Policy Gradient Algorithms

Apr 8, 2018 by Lilian Weng reinforcement-learning long-read

> Abstract: In this post, we are going to look deep into policy gradient, why it works, and many new policy gradient algorithms proposed in recent years: vanilla policy gradient, actor-critic, off-policy actor-critic, A3C, A2C, DPG, DDPG, D4PG, MADDPG, TRPO, PPO, ACER, ACTKR, SAC and TD3.

[Updated on 2018-06-30: add two new policy gradient methods, SAC and D4PG.]
[Updated on 2018-09-30: add an new policy gradient method, TD3.]
[Updated on 2019-02-09: add SAC with automatically adjusted temperature].
[Updated on 2019-05-01: Thanks to Wenhao, we have a version of this post in Chinese].

What is Policy Gradient

Policy gradient is an approach to solve reinforcement learning problems. If you haven't looked into the field of reinforcement learning, please first read the section "A (Long) Peek into Reinforcement Learning » Key Concepts" for the problem definition and key concepts.

Notations

Here is a list of notations to help you read through equations in the post easily.

Symbol	Meaning
$s \in \mathcal{S}$	States.
$a \in \mathcal{A}$	Actions.
$r \in \mathcal{R}$	Rewards.
S_t, A_t, R_t	State, action, and reward at time step t of one trajectory. I may occasionally use s_t, a_t, r_t as well.
γ	Discount factor; penalty to uncertainty of future rewards; $0 < \gamma \leq 1$.
G_t	Return; or discounted future reward; $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$.
$P(s', r \vert s, a)$	Transition probability of getting to the next state s' from the current state s with action a and reward r.
$\pi(a \vert s)$	Stochastic policy (agent behavior strategy); $\pi_\theta(.)$ is a policy parameterized by θ.
$\mu(s)$	Deterministic policy; we can also label this as $\pi(s)$, but using a different letter gives better distinction so that we can easily tell when the policy is stochastic or deterministic without further explanation. Either π or μ is what a reinforcement learning algorithm aims to learn.
$V(s)$	State-value function measures the expected return of state s; $V_w(.)$ is a value function parameterized by w.
$V^\pi(s)$	The value of state s when we follow a policy π; $V^\pi(s) = \mathbb{E}_{a\sim\pi}[G_t \vert S_t = s]$.
$Q(s, a)$	Action-value function is similar to $V(s)$, but it assesses the expected return of a pair of state and action (s, a); $Q_w(.)$ is a action value function parameterized by w.
$Q^\pi(s, a)$	Similar to $V^\pi(.)$, the value of (state, action) pair when we follow a policy π; $Q^\pi(s, a) = \mathbb{E}_{a\sim\pi}[G_t \vert S_t = s, A_t = a]$.
$A(s, a)$	Advantage function, $A(s, a) = Q(s, a) - V(s)$; it can be considered as another version of Q-value with lower variance by taking the state-value off as the baseline.

Policy Gradient

The goal of reinforcement learning is to find an optimal behavior strategy for the agent to obtain optimal rewards. The **policy gradient** methods target at modeling and optimizing the policy directly. The policy is usually modeled with a parameterized function respect to θ, $\pi_\theta(a \vert s)$. The value of the reward (objective) function depends on this policy and then various algorithms can be applied to optimize θ for the best reward.

The reward function is defined as:

$$J(\theta) = \sum_{s \in \mathcal{S}} d^\pi(s) V^\pi(s) = \sum_{s \in \mathcal{S}} d^\pi(s) \sum_{a \in \mathcal{A}} \pi_\theta(a \vert s) Q^\pi(s, a)$$

where $d^\pi(s)$ is the stationary distribution of Markov chain for π_θ (on-policy state distribution under π). For simplicity, the θ parameter would be omitted for the policy π_θ when the policy is present in the subscript of other functions; for example, d^π and Q^π should be d^{π_θ} and Q^{π_θ} if written in full. Imagine that you can travel along the Markov chain's states forever, and eventually, as the time progresses, the probability of you ending up with one state becomes unchanged — this is the stationary probability for π_θ. $d^\pi(s) = \lim_{t \to \infty} P(s_t = s \vert s_0, \pi_\theta)$ is the probability that $s_t = s$ when starting from s_0 and following policy π_θ for t steps. Actually, the existence of the stationary distribution of Markov chain is one main reason for why PageRank algorithm works. If you want to read more, check this.

It is natural to expect policy-based methods are more useful in the continuous space. Because there is an infinite number of actions and (or) states to estimate the values for and hence value-based approaches are way too expensive computationally in the continuous space. For example, in generalized policy iteration, the policy improvement step $\arg\max_{a\in\mathcal{A}} Q^\pi(s,a)$ requires a full scan of the action space, suffering from the curse of dimensionality.

Using *gradient ascent*, we can move θ toward the direction suggested by the gradient $\nabla_\theta J(\theta)$ to find the best θ for π_θ that produces the highest return.

Policy Gradient Theorem

Computing the gradient $\nabla_\theta J(\theta)$ is tricky because it depends on both the action selection (directly determined by π_θ) and the stationary distribution of states following the target selection behavior (indirectly determined by π_θ). Given that the environment is generally unknown, it is difficult to estimate the effect on the state distribution by a policy update.

Luckily, the **policy gradient theorem** comes to save the world! Woohoo! It provides a nice reformation of the derivative of the objective function to not involve the derivative of the state distribution $d^\pi(.)$ and simplify the gradient computation $\nabla_\theta J(\theta)$ a lot.

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a)\pi_\theta(a|s) \\
&\propto \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a)\nabla_\theta \pi_\theta(a|s)
\end{aligned}
$$

Proof of Policy Gradient Theorem

This session is pretty dense, as it is the time for us to go through the proof (Sutton & Barto, 2017; Sec. 13.1) and figure out why the policy gradient theorem is correct.

We first start with the derivative of the state value function:

$$
\begin{aligned}
& \nabla_\theta V^\pi(s) \\
=& \nabla_\theta \Big(\sum_{a\in\mathcal{A}} \pi_\theta(a|s)Q^\pi(s,a)\Big) & \\
=& \sum_{a\in\mathcal{A}} \Big(\nabla_\theta \pi_\theta(a|s)Q^\pi(s,a) + \pi_\theta(a|s) \color{red}{\nabla_\theta Q^\pi(s,a)}\Big) & \scriptstyle{\text{; Derivative product rule.}} \\
=& \sum_{a\in\mathcal{A}} \Big(\nabla_\theta \pi_\theta(a|s)Q^\pi(s,a) + \pi_\theta(a|s) \color{red}{\nabla_\theta \sum_{s', r} P(s',r|s,a)(r + V^\pi(s'))}\Big) & \scriptstyle{\text{; Extend } Q^\pi \text{ with future state value.}} \\
=& \sum_{a\in\mathcal{A}} \Big(\nabla_\theta \pi_\theta(a|s)Q^\pi(s,a) + \pi_\theta(a|s) \color{red}{\sum_{s', r} P(s',r|s,a) \nabla_\theta V^\pi(s')}\Big) & \scriptstyle{P(s',r|s,a) \text{ or } r \text{ is not a func of }\theta} \\
=& \sum_{a\in\mathcal{A}} \Big(\nabla_\theta \pi_\theta(a|s)Q^\pi(s,a) + \pi_\theta(a|s) \color{red}{\sum_{s'} P(s'|s,a) \nabla_\theta V^\pi(s')}\Big) & \scriptstyle{\text{; Because } P(s'|s,a) = \sum_r P(s',r|s,a)}
\end{aligned}
$$

Now we have:

$$
\color{red}{\nabla_\theta V^\pi(s)} = \sum_{a\in\mathcal{A}} \Big(\nabla_\theta \pi_\theta(a|s)Q^\pi(s,a) + \pi_\theta(a|s) \sum_{s'} P(s'|s,a) \color{red}{\nabla_\theta V^\pi(s')}\Big)
$$

This equation has a nice recursive form (see the red parts!) and the future state value function $V^\pi(s')$ can be repeated unrolled by following the same equation.

Let's consider the following visitation sequence and label the probability of transitioning from state s to state x with policy π_θ after k step as $\rho^\pi(s \to x, k)$.

$$
s \xrightarrow[]{a \sim \pi_\theta(.|s)} s' \xrightarrow[]{a \sim \pi_\theta(.|s')} s'' \xrightarrow[]{a \sim \pi_\theta(.|s'')} \dots
$$

- When k = 0: $\rho^\pi(s \to s, k=0) = 1$.
- When k = 1, we scan through all possible actions and sum up the transition probabilities to the target state: $\rho^\pi(s \to s', k=1) = \sum_a \pi_\theta(a \vert s) P(s' \vert s, a)$.
- Imagine that the goal is to go from state s to x after k+1 steps while following policy π_θ. We can first travel from s to a middle point s' (any state can be a middle point, $s' \in \mathcal{S}$) after k steps and then go to the final state x during the last step. In this way, we are able to update the visitation probability recursively: $\rho^\pi(s \to x, k+1) = \sum_{s'} \rho^\pi(s \to s', k) \rho^\pi(s' \to x, 1)$.

