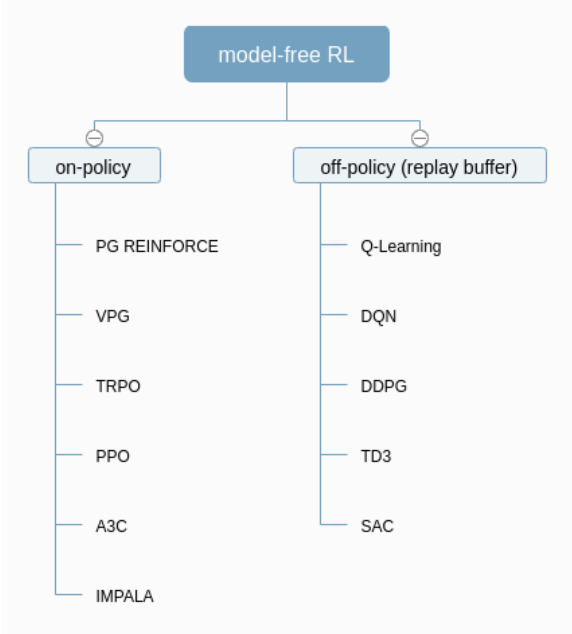
Model-free Reinforcement Learning Summary

2020.11.23 v1

李高

Contents

1 Algorithms



1) 基本概念

1. Observation:

A state s is a complete description of the state of the world.

An observation o is a partial description of a state, which may omit information.

2. Action:

Discrete action spaces, where only a finite number of moves are available to the agent.

Continuous action spaces, actions are real-valued vectors.

3. Policies:

deterministic policy

$$a_t = \mu(s_t)$$

stochastic policy

$$a_t \sim \pi(\cdot|s_t)$$

Categorical Policies (for discrete action): you have one final linear layer that gives you logits for each action, followed by a softmax to convert the logits into probabilities.

Diagonal Gaussian Policies (for contiune action): always has a neural network that maps from observations to mean actions, μθ (s).

4. Trajectories

trajectory τ 由一组 actions 及 states 组成:

$$\tau = (s_0, a_0, s_1, a_1, \ldots)$$

其中 s_0 通过 start-state distribution 采样获得： $s_0 = \rho_0(\cdot)$

5. Reward and Return

Reward function R： $r_t = R(s_t, a_t, s_{t+1})$

Return： $R(\tau) = \sum_{t=0}^{T} r_t$ 或 $R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$

2) 基本理论

The goal in RL is to select a policy which maximizes expected return when the agent acts according to it.

Let's suppose that both the environment transitions and the policy are stochastic. In this case, the probability of a T -step trajectory is:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t).$$

The expected return (for whichever measure), denoted by $J(\pi)$, is then:

$$J(\pi) = \int_{\tau} P(\tau|\pi)R(\tau) = \underset{\tau\sim\pi}{\mathrm{E}}[R(\tau)].$$

The central optimization problem in RL can then be expressed by

$$\pi^* = \arg\max_{\pi} J(\pi),$$

with π^* being the **optimal policy**.

Value Functions: value means the expected return if you start in that state or state-action pair, and then act according to a particular policy forever after.

There are four main functions of note here.

1. The **On-Policy Value Function**, $V^{\pi}(s)$, which gives the expected return if you start in state s and always act according to policy π:

$$V^{\pi}(s) = \underset{\tau \sim \pi}{\mathrm{E}}\left[R(\tau)\,|s_0 = s\right]$$

2. The **On-Policy Action-Value Function**, $Q^{\pi}(s,a)$, which gives the expected return if you start in state s, take an arbitrary action a (which may not have come from the policy), and then forever after act according to policy π:

$$Q^{\pi}(s,a) = \underset{\tau \sim \pi}{\mathrm{E}}\left[R(\tau)\,|s_0 = s, a_0 = a\right]$$

3. The **Optimal Value Function**, $V^{*}(s)$, which gives the expected return if you start in state s and always act according to the *optimal* policy in the environment:

$$V^{*}(s) = \max_{\pi} \underset{\tau \sim \pi}{\mathrm{E}}\left[R(\tau)\,|s_0 = s\right]$$

4. The **Optimal Action-Value Function**, $Q^{*}(s,a)$, which gives the expected return if you start in state s, take an arbitrary action a, and then forever after act according to the *optimal* policy in the environment:

$$Q^{*}(s,a) = \max_{\pi} \underset{\tau \sim \pi}{\mathrm{E}}\left[R(\tau)\,|s_0 = s, a_0 = a\right]$$

two key connections between the value function and the action-value function:

$$V^{\pi}(s) = \underset{a \sim \pi}{\mathrm{E}}\left[Q^{\pi}(s,a)\right],$$

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

The Optimal Q-Function and the Optimal Action:

The optimal policy in s will select whichever action maximizes the expected return from starting in s. As a result, if we have Q^{*}, we can directly obtain the optimal action, $a^{*}(s)$, via

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

Bellman Equations:

The Bellman equations for the on-policy value functions are

$$V^{\pi}(s) = \underset{\substack{a \sim \pi \\ s' \sim P}}{\mathrm{E}}\left[r(s,a) + \gamma V^{\pi}(s')\right],$$

$$Q^{\pi}(s,a) = \underset{s' \sim P}{\mathrm{E}}\left[r(s,a) + \gamma \underset{a' \sim \pi}{\mathrm{E}}\left[Q^{\pi}(s',a')\right]\right],$$

where $s' \sim P$ is shorthand for $s' \sim P(\cdot|s,a)$, indicating that the next state s' is sampled from the environment's transition rules; $a \sim \pi$ is shorthand for $a \sim \pi(\cdot|s)$; and $a' \sim \pi$ is shorthand for $a' \sim \pi(\cdot|s')$.

The Bellman equations for the optimal value functions are

$$V^{*}(s) = \max_{a} \underset{s' \sim P}{\mathrm{E}}\left[r(s,a) + \gamma V^{*}(s')\right],$$

$$Q^{*}(s,a) = \underset{s' \sim P}{\mathrm{E}}\left[r(s,a) + \gamma \max_{a'} Q^{*}(s',a')\right].$$

The crucial difference between the Bellman equations for the on-policy value functions and the optimal value functions, is the absence or presence of the max over actions. Its inclusion reflects the fact that whenever the agent gets to choose its action, in order to act optimally, it has to pick whichever action leads to the highest value.

on-policy 比较稳定，off-policy 不那么稳定的原因：

For more information about how and why Q-learning methods can fail, see 1) this classic paper by Tsitsiklis and van Roy, 2) the (much more recent) review by Szepesvari (in section 4.3.2), and 3) chapter 11 of Sutton and Barto, especially section 11.3 (on “the deadly triad” of function approximation, bootstrapping, and off-policy data, together causing instability in value-learning algorithms).

1.1 On-policy RL

1.1.1 Policy Gradient (PG) 1992

1) 理论介绍

直接根据 expected return 对 policy 进行优化： $J(\pi_\theta) = \underset{\tau\sim\pi_\theta}{E}[R(\tau)]$

由于目标是是要让 $J(\pi_\theta)$ 最大化，所以这种优化 policy 的方法被成为 policy ascent：

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_\theta)|_{\theta_k}$$

接下来就是怎么计算梯度 $\nabla_\theta J(\pi_\theta)$ 的问题，需要用的公式如下：

1. Probability of a Trajectory. The probability of a trajectory $\tau = (s_0, a_0, ..., s_{T+1})$ given that actions come from π_θ is

$$P(\tau|\theta) = \rho_0(s_0) \prod_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t).$$

2. The Log-Derivative Trick. The log-derivative trick is based on a simple rule from calculus: the derivative of $\log x$ with respect to x is $1/x$. When rearranged and combined with chain rule, we get:

$$\nabla_\theta P(\tau|\theta) = P(\tau|\theta)\nabla_\theta \log P(\tau|\theta).$$

3. Log-Probability of a Trajectory. The log-prob of a trajectory is just

$$\log P(\tau|\theta) = \log \rho_0(s_0) + \sum_{t=0}^{T} \Big(\log P(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t) \Big).$$

4. Gradients of Environment Functions. The environment has no dependence on θ, so gradients of $\rho_0(s_0)$, $P(s_{t+1}|s_t, a_t)$, and $R(\tau)$ are zero.

