首行

3-ALGORITHMS DOCS

单位：NWPU

时间 2018 年 12 月 7 日星期五

目录

1 Vanilla Policy Gradient

1.1 Background

基本策略梯度的关键思想是提高导致更高回报的行为的概率，并降低导致较低回报的行为的概率，直到您达到最优策略为止。

1.2 Quick Facts

VPG 是一种基于策略的算法.
VPG 可以用于具有离散或连续操作空间的环境。
VPG 的旋转实现支持 MPI 的并行化。

1.3 Key Equations

Let π_θ denote a policy with parameters θ, and $J(\pi_\theta)$ denote the expected finite-horizon undiscounted return of the policy. The gradient of $J(\pi_\theta)$ is

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

where τ is a trajectory and A^{π_θ} is the advantage function for the current policy.

The policy gradient algorithm works by updating policy parameters via stochastic gradient ascent on policy performance:

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

图 1-1

策略梯度实现通常基于无限水平折扣回报计算优势函数估计，尽管在其他情况下使用的是有

限水平未折扣策略梯度公式。

1.4 Exploration vs. Exploitation

VPG 以一种 on policy 的方式训练随机策略.这意味着它根据其随机政策的最新版本，通过抽样行动进行探索。行动选择的随机性程度取决于初始条件和训练过程。在培训过程中，策略通常变得越来越少随机性，因为更新规则鼓励它利用它已经发现的奖励。这可能导致政策陷入局部最优状态。

1.5 Pseudocode(伪代码)

Algorithm 1 Vanilla Policy Gradient Algorithm

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: Estimate policy gradient as

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

7: 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.
8: 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.
9: **end for**

φ是神经网络，具体参见 2 章 3.6。

1.6 Documentation

spinup.vpg(env_fn, actor_critic=<function mlp_actor_critic>, ac_kwargs={}, seed=0, steps_per_epoch=4000, epochs=50, gamma=0.99, pi_lr=0.0003, vf_lr=0.001, train_v_iters=80, lam=0.97, max_ep_len=1000, logger_kwargs={}, save_freq=10)

参数:
env_fn – 创建环境副本的函数。环境必须满足 OpenAI 健身房 API。

actor_critic –
一个函数，它接受状态、x_ph 和 action a_ph 的占位符符号，并从代理的 TensorFlow 计算图中返回主输出:

Symbol	Shape	Description
`pi`	(batch, act_dim)	Samples actions from policy given states.
`logp`	(batch,)	Gives log probability, according to the policy, of taking actions `a_ph` in states `x_ph`.
`logp_pi`	(batch,)	Gives log probability, according to the policy, of the action sampled by `pi`.
`v`	(batch,)	Gives the value estimate for states in `x_ph`. (Critical: make sure to flatten this!)

ac_kwargs (dict) – Any kwargs appropriate for the actor_critic function you provided to VPG.
seed (int) – Seed for random number generators.
steps_per_epoch (int) – Number of steps of interaction (state-action pairs) for the agent and the environment in each epoch.
epochs (int) – Number of epochs of interaction (equivalent to number of policy updates) to perform.
gamma (float) – Discount factor. (Always between 0 and 1.)
pi_lr (float) – Learning rate for policy optimizer.
vf_lr (float) – Learning rate for value function optimizer.
train_v_iters (int) – Number of gradient descent steps to take on value function per epoch.
lam (float) – Lambda for GAE-Lambda. (Always between 0 and 1, close to 1.)
max_ep_len (int) – Maximum length of trajectory / episode / rollout.
logger_kwargs (dict) – Keyword args for EpochLogger.
save_freq (int) – How often (in terms of gap between epochs) to save the current policy and value function.

1.7 Saved Model Contents

The computation graph saved by the logger includes:

Key	Value
`x`	Tensorflow placeholder for state input.
`pi`	Samples an action from the agent, conditioned on states in `x`.
`v`	Gives value estimate for states in `x`.

This saved model can be accessed either by

- running the trained policy with the test_policy.py tool,
- or loading the whole saved graph into a program with restore_tf_graph.

1.8 References

Relevant Papers

- Policy Gradient Methods for Reinforcement Learning with Function Approximation, Sutton et al. 2000
- Optimizing Expectations: From Deep Reinforcement Learning to Stochastic Computation Graphs, Schulman 2016(a)
- Benchmarking Deep Reinforcement Learning for Continuous Control, Duan et al. 2016
- High Dimensional Continuous Control Using Generalized Advantage Estimation, Schulman et al. 2016(b)

Why These Papers?

