1. Policy Optimization 策略优化

1.1 理论推导

将 policy 参数化为π_θ，系统经历的轨迹 trajectory 为$\tau=(s_0,a_0,...,s_{T+1})$，在策略$\pi_\theta$下系统可获得 Expected return 为 $J(\pi_\theta)=\underset{\tau\sim\pi_\theta}{\mathrm{E}}[R(\tau)]$。根据随机梯度下降法，可以得到参数$\theta$的更新如下：

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

策略梯度为$\nabla_\theta J(\pi_\theta)$。（通过此种方式优化策略的方法为策略梯度算法，policy gradient algorithms）。为了利用此算法，需要将策略梯度表示为可数值计算的形式，主要包括以下两步：1）策略梯度由期望的梯度变成一个值的期望；2）对该值进行采样取得其估计值。

策略梯度可以表征为以下期望形式：

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

进而，策略梯度可以估计（采样）为：

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

为了提高采样效率，或样本利用效率。可以通过以下两个方向进一步优化梯度更新。

一方面，$R(\tau)$中在 t 时刻（状态 s_t）之前的 rewards 与 a_t 的选择并无关系，这些 rewards 对于 a_t 选择的优化并没有作用，即这部分 rewards 对于梯度的期望没有影响，但是增加了方差。实际对于梯度估值有作用的样本是 a_t 选择之后的 rewards（备注：这些 rewards 累计后的期望就相当于 Q 值）。严格数学证明参见（https://spinningup.openai.com/en/latest/spinningup/extra_pg_proof2.html），即策略梯度的期望为：

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

策略梯度的估值为：

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

另一方面可以通过添加 baseline（期望为 0 的函数）来进一步减少样本的方差。即：

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

证明如下。首先，EGLP lemma 如下：

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:

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

然后，可以通过 EGLP lemma 设计相应的 baseline 函数，如下，b(s_t)。需要注意的是，策略梯度更新公式中的 b(s_t)应该与通过该公式进行梯度更新的θ无关。最常用的 b(s_t)是$V^\pi(s_t)$（AC 框架下），用另一网络参数进行表征，即$V_\phi(s_t)$，其参数更新的公式当然也不是策略梯度公式。

An immediate consequence of the EGLP lemma is that for any function b which only depends on state,

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

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**.

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}.

1.2 策略梯度方法小结

总体来说，策略梯度方法中，策略梯度的更新公式如下：

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

其中，Φ_t 有如下形式：

- $\Phi_t = R(\tau),$
- $\Phi_t = \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}),$
- $\Phi_t = \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t).$
- **1. On-Policy Action-Value Function.** The choice

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

  is also valid. See this page for an (optional) proof of this claim.
- **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,

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

  is also valid. The proof is that it's equivalent to using $\Phi_t = Q^{\pi_\theta}(s_t, a_t)$ and then using a value function baseline, which we are always free to do.

2. Vanilla Policy Gradient (REINFORCE)

2.1 VPG 算法流程

下面的算法是一个 on-policy 算法，也就是说更新所使用的样本都是基于更新前的策略参数产生的。

Algorithm 1 Vanilla Policy Gradient Algorithm

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
for $k = 0, 1, 2, \ldots$ **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

2.2 优势函数估计方法

http://www.360doc.com/content/20/0607/22/32196507_917085069.shtml

常见的优势函数：VPG、TRPO 等的优势函数也可以选择以下 A3C、PPO 的估计方式；

Advantage Function :

for A3C :

$$A(s_t, a_t; \theta, \theta_v) = \left(\sum_{i=0}^{k-1} \gamma^i r_{t+i} \right) + \gamma^k V(s_{t+k}; \theta_v) - V(s_t; \theta_v)$$

for PPO, a a truncated version of GAE :

$$\widehat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1}$$

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

T : *trajectory length*

头条 @布谷AI

这些常见的近似方式是优势函数定义的有偏估计(but not too biased)，但可以接受。

优势函数的一般估计(GAE)借鉴了 TD(lambda)思想，注意这里处理的是优势函数而不是 Value Function，通过调整 lambda，可以得到不同的近似估计。主要思路包含两方面，一是 Q(s,a)的近似，二是 V(s)的表达，二者差表征优势函数 A(s, a)。

$$\hat{A}_t^{(1)} := \delta_t^V \qquad = -V(s_t) + r_t + \gamma V(s_{t+1}) \tag{11}$$

$$\hat{A}_t^{(2)} := \delta_t^V + \gamma \delta_{t+1}^V \qquad = -V(s_t) + r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) \tag{12}$$

$$\hat{A}_t^{(3)} := \delta_t^V + \gamma \delta_{t+1}^V + \gamma^2 \delta_{t+2}^V = -V(s_t) + r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 V(s_{t+3}) \tag{13}$$