5. Grad-Log-Prob of a Trajectory. The gradient of the log-prob of a trajectory is thus

$$\begin{aligned} \nabla_\theta \log P(\tau|\theta) &= \cancel{\nabla_\theta \log \rho_0(s_0)} + \sum_{t=0}^{T} \Big(\cancel{\nabla_\theta \log P(s_{t+1}|s_t, a_t)} + \nabla_\theta \log \pi_\theta(a_t|s_t) \Big) \\ &= \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t). \end{aligned}$$

所以推导如下：

Derivation for Basic Policy Gradient

$$
\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \nabla_\theta \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}[R(\tau)] \\
&= \nabla_\theta \int_\tau P(\tau|\theta)R(\tau) && \text{Expand expectation} \\
&= \int_\tau \nabla_\theta P(\tau|\theta)R(\tau) && \text{Bring gradient under integral} \\
&= \int_\tau P(\tau|\theta)\nabla_\theta \log P(\tau|\theta)R(\tau) && \text{Log-derivative trick} \\
&= \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau|\theta)R(\tau)] && \text{Return to expectation form} \\
\therefore \nabla_\theta J(\pi_\theta) &= \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)\right] && \text{Expression for grad-log-prob}
\end{aligned}
$$

在 policy gradient 算法中可以对 Return 进行修改，但不影响 Expected Return mean 的原因（如具体算法中的 **reward-to-go Policy Gradient**， **Baselines Policy Gradients**）：

EGLP Lemma. Suppose that P_θ is a parameterized probability distribution over a random variable, x. Then:

$$\mathop{\mathrm{E}}_{x\sim P_\theta}[\nabla_\theta \log P_\theta(x)] = 0.$$

Proof

Recall that all probability distributions are *normalized*:

$$\int_x P_\theta(x) = 1.$$

Take the gradient of both sides of the normalization condition:

$$\nabla_\theta \int_x P_\theta(x) = \nabla_\theta 1 = 0.$$

Use the log derivative trick to get:

$$
\begin{aligned}
0 &= \nabla_\theta \int_x P_\theta(x) \\
&= \int_x \nabla_\theta P_\theta(x) \\
&= \int_x P_\theta(x)\nabla_\theta \log P_\theta(x) \\
\therefore 0 &= \mathop{\mathrm{E}}_{x\sim P_\theta}\left[\nabla_\theta \log P_\theta(x)\right].
\end{aligned}
$$

2) 具体算法

REINFORCE policy based

所以最基本的 Policy Gradient 算法 REINFORCE 就是可以根据下面这个公式获得：

$$\hat{g} = \frac{1}{|\mathcal{D}|}\sum_{\tau\in\mathcal{D}}\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau),$$

其中 $|D|$ 表示 trajectories 的数量。

具体的 REINFORCE 算法如下：

Initialize the policy parameter θ at random.

Generate one trajectory on policy π_θ: $s_1, a_1, r_2, s_2, a_2, \ldots, s_T$.

For t=1, 2, … , T:

Update policy parameters: $\theta \leftarrow \theta + \alpha R_t \nabla_\theta \log \pi_\theta(a_t|s_t)$

reward-to-go Policy Gradient（从统计上来看这样是不会改变 expectation 的）：

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1})\right]$$

Baselines Policy Gradients

根据 EGLP lemma 可以有：

$$\mathop{\mathrm{E}}_{a_t\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t) b(s_t)] = 0.$$

This allows us to add or subtract any number of terms like this from our expression for the policy gradient, without changing it in expectation:

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t)\right)\right].$$

Any function b used in this way is called a **baseline**.

baseline 是不能跟 a_t 有关，根据 EGLP，这样处理也是不会改变 expectation 的

3) 实现上的折中及不足

算法的的 expectation 是没有问题的，但每次都要去通过累加的方式计算 Return 还是有点麻烦的，并且这样单次累加的方式，算法的 variance 是会比较大的。

1.1.2 Vanilla Policy Gradient (VPG) 1999

1) 算法特点

VPG is an on-policy algorithm.

VPG can be used for environments with either discrete or continuous action spaces.

2) 产生的原因

主要是想通过 baseline 的方式及估计 Q 值去减小 REINFORCE 算法中 variance

3) 理论介绍

What we have seen so far is that the policy gradient has the general form

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\Phi_t\right],$$

where Φ_t could be any of

$$\Phi_t = R(\tau),$$

or

$$\Phi_t = \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}),$$

or

$$\Phi_t = \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t).$$

All of these choices lead to the same expected value for the policy gradient, despite having different variances. It turns out that there are two more valid choices of weights Φ_t which are important to know.

1. On-Policy Action-Value Function. The choice

$$\Phi_t = Q^{\pi_\theta}(s_t, a_t)$$

2. The Advantage Function. Recall that the advantage of an action, defined by $A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$, describes how much better or worse it is than other actions on average (relative to the current policy). This choice,

$$\Phi_t = A^{\pi_\theta}(s_t, a_t)$$

另外，关于 Q 或 V 的获得方式如下：

In practice, $V^\pi(s_t)$ cannot be computed exactly, so it has to be approximated. This is usually done with a neural network, $V_\phi(s_t)$, which is updated concurrently with the policy (so that the value network always approximates the value function of the most recent policy).

The simplest method for learning V_ϕ, used in most implementations of policy optimization algorithms (including VPG, TRPO, PPO, and A2C), is to minimize a mean-squared-error objective:

$$\phi_k = \arg\min_\phi \mathop{\mathrm{E}}_{s_t, \hat{R}_t\sim\pi_k}\left[\left(V_\phi(s_t) - \hat{R}_t\right)^2\right],$$

where π_k is the policy at epoch k. This is done with one or more steps of gradient descent, starting from the previous value parameters ϕ_{k-1}.

4) 具体算法

Algorithm 1 Vanilla Policy Gradient Algorithm

Input: initial policy parameters θ_0, initial value function parameters ϕ_0

for $k = 0, 1, 2, ...$ **do**

Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.

Compute rewards-to-go $\hat{R}_t$.

Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.

Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

Compute policy update, either using standard gradient ascent,

$$\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k,$$

or via another gradient ascent algorithm like Adam.

Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.

end for

5) 实现上的折中及不足

在训练的后期可能会存在 exploration 不足的问题：

VPG trains a stochastic policy in an on-policy way. This means that it explores by sampling actions according to the latest version of its stochastic policy. The amount of randomness in action selection depends on both initial conditions and the training procedure. Over the course of training, the policy typically becomes progressively less random, as the update rule encourages it to exploit rewards that it has already found. This may cause the policy to get trapped in local optima.

并且对于 policy 更新的步长是有比较高的要求的，不能设置太大，不然效果会比较差，设置太小则会很难收敛：

This method uses the first-order derivative and approximates the surface to be flat. If the surface has high curvature, we can make horrible moves.

Large policy change destroys training.

Cannot map changes between policy and parameter space easily.

Improper learning rate causes vanishing or exploding gradient, and Poor sample efficiency.

1.1.3 Trust Region Policy Optimization (TRPO) 2017

1) 算法特点

TRPO is an on-policy algorithm.

TRPO can be used for environments with either discrete or continuous action spaces.