Then we go back to unroll the recursive representation of $\nabla_\theta V^\pi(s)$! Let $\phi(s) = \sum_{a \in \mathcal{A}} \nabla_\theta \pi_\theta(a \vert s)Q^\pi(s, a)$ to simplify the maths. If we keep on extending $\nabla_\theta V^\pi(.)$ infinitely, it is easy to find out that we can transition from the starting state s to any state after any number of steps in this unrolling process and by summing up all the visitation probabilities, we get $\nabla_\theta V^\pi(s)$!

$$
\begin{aligned}
& {\color{red}{\nabla_\theta V^\pi(s)}} \\
=& \phi(s) + \sum_a \pi_\theta(a \vert s) \sum_{s'} P(s' \vert s,a) {\color{red}{\nabla_\theta V^\pi(s')}} \\
=& \phi(s) + \sum_{s'} \sum_a \pi_\theta(a \vert s) P(s' \vert s,a) {\color{red}{\nabla_\theta V^\pi(s')}} \\
=& \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1) {\color{red}{\nabla_\theta V^\pi(s')}} \\
=& \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1) {\color{red}{\nabla_\theta V^\pi(s')}} \\
=& \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1) {\color{red}{[\phi(s') + \sum_{s''} \rho^\pi(s' \to s'', 1) \nabla_\theta V^\pi(s'')]}} \\
=& \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1) \phi(s') + \sum_{s''} \rho^\pi(s \to s'', 2){\color{red}{\nabla_\theta V^\pi(s'')}} \scriptstyle{\text{ ; Consider }s'\text{ as the middle point for }s \to s''}\\
=& \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1) \phi(s') + \sum_{s''} \rho^\pi(s \to s'', 2)\phi(s'') + \sum_{s'''} \rho^\pi(s \to s''', 3){\color{red}{\nabla_\theta V^\pi(s''')}} \\
=& \dots \scriptstyle{\text{; Repeatedly unrolling the part of }\nabla_\theta V^\pi(.)} \\
=& \sum_{x\in\mathcal{S}}\sum_{k=0}^\infty \rho^\pi(s \to x, k) \phi(x)
\end{aligned}
$$

The nice rewriting above allows us to exclude the derivative of Q-value function, $\nabla_\theta Q^\pi(s, a)$. By plugging it into the objective function $J(\theta)$, we are getting the following:

$$
\begin{aligned}
\nabla_\theta J(\theta)
&= \nabla_\theta V^\pi(s_0) & \scriptstyle{\text{; Starting from a random state } s_0} \\
&= \sum_{s}{\color{blue}{\sum_{k=0}^\infty \rho^\pi(s_0 \to s, k)}} \phi(s) &\scriptstyle{\text{; Let }{\color{blue}{\eta(s) = \sum_{k=0}^\infty \rho^\pi(s_0 \to s, k)}}} \\
&= \sum_{s}\eta(s) \phi(s) & \\
&= \Big({\sum_s \eta(s)} \Big)\sum_{s}\frac{\eta(s)}{\sum_s \eta(s)} \phi(s) & \scriptstyle{\text{; Normalize } \eta(s), s\in\mathcal{S} \text{ to be a probability distribution.}}\\
&\propto \sum_s \frac{\eta(s)}{\sum_s \eta(s)} \phi(s) & \scriptstyle{\sum_s \eta(s) \text{ is a constant}} \\
&= \sum_s d^\pi(s) \sum_a \nabla_\theta \pi_\theta(a \vert s)Q^\pi(s, a) & \scriptstyle{d^\pi(s) = \frac{\eta(s)}{\sum_s \eta(s)}\text{ is stationary distribution.}}
\end{aligned}
$$

In the episodic case, the constant of proportionality ($\sum_s \eta(s)$) is the average length of an episode; in the continuing case, it is 1 (Sutton & Barto, 2017; Sec. 13.2). The gradient can be further written as:

$$
\begin{aligned}
\nabla_\theta J(\theta) &\propto \sum_{s \in \mathcal{S}} d^\pi(s) \sum_{a \in \mathcal{A}} Q^\pi(s,a) \nabla_\theta \pi_\theta(a|s) \\
&= \sum_{s \in \mathcal{S}} d^\pi(s) \sum_{a \in \mathcal{A}} \pi_\theta(a|s) Q^\pi(s,a) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \\
&= \mathbb{E}_\pi[Q^\pi(s,a) \nabla_\theta \ln \pi_\theta(a|s)] & \scriptstyle{\text{; Because } (\ln x)' = 1/x}
\end{aligned}
$$

Where $\mathbb{E}_\pi$ refers to $\mathbb{E}_{s \sim d_\pi, a \sim \pi_\theta}$ when both state and action distributions follow the policy π_θ (on policy).

The policy gradient theorem lays the theoretical foundation for various policy gradient algorithms. This vanilla policy gradient update has no bias but high variance. Many following algorithms were proposed to reduce the variance while keeping the bias unchanged.

$$
\nabla_\theta J(\theta) = \mathbb{E}_\pi[Q^\pi(s,a) \nabla_\theta \ln \pi_\theta(a|s)]
$$

Here is a nice summary of a general form of policy gradient methods borrowed from the GAE (general advantage estimation) paper (Schulman et al., 2016) and this post thoroughly discussed several components in GAE , highly recommended.

Policy gradient methods maximize the expected total reward by repeatedly estimating the gradient $g := \nabla_\theta \mathbb{E}\left[\sum_{t=0}^{\infty} r_t\right]$. There are several different related expressions for the policy gradient, which have the form

$$
g = \mathbb{E}\left[\sum_{t=0}^{\infty} \Psi_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\right], \tag{1}
$$

where Ψ_t may be one of the following:

1. $\sum_{t=0}^{\infty} r_t$: total reward of the trajectory.
2. $\sum_{t'=t}^{\infty} r_{t'}$: reward following action a_t.
3. $\sum_{t'=t}^{\infty} r_{t'} - b(s_t)$: baselined version of previous formula.
4. $Q^\pi(s_t, a_t)$: state-action value function.
5. $A^\pi(s_t, a_t)$: advantage function.
6. $r_t + V^\pi(s_{t+1}) - V^\pi(s_t)$: TD residual.

The latter formulas use the definitions

$$
V^\pi(s_t) := \mathbb{E}_{\substack{s_{t+1:\infty}, \\ a_{t:\infty}}}\left[\sum_{l=0}^{\infty} r_{t+l}\right] \qquad Q^\pi(s_t, a_t) := \mathbb{E}_{\substack{s_{t+1:\infty}, \\ a_{t+1:\infty}}}\left[\sum_{l=0}^{\infty} r_{t+l}\right] \tag{2}
$$

$$
A^\pi(s_t, a_t) := Q^\pi(s_t, a_t) - V^\pi(s_t), \quad \text{(Advantage function).} \tag{3}
$$

Fig. 1. A general form of policy gradient methods. (Image source: Schulman et al., 2016)

Policy Gradient Algorithms

Tons of policy gradient algorithms have been proposed during recent years and there is no way for me to exhaust them. I'm introducing some of them that I happened to know and read about.

REINFORCE

REINFORCE (Monte-Carlo policy gradient) relies on an estimated return by Monte-Carlo methods using episode samples to update the policy parameter θ. REINFORCE works because the expectation of the sample gradient is equal to the actual gradient:

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_\pi[Q^\pi(s,a) \nabla_\theta \ln \pi_\theta(a|s)] \\
&= \mathbb{E}_\pi[G_t \nabla_\theta \ln \pi_\theta(A_t|S_t)] & \scriptstyle{\text{; Because } Q^\pi(S_t,A_t) = \mathbb{E}_\pi[G_t|S_t,A_t]}
\end{aligned}
$$

Therefore we are able to measure G_t from real sample trajectories and use that to update our policy gradient. It relies on a full trajectory and that's why it is a Monte-Carlo method.

The process is pretty straightforward:

1. Initialize the policy parameter θ at random.
2. Generate one trajectory on policy π_θ: $S_1, A_1, R_2, S_2, A_2, \dots, S_T$.
3. For t=1, 2, … , T:
    1. Estimate the the return G_t;
    2. Update policy parameters: $\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla_\theta \ln \pi_\theta(A_t|S_t)$

A widely used variation of REINFORCE is to subtract a baseline value from the return G_t to *reduce the variance of gradient estimation while keeping the bias unchanged* (Remember we always want to do this when possible). For example, a common baseline is to subtract state-value from action-value, and if applied, we would use advantage $A(s,a) = Q(s,a) - V(s)$ in the gradient ascent update. This post nicely explained why a baseline works for reducing the variance, in addition to a set of fundamentals of policy gradient.

Actor-Critic

Two main components in policy gradient are the policy model and the value function. It makes a lot of sense to learn the value function in addition to the policy, since knowing the value function can assist the policy update, such as by reducing gradient variance in vanilla policy gradients, and that is exactly what the **Actor-Critic** method does.

Actor-critic methods consist of two models, which may optionally share parameters:

- **Critic** updates the value function parameters w and depending on the algorithm it could be action-value $Q_w(a|s)$ or state-value $V_w(s)$.
- **Actor** updates the policy parameters θ for $\pi_\theta(a|s)$, in the direction suggested by the critic.

Let's see how it works in a simple action-value actor-critic algorithm.

1. Initialize s, θ, w at random; sample $a \sim \pi_\theta(a|s)$.
2. For $t = 1 \dots T$:
    1. Sample reward $r_t \sim R(s,a)$ and next state $s' \sim P(s'|s,a)$;
    2. Then sample the next action $a' \sim \pi_\theta(a'|s')$;
    3. Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s,a)\nabla_\theta \ln \pi_\theta(a|s)$;
    4. Compute the correction (TD error) for action-value at time t:
       $\delta_t = r_t + \gamma Q_w(s',a') - Q_w(s,a)$
       and use it to update the parameters of action-value function:
       $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s,a)$
    5. Update $a \leftarrow a'$ and $s \leftarrow s'$.

Two learning rates, α_θ and α_w, are predefined for policy and value function parameter updates respectively.

Off-Policy Policy Gradient

Both REINFORCE and the vanilla version of actor-critic method are on-policy: training samples are collected according to the target policy — the very same policy that we try to optimize for. Off policy methods, however, result in several additional advantages:

1. The off-policy approach does not require full trajectories and can reuse any past episodes ("experience replay") for much better sample efficiency.
2. The sample collection follows a behavior policy different from the target policy, bringing better exploration.

Now let's see how off-policy policy gradient is computed. The behavior policy for collecting samples is a known policy (predefined just like a hyperparameter), labelled as $\beta(a|s)$. The objective function sums up the reward over the state distribution defined by this behavior policy:

$$J(\theta) = \sum_{s\in\mathcal{S}} d^\beta(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a)\pi_\theta(a|s) = \mathbb{E}_{s\sim d^\beta}\big[\sum_{a\in\mathcal{A}} Q^\pi(s,a)\pi_\theta(a|s)\big]$$

where $d^\beta(s)$ is the stationary distribution of the behavior policy β; recall that $d^\beta(s) = \lim_{t\to\infty} P(S_t = s|S_0, \beta)$; and Q^π is the action-value function estimated with regard to the target policy π (not the behavior policy!).