当k无限时，这一项近似为0

$$\hat{A}_t^{(k)} := \sum_{l=0}^{k-1} \gamma^l \delta_{t+l}^V = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{k-1} r_{t+k-1} + \gamma^k V(s_{t+k}) \tag{14}$$

$$\hat{A}_t^{(\infty)} = \sum_{l=0}^{\infty} \gamma^l \delta_{t+l}^V = -V(s_t) + \boxed{\sum_{l=0}^{\infty} \gamma^l r_{t+l},} \tag{15}$$

the empirical returns

The generalized advantage estimator $\mathrm{GAE}(\gamma, \lambda)$ is defined as the exponentially-weighted average of these k-step estimators:

$$\begin{aligned}
\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} &:= (1-\lambda)\left(\hat{A}_t^{(1)} + \lambda \hat{A}_t^{(2)} + \lambda^2 \hat{A}_t^{(3)} + \ldots\right) \\
&= (1-\lambda)\left(\delta_t^V + \lambda(\delta_t^V + \gamma\delta_{t+1}^V) + \lambda^2(\delta_t^V + \gamma\delta_{t+1}^V + \gamma^2\delta_{t+2}^V) + \ldots\right) \\
&= (1-\lambda)(\delta_t^V(1 + \lambda + \lambda^2 + \ldots) + \gamma\delta_{t+1}^V(\lambda + \lambda^2 + \lambda^3 + \ldots) \\
&\qquad + \gamma^2\delta_{t+2}^V(\lambda^2 + \lambda^3 + \lambda^4 + \ldots) + \ldots) \\
&= (1-\lambda)\left(\delta_t^V\left(\frac{1}{1-\lambda}\right) + \gamma\delta_{t+1}^V\left(\frac{\lambda}{1-\lambda}\right) + \gamma^2\delta_{t+2}^V\left(\frac{\lambda^2}{1-\lambda}\right) + \ldots\right) \\
&= \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V
\end{aligned} \tag{16}$$

头条 @布谷AI

from Berkeley GAE paper ICLR 2016

两个特例是：

TD Residual近似优势函数

$$\boxed{\mathrm{GAE}(\gamma, 0): \quad \hat{A}_t := \delta_t \qquad = r_t + \gamma V(s_{t+1}) - V(s_t)} \tag{17}$$

$$\mathrm{GAE}(\gamma, 1): \quad \hat{A}_t := \sum_{l=0}^{\infty} \gamma^l \delta_{t+l} = \boxed{\sum_{l=0}^{\infty} \gamma^l r_{t+l}} - V(s_t) \tag{18}$$

the empirical returns

头条 @布谷AI

要特别注意，TD Residual 是优势函数的一个不错的估计，但它们是两个概念，这很容易混淆，时序差分主要表达的是一种微分思想，优势函数描述的是一个相对量。

3. A2C 算法

3.1 单步更新（基于 TD error）与回合更新（基于轨迹）

详细推导见，https://www.zhihu.com/column/c_1266110382445654016

VPG 根据回合中轨迹的所有（s,a,r）进行一次性更新，根据轨迹的 rewards 确定更新(s,a)的权重，只能靠着大量的轨迹数据使其收敛到“早晚*优势的权重”平均。

AC 算法，可以让每一步有独属于自己的、能够衡量具体这一步的“好坏”以及“需要被学习的紧迫度”的权重。若 AC 算法单步更新(s,a)时所使用的优势函数考虑了该步(s,a)在轨迹中所处的时间位置，则需要记录每步在轨迹中的位置。若 AC 算法单步更新(s,a)时所使用的优势函数并未考虑该步(s,a)在轨迹中所处的时间位置，则仅在在衰减因子γ不等于 1 时等同于策略梯度算法。

单步更新相对于回合更新的优势在于用于训练的样本可以不是一个（具有时间先后关联关系）轨迹，而是一个个独立的 slot 数据，如此，不需要产生大量轨迹样本，也可使用 replay buffer。