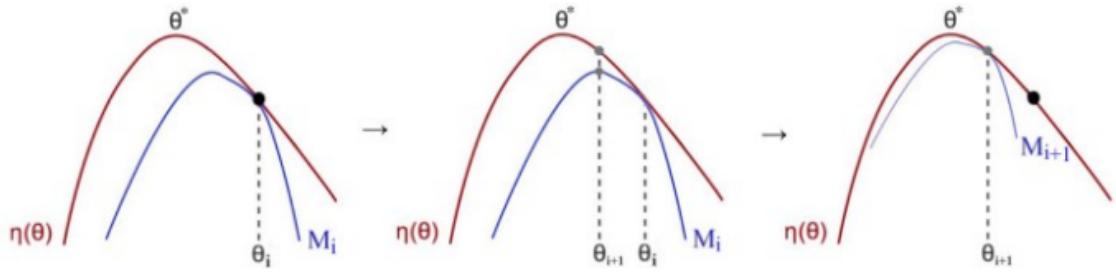
2) 产生的原因

TRPO 主要解决的就是 PG 难收敛的问题，可以保证算法在训练过程中一定能往好的方向优化，但也并不是说保证收敛到全局最优。

相比于 PG，TRPO 是可以多次利用 trajectories，只要能满足相关约束就行。

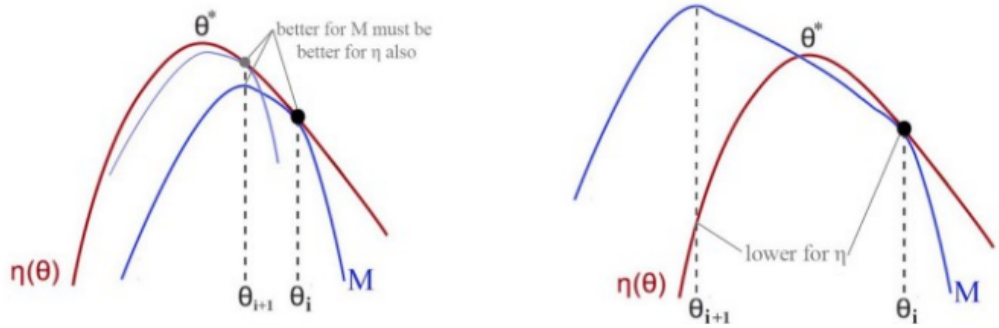
3) 理论介绍

1. Minorize-Maximization MM algorithm



如上图，假如 object function 是 $\eta(\theta)$ ，想要优化 $\eta(\theta)$ 中的 θ ，可以先找到 $\eta(\theta)$ 的一个 lower bound function M_i ，可以去优化 M_i 中的 θ 来保证 $\eta(\theta)$ 中的 θ 也是有被优化到的，迭代这个过程，最终是是可以获得 $\eta(\theta)$ 上的局部或全局最有 θ 的。

但是非 lower bound function 是没有这样的保证的，如下图所示:



2. 然后就是构造 TRPO 的 object function：

$$\underset{\pi'}{maximize}\ J(\pi') = \underset{\pi'}{maximize}\ J(\pi') - J(\pi)$$

上面公式的右边是 TRPO 要优化的 object function，其中 π' 是要更新的 policy（是一个变量），π 是要旧的 policy（是一个常量，用于获得 trajectories），$J(\pi)$ 是一般 RL 的目标函数。由于 $J(\pi)$ 是常量，所以这个等式是成立的。

3. 接下来就是寻找 $J(\pi')-J(\pi)$ 的 lower bound function：

首先用 Importance sampling 获得一个 $L_\pi(\pi')$

$$\mathcal{L}_\pi(\pi') = \frac{1}{1-\gamma}\mathop{\mathrm{E}}_{\substack{s\sim d^\pi \\ a\sim\pi}}\left[\frac{\pi'(a|s)}{\pi(a|s)}A^\pi(s,a)\right] = \mathop{\mathrm{E}}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t\frac{\pi'(a_t|s_t)}{\pi(a_t|s_t)}A^\pi(s_t,a_t)\right]$$

where

$$d^\pi(s) = (1-\gamma)\sum_{t=0}^{\infty}\gamma^t P(s_t = s|\pi)$$

d is the discounted future state distribution.

The appendix A of the TRPO paper provides a 2-page proof that establishes the following boundary:

$$|J(\pi') - (J(\pi) + \mathcal{L}_\pi(\pi'))| \le C\sqrt{\mathop{\mathrm{E}}_{s\sim d^\pi}[D_{KL}(\pi'||\pi)[s]]}$$

With some twitting, this is our final lower bound *M*.

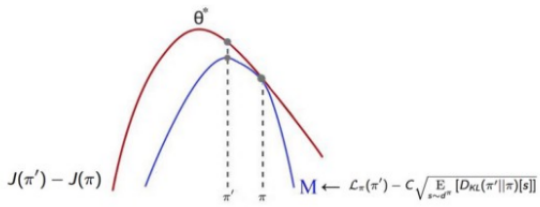
$$J(\pi') - J(\pi) \ge \underbrace{\mathcal{L}_\pi(\pi') - C\sqrt{\mathop{\mathrm{E}}_{s\sim d^\pi}[D_{KL}(\pi'||\pi)[s]]}}_{\mathrm{M}}$$

for our objective function

$$\mathop{maximize}_{\pi'} \; J(\pi') - J(\pi)$$

To summarize, we will use MM algorithms to maximize:

$$\max_{\pi'} \mathcal{L}_\pi\ (\pi') - C\sqrt{\mathop{\mathrm{E}}_{s\sim d^{\pi_k}}[D_{KL}(\pi'||\pi)[s]]}$$



为了后面的简化处理，还可以将这个 lower bound function 转化为 trust region 的形式（这也是 TPRO 名字的由来）：

The inequality equation below is important because we can establish an upper bound error for the objective calculation. This establishes a trust region on whether we can trust the result.

upper bound error

$$J(\pi') - J(\pi) \geq \mathcal{L}_{\pi}(\pi') - C\sqrt{\mathop{\mathrm{E}}_{s\sim d^{\pi}}[D_{KL}(\pi'||\pi)[s]]}$$

In fact, with the Lagrangian Duality, our objective is mathematically the same as the following using a trust region constraint.

trust region

$$\max_{\pi'} \mathcal{L}_{\pi}(\pi')$$
$$\text{s.t.} \mathop{\mathrm{E}}_{s\sim d^{\pi}}[D_{KL}(\pi'||\pi)[s]] \leq \delta$$

4. 根据上面的推导我们就获得了 $J(\pi')-J(\pi)$ 的 lower bound function，然后就是要证明这个 lower bound function 是可以持续被优化的：

TRPO 文章中证明了 $L_{\pi}(\pi)$ 是为 0 的，所以

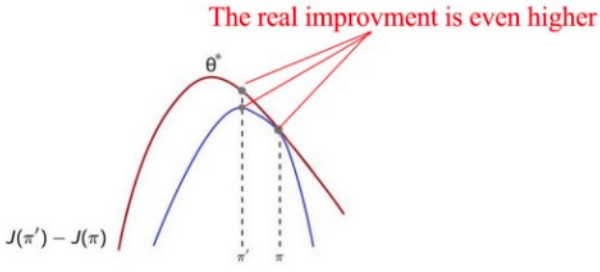
$$\mathcal{L}_{\pi}(\pi) = 0$$

The R.H.S. term below equals to zero when π' = π. Therefore, the L.H.S. is always greater or equal to 0.

$$J(\pi') - J(\pi) \geq \mathcal{L}_{\pi}(\pi') - C\sqrt{\mathop{\mathrm{E}}_{s\sim d^{\pi}}[D_{KL}(\pi'||\pi)[s]]}$$
$$\underset{=0}{\mathcal{L}_{\pi}(\pi)} - C\sqrt{\mathop{\mathrm{E}}_{s\sim d^{\pi}}[\underset{=0}{D_{KL}(\pi||\pi)[s]}]} = 0$$
$$J(\pi') - J(\pi) \geq 0$$

i.e. the new policy is always better than the old one. In fact, the new policy will have greater improvement in the real objective function than the lower bound approximation.

$$J(\pi') - J(\pi) \geq \text{improvement in M over } \pi'$$



有了上面的说明，那么 TRPO 算法中去优化 $L_\pi(\pi')$ 是可以逐渐提高 policy 的性能的，但直接去优化 $J(\pi')-J(\pi)$ 就跟一般的 PG 算法没什么区别了。

5. 接下来就是 $L_\pi(\pi')$ 的具体计算过程：

通过泰勒公式将目标函数转化为二次方程，然后就可以直接解了。

To solve it, we can use Taylor's series to expand both terms above up to the second-order. But the second-order of $\mathcal{L}$ is much smaller than the KL-divergence term and will be ignored.