Given that the training observations are sampled by $a \sim \beta(a|s)$, we can rewrite the gradient as:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \mathbb{E}_{s\sim d^\beta}\Big[\sum_{a\in\mathcal{A}} Q^\pi(s,a)\pi_\theta(a|s)\Big] & \\
&= \mathbb{E}_{s\sim d^\beta}\Big[\sum_{a\in\mathcal{A}} \big(Q^\pi(s,a)\nabla_\theta\pi_\theta(a|s) + \color{red}{\pi_\theta(a|s)\nabla_\theta Q^\pi(s,a)}\big)\Big] & \scriptstyle{\text{; Derivative product rule.}} \\
&\overset{(i)}{\approx} \mathbb{E}_{s\sim d^\beta}\Big[\sum_{a\in\mathcal{A}} Q^\pi(s,a)\nabla_\theta\pi_\theta(a|s)\Big] & \scriptstyle{\text{; Ignore the red part: } \color{red}{\pi_\theta(a|s)\nabla_\theta Q^\pi(s,a)}}. \\
&= \mathbb{E}_{s\sim d^\beta}\Big[\sum_{a\in\mathcal{A}} \beta(a|s)\frac{\pi_\theta(a|s)}{\beta(a|s)} Q^\pi(s,a)\frac{\nabla_\theta\pi_\theta(a|s)}{\pi_\theta(a|s)}\Big] & \\
&= \mathbb{E}_\beta\Big[\frac{\color{blue}{\pi_\theta(a|s)}}{\color{blue}{\beta(a|s)}} Q^\pi(s,a)\nabla_\theta\ln\pi_\theta(a|s)\Big] & \scriptstyle{\text{; The blue part is the importance weight.}}
\end{aligned}$$

where $\frac{\pi_\theta(a|s)}{\beta(a|s)}$ is the importance weight. Because Q^π is a function of the target policy and thus a function of policy parameter θ, we should take the derivative of $\nabla_\theta Q^\pi(s,a)$ as well according to the product rule. However, it is super hard to compute $\nabla_\theta Q^\pi(s,a)$ in reality. Fortunately if we use an approximated gradient with the gradient of Q ignored, we still guarantee the policy improvement and eventually achieve the true local minimum. This is justified in the proof here (Degris, White & Sutton, 2012).

In summary, when applying policy gradient in the off-policy setting, we can simple adjust it with a weighted sum and the weight is the ratio of the target policy to the behavior policy, $\frac{\pi_\theta(a|s)}{\beta(a|s)}$.

A3C

[paper|code]

Asynchronous Advantage Actor-Critic (Mnih et al., 2016), short for **A3C**, is a classic policy gradient method with a special focus on parallel training.

In A3C, the critics learn the value function while multiple actors are trained in parallel and get synced with global parameters from time to time. Hence, A3C is designed to work well for parallel training.

Let's use the state-value function as an example. The loss function for state value is to minimize the mean squared error, $J_v(w) = (G_t - V_w(s))^2$ and gradient descent can be applied to find the optimal w. This state-value function is used as the baseline in the policy gradient update.

1. We have global parameters, θ and w; similar thread-specific parameters, θ' and w'.
2. Initialize the time step $t = 1$
3. While $T <= T_{\text{MAX}}$:
    1. Reset gradient: dθ = 0 and dw = 0.
    2. Synchronize thread-specific parameters with global ones: θ' = θ and w' = w.
    3. t_{start} = t and sample a starting state s_t.
    4. While (s_t != TERMINAL) and $t - t_{\text{start}} <= t_{\text{max}}$:
        1. Pick the action $A_t \sim \pi_{\theta'}(A_t|S_t)$ and receive a new reward R_t and a new state s_{t+1}.
        2. Update t = t + 1 and T = T + 1
    5. Initialize the variable that holds the return estimation

    $$R = \begin{cases} 0 & \text{if } s_t \text{ is TERMINAL} \\ V_{w'}(s_t) & \text{otherwise} \end{cases}$$

    6. For $i = t-1, \dots, t_{\text{start}}$:
        1. $R \leftarrow \gamma R + R_i$; here R is a MC measure of G_i.
        2. Accumulate gradients w.r.t. θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi_{\theta'}(a_i|s_i)(R - V_{w'}(s_i))$;
           Accumulate gradients w.r.t. w': $dw \leftarrow dw + 2(R - V_{w'}(s_i))\nabla_{w'}(R - V_{w'}(s_i))$.
    7. Update asynchronously θ using dθ, and w using dw.

A3C enables the parallelism in multiple agent training. The gradient accumulation step (6.2) can be considered as a parallelized reformation of minibatch-based stochastic gradient update: the values of w or θ get corrected by a little bit in the direction of each training thread independently.

A2C

[paper|code]

A2C is a synchronous, deterministic version of A3C; that's why it is named as "A2C" with the first "A" ("asynchronous") removed. In A3C each agent talks to the global parameters independently, so it is possible sometimes the thread-specific agents would be playing with policies of different versions and therefore the aggregated update would not be optimal. To resolve the inconsistency, a coordinator in A2C waits for all the parallel actors to finish their work before updating the global parameters and then in the next iteration parallel actors starts from the same policy. The synchronized gradient update keeps the training more cohesive and potentially to make convergence faster.

A2C has been shown to be able to utilize GPUs more efficiently and work better with large batch sizes while achieving same or better performance than A3C.

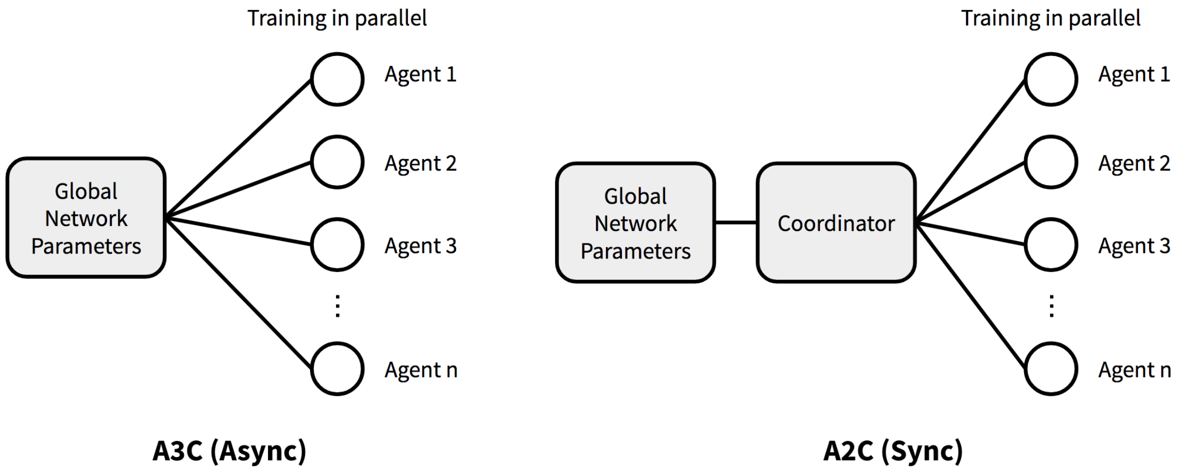

Fig. 2. The architecture of A3C versus A2C.

[paper|code]

In methods described above, the policy function $\pi(.|s)$ is always modeled as a probability distribution over actions $\mathcal{A}$ given the current state and thus it is *stochastic*. **Deterministic policy gradient (DPG)** instead models the policy as a deterministic decision: $a = \mu(s)$. It may look bizarre — how can you calculate the gradient of the policy function when it outputs a single action? Let's look into it step by step.

Refresh on a few notations to facilitate the discussion:

- $\rho_0(s)$: The initial distribution over states
- $\rho^\mu(s \to s', k)$: Starting from state s, the visitation probability density at state s' after moving k steps by policy μ.
- $\rho^\mu(s')$: Discounted state distribution, defined as $\rho^\mu(s') = \int_\mathcal{S} \sum_{k=1}^\infty \gamma^{k-1} \rho_0(s) \rho^\mu(s \to s', k) ds$.

The objective function to optimize for is listed as follows:

$$J(\theta) = \int_\mathcal{S} \rho^\mu(s) Q(s, \mu_\theta(s)) ds$$

Deterministic policy gradient theorem: Now it is the time to compute the gradient! According to the chain rule, we first take the gradient of Q w.r.t. the action a and then take the gradient of the deterministic policy function μ w.r.t. θ:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \int_\mathcal{S} \rho^\mu(s) \nabla_a Q^\mu(s, a) \nabla_\theta \mu_\theta(s) \rvert_{a=\mu_\theta(s)} ds \\
&= \mathbb{E}_{s \sim \rho^\mu} [\nabla_a Q^\mu(s, a) \nabla_\theta \mu_\theta(s) \rvert_{a=\mu_\theta(s)}]
\end{aligned}$$

We can consider the deterministic policy as a *special case* of the stochastic one, when the probability distribution contains only one extreme non-zero value over one action. Actually, in the DPG paper, the authors have shown that if the stochastic policy $\pi_{\mu_\theta, \sigma}$ is re-parameterized by a deterministic policy μ_θ and a variation variable σ, the stochastic policy is eventually equivalent to the deterministic case when $\sigma=0$. Compared to the deterministic policy, we expect the stochastic policy to require more samples as it integrates the data over the whole state and action space.

The deterministic policy gradient theorem can be plugged into common policy gradient frameworks.

Let's consider an example of on-policy actor-critic algorithm to showcase the procedure. In each iteration of on-policy actor-critic, two actions are taken deterministically $a = \mu_\theta(s)$ and the SARSA update on policy parameters relies on the new gradient that we just computed above:

$$\begin{aligned}
\delta_t &= R_t + \gamma Q_w(s_{t+1}, a_{t+1}) - Q_w(s_t, a_t) & \text{; TD error in SARSA}\\
w_{t+1} &= w_t + \alpha_w \delta_t \nabla_w Q_w(s_t, a_t) & \\
\theta_{t+1} &= \theta_t + \alpha_\theta \color{red}{\nabla_a Q_w(s_t, a_t) \nabla_\theta \mu_\theta(s) \rvert_{a=\mu_\theta(s)}} & \text{; Deterministic policy gradient theorem}
\end{aligned}$$

However, unless there is sufficient noise in the environment, it is very hard to guarantee enough exploration due to the determinacy of the policy. We can either add noise into the policy (ironically this makes it nondeterministic!) or learn it off-policy-ly by following a different stochastic behavior policy to collect samples.

Say, in the off-policy approach, the training trajectories are generated by a stochastic policy $\beta(a|s)$ and thus the state distribution follows the corresponding discounted state density ρ^β:

$$J_\beta(\theta) = \int_\mathcal{S} \rho^\beta Q^\mu(s, \mu_\theta(s)) ds$$
$$\nabla_\theta J_\beta(\theta) = \mathbb{E}_{s\sim\rho^\beta}[\nabla_a Q^\mu(s,a)\nabla_\theta \mu_\theta(s)|_{a=\mu_\theta(s)}]$$

Note that because the policy is deterministic, we only need $Q^\mu(s, \mu_\theta(s))$ rather than $\sum_a \pi(a|s)Q^\pi(s,a)$ as the estimated reward of a given state s. In the off-policy approach with a stochastic policy, importance sampling is often used to correct the mismatch between behavior and target policies, as what we have described above. However, because the deterministic policy gradient removes the integral over actions, we can avoid importance sampling.

