Reinforcement learning

Ying Nian Wu and Quanshi Zhang

Contents

1 Reinforcement learning

1.1 Alpha Go

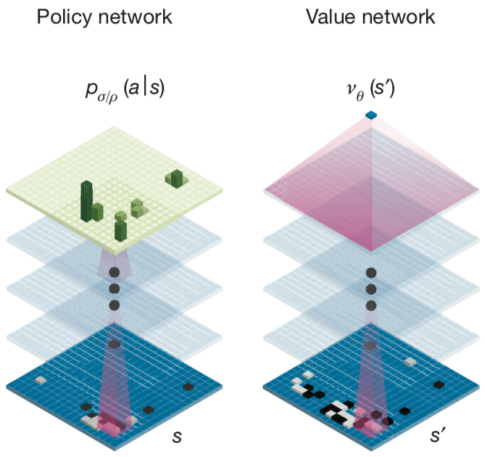

Figure 1: Policy network and value network. The input is a 19×19 image. The policy network is a classification network. The value network is a regression network.

Alpha Go[**?**] is a good starting point to learn reinforcement learning. Let s be the current state, which is a 19×19 image. Let a be the action, i.e., where to place the stone. There are 19×19 choices (although some are forbidden by the rule). We want to decide the action a.

Supervised learning or behavior cloning (learning from human annotations)

We can learn a policy network $p_\sigma(a|s)$, with parameter σ (stands for supervised) from the data $\{(s,a)\}$ collected from human players. See Figure 1 for an illustration of the policy network. This is a classification problem and is a supervised learning problem. It is also called behavior cloning. The learning rule is

$$\Delta\sigma \propto \frac{\partial}{\partial\sigma} \log p_\sigma(a|s),$$

which is to maximize the log-likelihood $\log p_\sigma(a|s)$ over σ.

We can also learn a simpler roll out network to be used in Monte Carlo tree search.

Reinforcement learning by policy gradient (learning from the competition with p_σ)

After learning $p_\sigma(a|s)$, we can learn another policy network $p_\rho(a|s)$ by reinforcement learning. Starting from $\rho = \sigma$, we let p_ρ play against p_σ, until the end. Let $z \in \{+1, -1\}$ be whether ρ wins the game. We update ρ by

$$\Delta\rho \propto \frac{\partial}{\partial\rho} \log p_\rho(a|s) z.$$

The above updating rule is called policy gradient. It is similar to the above maximum likelihood updating rule, except for the following two aspects: (1) the derivative of the log-likelihood is weighted by the reward z. (2) the action a is generated by the current policy instead of human expert.

The above updating rule maximizes $\mathrm{E}_\rho[z]$.

$$E_\rho[z] = \int_{a_1,\ldots,a_n,s_2,\ldots,s_n} p(z=+1|a_1,\ldots,a_n,s_2,\ldots,s_{n+1}) \cdot p_\rho(a_1,\ldots,a_n,s_2,\ldots,s_{n+1}|s_1) da_1 \cdots da_n ds_2 \cdots ds_{n+1}$$

where

$$p_\rho(a_1,\ldots,a_n,s_2,\ldots,s_n,s_{n+1}|s_1) = \prod_{i=1}^{n} p(s_{i+1}|a_i,s_i) p_\rho(a_i|s_i)$$

It is called REINFORCE algorithm. We will justify this algorithm later on.

Value network (learning with the help of a value net)

After learning $p_\rho(a|s)$, we can let ρ play with itself from a random starting state s, until we reach the end and get z. We then update the value network $v_\theta(s)$ by

$$\Delta\theta \propto -\frac{\partial}{\partial\theta}[z - v_\theta(s)]^2.$$

See Figure 1 for an illustration of the value network.