$$\mathcal{L}_{\theta_k}(\theta) \approx \cancelto{0}{\mathcal{L}_{\theta_k}(\theta_k)} + g^T(\theta-\theta_k) + \dots$$
$$\bar{D}_{KL}(\theta||\theta_k) \approx \cancelto{0}{\bar{D}_{KL}(\theta_k||\theta_k)} + \cancelto{0}{\nabla_\theta \bar{D}_{KL}(\theta||\theta_k)|_{\theta_k}}(\theta-\theta_k) + \frac{1}{2}(\theta-\theta_k)^T H(\theta-\theta_k)$$

Taking out all the zero terms, it becomes:

$$\mathcal{L}_{\theta_k}(\theta) \approx g^T(\theta-\theta_k) \qquad g \doteq \nabla_\theta \mathcal{L}_{\theta_k}(\theta)\,|_{\theta_k}$$
$$\bar{D}_{KL}(\theta||\theta_k) \approx \frac{1}{2}(\theta-\theta_k)^T H(\theta-\theta_k) \qquad H \doteq \nabla^2_\theta \bar{D}_{KL}(\theta||\theta_k)\,|_{\theta_k}$$

where $\boldsymbol{g}$ is the policy gradient (the same we learn in Policy Gradient) and $\boldsymbol{H}$ measure the sensitivity (curvature) of the policy relative to the model parameter $\boldsymbol{\theta}$. Our objective turns to:

$$\theta_{k+1} = \arg\max_\theta \; g^T(\theta-\theta_k)$$
$$\text{s.t.} \;\; \frac{1}{2}(\theta-\theta_k)^T H(\theta-\theta_k) \le \delta$$

This is a quadratic equation and can be solved analytically:

$$\theta_{k+1} = \theta_k + \underbrace{\sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g}_{\text{natural policy gradient}}$$

6. 上面的优化公式就构成了 Natural Policy Gradient 算法，但是要计算出 Hessian 矩阵及其逆，是没办法用在神经网络上的，所以下面要将 Hessian 矩阵去除：

Instead of finding the inverse of FIM, we want to compute the following combined term directly

$$x_k \approx \hat{H}_k^{-1}\hat{g}_k$$

where $\boldsymbol{x}$ can be solved as:

$$\hat{H}_k x_k \approx \hat{g}$$

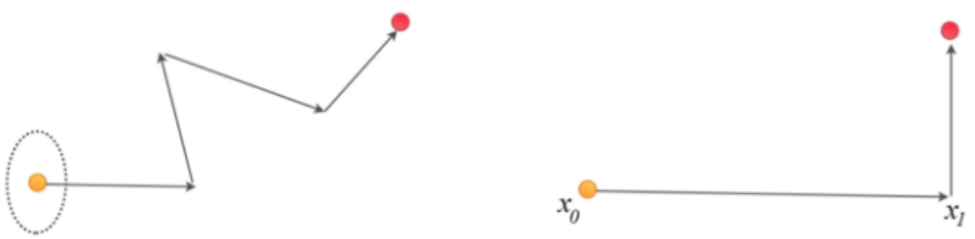
We convert this equation into an optimization problem for a quadratic equation:

$$\begin{aligned} &\textit{Solving} \quad Ax = b \\ &\textit{is equivalent to} \\ &\underset{x}{\textit{minimize}} \quad f(x) = \frac{1}{2}x^T Ax - b^T x \\ &\textit{since} \quad f'(x) = Ax - b = 0 \end{aligned}$$

In short, we can transform our problem as optimizing the quadratic equation below:

$$\min_{x \in \mathbb{R}^n} \frac{1}{2}x^T Hx - g^T x$$

这样的形式就可以不求逆了，由于上式是一个二次方程，所以用 conjugate gradient 来进行优化会比 gradient ascent 要快（至于二次方程为什么不会受系数相关性的影响我还不是很清楚），这样我们就有 Truncated Natural Policy Gradient (TNPG)算法了。



Gradient ascent　　Conjugate gradient

7. 但是上面的做法中 trust region 是比较小的，所以收敛会比较慢，TRPO 会放宽这个条件：

做法是添加一个步长 α 系数，使优化的速度可以更快一点。但如果这个步长随便给的话，算法就会回退到到一般的 PG 算法，所以 TRPO 上的做法是加入一定限制的。这种做法叫 backtracking line search，具体如下：

Algorithm 2 Line Search for TRPO

Compute proposed policy step $\Delta_k = \sqrt{\frac{2\delta}{\hat{g}_k^T \hat{H}_k^{-1} \hat{g}_k}} \hat{H}_k^{-1} \hat{g}_k$
for $j = 0, 1, 2, ..., L$ **do**
  Compute proposed update $\theta = \theta_k + \alpha^j \Delta_k$
  if $\mathcal{L}_{\theta_k}(\theta) \geq 0$ and $\bar{D}_{KL}(\theta || \theta_k) \leq \delta$ **then**
    accept the update and set $\theta_{k+1} = \theta_k + \alpha^j \Delta_k$
    break
  end if
end for

4) 具体算法

Algorithm 2 Trust Region Policy Optimization

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
Hyperparameters: KL-divergence limit δ, backtracking coefficient α, maximum number of backtracking steps K
for $k = 0, 1, 2, ...$ **do**
  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
  Compute rewards-to-go $\hat{R}_t$.
  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
  Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

  Use the conjugate gradient algorithm to compute

$$\hat{x}_k \approx \hat{H}_k^{-1} \hat{g}_k,$$

  where $\hat{H}_k$ is the Hessian of the sample average KL-divergence.
  Update the policy by backtracking line search with

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k,$$

  where $j \in \{0, 1, 2, ...K\}$ is the smallest value which improves the sample loss and satisfies the sample KL-divergence constraint.
  Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

  typically via some gradient descent algorithm.
end for

5) 实现上的折中及不足

同样到训练的后期会存在 exploration 不足的问题，一般会在 policy 的 object function 中加入 entropy。

并且实现起来会相对麻烦一点，并且要计算 Hessian 等原因，在 deep neural network 上是不是很适合的。

6) 参考

https://jonathan-hui.medium.com/rl-trust-region-policy-optimization-trpo-explained-a6ee04eeeee9

https://jonathan-hui.medium.com/rl-trust-region-policy-optimization-trpo-part-2-f51e3b2e373a

1.1.4 Proximal Policy Optimization (PPO) 2017

1) 算法特点

PPO is an on-policy algorithm.

PPO can be used for environments with either discrete or continuous action spaces.

There are two primary variants of PPO: PPO-Penalty and PPO-Clip.

PPO-Penalty approximately solves a KL-constrained update like TRPO, but penalizes the KL-divergence in the objective function instead of making it a hard constraint, and automatically adjusts the penalty coefficient over the course of training so that it's scaled appropriately.

PPO-Clip doesn't have a KL-divergence term in the objective and doesn't have a constraint at all. Instead relies on specialized clipping in the objective function to remove incentives for the new policy to get far from the old policy.

2) 产生的原因

算是 TRPO 实现的一种简化版本，但效果还是比较好的。

Because of this scalability issue, TRPO is not practical for large deep networks. PPO&ACKTR are introduced to address these problems.

TRPO tries to solve this problem with a complex second-order method, PPO is a family of first-order methods that use a few other tricks to keep new policies close to old.