DDPG

[paper|code]

DDPG (Lillicrap, et al., 2015), short for **Deep Deterministic Policy Gradient**, is a model-free off-policy actor-critic algorithm, combining DPG with DQN. Recall that DQN (Deep Q-Network) stabilizes the learning of Q-function by experience replay and the frozen target network. The original DQN works in discrete space, and DDPG extends it to continuous space with the actor-critic framework while learning a deterministic policy.

In order to do better exploration, an exploration policy μ' is constructed by adding noise $\mathcal{N}$:

$$\mu'(s) = \mu_\theta(s) + \mathcal{N}$$

In addition, DDPG does soft updates ("conservative policy iteration") on the parameters of both actor and critic, with $\tau \ll 1$: $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$. In this way, the target network values are constrained to change slowly, different from the design in DQN that the target network stays frozen for some period of time.

One detail in the paper that is particularly useful in robotics is on how to normalize the different physical units of low dimensional features. For example, a model is designed to learn a policy with the robot's positions and velocities as input; these physical statistics are different by nature and even statistics of the same type may vary a lot across multiple robots. Batch normalization is applied to fix it by normalizing every dimension across samples in one minibatch.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Fig 3. DDPG Algorithm. (Image source: Lillicrap, et al., 2015)

D4PG

[paper|code (Search "github d4pg" and you will see a few.)]

Distributed Distributional DDPG (D4PG) applies a set of improvements on DDPG to make it run in the distributional fashion.

(1) **Distributional Critic**: The critic estimates the expected Q value as a random variable ~ a distribution Z_w parameterized by w and therefore $Q_w(s, a) = \mathbb{E}Z_w(x, a)$. The loss for learning the distribution parameter is to minimize some measure of the distance between two distributions — distributional TD error: $L(w) = \mathbb{E}[d(\mathcal{T}_{\mu_\theta}, Z_{w'}(s, a), Z_w(s, a)]$, where $\mathcal{T}_{\mu_\theta}$ is the Bellman operator.

The deterministic policy gradient update becomes:

$$\begin{aligned} \nabla_\theta J(\theta) &\approx \mathbb{E}_{\rho^\mu}[\nabla_a Q_w(s, a)\nabla_\theta \mu_\theta(s)|_{a=\mu_\theta(s)}] && \text{; gradient update in DPG} \\ &= \mathbb{E}_{\rho^\mu}[\mathbb{E}[\nabla_a Q_w(s, a)]\nabla_\theta \mu_\theta(s)|_{a=\mu_\theta(s)}] && \text{; expectation of the Q-value distribution.} \end{aligned}$$

(2) **N-step returns**: When calculating the TD error, D4PG computes N-step TD target rather than one-step to incorporate rewards in more future steps. Thus the new TD target is:

$$r(s_0, a_0) + \mathbb{E}[\sum_{n=1}^{N-1} r(s_n, a_n) + \gamma^N Q(s_N, \mu_\theta(s_N))|s_0, a_0]$$

(3) **Multiple Distributed Parallel Actors**: D4PG utilizes K independent actors, gathering experience in parallel and feeding data into the same replay buffer.

(4) **Prioritized Experience Replay (PER)**: The last piece of modification is to do sampling from the replay buffer of size R with an non-uniform probability p_i. In this way, a sample i has the probability $(Rp_i)^{-1}$ to be selected and thus the importance weight is $(Rp_i)^{-1}$.

Algorithm 1 D4PG

Input: batch size M, trajectory length N, number of actors K, replay size R, exploration constant ϵ, initial learning rates α_0 and β_0

Initialize network weights (θ, w) at random
Initialize target weights $(\theta', w') \leftarrow (\theta, w)$
Launch K actors and replicate network weights (θ, w) to each actor
for $t = 1, \dots, T$ **do**
    Sample M transitions $(\mathbf{x}_{i:i+N}, \mathbf{a}_{i:i+N-1}, r_{i:i+N-1})$ of length N from replay with priority p_i
    Construct the target distributions $Y_i = \sum_{n=0}^{N-1} \gamma^n r_{i+n} + \gamma^N Z_{w'}(\mathbf{x}_{i+N}, \pi_{\theta'}(\mathbf{x}_{i+N}))$ TD(n)
    Compute the actor and critic updates

importance weight

$$\text{Q update:} \quad \delta_w = \frac{1}{M} \sum_i \nabla_w \boxed{(Rp_i)^{-1}} d(Y_i, Z_w(\mathbf{x}_i, \mathbf{a}_i))$$

Q estimated as a random variable ~ distribution Z_w

$$\text{Policy update:} \quad \delta_\theta = \frac{1}{M} \sum_i \nabla_\theta \pi_\theta(\mathbf{x}_i) \, \mathbb{E}[\nabla_\mathbf{a} Z_w(\mathbf{x}_i, \mathbf{a})] \big|_{\mathbf{a}=\pi_\theta(\mathbf{x}_i)}$$

    Update network parameters $\theta \leftarrow \theta + \alpha_t \delta_\theta$, $w \leftarrow w + \beta_t \delta_w$
    If $t = 0 \mod t_{\text{target}}$, update the target networks $(\theta', w') \leftarrow (\theta, w)$
    If $t = 0 \mod t_{\text{actors}}$, replicate network weights to the actors
end for
return policy parameters θ

Actor (Periodically updated)

repeat
    Sample action $\mathbf{a} = \pi_\theta(\mathbf{x}) + \epsilon \mathcal{N}(0, 1)$
    Execute action $\mathbf{a}$, observe reward r and state $\mathbf{x}'$
    Store $(\mathbf{x}, \mathbf{a}, r, \mathbf{x}')$ in replay
until learner finishes

Fig. 4. D4PG algorithm (Image source: Barth-Maron, et al. 2018); Note that in the original paper, the variable letters are chosen slightly differently from what in the post; i.e. I use $\mu(.)$ for representing a deterministic policy instead of $\pi(.)$.

MADDPG

[paper|code]

Multi-agent DDPG (**MADDPG**) (Lowe et al., 2017)extends DDPG to an environment where multiple agents are coordinating to complete tasks with only local information. In the viewpoint of one agent, the environment is non-stationary as policies of other agents are quickly upgraded and remain unknown. MADDPG is an actor-critic model redesigned particularly for handling such a changing environment and interactions between agents.

The problem can be formalized in the multi-agent version of MDP, also known as *Markov games*. Say, there are N agents in total with a set of states $\mathcal{S}$. Each agent owns a set of possible action, $\mathcal{A}_1, \dots, \mathcal{A}_N$, and a set of observation, $\mathcal{O}_1, \dots, \mathcal{O}_N$. The state transition function involves all states, action and observation spaces $\mathcal{T} : \mathcal{S} \times \mathcal{A}_1 \times \dots \mathcal{A}_N \mapsto \mathcal{S}$. Each agent's stochastic policy only involves its own state and action: $\pi_{\theta_i} : \mathcal{O}_i \times \mathcal{A}_i \mapsto [0, 1]$, a probability distribution over actions given its own observation, or a deterministic policy: $\mu_{\theta_i} : \mathcal{O}_i \mapsto \mathcal{A}_i$.

Let $\vec{o} = o_1, \dots, o_N$, $\vec{\mu} = \mu_1, \dots, \mu_N$ and the policies are parameterized by $\vec{\theta} = \theta_1, \dots, \theta_N$.

The critic in MADDPG learns a centralized action-value function $Q^{\vec{\mu}}_i(\vec{o}, a_1, \dots, a_N)$ for the i-th agent, where $a_1 \in \mathcal{A}_1, \dots, a_N \in \mathcal{A}_N$ are actions of all agents. Each $Q^{\vec{\mu}}_i$ is learned separately for $i = 1, \dots, N$ and therefore multiple agents can have arbitrary reward structures, including conflicting rewards in a competitive setting. Meanwhile, multiple actors, one for each agent, are exploring and upgrading the policy parameters θ_i on their own.

Actor update:

$$\nabla_{\theta_i} J(\theta_i) = \mathbb{E}_{\vec{o}, a \sim \mathcal{D}} [\nabla_{a_i} Q^{\vec{\mu}}_i (\vec{o}, a_1, \dots, a_N) \nabla_{\theta_i} \mu_{\theta_i}(o_i) \rvert_{a_i=\mu_{\theta_i}(o_i)}]$$

Where $\mathcal{D}$ is the memory buffer for experience replay, containing multiple episode samples $(\vec{o}, a_1, \dots, a_N, r_1, \dots, r_N, \vec{o}')$ — given current observation $\vec{o}$, agents take action $a_1, \dots, a_N$ and get rewards $r_1, \dots, r_N$, leading to the new observation $\vec{o}'$.

Critic update:

$$\begin{aligned} \mathcal{L}(\theta_i) &= \mathbb{E}_{\vec{o}, a_1, \dots, a_N, r_1, \dots, r_N, \vec{o}'}[(Q^{\vec{\mu}}_i(\vec{o}, a_1, \dots, a_N) - y)^2] & \\ \text{where } y &= r_i + \gamma Q^{\vec{\mu}'}_i (\vec{o}', a'_1, \dots, a'_N) \rvert_{a'_j = \mu'_{\theta_j}} & \scriptstyle{\text{; TD target!}} \end{aligned}$$

where $\vec{\mu}'$ are the target policies with delayed softly-updated parameters.

If the policies $\vec{\mu}$ are unknown during the critic update, we can ask each agent to learn and evolve its own approximation of others' policies. Using the approximated policies, MADDPG still can learn efficiently although the inferred policies might not be accurate.

To mitigate the high variance triggered by the interaction between competing or collaborating agents in the environment, MADDPG proposed one more element - *policy ensembles*:

1. Train K policies for one agent;
2. Pick a random policy for episode rollouts;
3. Take an ensemble of these K policies to do gradient update.

In summary, MADDPG added three additional ingredients on top of DDPG to make it adapt to the multi-agent environment:

- Centralized critic + decentralized actors;
- Actors are able to use estimated policies of other agents for learning;
- Policy ensembling is good for reducing variance.