The learned v can also be used as a baseline for policy gradient

$$\Delta\rho \propto \frac{\partial}{\partial\rho} \log p_\rho(a|s)[z - v_\theta(s)],$$

where $v_\theta(s)$ serves to reduce the variance of the gradient. Here v_θ is the critic, and p_ρ is the actor.

v_θ	critic
p_ρ	actor

Learning from human annotations

$$\Delta\sigma \propto \frac{\partial}{\partial\sigma} \log p_\sigma(a|s),$$

Learning from the competition with p_σ

$$\Delta\rho \propto \frac{\partial}{\partial\rho} \log p_\rho(a|s) z,$$

Learning with the help of a value net

$$\Delta\rho \propto \frac{\partial}{\partial\rho} \log p_\rho(a|s)[z - v_\theta(s)].$$

Monte Carlo tree search (in order to use p_σ, p_ρ, v_θ to play the Go game)

After learning the policy networks p_σ, p_ρ, and value network v_θ, we can use either the policy network or the value network to play the game. But this will not work well. Instead we use them to help us look ahead and plan the next action using Monte Carlo tree search (MCTS). Specifically, we want to estimate $Q(s,a)$, the value of action a at state s.

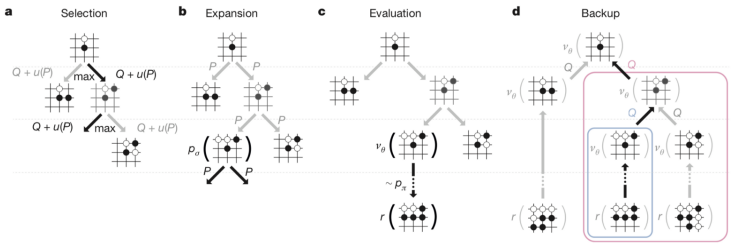

Figure 2: Monte Carlo tree search grows a tree by repeating the following four steps: selection, expansion, evaluation, and backup.

Treating the current state as the root, MCTS grows a tree by repeating the following four steps: selection, expansion, evaluation, and backup. For each node s of the tree, and each action a from this node, we record the number of visits $N(s,a)$ and the action value $Q(s,a)$. During each pass of MCTS, starting from the root state, we go down the tree until we get to a leaf node. At each non-leaf node s, we choose an action a that balances $Q(s,a)$ and $N(s,a)$. We want to choose a with a high $Q(s,a)$ for exploitation, meanwhile we also want to choose a with low $N(s,a)$ for exploration. When we come to a leaf node, we expand the tree using all the possible moves from this node. Then we choose an expanded node, and use a roll out policy to play the game until the end. Finally we backup the roll out result. For all the branches (s,a) we go through in this pass of MCTS, we increase the visit count $N(s,a)$ by 1, and increase or decrease the total value of (s,a) by 1 according to the roll out result. $Q(s,a)$ is then updated as the current total value divided by the current visit count. MCTS is expected to converge to the minimax solution to the game.

$N(s,a)$	the number of visit of (s,a)	If s is visited and a is taken, then $N(s,a) \leftarrow N(s,a)+1$
$Q(s,a)$	the action value	$Q(s,a) = \frac{\text{total value (total number of winning)}}{N(s,a)}$
		If wining the game, then total number of winning plus one.

The policy network can help guide the selection step. For each node s, we select the action according to

$$Q(s,a) + C p_\sigma(a|s) / \sqrt{N(s,a)}$$

where the constant C balances the exploration and exploitation. We use the supervised p_σ instead of reinforcement learning p_ρ, because p_σ is more diverse.

The value network can help avoid the reliance on roll out. For an expanded node s, instead of rolling out to the end, and then backup, we can simply backup $v_\theta(s)$ by adding the total value of a traveled branch (s,a) the value $v_\theta(s)$. The Alpha Go uses a linear combination of roll out result from s and $v_\theta(s)$. Since s is closer to the end of the game than the root node, the value $v_\theta(s)$ can be a more precise estimate than the value at the root node.

After many passes of MCTS, at the current root node s, we choose a with the maximal $Q(s,a)$. We can also choose a with the minimum $N(s,a)$.

The MCTS is a planning process. Its purpose is to select the next move a from the current root node s. After we make the move a in real game play, we can then discard the tree. When we need to choose the next move in real game play, we start to grow another tree.