3) 理论介绍

优化的主要目标还是跟 TRPO 一样的，但处理 KL 的方式有点不一样：

PPO-clip updates policies via

$$\theta_{k+1} = \arg\max_{\theta} \underset{s,a\sim\pi_{\theta_k}}{\mathrm{E}} \left[L(s, a, \theta_k, \theta)\right],$$

typically taking multiple steps of (usually minibatch) SGD to maximize the objective. Here L is given by

$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a),\ \ \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}(s, a)\right),$$

也有更简化的写法：

PPO-clip updates policies via

$$\theta_{k+1} = \arg\max_{\theta} \mathop{\mathrm{E}}_{s,a\sim\pi_{\theta_k}} \left[L(s, a, \theta_k, \theta) \right],$$

typically taking multiple steps of (usually minibatch) SGD to maximize the objective. Here L is given by

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a),\ \ \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon \right) A^{\pi_{\theta_k}}(s, a) \right),$$

直观上的表示如下：

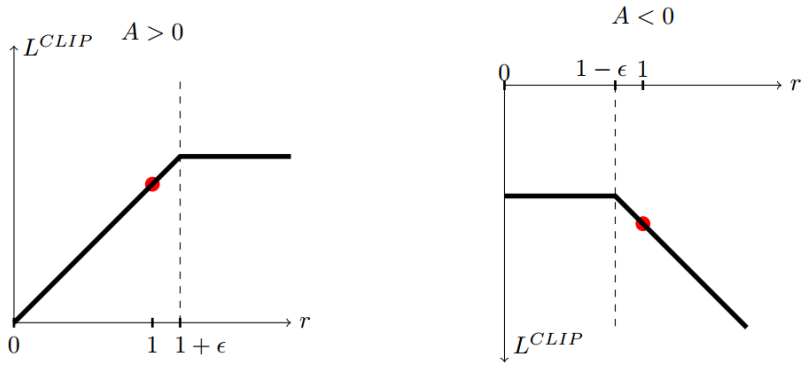


Figure 1: Plots showing one term (i.e., a single timestep) of the surrogate function L^{CLIP} as a function of the probability ratio r, for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimization, i.e., $r = 1$. Note that L^{CLIP} sums many of these terms.

4) 具体算法

Algorithm 3 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau\in\mathcal{D}_k} \sum_{t=0}^{T} \min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t),\ \ g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau\in\mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

5) 实现上的折中及不足

同样到训练的后期会存在 exploration 不足的问题

1.1.5 Synchronous Advantage Actor-Critic (A2C), Asynchronous Advantage Actor-Critic (A3C) 2016

1) 算法特点

an on-policy algorithm.

can be used for environments with either discrete or continuous action spaces.

2) 产生的原因

主要是为了进行并行运算，弥补 PG 算法 sampling 不足的问题。算法如果使用了神经网络，通过并行也可以缓解数据之间的相关性， 防止过拟合。

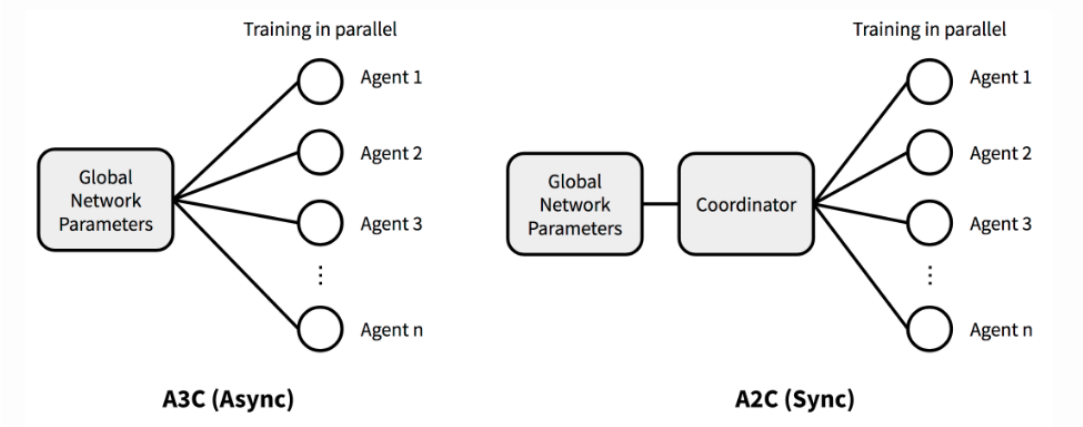
3) 理论介绍

并行部分属于偏重实验的工作

优化部分主要还是基于原有 PG 的理论

4) 具体算法

A3C 采用异步的并行方式，A2C 是同步的并行方式：



Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ *and* θ_v *and global shared counter* $T = 0$
// Assume thread-specific parameter vectors θ' *and* θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial \left(R - V(s_i;\theta'_v)\right)^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

1.1.6 Importance Weighted Actor-Learner Architecture (IMPALA) 2018

1) 算法特点

算是一种 off-policy 的算法，但这边还是放在 on-policy 这个部分来介绍，是因为其基本理论还是 PG 那一套。

算法可用于 continue 及 discrete action 的任务上。

2) 产生的原因

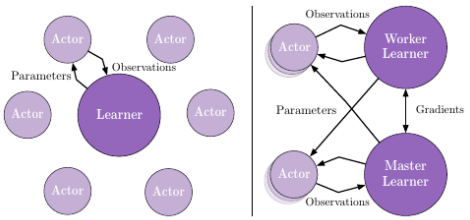


Figure 1. **Left: Single Learner.** Each *actor* generates trajectories and sends them via a queue to the *learner*. Before starting the next trajectory, *actor* retrieves the latest policy parameters from *learner*. **Right: Multiple Synchronous Learners.** Policy parameters are distributed across multiple *learners* that work synchronously.

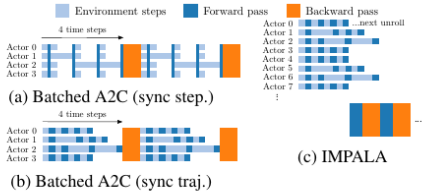


Figure 2. Timeline for one unroll with 4 steps using different architectures. Strategies shown in **(a)** and **(b)** can lead to low GPU utilisation due to rendering time variance within a batch. In **(a)**, the actors are synchronised after every step. In **(b)** after every n steps. IMPALA **(c)** decouples acting from learning.

IMPLALA 是一个大规模强化学习训练的框架，具有较高的性能（high throughput）、较好的扩展性（scalability）和较高的效率（data-efficiency）。

在大规模异步的并行计算框架下，采样（behavior policy 的 V_{μ} ）和策略（target policy V_{π} ）更新会有一些错位（不再是完全的 on-policy），在这种情况下，文章通过 V-trace 技术来使用这种 off-policy 样本进行训练。

A3C 是传递的 gradient，IMPALA 传递的是 trajectories。

3) 理论介绍

V- TRACE 就是为了解决 V 值在这种高并行的架构下估计不准的问题，具体如下：

根本的做法还是基于 Importance Sampling

Consider a trajectory $(x_t, a_t, r_t)_{t=s}^{t=s+n}$ generated by the actor following some policy μ. We define the n-steps V-trace target for $V(x_s)$, our value approximation at state x_s, as:

$$v_s \overset{\text{def}}{=} V(x_s) + \sum_{t=s}^{s+n-1} \gamma^{t-s} \Big(\prod_{i=s}^{t-1} c_i \Big) \delta_t V, \quad (1)$$

where $\delta_t V \overset{\text{def}}{=} \rho_t \big(r_t + \gamma V(x_{t+1}) - V(x_t) \big)$ is a temporal difference for V, and $\rho_t \overset{\text{def}}{=} \min \big(\bar{\rho}, \frac{\pi(a_t|x_t)}{\mu(a_t|x_t)} \big)$ and $c_i \overset{\text{def}}{=} \min \big(\bar{c}, \frac{\pi(a_i|x_i)}{\mu(a_i|x_i)} \big)$ are truncated importance sampling (IS) weights (we make use of the notation $\prod_{i=s}^{t-1} c_i = 1$ for $s = t$). In addition we assume that the truncation levels are such that $\bar{\rho} \geq \bar{c}$.