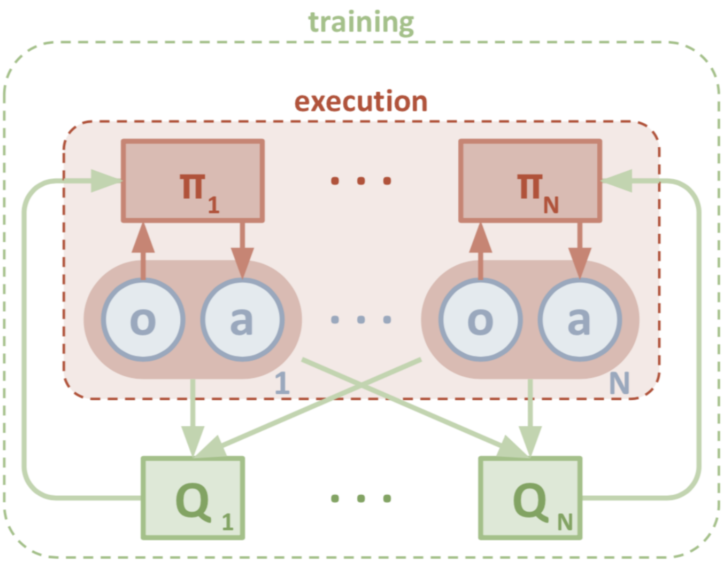

Fig. 5. The architecture design of MADDPG. (Image source: Lowe et al., 2017)

TRPO

[paper|code]

To improve training stability, we should avoid parameter updates that change the policy too much at one step. **Trust region policy optimization (TRPO)** (Schulman, et al., 2015) carries out this idea by enforcing a KL divergence constraint on the size of policy update at each iteration.

If off policy, the objective function measures the total advantage over the state visitation distribution and actions, while the rollout is following a different behavior policy $\beta(a|s)$:

$$\begin{aligned}
J(\theta) &= \sum_{s\in\mathcal{S}} \rho^{\pi_{\theta_\text{old}}} \sum_{a\in\mathcal{A}} \big(\pi_\theta(a|s) \hat{A}_{\theta_\text{old}}(s,a) \big) & \\
&= \sum_{s\in\mathcal{S}} \rho^{\pi_{\theta_\text{old}}} \sum_{a\in\mathcal{A}} \big(\beta(a|s) \frac{\pi_\theta(a|s)}{\beta(a|s)} \hat{A}_{\theta_\text{old}}(s,a) \big) & \scriptstyle{\text{; Importance sampling}} \\
&= \mathbb{E}_{s \sim \rho^{\pi_{\theta_\text{old}}}, a \sim \beta} \big[\frac{\pi_\theta(a|s)}{\beta(a|s)} \hat{A}_{\theta_\text{old}}(s,a) \big] &
\end{aligned}$$

where θ_old is the policy parameters before the update and thus known to us; $\rho^{\pi_{\theta_\text{old}}}$ is defined in the same way as above; $\beta(a|s)$ is the behavior policy for collecting trajectories. Noted that we use an estimated advantage $\hat{A}(.)$ rather than the true advantage function $A(.)$ because the true rewards are usually unknown.

If on policy, the behavior policy is $\pi_{\theta_\text{old}}(a|s)$:

$$J(\theta) = \mathbb{E}_{s \sim \rho^{\pi_{\theta_\text{old}}}, a \sim \pi_{\theta_\text{old}}} \big[\frac{\pi_\theta(a|s)}{\pi_{\theta_\text{old}}(a|s)} \hat{A}_{\theta_\text{old}}(s,a) \big]$$

TRPO aims to maximize the objective function $J(\theta)$ subject to, *trust region constraint* which enforces the distance between old and new policies measured by KL-divergence to be small enough, within a parameter δ:

$$\mathbb{E}_{s \sim \rho^{\pi_{\theta_\text{old}}}} [D_\text{KL}(\pi_{\theta_\text{old}}(.|s) \| \pi_\theta(.|s)] \leq \delta$$

In this way, the old and new policies would not diverge too much when this hard constraint is met. While still, TRPO can guarantee a monotonic improvement over policy iteration (Neat, right?). Please read the proof in the paper if interested :)

PPO

[paper|code]

Given that TRPO is relatively complicated and we still want to implement a similar constraint, **proximal policy optimization (PPO)** simplifies it by using a clipped surrogate objective while retaining similar performance.

First, let's denote the probability ratio between old and new policies as:

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_\text{old}}(a|s)}$$

Then, the objective function of TRPO (on policy) becomes:

$$J^\text{TRPO} (\theta) = \mathbb{E} [r(\theta) \hat{A}_{\theta_\text{old}}(s, a)]$$

Without a limitation on the distance between θ_old and θ, to maximize $J^\text{TRPO} (\theta)$ would lead to instability with extremely large parameter updates and big policy ratios. PPO imposes the constraint by forcing r(θ) to stay within a small interval around 1, precisely [1-ε, 1+ε], where ε is a hyperparameter.

$$J^\text{CLIP} (\theta) = \mathbb{E} [\min(r(\theta) \hat{A}_{\theta_\text{old}}(s, a), \text{clip}(r(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_{\theta_\text{old}}(s, a))]$$

The function $\text{clip}(r(\theta), 1-\epsilon, 1+\epsilon)$ clips the ratio within [1-ε, 1+ε]. The objective function of PPO takes the minimum one between the original value and the clipped version and therefore we lose the motivation for increasing the policy update to extremes for better rewards.

When applying PPO on the network architecture with shared parameters for both policy (actor) and value (critic) functions, in addition to the clipped reward, the objective function is augmented with an error term on the value estimation (formula in red) and an entropy term (formula in blue) to encourage sufficient exploration.

$$J^{\text{CLIP'}}(\theta) = \mathbb{E}[J^{\text{CLIP}}(\theta) - c_1(V_\theta(s) - V_{\text{target}})^2 + c_2 H(s, \pi_\theta(.))]$$

where Both c_1 and c_2 are two hyperparameter constants.

PPO has been tested on a set of benchmark tasks and proved to produce awesome results with much greater simplicity.

ACER

[paper|code]

ACER, short for **actor-critic with experience replay** (Wang, et al., 2017), is an off-policy actor-critic model with experience replay, greatly increasing the sample efficiency and decreasing the data correlation. A3C builds up the foundation for ACER, but it is on policy; ACER is A3C's off-policy counterpart. The major obstacle to making A3C off policy is how to control the stability of the off-policy estimator. ACER proposes three designs to overcome it:

- Use Retrace Q-value estimation;
- Truncate the importance weights with bias correction;
- Apply efficient TRPO.

Retrace Q-value Estimation

Retrace is an off-policy return-based Q-value estimation algorithm with a nice guarantee for convergence for any target and behavior policy pair (π, β), plus good data efficiency.

Recall how TD learning works for prediction:

1. Compute TD error: $\delta_t = R_t + \gamma \mathbb{E}_{a \sim \pi} Q(S_{t+1}, a) - Q(S_t, A_t)$; the term $r_t + \gamma \mathbb{E}_{a \sim \pi} Q(s_{t+1}, a)$ is known as "TD target". The expectation $\mathbb{E}_{a \sim \pi}$ is used because for the future step the best estimation we can make is what the return would be if we follow the current policy π.
2. Update the value by correcting the error to move toward the goal: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \delta_t$. In other words, the incremental update on Q is proportional to the TD error: $\Delta Q(S_t, A_t) = \alpha \delta_t$.

When the rollout is off policy, we need to apply importance sampling on the Q update:

$$\Delta Q^{\text{imp}}(S_t, A_t) = \gamma^t \prod_{1 \leq \tau \leq t} \frac{\pi(A_\tau | S_\tau)}{\beta(A_\tau | S_\tau)} \delta_t$$

The product of importance weights looks pretty scary when we start imagining how it can cause super high variance and even explode. Retrace Q-value estimation method modifies ΔQ to have importance weights truncated by no more than a constant c:

$$\Delta Q^{\text{ret}}(S_t, A_t) = \gamma^t \prod_{1 \leq \tau \leq t} \min(c, \frac{\pi(A_\tau | S_\tau)}{\beta(A_\tau | S_\tau)}) \delta_t$$

ACER uses Q^{ret} as the target to train the critic by minimizing the L2 error term: $(Q^{\text{ret}}(s,a) - Q(s,a))^2$.

Importance weights truncation

To reduce the high variance of the policy gradient $\hat{g}$, ACER truncates the importance weights by a constant c, plus a correction term. The label $\hat{g}_t^{\text{acer}}$ is the ACER policy gradient at time t.

$$
\begin{aligned}
\hat{g}_t^{\text{acer}} = & \omega_t \big(Q^{\text{ret}}(S_t, A_t) - V_{\theta_v}(S_t) \big) \nabla_\theta \ln \pi_\theta(A_t \vert S_t) & \scriptstyle{\text{; Let }\omega_t=\frac{\pi(A_t\vert S_t)}{\beta(A_t\vert S_t)}} \\
= & \color{blue}{\min(c, \omega_t) \big(Q^{\text{ret}}(S_t, A_t) - V_w(S_t) \big) \nabla_\theta \ln \pi_\theta(A_t \vert S_t)} \\
& + \color{red}{\mathbb{E}_{a \sim \pi} \big[\max(0, \frac{\omega_t(a) - c}{\omega_t(a)}) \big(Q_w(S_t, a) - V_w(S_t) \big) \nabla_\theta \ln \pi_\theta(a \vert S_t) \big]} & \scriptstyle{\text{; Let }\omega_t(a)=\frac{\pi(a\vert S_t)}{\beta(a\vert S_t)}}
\end{aligned}
$$

where $Q_w(.)$ and $V_w(.)$ are value functions predicted by the critic with parameter w. The first term (blue) contains the clipped important weight. The clipping helps reduce the variance, in addition to subtracting state value function $V_w(.)$ as a baseline. The second term (red) makes a correction to achieve unbiased estimation.

Efficient TRPO

Furthermore, ACER adopts the idea of TRPO but with a small adjustment to make it more computationally efficient: rather than measuring the KL divergence between policies before and after one update, ACER maintains a running average of past policies and forces the updated policy to not deviate far from this average.

The ACER paper is pretty dense with many equations. Hopefully, with the prior knowledge on TD learning, Q-learning, importance sampling and TRPO, you will find the paper slightly easier to follow :)

ACTKR

[paper|code]

ACKTR (actor-critic using Kronecker-factored trust region) (Yuhuai Wu, et al., 2017) proposed to use Kronecker-factored approximation curvature (K-FAC) to do the gradient update for both the critic and actor. K-FAC made an improvement on the computation of *natural gradient*, which is quite different from our *standard gradient*. Here is a nice, intuitive explanation of natural gradient. One sentence summary is probably:

> "we first consider all combinations of parameters that result in a new network a constant KL divergence away from the old network. This constant value can be viewed as the step size or learning rate. Out of all these possible combinations, we choose the one that minimizes our loss function."