Summarization

- **Building a tree → dreaming of the future play:** For each state s, build a tree. When we get a new state s' after taking an action, we need to build a new tree.
- **The real play:** Choosing $\hat{a} = \arg\min_a N(s,a)$ or $\hat{a} = \arg\max_a Q(s,a)$ to play the game.
- How to build a tree?

  Strategy 1: Choose a with a large $Q(s,a)$ and/or a small $N(s,a)$. Roll out until the end of the game. Update Q and N for all states and actions along the trajectory after s.

  $s_1 \to a_1 \to s_2 \to a_2 \to s_3 \to a_3 \to s_4, \ldots$ rolling out until the end of the game. Then use the result $z \in \{-1, +1\}$ to update $Q(s_i, a_i)$ and $N(s_i, a_i)$ along the entire the trajectory of (a, s).

  Strategy 2: Using the pre-trained actor p_σ for action selection. Compared to Strategy 1, choose the action via $\max_a[Q(s,a) + Cp_\sigma(a|s)/\sqrt{N(s,a)}]$.

  Strategy 3: Using the value net $v_\theta(s)$ to avoid rolling out to the end of the game. $s_1 \to a_1 \to s_2 \to a_2 \to s_3 \to a_3 \to s_4$. Then directly use $v_\theta(s_4)$ to update $Q(s_1, a_1)$, $Q(s_2, a_2)$, $Q(s_3, a_3)$, i.e., the total value of each (s_i, a_i) is increased by $v_\theta(s_4)$, and the visit number of each (s_i, a_i), $N(s_i, a_i)$, is increased by 1. $Q(s_i, a_i) = \frac{\text{total value}}{N(s_i, a_i)}$.

1.2 Alpha Go Zero

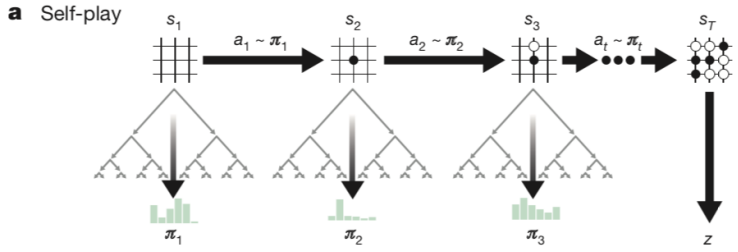

Figure 3: Alpha Go Zero. In self-play, we use MCTS to plan each move, where MCTS is guided by policy and value networks. After the game is over, the game result is used to train the policy and value networks.

- **Alpha Go:** Do not update $p(a|s)$ or $v(s)$ after the learning.
- **Alpha Go Zero:** No learning of $p(a|s)$ or $v(s)$ before MCTS. Update $p(a|s)$ and $v(s)$ via self play.

In Alpha Go, after learning the policy and value networks, we will not update it, and they are used for MCTS for playing the game. This is a big waste, because when we use MCTS to play the game, the results of the game play should be used to further update the policy and value networks. In fact, the policy and value networks can be learned solely from the games where the moves are planned by MCTS.

In Alpha Go Zero[**?**], the learning of p_σ, p_ρ and v_θ before MCTS is discarded. Instead, p and v are learned from the results of the self-play, and p and v in turn guides the MCTS in planning each move of the self-play. Specifically, we iterate the following two steps:

- **(1) Real play:** Given p and v, play a game where each move is planned by MCTS. Specifically, at each state s of the game, we grow a tree in order to evaluate $N(s,a)$ and $Q(s,a)$ (in fact, $N(s,a)$ and $Q(s,a)$ are computed for all the nodes in the tree). We let $\pi(a|s) \propto N(s,a)^{1/\tau}$, where τ is the temperature parameter. We then discard the tree and use $\pi(a|s)$ to select the move a to play the game. We play the game until the end.
- **(2) Model updating:** We use $\pi(a|s)$ to train the policy $p(a|s)$. We use the end result $z \in \{+1,-1\}$ to train the value network $v(s)$ for all the states of the game.

In Alpha Go Zero, the policy and value networks share the same body, which consists of many residual blocks. Each network has a head. Such a design makes sense because the two networks are consistent with each other.

