty, we will ignore the important features. So instead of the states, we can predict the features of the state, and $r_t^i = \text{diff}(\phi(\hat{s}_{t+1}), \phi(s_{t+1}))$ where ϕ could be a small network (linear or non-linear transformation) that takes states and output the features of the states. How to learn the ϕ function, we can use another network that takes $\phi(s_t)$ and $\phi(s_{t+1})$ and outputs $\hat{a}_t$, ensuring $\hat{a}_t$ and a_t is as close as possible. Because s_t takes action a_t to reach s_{t+1}, and if the ϕs_t and ϕs_{t+1} can be used to recover a_t, we can say ϕ function outputs the main features of the states.

Curriculum Learning The basic idea of curriculum learning is that we first train model on the simple cases and then the model will be trained on harder cases. The question is how to identify which cases are simple or hard. For example, we can use reverse curriculum generation. Starts from the goal state, we sample several states around the goal states and pick the top-k nearest states, and then we generate another set of several states around the top-k nearest states, and so on. In the end, we can get the orders of the learning cases from simplest to hardest.

Hierarchical RL We can divide the task to several small task and for each task, we apply RL methods.

0.4 No Rewards (Imitation Learning)

Imitation learning is to teach model behavor according to the examples, though we do not have rewards.

0.4.1 Behavior Cloning

We can collect a lots of (s, a) and train a model $\pi(s)$ that takes s and outputs correct a. The problems are that the observation is limited, and that we cannot have global optimization. Noted that in RL, the rewards are the expectation of the whole trajectory. One more problem is that the model will not learn the main features that leads the correct actions.

0.4.2 Inverse Reinforcement Learning

Rewards can be learnt from the correct (s, a), The idea is really simple, we can learn the reward function from the expert trajectory τ, and use the reward function to predict trajectory $\bar{\tau}$. The objective is the rewards of the expert trajectory is always larger then the rewards of the predicted trajectory, i.e., $\sum_{n=1}^{N} R(\tau_n) > \sum_{n=1}^{N} R(\bar{\tau}_n)$.

References

Mauricio Araya-López, Vincent Thomas, and Olivier Buffet. 2012. Near-optimal brl using optimistic local transitions. In Proceedings of the 29th International Coference on International Conference on Machine Learning, pages 515–522.

John Asmuth. 2009. A bayesian sampling approach to exploration in reinforcement learning. In Proceedings of The 25th Conference on Uncertainty in Artificial Intelligence (UAI-09), June.

Jimmy Ba, Volodymyr Mnih, and Koray Kavukcuoglu. 2015. Multiple object recognition with visual attention. In ICLR (Poster).

Richard Bellman. 1957. A markovian decision process. Journal of mathematics and mechanics, 6(5):679–684.

Dimitri Bertsekas and John Tsitsiklis. 1996. Neuro-Dynamic Programming, volume 27.

Ronen I Brafman and Moshe Tennenholtz. 2002. R-max-a general polynomial time algorithm for near-optimal reinforcement learning. Journal of Machine Learning Research, 3(Oct):213–231.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V Le, Mark Z Mao, Marc'Aurelio Ranzato, Andrew W Senior, Paul A Tucker, et al. 2012. Large scale distributed deep networks. In NIPS.

Hado Hasselt. 2010. Double q-learning. Advances in neural information processing systems, 23:2613–2621.

Hado Hasselt, Arthur Guez, and David Silver. 2016. Deep reinforcement learning with double q-learning. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 30.

Ronald A Howard. 1960. Dynamic programming and markov processes.

Michael Kearns and Daphne Koller. 1999. Efficient reinforcement learning in factored mdps. In IJCAI, volume 16, pages 740–747.

Michael Kearns and Satinder Singh. 2002. Near-optimal reinforcement learning in polynomial time. Machine learning, 49(2):209–232.

J Zico Kolter and Andrew Y Ng. 2009. Near-bayesian exploration in polynomial time. In Proceedings of the 26th annual international conference on machine learning, pages 513–520.

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. 2016. End-to-end training of deep visuomotor policies. The Journal of Machine Learning Research, 17(1):1334–1373.

Chris J Maddison, Aja Huang, Ilya Sutskever, and David Silver. 2015. Move evaluation in go using deep convolutional neural networks. In NIPS.

Volodymyr Mnih, Nicolas Heess, Alex Graves, and Koray Kavukcuoglu. 2014. Recurrent models of visual attention. In Proceedings of the 27th International Conference on Neural Information Processing Systems-Volume 2, pages 2204–2212.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. nature, 518(7540):529–533.

Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. 2015. Massively parallel methods for deep reinforcement learning. arXiv preprint arXiv:1507.04296.

Pascal Poupart, Nikos Vlassis, Jesse Hoey, and Kevin Regan. 2006. An analytic solution to discrete bayesian reinforcement learning. In Proceedings of the 23rd international conference on Machine learning, pages 697–704.

Arthur L Samuel. 1959. Some studies in machine learning using the game of checkers. IBM Journal of research and development, 3(3):210–229.

Claude E Shannon. 1950. Xxii. programming a computer for playing chess. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, 41(314):256–275.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of go with deep neural networks and tree search. nature, 529(7587):484–489.

William D Smart and Leslie Pack Kaelbling. 2000. Practical reinforcement learning in continuous spaces. In ICML, pages 903–910. Citeseer.

Jonathan Sorg, Satinder Singh, and Richard L Lewis. 2010. Variance-based rewards for approximate bayesian reinforcement learning. In Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence, pages 564–571.

Richard S Sutton. 1985. Temporal credit assignment in reinforcement learning.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. 2016. Dueling network architectures for deep reinforcement learning. In International conference on machine learning, pages 1995–2003. PMLR.

Christopher John Cornish Hellaby Watkins. 1989. Learning from delayed rewards.

Manuel Watter, Jost Tobias Springenberg, Joschka Boedecker, and Martin Riedmiller. 2015. Embed to control: a locally linear latent dynamics model for control from raw images. In Proceedings of the 28th International Conference on Neural Information Processing Systems-Volume 2, pages 2746–2754.