I listed ACTKR here mainly for the completeness of this post, but I would not dive into details, as it involves a lot of theoretical knowledge on natural gradient and optimization methods. If interested, check these papers/posts, before reading the ACKTR paper:

- Amari. Natural Gradient Works Efficiently in Learning. 1998
- Kakade. A Natural Policy Gradient. 2002
- A intuitive explanation of natural gradient descent
- Wiki: Kronecker product
- Martens & Grosse. Optimizing neural networks with kronecker-factored approximate curvature. 2015.

> "This approximation is built in two stages. In the first, the rows and columns of the Fisher are divided into groups, each of which corresponds to all the weights in a given layer, and this gives rise to a block-partitioning of the matrix. These blocks are then approximated as Kronecker products between much smaller matrices, which we show is equivalent to making certain approximating assumptions regarding the statistics of the network's gradients.

> In the second stage, this matrix is further approximated as having an inverse which is either block-diagonal or block-tridiagonal. We justify this approximation through a careful examination of the relationships between inverse covariances, tree-structured graphical models, and linear regression. Notably, this justification doesn't apply to the Fisher itself, and our experiments confirm that while the inverse Fisher does indeed possess this structure (approximately), the Fisher itself does not."

SAC

[paper|code]

Soft Actor-Critic (SAC) (Haarnoja et al. 2018) incorporates the entropy measure of the policy into the reward to encourage exploration: we expect to learn a policy that acts as randomly as possible while it is still able to succeed at the task. It is an off-policy actor-critic model following the maximum entropy reinforcement learning framework. A precedent work is Soft Q-learning.

Three key components in SAC:

- An actor-critic architecture with separate policy and value function networks;
- An off-policy formulation that enables reuse of previously collected data for efficiency;
- Entropy maximization to enable stability and exploration.

The policy is trained with the objective to maximize the expected return and the entropy at the same time:

$$J(\theta) = \sum_{t=1}^{T} \mathbb{E}_{(s_t, a_t) \sim \rho_{\pi_\theta}} [r(s_t, a_t) + \alpha \mathcal{H}(\pi_\theta(.|s_t))]$$

where $\mathcal{H}(.)$ is the entropy measure and α controls how important the entropy term is, known as *temperature* parameter. The entropy maximization leads to policies that can (1) explore more and (2) capture multiple modes of near-optimal strategies (i.e., if there exist multiple options that seem to be equally good, the policy should assign each with an equal probability to be chosen).

Precisely, SAC aims to learn three functions:

- The policy with parameter θ, π_θ.
- Soft Q-value function parameterized by w, Q_w.
- Soft state value function parameterized by ψ, V_ψ; theoretically we can infer V by knowing Q and π, but in practice, it helps stabilize the training.

Soft Q-value and soft state value are defined as:

$$\begin{aligned} Q(s_t, a_t) &= r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim \rho_\pi(s)} [V(s_{t+1})] && \text{; according to Bellman equation.} \\ \text{where } V(s_t) &= \mathbb{E}_{a_t \sim \pi} [Q(s_t, a_t) - \alpha \log \pi(a_t|s_t)] && \text{; soft state value function.} \end{aligned}$$

$$\text{Thus, } Q(s_t, a_t) = r(s_t, a_t) + \gamma \mathbb{E}_{(s_{t+1}, a_{t+1}) \sim \rho_\pi} [Q(s_{t+1}, a_{t+1}) - \alpha \log \pi(a_{t+1} | s_{t+1})]$$

$\rho_\pi(s)$ and $\rho_\pi(s, a)$ denote the state and the state-action marginals of the state distribution induced by the policy $\pi(a|s)$; see the similar definitions in DPG section.

The soft state value function is trained to minimize the mean squared error:

$$\begin{aligned} J_V(\psi) &= \mathbb{E}_{s_t \sim \mathcal{D}} [\frac{1}{2} \big(V_\psi(s_t) - \mathbb{E}[Q_w(s_t, a_t) - \log \pi_\theta(a_t | s_t)]\big)^2] \\ \text{with gradient: } \nabla_\psi J_V(\psi) &= \nabla_\psi V_\psi(s_t)\big(V_\psi(s_t) - Q_w(s_t, a_t) + \log \pi_\theta (a_t | s_t) \big) \end{aligned}$$

where $\mathcal{D}$ is the replay buffer.

The soft Q function is trained to minimize the soft Bellman residual:

$$\begin{aligned} J_Q(w) &= \mathbb{E}_{(s_t, a_t) \sim \mathcal{D}} [\frac{1}{2}\big(Q_w(s_t, a_t) - (r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim \rho_\pi(s)}[V_{\bar{\psi}}(s_{t+1})]) \big)^2] \\ \text{with gradient: } \nabla_w J_Q(w) &= \nabla_w Q_w(s_t, a_t) \big(Q_w(s_t, a_t) - r(s_t, a_t) - \gamma V_{\bar{\psi}}(s_{t+1})\big) \end{aligned}$$

where $\bar{\psi}$ is the target value function which is the exponential moving average (or only gets updated periodically in a "hard" way), just like how the parameter of the target Q network is treated in DQN to stabilize the training.

SAC updates the policy to minimize the KL-divergence:

$$\begin{aligned} \pi_\text{new} &= \arg\min_{\pi' \in \Pi} D_\text{KL} \Big(\pi'(.\vert s_t) \| \frac{\exp(Q^{\pi_\text{old}}(s_t, .))}{Z^{\pi_\text{old}}(s_t)} \Big) \\[6pt] &= \arg\min_{\pi' \in \Pi} D_\text{KL} \big(\pi'(.\vert s_t) \| \exp(Q^{\pi_\text{old}}(s_t, .) - \log Z^{\pi_\text{old}}(s_t)) \big) \\[6pt] \text{objective for update: } J_\pi(\theta) &= \nabla_\theta D_\text{KL} \big(\pi_\theta(. \vert s_t) \| \exp(Q_w(s_t, .) - \log Z_w(s_t)) \big) \\[6pt] &= \mathbb{E}_{a_t\sim\pi} \Big[- \log \big(\frac{\exp(Q_w(s_t, a_t) - \log Z_w(s_t))}{\pi_\theta(a_t \vert s_t)} \big) \Big] \\[6pt] &= \mathbb{E}_{a_t\sim\pi} [\log \pi_\theta(a_t \vert s_t) - Q_w(s_t, a_t) + \log Z_w(s_t)] \end{aligned}$$

where Π is the set of potential policies that we can model our policy as to keep them tractable; for example, Π can be the family of Gaussian mixture distributions, expensive to model but highly expressive and still tractable. $Z^{\pi_\text{old}}(s_t)$ is the partition function to normalize the distribution. It is usually intractable but does not contribute to the gradient. How to minimize $J_\pi(\theta)$ depends our choice of Π.

This update guarantees that $Q^{\pi_\text{new}}(s_t, a_t) \geq Q^{\pi_\text{old}}(s_t, a_t)$, please check the proof on this lemma in the Appendix B.2 in the original paper.

Once we have defined the objective functions and gradients for soft action-state value, soft state value and the policy network, the soft actor-critic algorithm is straightforward:

Algorithm 1 Soft Actor-Critic

Inputs: The learning rates, λ_π, λ_Q, and λ_V for functions π_θ, Q_w, and V_ψ respectively; the weighting factor τ for exponential moving average.

Initialize parameters θ, w, ψ, and $\bar{\psi}$.
for each iteration **do**
    (In practice, a combination of a single environment step and multiple gradient steps is found to work best.)
    for each environment setup **do**
        $a_t \sim \pi_\theta(a_t|s_t)$
        $s_{t+1} \sim \rho_\pi(s_{t+1}|s_t, a_t)$
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t, a_t, r(s_t, a_t), s_{t+1}\}$
    for each gradient update step **do**
        $\psi \leftarrow \psi - \lambda_V \nabla_\psi J_V(\psi)$.
        $w \leftarrow w - \lambda_Q \nabla_w J_Q(w)$.
        $\theta \leftarrow \theta - \lambda_\pi \nabla_\theta J_\pi(\theta)$.
        $\bar{\psi} \leftarrow \tau\psi + (1-\tau)\bar{\psi}$.

Fig. 6. The soft actor-critic algorithm. (Image source: original paper)

SAC with Automatically Adjusted Temperature

[paper|code]

SAC is brittle with respect to the temperature parameter. Unfortunately it is difficult to adjust temperature, because the entropy can vary unpredictably both across tasks and during training as the policy becomes better. An improvement on SAC formulates a constrained optimization problem: while maximizing the expected return, the policy should satisfy a minimum entropy constraint:

$$\max_{\pi_0, \dots, \pi_T} \mathbb{E}\Big[\sum_{t=0}^T r(s_t, a_t)\Big] \text{s.t. } \forall t, \mathcal{H}(\pi_t) \geq \mathcal{H}_0$$

where $\mathcal{H}_0$ is a predefined minimum policy entropy threshold.

The expected return $\mathbb{E}\Big[\sum_{t=0}^T r(s_t, a_t)\Big]$ can be decomposed into a sum of rewards at all the time steps. Because the policy π_t at time t has no effect on the policy at the earlier time step, π_{t-1}, we can maximize the return at different steps backward in time — this is essentially **DP**.

$$\underbrace{\max_{\pi_0} \Big(\mathbb{E}[r(s_0, a_0)] + \underbrace{\max_{\pi_1} \Big(\mathbb{E}[\dots] + \underbrace{\max_{\pi_T} \mathbb{E}[r(s_T, a_T)]}_\text{1st maximization} \Big)}_\text{second but last maximization} \Big)}_\text{last maximization}$$

where we consider $\gamma = 1$.