如果 $\rho_i=1$ $c_i=1$ ，那么上述公式是会回退到 on-policy n-step Bellman 方程。

另外就是系数 $\bar{\rho}$ $\bar{c}$ 的作用的说明：

$\bar{\rho}$ 影响的是 Value 能收敛到的位置，如过 $\bar{\rho}=\infty$ ，那么 value function 将会收敛到 target policy 的 V_π ；而如果 $\bar{\rho}$ 设成接近于 0 的值，那么 value function 将会收敛到 behavior policy 的 V_μ ；如过介于 0 与 ∞ ，那么 value function 会收敛到介于 target policy 与 behavior policy 的 V 上。

$\bar{c}$ 主要影响的是收敛的速度，当然也是不能太大或太小的。（具体证明可以参考原文）

另外就是可以将上述公式写成一个递归的解法：

Remark 1. *V-trace targets can be computed recursively:*

$$v_s = V(x_s) + \delta_s V + \gamma c_s\big(v_{s+1} - V(x_{s+1})\big).$$

4) 具体算法

Consider a parametric representation V_θ of the value function and the current policy π_ω. Trajectories have been generated by actors following some behaviour policy μ. The V-trace targets v_s are defined by (1). At training time s, the value parameters θ are updated by gradient descent on the $l2$ loss to the target v_s, i.e., in the direction of

$$\big(v_s - V_\theta(x_s)\big)\nabla_\theta V_\theta(x_s),$$

and the policy parameters ω in the direction of the policy gradient:

$$\rho_s \nabla_\omega \log \pi_\omega(a_s|x_s)\big(r_s + \gamma v_{s+1} - V_\theta(x_s)\big).$$

In order to prevent premature convergence we may add an entropy bonus, like in A3C, along the direction

$$-\nabla_\omega \sum_a \pi_\omega(a|x_s) \log \pi_\omega(a|x_s).$$

The overall update is obtained by summing these three gradients rescaled by appropriate coefficients, which are hyperparameters of the algorithm.

1.2 Off-policy RL

1.2.1 Q-learning 1992

1) 算法特点

off-policy，说以会有更高效 sampling

主要是用于处理 discrete action 的问题

Policy 是隐式的，action 的选取直接依据于 Q 值的大小

2) 产生的原因

Q-learning 源于之前提到的 optimal value function 的 Bellman Equation。

3) 理论介绍

optimal value function 的 Bellman Equation:

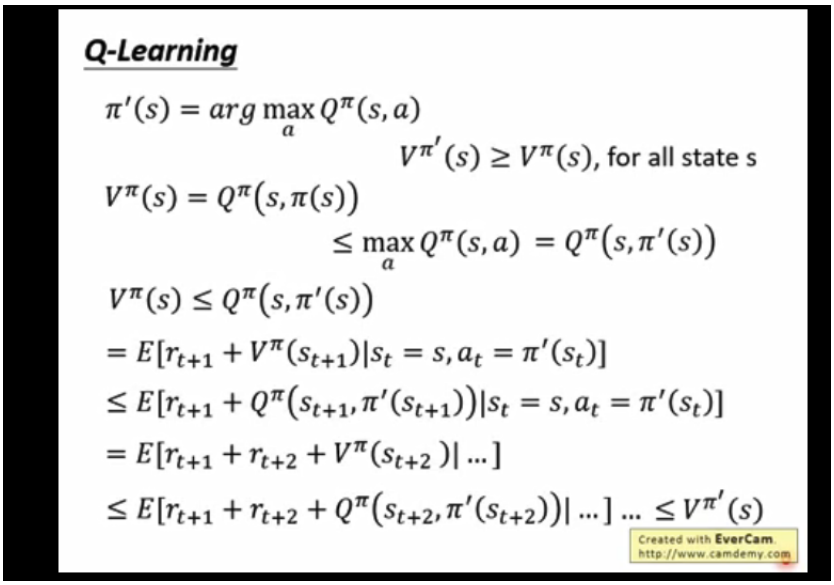
The Bellman equations for the optimal value functions are

$$V^*(s) = \max_a \mathop{\mathrm{E}}_{s' \sim P} \left[r(s,a) + \gamma V^*(s') \right],$$

$$Q^*(s,a) = \mathop{\mathrm{E}}_{s' \sim P} \left[r(s,a) + \gamma \max_{a'} Q^*(s',a') \right].$$

从 Dynamic Programing 到上面的 Bellman Equation 还是有一些推导过程的。

收敛证明：



4) 具体算法

下面给出 Q-learning 跟 Sarsa 的对比：

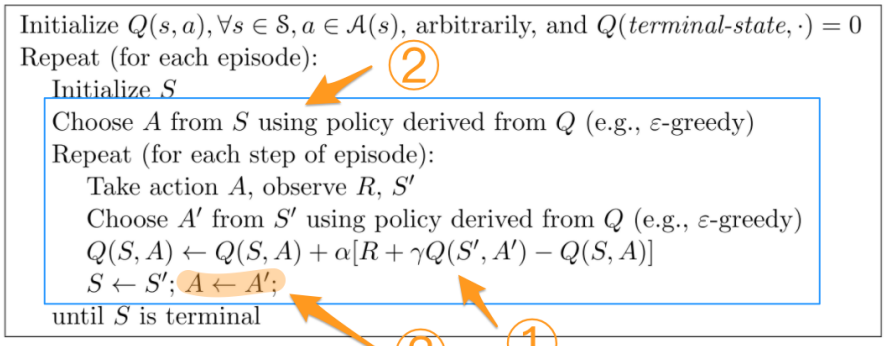


Figure 6.9: Sarsa: An on-policy TD control algorithm.

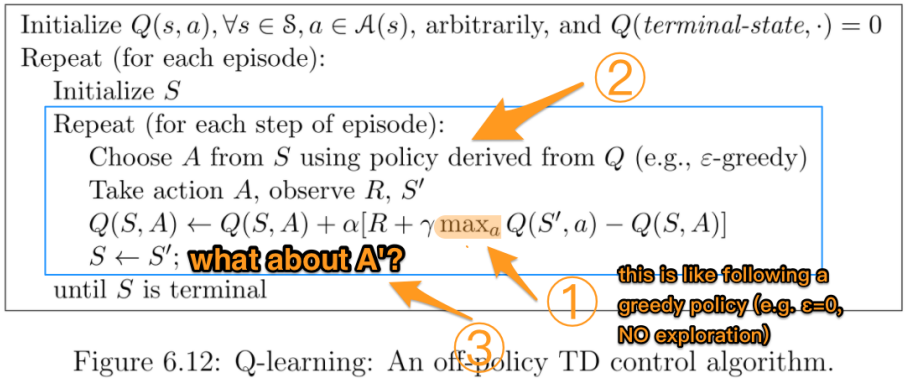


Figure 6.12: Q-learning: An off-policy TD control algorithm.

5) 实现上的折中及不足

处理 contiune action 的任务会比较麻烦

Q 容易被高估

适用于 tabular 的问题，所以 observation space 是不能太大的

1.2.2 Deep Q-network (DQN) 2015

1) 算法特点

off-policy

适用于 discrete action 的问题

2) 产生的原因

为了能直接处理图像输入的 RL 问题

3) 理论介绍

想处理图像作为 observation 的问题，就需要借助 CNN。如果 batch 里面的数据关联性太强，就会造成过拟合，所以就需要采用 replay buffer，并且 DQN 的 replay buffer 通常会是比较大的。

另外就是 target value 需要使用一个 separated Network 来估计，主要是原因如下：

如果不用一个 separated network 来估计 target value，那么 target value 是跟 value network 相关的，但 Q-learning 的形式又是采用 supervised learning 的方式来进行优化的（supervised learning 中的 label 数据需要是独立于网络参数），同时 target value 上并也没有去计算 gradient，这样算法的收敛会很难保证。

4) 具体算法

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For $t = 1$,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

5) 实现上的折中及不足

因为使用了神经网络，loss 是无法达到 0 的，所以相对于 tabular 的方式，policy 也是无法达到最优的，所以虽然 Bellman Equation 跟 L2（supervised learning 的形式）都是可以保证收敛的，但这三者结合起来，却是无法保证收敛的（具体证明可以参考 UCB CS285 Lecture 7: Value Function Methods 及 Lecture 8: Deep RL with Q-functions）

1.2.3 Deep Deterministic policy Gradient (DDPG) 2016

1) 算法特点

off policy

只能用于处理 continue action 的问题

2) 产生的原因

算是 Q-learning 用于处理 continue action 问题的一种形式

3) 理论介绍

Critic 部分就 mean-squared Bellman error (MSBE) function：

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}} \left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d) \max_{a'} Q_\phi(s',a') \right) \right)^2 \right]$$

Policy 部分：

Policy learning in DDPG is fairly simple. We want to learn a deterministic policy $\mu_\theta(s)$ which gives the action that maximizes $Q_\phi(s,a)$. Because the action space is continuous, and we assume the Q-function is differentiable with respect to action, we can just perform gradient ascent (with respect to policy parameters only) to solve

$$\max_\theta \underset{s\sim\mathcal{D}}{\mathrm{E}} \left[Q_\phi(s, \mu_\theta(s)) \right].$$

Note that the Q-function parameters are treated as constants here.