1.3 Key elements of reinforcement learning

The Alpha Go contains almost all the elements in reinforcement learning.

(1) Policy network $p(a|s)$.

(2) Value network $v(s)$.

(3) Action value $Q(s,a)$.

(4) Model based planning by Monte Carlo tree search.

(5) Dynamics model $p(s_{t+1}|s_t,a)$, which is given by the rule of the game.

The difference between Go and some of the other problems in reinforcement learning is that there is no immediate reward.

Alpha Go is ultimately about model-based planning, i.e., given the dynamics model, we find the actions by maximizing the return. The policy network and the value network only play an assisting role for this planning process.

Most of the RL is about model-free learning, i.e., we do not learn about the dynamics. Instead we learning $Q(s,a)$ directly in Q-learning[**?**, **?**, **?**], or we learn $p(a|s)$ directly by policy gradient. Usually we need big training data for model-free learning.

- **Q-learning:** learning $Q(s,a)$
- **Policy gradient:** learning $p(a|s)$

Currently most of the robotics are about model-based planning or control. The self-driving cars are based on model-based control.

In the following, we shall formulate RL with immediate reward. For further reading, one can refer to the book "Reinforcement Learning"[**?**] by Sutton.

1.4 Reward and return

At state s_t, we take an action a_t, according to the policy $\pi(a|s)$, and arrive at state s_{t+1}, according to the dynamics $p(s_{t+1}|s_t,a_t)$. The reward is $r_t = r(s_t,a_t,s_{t+1})$. The accumulated return is

$$R_t = \sum_{k=0}^{\infty} r_{t+k},$$

Let us ignore the discount factor γ for simplicity. We want to maximize the expected return.

If we do not ignore the discount factor $0 \leq \gamma \leq 1$. Then,

$$R_t = \sum_{k=0}^{\infty} r_{t+k}\gamma^k,$$

which considers the reward of near states more reliable than the reward of far states.

1.5 Value

The value of taking action a at state s and then follow the policy π is

$$Q^{\pi}(s,a) = \mathrm{E}[R_t | s_t = s, a_t = a],$$

where the expectation is with respect to the trajectory $(s_{t+k}, a_{t+k}, k = 1,2,...)$ generated from $(s_t = s, a_t = a)$ by the policy $\pi(a|s)$ and the dynamics $p(s_{t+1}|s_t, a_t)$.

The optimal value is

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a).$$

Similarly, we can define the value of a state

$$\begin{aligned} V^{\pi}(s) =& \mathrm{E}[R_t | s_t = s] \\ =& E_{s_{t+1}, s_{t+2}, \ldots, a_{t+1}, a_{t+2}, \ldots}[\sum_{a} R_t |_{s_t = s, a_t = a} \cdot p(a|s_t = s)] \end{aligned}$$

1.6 REINFORCE for policy gradient

$$E_{\rho}[h(X)] \stackrel{def}{=} \sum_{a_1, a_2, \ldots, s_1, s_2, \ldots} [h(X) \cdot p(a_1, a_2, \ldots, s_1, s_2, \ldots | \rho)],$$

Here,

- $X = (a_1, a_2, \ldots, s_1, s_2, \ldots)$
- $h(X) = \sum_{t=0}^{+\infty} r_t$, because each local reward r_t depends on s_t.

We can learn a policy network $\pi_{\rho}(a|s)$ by maximizing the expected R_0. Assuming we start from s_0. The