So we start the optimization from the last timestep T:

$$\text{maximize } \mathbb{E}_{(s_T, a_T) \sim \rho_{\pi}} [r(s_T, a_T)] \text{ s.t. } \mathcal{H}(\pi_T) - \mathcal{H}_0 \geq 0$$

First, let us define the following functions:

$$\begin{aligned}
h(\pi_T) &= \mathcal{H}(\pi_T) - \mathcal{H}_0 = \mathbb{E}_{(s_T, a_T) \sim \rho_{\pi}} [-\log \pi_T(a_T|s_T)] - \mathcal{H}_0\\
f(\pi_T) &= \begin{cases}
\mathbb{E}_{(s_T, a_T) \sim \rho_{\pi}} [r(s_T, a_T)], & \text{if }h(\pi_T) \geq 0 \\
-\infty, & \text{otherwise}
\end{cases}
\end{aligned}$$

And the optimization becomes:

$$\text{maximize } f(\pi_T) \text{ s.t. } h(\pi_T) \geq 0$$

To solve the maximization optimization with inequality constraint, we can construct a Lagrangian expression with a Lagrange multiplier (also known as "dual variable"), α_T:

$$L(\pi_T, \alpha_T) = f(\pi_T) + \alpha_T h(\pi_T)$$

Considering the case when we try to *minimize* $L(\pi_T, \alpha_T)$ *with respect to* α_T - given a particular value π_T,

- If the constraint is satisfied, $h(\pi_T) \geq 0$, at best we can set $\alpha_T = 0$ since we have no control over the value of $f(\pi_T)$. Thus, $L(\pi_T, 0) = f(\pi_T)$.
- If the constraint is invalidated, $h(\pi_T) < 0$, we can achieve $L(\pi_T, \alpha_T) \to -\infty$ by taking $\alpha_T \to \infty$. Thus, $L(\pi_T, \infty) = -\infty = f(\pi_T)$.

In either case, we can recover the following equation,

$$f(\pi_T) = \min_{\alpha_T \geq 0} L(\pi_T, \alpha_T)$$

At the same time, we want to maximize $f(\pi_T)$,

$$\max_{\pi_T} f(\pi_T) = \min_{\alpha_T \geq 0} \max_{\pi_T} L(\pi_T, \alpha_T)$$

Therefore, to maximize $f(\pi_T)$, the dual problem is listed as below. Note that to make sure $\max_{\pi_T} f(\pi_T)$ is properly maximized and would not become $-\infty$, the constraint has to be satisfied.

$$\begin{aligned}
\max_{\pi_T} \mathbb{E}[r(s_T, a_T)] &= \max_{\pi_T} f(\pi_T) \\
&= \min_{\alpha_T \geq 0} \max_{\pi_T} L(\pi_T, \alpha_T) \\
&= \min_{\alpha_T \geq 0} \max_{\pi_T} f(\pi_T) + \alpha_T h(\pi_T) \\
&= \min_{\alpha_T \geq 0} \max_{\pi_T} \mathbb{E}_{(s_T, a_T) \sim \rho_\pi}[r(s_T, a_T)] + \alpha_T (\mathbb{E}_{(s_T, a_T) \sim \rho_\pi}[-\log \pi_T(a_T|s_T)] - \mathcal{H}_0) \\
&= \min_{\alpha_T \geq 0} \max_{\pi_T} \mathbb{E}_{(s_T, a_T) \sim \rho_\pi}[r(s_T, a_T) - \alpha_T \log \pi_T(a_T|s_T)] - \alpha_T \mathcal{H}_0 \\
&= \min_{\alpha_T \geq 0} \max_{\pi_T} \mathbb{E}_{(s_T, a_T) \sim \rho_\pi}[r(s_T, a_T) + \alpha_T \mathcal{H}(\pi_T) - \alpha_T \mathcal{H}_0]
\end{aligned}$$

We could compute the optimal π_T and α_T iteratively. First given the current α_T, get the best policy π_T^* that maximizes $L(\pi_T^*, \alpha_T)$. Then plug in π_T^* and compute α_T^* that minimizes $L(\pi_T^*, \alpha_T)$. Assuming we have one neural network for policy and one network for temperature parameter, the iterative update process is more aligned with how we update network parameters during training.

$$\begin{aligned}
\pi_T^* &= \arg\max_{\pi_T} \mathbb{E}_{(s_T, a_T) \sim \rho_\pi}[r(s_T, a_T) + \alpha_T \mathcal{H}(\pi_T) - \alpha_T \mathcal{H}_0] \\
\alpha_T^* &= \arg\min_{\alpha_T \geq 0} \mathbb{E}_{(s_T, a_T) \sim \rho_{\pi^*}}[\alpha_T \mathcal{H}(\pi_T^*) - \alpha_T \mathcal{H}_0]
\end{aligned}$$

$$\text{Thus, } \max_{\pi_T} \mathbb{E}[r(s_T, a_T)] = \mathbb{E}_{(s_T, a_T) \sim \rho_{\pi^*}}[r(s_T, a_T) + \alpha_T^* \mathcal{H}(\pi_T^*) - \alpha_T^* \mathcal{H}_0]$$

Now let's go back to the soft Q value function:

$$\begin{aligned}
Q_{T-1}(s_{T-1}, a_{T-1}) &= r(s_{T-1}, a_{T-1}) + \mathbb{E}[Q(s_T, a_T) - \alpha_T \log \pi(a_T|s_T)] \\
&= r(s_{T-1}, a_{T-1}) + \mathbb{E}[r(s_T, a_T)] + \alpha_T \mathcal{H}(\pi_T) \\
Q_{T-1}^*(s_{T-1}, a_{T-1}) &= r(s_{T-1}, a_{T-1}) + \max_{\pi_T} \mathbb{E}[r(s_T, a_T)] + \alpha_T \mathcal{H}(\pi_T^*) & \text{; plug in the optimal } \pi_T^*
\end{aligned}$$

Therefore the expected return is as follows, when we take one step further back to the time step $T-1$:

$$
\begin{aligned}
&\max_{\pi_{T-1}} \Big(\mathbb{E}[r(s_{T-1}, a_{T-1})] + \max_{\pi_T} \mathbb{E}[r(s_T, a_T)]\Big) \\
&= \max_{\pi_{T-1}} \Big(Q^*_{T-1}(s_{T-1}, a_{T-1}) - \alpha^*_T \mathcal{H}(\pi^*_T)\Big) & \text{; should s.t. } \mathcal{H}(\pi_{T-1}) - \\
&= \min_{\alpha_{T-1} \geq 0} \max_{\pi_{T-1}} \Big(Q^*_{T-1}(s_{T-1}, a_{T-1}) - \alpha^*_T \mathcal{H}(\pi^*_T) + \alpha_{T-1}\big(\mathcal{H}(\pi_{T-1}) - \mathcal{H}_0\big)\Big) & \text{; dual problem w/ Lagr} \\
&= \min_{\alpha_{T-1} \geq 0} \max_{\pi_{T-1}} \Big(Q^*_{T-1}(s_{T-1}, a_{T-1}) + \alpha_{T-1}\mathcal{H}(\pi_{T-1}) - \alpha_{T-1}\mathcal{H}_0\Big) - \alpha^*_T \mathcal{H}(\pi^*_T)
\end{aligned}
$$

Similar to the previous step,

$$
\begin{aligned}
\pi^*_{T-1} &= \arg\max_{\pi_{T-1}} \mathbb{E}_{(s_{T-1}, a_{T-1}) \sim \rho_\pi} [Q^*_{T-1}(s_{T-1}, a_{T-1}) + \alpha_{T-1}\mathcal{H}(\pi_{T-1}) - \alpha_{T-1}\mathcal{H}_0] \\
\alpha^*_{T-1} &= \arg\min_{\alpha_{T-1} \geq 0} \mathbb{E}_{(s_{T-1}, a_{T-1}) \sim \rho_{\pi^*}} [\alpha_{T-1}\mathcal{H}(\pi^*_{T-1}) - \alpha_{T-1}\mathcal{H}_0]
\end{aligned}
$$

The equation for updating α_{T-1} in green has the same format as the equation for updating α_{T-1} in blue above. By repeating this process, we can learn the optimal temperature parameter in every step by minimizing the same objective function:

$$
J(\alpha) = \mathbb{E}_{a_t \sim \pi_t} [-\alpha \log \pi_t(a_t \mid \pi_t) - \alpha \mathcal{H}_0]
$$

The final algorithm is same as SAC except for learning α explicitly with respect to the objective $J(\alpha)$ (see Fig. 7):

Algorithm 1 Soft Actor-Critic

Input: θ_1, θ_2, ϕ ▷ Initial parameters
 $\bar{\theta}_1 \leftarrow \theta_1, \bar{\theta}_2 \leftarrow \theta_2$ ▷ Initialize target network weights
 $\mathcal{D} \leftarrow \emptyset$ ▷ Initialize an empty replay pool
 for each iteration **do**
  for each environment step **do**
   $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$ ▷ Sample action from the policy
   $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ ▷ Sample transition from the environment
   $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$ ▷ Store the transition in the replay pool
  end for
  for each gradient step **do**
   $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$ ▷ Update the Q-function parameters
   $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$ ▷ Update policy weights
   $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$ ▷ Adjust temperature
   $\bar{\theta}_i \leftarrow \tau\theta_i + (1 - \tau)\bar{\theta}_i$ for $i \in \{1, 2\}$ ▷ Update target network weights
  end for
 end for
Output: θ_1, θ_2, ϕ ▷ Optimized parameters

Fig. 7. The soft actor-critic algorithm with automatically adjusted temperature. (Image source: original paper)

TD3

[paper|code]

The Q-learning algorithm is commonly known to suffer from the overestimation of the value function. This overestimation can propagate through the training iterations and negatively affect the policy. This property directly motivated Double Q-learning and Double DQN: the action selection and Q-value update are decoupled by using two value networks.

Twin Delayed Deep Deterministic (short for **TD3**; Fujimoto et al., 2018) applied a couple of tricks on DDPG to prevent the overestimation of the value function:

(1) **Clipped Double Q-learning**: In Double Q-Learning, the action selection and Q-value estimation are made by two networks separately. In the DDPG setting, given two deterministic actors $(\mu_{\theta_1}, \mu_{\theta_2})$ with two corresponding critics (Q_{w_1}, Q_{w_2}), the Double Q-learning Bellman targets look like:

$$\begin{aligned}
y_1 &= r + \gamma Q_{w_2}(s', \mu_{\theta_1}(s')) \\
y_2 &= r + \gamma Q_{w_1}(s', \mu_{\theta_2}(s'))
\end{aligned}$$

However, due to the slow changing policy, these two networks could be too similar to make independent decisions. The *Clipped Double Q-learning* instead uses the minimum estimation among two so as to favor underestimation bias which is hard to propagate through training:

$$\begin{aligned}
y_1 &= r + \gamma \min_{i=1,2} Q_{w_i}(s', \mu_{\theta_1}(s')) \\
y_2 &= r + \gamma \min_{i=1,2} Q_{w_i}(s', \mu_{\theta_2}(s'))
\end{aligned}$$

(2) **Delayed update of Target and Policy Networks**: In the actor-critic model, policy and value updates are deeply coupled: Value estimates diverge through overestimation when the policy is poor, and the policy will become poor if the value estimate itself is inaccurate.