4) 具体算法

Algorithm 4 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1-d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} \left(Q_\phi(s,a) - y(r,s',d) \right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s\in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho \phi_{\text{targ}} + (1-\rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1-\rho)\theta$$

    end for
  end if
until convergence

5) 实现上的折中及不足

具体存在的问题就是跟 Q-learning 一样的。

Frequently brittle with respect to hyperparameters and other kinds of tuning.

1.2.4 Twin Delayed DDPG （TD3）2018

1) 算法特点

off policy

只能用在 continue action 的任务中

2) 产生的原因

主要是为了解决 DDPG 中存在的一些问题，如 Q 值的估计不准，并由此使 policy 很容易学得不好。

3) 理论介绍

主要的解决方法如下：

Trick One: Clipped Double-Q Learning. TD3 learns *two* Q-functions instead of one (hence "twin"), and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.

Trick Two: "Delayed" Policy Updates. TD3 updates the policy (and target networks) less frequently than the Q-function. The paper recommends one policy update for every two Q-function updates.

Trick Three: Target Policy Smoothing. TD3 adds noise to the target action, to make it harder for the policy to exploit Q-function errors by smoothing out Q along changes in action.

Together, these three tricks result in substantially improved performance over baseline DDPG.

First: **target policy smoothing**. Actions used to form the Q-learning target are based on the target policy, $\mu_{\theta_{\text{targ}}}$, but with clipped noise added on each dimension of the action. After adding the clipped noise, the target action is then clipped to lie in the valid action range (all valid actions, a, satisfy $a_{Low} \leq a \leq a_{High}$). The target actions are thus:

$$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \qquad \epsilon \sim \mathcal{N}(0, \sigma)$$

Target policy smoothing essentially serves as a regularizer for the algorithm. It addresses a particular failure mode that can happen in DDPG: if the Q-function approximator develops an incorrect sharp peak for some actions, the policy will quickly exploit that peak and then have brittle or incorrect behavior. This can be averted by smoothing out the Q-function over similar actions, which target policy smoothing is designed to do.

Next: **clipped double-Q learning**. Both Q-functions use a single target, calculated using whichever of the two Q-functions gives a smaller target value:

$$y(r, s', d) = r + \gamma(1-d) \min_{i=1,2} Q_{\phi_{i,\text{targ}}}(s', a'(s')),$$

and then both are learned by regressing to this target:

$$L(\phi_1, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\text{E}} \left[\left(Q_{\phi_1}(s,a) - y(r,s',d) \right)^2 \right],$$

$$L(\phi_2, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\text{E}} \left[\left(Q_{\phi_2}(s,a) - y(r,s',d) \right)^2 \right].$$

Using the smaller Q-value for the target, and regressing towards that, helps fend off overestimation in the Q-function.

Lastly: the policy is learned just by maximizing Q_{ϕ_1}:

$$\max_{\theta} \underset{s\sim\mathcal{D}}{\text{E}} \left[Q_{\phi_1}(s, \mu_\theta(s)) \right],$$

4) 具体算法

Algorithm 5 Twin Delayed DDPG

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute target actions

$$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \qquad \epsilon \sim \mathcal{N}(0, \sigma)$$

      Compute targets

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', a'(s'))$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      if $j \mod \texttt{policy\_delay} = 0$ **then**
        Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_{\phi_1}(s, \mu_\theta(s))$$

        Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$
$$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta$$

      end if
    end for
  end if
until convergence

5) 实现上的折中及不足

具体存在的问题就是跟 Q-learning 一样的。

另外由于是 deterministic policy，算法的开始的时候通常是探索不足的，最好在 action 上加入一定的 noise，但后期为了可以获得更好的 trajectory，这个 noise 可以逐渐去减小。

1.2.5 Soft Actor-Critic (ASC) 2018

1) 算法特点

off policy

可以用于 discrete 跟 continue action 的任务中

2) 产生的原因

采用 stochastic policies，所以 policy 可以更灵活一点，可以用来处理 discrete 跟 continue action 的问题

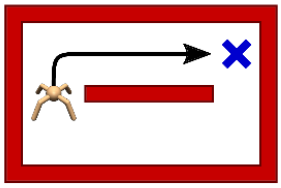
最主要还是 entropy regularization，这样可以更好平衡 exploration 跟 exploitation

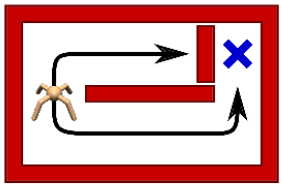
具有 on policy 算法的稳定性及 off policy 算法的高效性

3) 理论介绍

Energy-based policy

使用 energy-based policy 的原因：

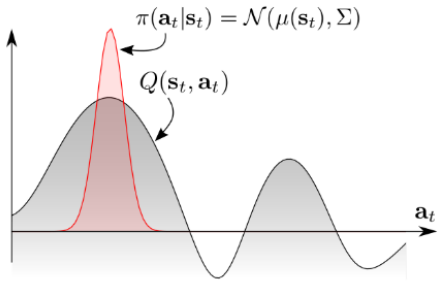




RL 常规的目标函数定义如下：

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\pi}\left[\sum_{t=0}^{T} r_t\right]$$

按照这种方式获得的 policy （如 Q-learning），是一种 unimodal policy，这种 policy 在解决左图的 task 是没什么问题的，但比较难处理右图的 task

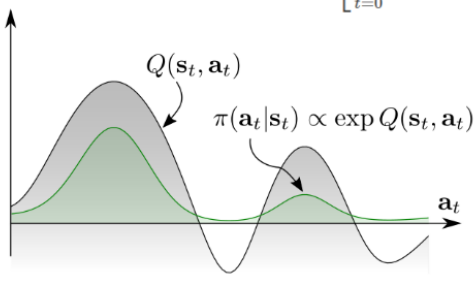


为了解决右边这种 task，需要一种 multi-modal 的 policy：

$$\pi(\mathbf{a}|\mathbf{s}) \propto \exp Q(\mathbf{s}, \mathbf{a})$$

在这种 multimodel policy 的定义下是可以推出一种 entropy-based 的 policy 的：

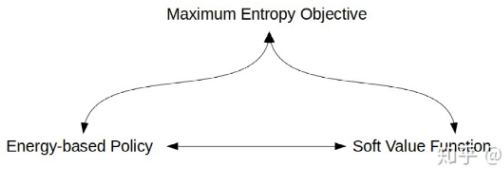
$$\pi^*_{\text{MaxEnt}} = \arg\max_{\pi} \mathbb{E}_{\pi}\left[\sum_{t=0}^{T} r_t + \mathcal{H}(\pi(\cdot|\mathbf{s}_t))\right]$$



从 Maximum Entropy Object，我们可以推出 Q、V 的形式及更 policy 的关系：

Maximum Entropy Objective ← → Energy-based Policy

$$\pi^*_{\text{MaxEnt}} = \arg\max_{\pi} \mathbb{E}_{\pi}\left[\sum_{t=0}^{T} r_t + \mathcal{H}(\pi(\cdot|s_t))\right] \quad \leftarrow\rightarrow \quad \pi(\mathbf{a}|\mathbf{s}) \propto \exp Q(\mathbf{s},\mathbf{a})$$



$$r_{soft}(s_t,a_t) = r(s_t,a_t) + \gamma\alpha\mathbb{E}_{s_{t+1}\sim\rho}H(\pi(\cdot|s_{t+1})) \quad (7)$$

$$Q(s_t,a_t) = r(s_t,a_t) + \gamma\mathbb{E}_{s_{t+1},a_{t+1}}[Q(s_{t+1},a_{t+1})] \quad (5)$$