gradient [**?**] is

$$
\begin{aligned}
\frac{\partial}{\partial\rho}\mathrm{E}_\rho[R_0] &= \frac{\partial}{\partial\rho}\mathrm{E}_\rho\left[\sum_{t=0}^{\infty} r_t\right] \\
&= \frac{\partial}{\partial\rho}\sum_{a_0,a_1,\ldots,s_0,s_1,\ldots}\left[\left(\sum_{t=0}^{+\infty} r_t\right)\cdot p(a_0,a_1,\ldots,s_0,s_1,\ldots|\rho)\right] \\
&= \sum_{a_0,a_1,\ldots,s_0,s_1,\ldots}\left[\left(\sum_{t=0}^{+\infty} r_t\right)\cdot \frac{\partial}{\partial\rho}p(a_0,a_1,\ldots,s_0,s_1,\ldots|\rho)\right], \qquad \text{//Considering } r_t \text{ a constant w.r.t. } \rho \\
&= \sum_{a_0,a_1,\ldots,s_0,s_1,\ldots}\left[\left(\sum_{t=0}^{+\infty} r_t\right)\cdot p(a_0,a_1,\ldots,s_0,s_1,\ldots|\rho)\frac{\partial}{\partial\rho}\log p(a_0,a_1,\ldots,s_0,s_1,\ldots|\rho)\right] \\
&= \sum_{a_0,a_1,\ldots,s_0,s_1,\ldots}\left[\left(\sum_{t=0}^{+\infty} r_t\right)\cdot p(a_0,a_1,\ldots,s_0,s_1,\ldots|\rho)\sum_{t=0}^{+\infty}\frac{\partial}{\partial\rho}\log p(s_{t+1}|a_t,s_t)p_\rho(a_t|s_t)\right] \\
&= \mathrm{E}_\rho\left[\left(\sum_{t=0}^{\infty} r_t\right)\frac{\partial}{\partial\rho}\log\left(\prod_{t=0}^{\infty}\pi_\rho(a_t|s_t)p(s_{t+1}\mid s_t,a_t)\right)\right] \\
&= \mathrm{E}_\rho\left[\left(\sum_{t=0}^{\infty} r_t\right)\left(\sum_{t=0}^{\infty}\frac{\partial}{\partial\rho}\log\pi_\rho(a_t|s_t)\right)\right] \\
&= \mathrm{E}_\rho\left[\sum_{t=0}^{\infty}\left(\frac{\partial}{\partial\rho}\log\pi_\rho(a_t|s_t)\sum_{t'=0}^{\infty} r_{t'}\right)\right].
\end{aligned}
$$

Using simplified notation, for any $h(x)$ and $p_\rho(x)$, we have

$$
\begin{aligned}
\frac{\partial}{\partial\rho}\mathrm{E}_\rho[h(X)] &= \frac{\partial}{\partial\rho}\int h(x)p_\rho(x)dx \\
&= \int h(x)\frac{\partial}{\partial\rho}p_\rho(x)dx \\
&= \int\left[h(x)\frac{\partial}{\partial\rho}\log p_\rho(x)\right]p_\rho(x)dx \\
&= \mathrm{E}_\rho\left[h(X)\frac{\partial}{\partial\rho}\log p_\rho(X)\right].
\end{aligned}
$$

REINFORCE is an application of the above identity. Writing the derivative in terms of E_ρ enables us to approximate E_ρ by Monte Carlo sampling, i.e., we update ρ by

$$\Delta\rho \propto h(X)\frac{\partial}{\partial\rho}\log p_\rho(X),$$

where $X \sim p_\rho(x)$. We can sample multiple copies of X and average over the multiple copies.

In the case of policy gradient, Monte Carlo sampling means we run the policy forward to obtain the trajectory. We can let multiple agents run parallel trajectories following the current policy, and then average over the multiple trajectories.

1.7 Relationship with maximum likelihood

If $X \sim p_{\rho_*}(x)$, where p_{ρ_*} is a teacher, then we can estimate ρ by minimizing the Kullback-Leibler divergence $\mathrm{KL}(p_{\rho_*}|p_\rho)$ over ρ, which is equivalent to maximizing the log-likelihood $\mathrm{E}_{\rho_*}[\log p_\rho(X)]$ (the negative of which is also called cross entropy). The gradient is $\mathrm{E}_{\rho_*}[\frac{\partial}{\partial\rho}\log p_\rho(X)]$, and the stochastic gradient is

$$\Delta\rho \propto \sum_X p_{\rho_*}(X)\frac{\partial}{\partial\rho}\log p_\rho(X) = E_{X\sim p_{\rho_*}}\left[\frac{\partial}{\partial\rho}\log p_\rho(X)\right], \qquad \text{i.e., } (-1)\times \text{ gradients of cross entropy, minimizing } \mathrm{KL}(p_{\rho_*}|p_\rho)$$

The above MLE gradient and the REINFORCE gradient share the term $\frac{\partial}{\partial\rho}\log p_\rho(X)$. However, they differ in the following two aspects.