To reduce the variance, TD3 updates the policy at a lower frequency than the Q-function. The policy network stays the same until the value error is small enough after several updates. The idea is similar to how the periodically-updated target network stay as a stable objective in DQN.

(3) **Target Policy Smoothing**: Given a concern with deterministic policies that they can overfit to narrow peaks in the value function, TD3 introduced a smoothing regularization strategy on the value function: adding a small amount of clipped random noises to the selected action and averaging over mini-batches.

$$\begin{aligned}
y &= r + \gamma Q_w(s', \mu_\theta(s') + \epsilon) & \\
\epsilon &\sim \text{clip}(\mathcal{N}(0, \sigma), -c, +c) & \scriptstyle{\text{; clipped random noises.}}
\end{aligned}$$

This approach mimics the idea of SARSA update and enforces that similar actions should have similar values.

Here is the final algorithm:

Algorithm 1 TD3

Initialize critic networks Q_{θ_1}, Q_{θ_2}, and actor network π_ϕ with random parameters θ_1, θ_2, ϕ

Initialize target networks $\theta'_1 \leftarrow \theta_1, \theta'_2 \leftarrow \theta_2, \phi' \leftarrow \phi$

Initialize replay buffer $\mathcal{B}$

for $t = 1$ **to** T **do**

  Select action with exploration noise $a \sim \pi(s) + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward r and new state s'

  Store transition tuple (s, a, r, s') in $\mathcal{B}$

  Sample mini-batch of N transitions (s, a, r, s') from $\mathcal{B}$

  $\tilde{a} \leftarrow \pi_{\phi'}(s) + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$ Target policy smoothing

  $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$ Clipped Double Q-learning

  Update critics $\theta_i \leftarrow \min_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$

  if t mod d **then** Delayed update of target and policy networks

    Update ϕ by the deterministic policy gradient:

    $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$

    Update target networks:

    $\theta'_i \leftarrow \tau\theta_i + (1 - \tau)\theta'_i$

    $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$

  end if

end for

Fig 8. TD3 Algorithm. (Image source: Fujimoto et al., 2018)

Quick Summary

After reading through all the algorithms above, I list a few building blocks or principles that seem to be common among them:

- Try to reduce the variance and keep the bias unchanged to stabilize learning.
- Off-policy gives us better exploration and helps us use data samples more efficiently.
- Experience replay (training data sampled from a replay memory buffer);
- Target network that is either frozen periodically or updated slower than the actively learned policy network;
- Batch normalization;
- Entropy-regularized reward;
- The critic and actor can share lower layer parameters of the network and two output heads for policy and value functions.
- It is possible to learn with deterministic policy rather than stochastic one.
- Put constraint on the divergence between policy updates.
- New optimization methods (such as K-FAC).
- Entropy maximization of the policy helps encourage exploration.
- Try not to overestimate the value function.
- TBA more.

If you notice mistakes and errors in this post, don't hesitate to contact me at [lilian dot wengweng at gmail dot com] and I would be very happy to correct them right away!

See you in the next post :D

References

[1] jeremykun.com Markov Chain Monte Carlo Without all the Bullshit

[2] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction; 2nd Edition. 2017.

[3] John Schulman, et al. "High-dimensional continuous control using generalized advantage estimation." ICLR 2016.

[4] Thomas Degris, Martha White, and Richard S. Sutton. "Off-policy actor-critic." ICML 2012.

[5] timvieira.github.io Importance sampling

[6] Mnih, Volodymyr, et al. "Asynchronous methods for deep reinforcement learning." ICML. 2016.

[7] David Silver, et al. "Deterministic policy gradient algorithms." ICML. 2014.

[8] Timothy P. Lillicrap, et al. "Continuous control with deep reinforcement learning." arXiv preprint arXiv:1509.02971 (2015).

[9] Ryan Lowe, et al. "Multi-agent actor-critic for mixed cooperative-competitive environments." NIPS. 2017.

[10] John Schulman, et al. "Trust region policy optimization." ICML. 2015.

[11] Ziyu Wang, et al. "Sample efficient actor-critic with experience replay." ICLR 2017.

[12] Rémi Munos, Tom Stepleton, Anna Harutyunyan, and Marc Bellemare. "Safe and efficient off-policy reinforcement learning" NIPS. 2016.

[13] Yuhuai Wu, et al. "Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation." NIPS. 2017.

[14] kvfrans.com A intuitive explanation of natural gradient descent

[15] Sham Kakade. "A Natural Policy Gradient.". NIPS. 2002.

[16] "Going Deeper Into Reinforcement Learning: Fundamentals of Policy Gradients." - Seita's Place, Mar 2017.

[17] "Notes on the Generalized Advantage Estimation Paper." - Seita's Place, Apr, 2017.

[18] Gabriel Barth-Maron, et al. "Distributed Distributional Deterministic Policy Gradients." ICLR 2018 poster.

[19] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor." arXiv preprint arXiv:1801.01290 (2018).

[20] Scott Fujimoto, Herke van Hoof, and Dave Meger. "Addressing Function Approximation Error in Actor-Critic Methods." arXiv preprint arXiv:1802.09477 (2018).

[21] Tuomas Haarnoja, et al. "Soft Actor-Critic Algorithms and Applications." arXiv preprint arXiv:1812.05905 (2018).

[22] David Knowles. "Lagrangian Duality for Dummies" Nov 13, 2010.

← A (Long) Peek into Reinforcement Learning

yafan tao • 4 months ago

小姐姐好厉害，看到你写的文章，涉及这么多知识，我要视你为榜样，向你学习~

3 • Reply • Share ›

王健树 • 11 days ago

nice blog! I want to use some content of this page in my blog, hope to get your permission.

1 • Reply • Share ›

Jack Wang • 2 months ago

Can you introduce the DDPG in detail? Expect to see

1 • Reply • Share ›

민철희 • 21 days ago

Thanks for the insightful explanation and consistent update with SOTA algorithms!

I think there's a typo in the description of the loss function of EC-SAC!

J(\alpha) = \mathbb{E}_{a_t \sim \pi_t} [-\alpha \log \pi_t(a_t\mid\pi_t) - \alpha \mathcal{H}_0]

which I think should be

J(\alpha) = \mathbb{E}_{a_t \sim \pi_t} [-\alpha \log \pi_t(a_t\mid\s_t) - \alpha \mathcal{H}_0].

• Reply • Share ›

Tessa van der Heiden • a month ago

Hi Lilian,

Thanks for your well described overview!

Why are there different terms used to describe the update steps for the two networks in an actor-critic network (DDPG). For the critic they use the LOSS, while for the actor they use the GRADIENT?

In code the LOSS for the actor is simply the negative LOSS of the critic.
In algorithm description it is the product of the GRADIENT of the critic and actor.

• Reply • Share ›

> **민철희** ➦ Tessa van der Heiden • a month ago
>
> I think the original authors described in this way "on purpose", to emphasize the result of the DPG theorem, since the paper was the first deep RL method that used DPG theorem.
>
> • Reply • Share ›
>
> > **Tessa van der Heiden** ➦ 민철희 • a month ago
> >
> > Hi, thanks.
> >
> > What bothers me, is that the implementation differs from the GRADIENT of the actor. So in the paper you see the multiplication of both GRADIENTS, while in the code you only see the LOSS, being the output of the critic. I think this is a better description: https://spinningup.openai.c...

Show more replies

Chunpai • a month ago

In your derivation of policy gradient theorem, it is not aligned with the generalized advantage estimation. There is not summation in your formula. Could you explain more on this part ? Thanks.

• Reply • Share ›

Lilian Weng Mod → Chunpai • a month ago

You can consider the equation in GAE as a Monte-Carlo way to estimate the return from a full trajectory, while the equations in policy gradient theorem work on expectation across transitions. I say they are same, but precisely which equation to use would depend on what's in your replay buffer.

• Reply • Share ›

尹一航 • a month ago

Excellent blog post!

• Reply • Share ›

Alvin Ayeni • 2 months ago

Very informative, any plans to do the same with evolutionary algorithms?

• Reply • Share ›

Lilian Weng Mod → Alvin Ayeni • a month ago

Noted. Nice suggestion. I do like EA a lot :)

• Reply • Share ›

Norio • 2 months ago

this is the best explanation i have seen!

• Reply • Share ›

Young Rick Choi • 2 months ago

This is exactly what i was looking for. Thank you sir.

• Reply • Share ›

郭帅 • 2 months ago

感谢您的文章！

• Reply • Share ›

PF • 6 months ago

Hello, why is the proportionality constant the average length in the episodic case? Maybe I misunderstood something, but we are adding up probabilities, it should be 1, shouldn't it? Thanks for the great post!

• Reply • Share ›

Lilian Weng Mod → PF • 5 months ago

Sorry i should provide more context. I saw this from Sutton & Barto book.
In the continuous task, an episode never ends, so eta can only be measured as a probability. But in the episodic task, a task can finish and eta measures the total amount of time spent in a state, so the sum is the average length of episodes. --- I believe this is how the book defines them.

I will try to clarify that more. Or just remove that sentence :P

1 • Reply • Share ›

Miguel Morales • 6 months ago

"N -step returns: When calculating the TD error, D4PG computes TD(N) rather than TD(1) to incorporate rewards in more future steps." This notation can be confusing. TD(1) is usually TD(lambda) when lambda is set to 1. TD(1) is equivalent to MC, which is the opposite of TD(0), which

^ | v • Reply • Share ›

Lilian Weng Mod → Miguel Morales • 5 months ago

You are totally right. Thank you for suggesting the correction.

^ | v • Reply • Share ›

Alexander Yau • 7 months ago

Thank you for writing such a good post.

^ | v • Reply • Share ›

ALSO ON **LILIANWENG.GITHUB.IO/LIL-LOG**

From Autoencoder to Beta-VAE

7 comments • 7 months ago

Lilian Weng — Fixed :)

Predict Stock Prices Using RNN: Part 1

4 comments • 2 years ago

Aleksandr — Thank you for your post. I have a question on input data: Why would you choose input_size=3, num_step=2 in your example …

Learning Word Embedding

3 comments • 7 months ago

Liling Tan — Thanks for the answer! =)

A (Long) Peek into Reinforcement Learning

11 comments • 7 months ago

Lilian Weng — Thank you for noticing that! fixed.