我们将（7）带入到公式（5）：

$$\begin{aligned}
Q_{soft}(s_t,a_t) &= r(s_t,a_t) + \gamma\alpha\mathbb{E}_{s_{t+1}\sim\rho}H(\pi(\cdot|s_{t+1})) + \gamma\mathbb{E}_{s_{t+1},a_{t+1}}[Q_{soft}(s_{t+1},a_{t+1})] \\
&= r(s_t,a_t) + \gamma\mathbb{E}_{s_{t+1}\sim\rho,a_{t+1}\sim\pi}[Q_{soft}(s_{t+1},a_{t+1})] + \gamma\alpha\mathbb{E}_{s_{t+1}\sim\rho}H(\pi(\cdot|s_{t+1})) \\
&= r(s_t,a_t) + \gamma\mathbb{E}_{s_{t+1}\sim\rho,a_{t+1}\sim\pi}[Q_{soft}(s_{t+1},a_{t+1})] + \gamma\mathbb{E}_{s_{t+1}\sim\rho}\mathbb{E}_{a_{t+1}\sim\pi}[-\alpha\log\pi(a_{t+1}|s_{t+1})] \\
&= r(s_t,a_t) + \gamma\mathbb{E}_{s_{t+1}\sim\rho}[\mathbb{E}_{a_{t+1}\sim\pi}[Q_{soft}(s_{t+1},a_{t+1}) - \alpha\log(\pi(a_{t+1}|s_{t+1}))]] \\
&= r(s_t,a_t) + \gamma\mathbb{E}_{s_{t+1},a_{t+1}}[Q_{soft}(s_{t+1},a_{t+1}) - \alpha\log(\pi(a_{t+1}|s_{t+1}))]
\end{aligned}$$

与此同时，我们知道：

$$Q(s_t,a_t) = r(s_t,a_t) + \gamma\mathbb{E}_{s_{t+1}\sim\rho}[V(s_{t+1})] \quad (9)$$

因此，我们有：

$$V_{soft}(s_t) = \mathbb{E}_{a_t\sim\pi}[Q_{soft}(s_t,a_t) - \alpha\log\pi(a_t|s_t)] \quad (10)$$

前面我们已经推导得到了公式（10），那么根据公式（10），我们可以直接推导得到policy的形式：

$$\begin{aligned}
\pi(s_t,a_t) &= \exp(\frac{1}{\alpha}(Q_{soft}(s_t,a_t) - V_{soft}(s_t))) \\
&= \frac{\exp(\frac{1}{\alpha}Q_{soft}(s_t,a_t))}{\exp(\frac{1}{\alpha}V_{soft}(s_t))}
\end{aligned} \quad (14)$$

接下来就可以设计具体的算法了

Q-function 的训练，具体做的时候，SAC 也跟 TD3 一样，用了两个 Q network 来估计 Q 值：

Before we give the final form of the Q-loss, let's take a moment to discuss how the contribution from entropy regularization comes in. We'll start by taking our recursive Bellman equation for the entropy-regularized Q^π from earlier, and rewriting it a little bit by using the definition of entropy:

$$\begin{aligned}
Q^\pi(s,a) &= \mathop{\mathrm{E}}_{\substack{s'\sim P\\ a'\sim\pi}}[R(s,a,s') + \gamma\left(Q^\pi(s',a') + \alpha H\left(\pi(\cdot|s')\right)\right)] \\
&= \mathop{\mathrm{E}}_{\substack{s'\sim P\\ a'\sim\pi}}[R(s,a,s') + \gamma\left(Q^\pi(s',a') - \alpha\log\pi(a'|s')\right)]
\end{aligned}$$

The RHS is an expectation over next states (which come from the replay buffer) and next actions (which come from the current policy, and **not** the replay buffer). Since it's an expectation, we can approximate it with samples:

$$Q^\pi(s,a) \approx r + \gamma\left(Q^\pi(s',\tilde{a}') - \alpha\log\pi(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi(\cdot|s').$$

policy 的训练：

Learning the Policy. The policy should, in each state, act to maximize the expected future return plus expected future entropy. That is, it should maximize $V^\pi(s)$, which we expand out into

$$\begin{aligned}
V^\pi(s) &= \mathop{\mathrm{E}}_{a\sim\pi}[Q^\pi(s,a)] + \alpha H\left(\pi(\cdot|s)\right) \\
&= \mathop{\mathrm{E}}_{_}[Q^\pi(s,a) - \alpha\log\pi(a|s)].
\end{aligned}$$

reparameterization trick，SAC 中 action 的产生方式跟 PG 算法有点不一样，一是对 variance σ 添加了另外的 noise，另外使用 tanH 对 action 进行了归一化（为什么要这么做，文章没有给出解释。我猜测是在 σ 上添加 noise，是因为当网络输出的 σ 还不是很好的时候，计算出来的 action 还是可以用的；tanh 可以将 action 限制在一定范围，可以让训练比较稳定）：

The way we optimize the policy makes use of the **reparameterization trick**, in which a sample from $\pi_\theta(\cdot|s)$ is drawn by computing a deterministic function of state, policy parameters, and independent noise. To illustrate: following the authors of the SAC paper, we use a squashed Gaussian policy, which means that samples are obtained according to

$$\tilde{a}_\theta(s, \xi) = \tanh\left(\mu_\theta(s) + \sigma_\theta(s) \odot \xi\right), \quad \xi \sim \mathcal{N}(0, I).$$

所以 Q-function 跟 policy 最终的优化目标如下：

The reparameterization trick allows us to rewrite the expectation over actions (which contains a pain point: the distribution depends on the policy parameters) into an expectation over noise (which removes the pain point: the distribution now has no dependence on parameters):

$$\mathop{\mathrm{E}}_{a \sim \pi_\theta}\left[Q^{\pi_\theta}(s,a) - \alpha \log \pi_\theta(a|s)\right] = \mathop{\mathrm{E}}_{\xi \sim \mathcal{N}}\left[Q^{\pi_\theta}(s, \tilde{a}_\theta(s,\xi)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s,\xi)|s)\right]$$

To get the policy loss, the final step is that we need to substitute Q^{π_θ} with one of our function approximators. Unlike in TD3, which uses Q_{ϕ_1} (just the first Q approximator), SAC uses $\min_{j=1,2} Q_{\phi_j}$ (the minimum of the two Q approximators). The policy is thus optimized according to

$$\max_\theta \mathop{\mathrm{E}}_{\substack{s \sim \mathcal{D} \\ \xi \sim \mathcal{N}}}\left[\min_{j=1,2} Q_{\phi_j}(s, \tilde{a}_\theta(s,\xi)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s,\xi)|s)\right],$$

which is almost the same as the DDPG and TD3 policy optimization, except for the min-double-Q trick, the stochasticity, and the entropy term.

4) 具体算法

Algorithm 6 Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1-d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \qquad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s,a) - y(r,s',d)\right)^2 \qquad \text{for } i = 1, 2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \Big(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right) \Big),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
      Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1-\rho)\phi_i \qquad \text{for } i = 1, 2$$

    end for
  end if
until convergence

5) 实现上的折中及不足

算是目前比较好的 model free RL 算法

6) 参考

Haarnoja, Tuomas, et al. "Reinforcement learning with deep energy-based policies." arXiv preprint arXiv:1702.08165 (2017).

Haarnoja, Tuomas, et al. "Reinforcement learning with deep energy-based policies." arXiv preprint arXiv:1702.08165 (2017).

https://bair.berkeley.edu/blog/2017/10/06/soft-q-learning/

https://zhuanlan.zhihu.com/p/70360272

https://blog.csdn.net/geter_CS/article/details/85068407

2 References

https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html

https://sites.google.com/view/deep-rl-bootcamp/lectures

http://rail.eecs.berkeley.edu/deeprlcourse/

https://www.davidsilver.uk/teaching/