(1) In MLE, $X \sim p_{\rho_*}$, which is a teacher or an expert. In REINFORCE, $X \sim p_\rho$, which is the current policy or student.

(2) In MLE, there is no reward. In REINFORCE, there is a reward.

The student has nothing to learn from itself, i.e.,

$$\mathrm{E}_\rho\left[\frac{\partial}{\partial\rho}\log p_\rho(X)\right] = 0,$$

which is a direct result if we take $h(x) = 1$ or a constant.

Proof:

$$\begin{aligned}
\mathrm{E}_\rho\left[\frac{\partial}{\partial\rho}\log p_\rho(X)\right] &= \int_x p_\rho(X)\frac{\partial}{\partial\rho}\log p_\rho(X)dx \\
&= \int_x p_\rho(X)\frac{1}{p_\rho(X)}\frac{\partial}{\partial\rho}p_\rho(X)dx \\
&= \int_x \frac{\partial}{\partial\rho}p_\rho(X)dx \\
&= \frac{\partial}{\partial\rho}\int_x p_\rho(X)dx \\
&= \frac{\partial}{\partial\rho}1 \\
&= 0
\end{aligned}$$

Therefore, considering $\frac{\partial}{\partial\rho}\mathrm{E}_\rho[h(X)] = \mathrm{E}_\rho\left[h(X)\frac{\partial}{\partial\rho}\log p_\rho(X)\right]$ that have been proved before,

$$\frac{\partial}{\partial\rho}\mathrm{E}_\rho[1] = \mathrm{E}_\rho\left[\frac{\partial}{\partial\rho}\log p_\rho(X)\right] = 0.$$

We can also obtain the above result directly by differentiating $\int p_\rho(x)dx = 1$.

In order to have a non-zero gradient, either X is sampled from a teacher, or there is a reward. The former is MLE, and the latter is REINFORCE.

For REINFORCE, we can change $h(x)$ to $h(x) - b$ for any baseline, because

$$\frac{\partial}{\partial\rho}\mathrm{E}_\rho[h(X)] = \frac{\partial}{\partial\rho}\mathrm{E}_\rho[h(X) - b].$$

- $X \longrightarrow (a_0, a_1, \ldots, s_0, s_2, \ldots)$
- $b \longrightarrow V_\theta(X) \longrightarrow V_\theta(s_0)$
- $h(X) \longrightarrow z$ or $\sum_{t=0}^{+\infty} r_t$

Thus,

$$\Delta\rho = \frac{\partial}{\partial\rho}\mathrm{E}_\rho[h(X)] = \frac{\partial}{\partial\rho}\mathrm{E}_\rho[h(X)-b] = \mathrm{E}_\rho[(h(X)-b)\frac{\partial}{\partial\rho}\log p_\rho(X)]$$

corresponds to the policy gradient

$$\Delta\rho \propto [z - v_\theta(s)]\frac{\partial}{\partial\rho}\log p_\rho(a|s).$$