- VPG 采用回合更新，其 actor 网络的更新是基于最大化轨迹累积 return（也可由 critic 的 V 函数对累计 return 进行进一步修正得到 advantage 函数）进行更新的，即朝着最大化累计 return 的期望对 actor 网络参数按照策略梯度方法进行求导并更新，即在 tensorflow 中将损失函数设计为 advantage 函数与交叉熵损失函数的乘积。注意累计 return 的期望即是 Q 函数，回合更新相当于对“累计 return”进行了不同轨迹下的采样，然后直接朝着最大化“累计 return”期望的方向上根据策略梯度方法更新 actor 网络参数。Critic 网络参数根据“累计 return”与 critic V 值的方差最小化来进行跟新。
- DDPG 算法采用单步更新，其 actor 网络的学习目标是针对每步的(s,a) 最大化 critic 网络中 Q(s,a)，即对 Q(s,a)求 actor 网络参数的导数。Critic 网络参数根据 Q(s,a)与目标值（即单步 reward+Target Q(s’,a’)）的方差最小化进行更新。
- AC 算法可以使用单步更新，但为了提升训练效率，本文实现的 A2C 算法使用 mini-batch 更新，其 actor/critic 网络更新类似于回合更新。

3.2 A2C 代码

The code is based on Python 3.8, Tensorflow 2.0, keras. It can be found in my github webpage.

本代码中 A2C 使用 mini-batch 更新。

4. DDPG

4.1 确定策略梯度（DPG） 和随机策略梯度（SPG）

SPG:

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

注意，目标函数为$E_{s\sim\rho^\pi}E_{a\sim\pi_\theta}\pi_\theta(a|s)Q^\pi(s,a)$，其中策略$\pi$为状态$s$下采取动作$a$（离散值）概率，对此目标函数的求导可类似 1.1 小节中的数学方法。

DPG:

$$\nabla_\theta J(\mu_\theta) = E_{s\sim\rho^\mu}\left[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)}\right]$$

注意，目标函数为$E_{s\sim\rho^\mu}Q^\mu(s,a=\mu_\theta(s))$，其中策略$\mu$为状态$s$下采取动作值为$a$（连续

值）。对此函数的求导是根据链式法则求复合函数的导数。

DDPG：在 DPG 上做 NN 参数化时使用经验池和目标网络来提升训练效率。

4.2 经验池和目标网络

类似 DQN，可以利用验池和目标网络来提升网络训练效率。经验池用于存放历史数据，随机从经验池中抽样有助于降低样本的时间关联性，增加样本独立分布的特性。相比与从当前网络推导出当前网络应该逼近的目标值，设计一个更新速度较慢的目标网络来产生当前网络的逼近目标值有助于网络训练的稳定性。当前网络与目标网络结构和表征含义一样，只不过目标网络迭代更新速度小于当前网络。

4.2 DDPG 算法流程

https://spinningup.openai.com/en/latest/algorithms/ddpg.html

DDPG 算法采用单步更新，其 actor 网络的学习目标是针对每步的(s,a) 最大化 critic 网络中 Q(s,a)，即对 Q(s,a)求 actor 网络参数的导数（其中的 a 是根据当前 actor 网络输入 s 后得到的），会根据链式法则求复合函数的导数。Critic 网络参数根据 Q(s,a)与目标值（即单步 reward+Target Q(s',a')）的方差最小化进行更新。

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

知乎 @飞翔

5. TD3 (Twin Delayed DDPG)

5.1 相对于 DDPG 的修改

First: **target policy smoothing**. 加上噪声 ε（clipped），避免 policy 网络在训练时过于利用出了偏差的 Q，如下式。

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

Next: **clipped double-Q learning**. 当前 Q 和目标 Q 中的任意一个，都使用两个（Twins）Q 网络。计算 Target 时，在 Twins 中选择 Q 值更小的那个。Twins Q 网络均使用这个 Target 进行训练。避免对 Q 的估计过高。

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

Lastly: Policy 网络的更新频度小于 Q 网络。相比于 DDPG 中 policy 频繁变化会影响 Target，这个措施有助于提升训练稳定性，

5.2 TD3 算法流程

Algorithm 1 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

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

            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 $j \mod$ `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

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

            end if
        end for
    end if
until convergence