1.8 Causal (in order to prove that the learning process only needs to consider the reward after the current action)

$$\begin{aligned}
\frac{\partial}{\partial\rho}\mathrm{E}_\rho[R_0] &= \mathrm{E}_\rho\left[\sum_{t=0}^{\infty}\left(\frac{\partial}{\partial\rho}\log\pi_\rho(a_t \mid s_t)\sum_{t'=0}^{\infty} r_{t'}\right)\right] \\
&= \sum_{t=0}^{\infty}\mathrm{E}_\rho\left[\left(\frac{\partial}{\partial\rho}\log\pi_\rho(a_t \mid s_t)\left(\sum_{t'=0}^{t-1} r_{t'} + \sum_{t'=t}^{\infty} r_{t'}\right)\right)\right] \\
&= \sum_{t=0}^{\infty}\mathrm{E}_\rho\left[\frac{\partial}{\partial\rho}\log\pi_\rho(a_t \mid s_t)(R_t + C)\right] \quad \text{where } C = \sum_{t'=0}^{t-1} r_{t'} \\
&= \sum_{t=0}^{\infty}\mathrm{E}_\rho\left[\frac{\partial}{\partial\rho}\log\pi_\rho(a_t \mid s_t)R_t\right] + C\cdot E_\rho\sum_{t=0}^{\infty}\frac{\partial}{\partial\rho}\log\pi_\rho(a_t \mid s_t) \\
&= \sum_{t=0}^{\infty}\mathrm{E}_\rho\left[\frac{\partial}{\partial\rho}\log\pi_\rho(a_t \mid s_t)R_t\right] + C\cdot E_\rho\frac{\partial}{\partial\rho}\left[\sum_{t=0}^{\infty}\log\pi_\rho(a_t \mid s_t) + \log p(s_{t+1}|a_t, s_t)\right] \\
&= \sum_{t=0}^{\infty}\mathrm{E}_\rho\left[\frac{\partial}{\partial\rho}\log\pi_\rho(a_t \mid s_t)R_t\right] + C\cdot E_\rho\frac{\partial}{\partial\rho}\log p_\rho(a_0, \ldots, a_\infty, s_0, \ldots, s_\infty \mid s_0) \\
&= \sum_{t=0}^{\infty}\mathrm{E}_\rho\left[\frac{\partial}{\partial\rho}\log\pi_\rho(a_t \mid s_t)R_t\right] + C\cdot E_\rho\frac{\partial}{\partial\rho}\log p_\rho(X) \quad \because p(s_0) = 1 \\
&= \sum_{t=0}^{\infty}\mathrm{E}_\rho\left[\frac{\partial}{\partial\rho}\log\pi_\rho(a_t \mid s_t)R_t\right] \quad \because C\cdot E_\rho\frac{\partial}{\partial\rho}\log p_\rho(X) = 0
\end{aligned}$$

where

$$R_t = \sum_{t'=t}^{\infty} r_{t'}$$

is the reward to go after action a_t, whereas $\sum_{t'=0}^{t-1} r_{t'}$ is constant relative to a_t, because a_t can only cause the change in the future, but it cannot change the past. Therefore we can remove it as a baseline.

1.9 Advantage

$$\frac{\partial}{\partial \rho} \mathrm{E}_\rho[R_0] \quad = \quad \sum_{t=0}^{\infty} \mathrm{E}_\rho \left[\frac{\partial}{\partial \rho} \log \pi_\rho(a_t|s_t)(R_t - V(s_t)) \right]$$

for any baseline $V(s_t)$, which cannot be changed by a_t. Here we take

$$V(s_t) = V^\pi(s_t) = \mathrm{E}[R_t \mid s_t] = \mathrm{E}_{a_t, s_{t+1}, a_{t+1}, s_{t+2}, \ldots}[R_t \cdot p(a_t, s_{t+1}, a_{t+1}, s_{t+2}, \ldots | s_t)],$$

which is the value of s_t, averaged over all possible actions $a_t \sim \pi(a|s_t)$.

$A_t = R_t - V(s_t)$, related to the term $(z - v_\theta(s))$ in the computation of gradient policy

is called the **advantage of taking action** a_t relative to other possible actions at s_t. Subtracting $V(s_t)$ as a baseline helps reduce the variance of the gradient.

When computing the gradient, we fix π at the current policy.

1.10 Value network

We can learn ρ by stochastic gradient

$$\Delta\rho \propto \sum_{t=0}^{\infty} \left[\frac{\partial}{\partial \rho} \log \pi_\rho(a_t|s_t)(R_t - V_\theta(s_t)) \right], \qquad \text{this corresponds to the policy gradient}$$

where we run the policy π_ρ to the end. We also need to learn a value network $V_\theta(s)$ to approximate $V^\pi(s)$.

1.11 Temporal difference (in order to get a more accurate value than $V_\theta(s_t)$)

We may also estimate R_t based on

$$V^\pi(s_t) = \mathrm{E}[R_t] = \mathrm{E}\left[\sum_{k=0}^{n} r_{t+k} + V^\pi(s_{t+n+1}) \right].$$

Thus we can estimate R_t without running the policy to the very end, and we estimate R_t by

$$\hat{R}_t = \sum_{k=0}^{n} r_{t+k} + V_\theta(s_{t+n+1}),$$

which consists of two parts:

- Monte Carlo unrolling: run the policy π_ρ for n steps to accumulate $\sum_{k=0}^{n} r_{t+k}$.
- Bootstrapping: prediction the rest of the accumulated reward by the current value network $V_\theta(s_{t+n+1})$.

This is a temporal difference scheme.

$V^\pi(s)$ is more accurate than $V_\theta(s)$, because in the computation of $V^\pi(s)$, the value net is applied to the state s_{t+n+1}, which is closer to the end of the game than the state s_t.

1.12 Policy update

Thus we can use gradient descent

$$\Delta\rho \propto \sum_{t=0}^{\infty} \left[\frac{\partial}{\partial \rho} \log \pi_\rho(a_t|s_t) \left(\left(\sum_{k=0}^{n} r_{t+k} \right) + V_\theta(s_{t+n+1}) - V_\theta(s_t) \right) \right]$$

1.13 Value update

Meanwhile we can update θ by

$$\Delta\theta \propto \left[\left(\sum_{k=0}^{n} r_{t+k}\right) + V_\theta(s_{t+n+1}) - V_\theta(s_t)\right] \frac{\partial V}{\partial \theta}, \qquad \text{//considering } Loss(\theta) = \left[V_\theta(s_t) - \underbrace{\left(\left(\sum_{k=0}^{n} r_{t+k}\right) + V_\theta(s_{t+n+1})\right)}_{\text{Temporarily consider this is independent with } \theta}\right]^2$$

which seeks to approach the fixed point

$$\hat{R}_t \overset{def}{=} V^\pi(s_t) = \mathrm{E}\left[\sum_{k=0}^{n} r_{t+k} + V^\pi(s_{t+n+1})\right].$$

by gradient descent on

$$[\hat{R}_t - V_\theta(s_t)]^2.$$

The above is not a real loss function because the target $\hat{R}_t = (\sum_{k=0}^{n} r_{t+k}) + V_\theta(s_{t+n+1})$ is based on the value-net-based policy V_θ itself, and this bootstrapped target keeps changing. Without Monte Carlo, it becomes a self-fulfilling prophecy, and there is nothing for the value network to learn.

1.14 Q learning

The value update also underlies the Q learning[**?**, **?**, **?**], which seeks to approach $Q(s,a) = \max_\pi Q^\pi(s,a)$ (here we drop the subscript $*$ for convenience). $Q(s,a)$ is again a fixed point

$$Q(s,a) = \mathrm{E}_{s'}\left[r(s,a,s') + \max_{a'} Q(s',a')\right],$$

where the expectation is with respect to the dynamics $p(s'|s,a)$. We can learn a Q network $Q_\alpha(s,a)$, and update α by

$$\Delta\alpha \propto \left[r(s,a,s') + \max_{a'} Q_\alpha(s',a') - Q_\alpha(s,a)\right] \frac{\partial Q}{\partial \alpha}.$$

Again this algorithm seeks the fixed point by gradient descent on the loss

$$L = [R - Q_\alpha(s,a)]^2,$$

where

$$R = r(s,a,s') + \max_{a'} Q_\alpha(s',a')$$

is the bootstrapped changing target (we temporarily consider R is independent with α for bootstrapping).

1.15 SARSA

The policy gradient method is an on-policy method, because we need to run Monte Carlo according to the current policy. The Q learning is an off-policy method, where when taking the action a at the current state s, we do not need to following the current policy.

SARSA[**?**] is an on-policy version of Q learning, where at state s, we run the current policy to get a, and in the sub-sequent states, we also follow the current policy to get a', instead of using $\max_{a'}$ as in Q learning. SARSA stands for s,a,r,s',a', which is a sequence of terms to be considered.