Sutton 2000 之所以包括在内，是因为它是强化学习理论的永恒经典，它包含了对导致现代政策梯度的早期工作的参考。Schulman 2016(A)之所以包括在内，是因为第二章清楚地介绍了包括伪码在内的策略梯度算法的理论。段 2016 是一篇清晰的、最近的基准论文，它展示了深度 rl 设置中的香草策略梯度(例如，使用神经网络策略和 adam 作为优化器)与其他深度 rl 算法的比较。Schulman 2016(B)之所以包括在内，是因为我们的 VPG 实现使用了广义优势估计来计算策略梯度。

2 Trust Region Policy Optimization

2.1 Background

trpo 通过采取尽可能大的步骤来提高性能来更新策略，同时满足新和旧策略允许有多近的特殊限制。约束用 KL-散度来表示，KL-散度是概率分布之间距离的一种度量。
这不同于常规的策略梯度，即在参数空间中保持新旧策略的紧密性。但是，即使参数空间上看似很小的差异，在性能上也会有很大的差异-因此，一个糟糕的步骤可能会使策略绩效崩溃。这使得使用大步长与普通策略梯度一起使用是危险的，从而损害了它的样本效率。trpo 很好地避免了这种崩溃，并且趋向于快速和单调地提高表现。

2.2 Quick Facts

trpo 是一种在策略上的算法.
TRPO 可用于具有离散或连续操作空间的环境。
trpo 的旋转实现支持 MPI 的并行化。

2.3 Key Equations

Let π_θ denote a policy with parameters θ. The theoretical TRPO update is:

$$\theta_{k+1} = \arg\max_\theta \mathcal{L}(\theta_k, \theta)$$
$$\text{s.t. } \bar{D}_{KL}(\theta||\theta_k) \leq \delta$$

where $\mathcal{L}(\theta_k, \theta)$ is the *surrogate advantage*, a measure of how policy π_θ performs relative to the old policy π_{θ_k} using data from the old policy:

$$\mathcal{L}(\theta_k, \theta) = \underset{s,a\sim\pi_{\theta_k}}{\mathrm{E}}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a)\right],$$

and $D_{KL}(\theta||\theta_k)$ is an average KL-divergence between policies across states visited by the old policy:

$$D_{KL}(\theta||\theta_k) = \underset{s\sim\pi_{\theta_k}}{\mathrm{E}}\left[D_{KL}\left(\pi_\theta(\cdot|s)||\pi_{\theta_k}(\cdot|s)\right)\right].$$

当θ =θ _k 时，你应该知道目标和约束都是零的。此外，当θ =θ _k 时，对θ 的约束梯度为零。证明这些事实需要一些微妙的掌握相关的数学-这是一个练习值得做，只要你觉得准备！

理论上的 trpo 更新并不是最容易使用的，所以 trpo 做了一些近似来快速得到答案。我们把目标和约束扩展到θ _k 周围的主导秩序。

这个近似问题可以用拉格朗日对偶的方法解析求解，得到的解如下：

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g.$$

如果我们在这里停下来，使用这个最后的结果，算法就会精确地计算出自然政策的梯度。一个问题是，由于泰勒展开引入的逼近误差，这可能不满足 KL 约束，或者实际上改善了代理优势。trpo 添加了对此更新规则的修改：回溯行搜索，

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

其中α in(0，1)是回溯系数，j 是最小的非负整数，使得 pi_{theta_{k1}满足 KL 约束并产生正的代理优势。
最后：计算和存储矩阵逆 h^{-1}，在处理数千或数百万参数的神经网络策略时，代价高昂。Trpo 采用共轭梯度算法求解 x=h^{-1}的 hx=g，只需要一个能计算矩阵向量积 hx 的函数，而不需要直接计算和存储整个矩阵 h。这并不难：我们设置了一个符号运算来计算。

$$Hx = \nabla_\theta \left(\left(\nabla_\theta \bar{D}_{KL}(\theta || \theta_k) \right)^T x \right),$$

这给了我们正确的输出，而不需要计算整个矩阵。

2.4 Exploration vs. Exploitation

trpo 以一种政策上的方式训练随机政策.这意味着它根据其随机政策的最新版本，通过抽样行动进行探索。行动选择的随机性程度取决于初始条件和训练过程。在培训过程中，策略通常变得越来越少随机性，因为更新规则鼓励它利用它已经发现的奖励。这可能导致政策陷入局部最优状态。

2.5 Pseudocode

Algorithm 1 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

2.6 Documentation

2.7 Saved Model Contents

2.8 References

2.9 Relevant Papers

- Trust Region Policy Optimization, Schulman et al. 2015
- High Dimensional Continuous Control Using Generalized Advantage Estimation, Schulman et al. 2016
- Approximately Optimal Approximate Reinforcement Learning, Kakade and Langford 2002

2.10 Why These Papers?

Schulman 2015 之所以包括在内，是因为它是描述 TRPO 的原始论文。Schulman 2016 之所以包括在内，是因为我们的 Trpo 实现使用了广义优势估计来计算策略梯度。Kakade 和 Langford 2002 之所以包括在内，是因为它包含了激励和深入联系 TRPO 理论基础的理论成果。

2.11 Other Public Implementations

- Baselines
- ModularRL
- rllab

3 Proximal Policy Optimization（近端策略优化）

3.1 Background

PPO 的动机与 trpo 相同：我们如何才能在使用现有数据的政策上采取最大可能的改进步骤，而不采取意外导致性能崩溃的步骤呢？在 trpo 试图用复杂的二阶方法解决这个问题的地方，ppo 是一个一阶方法的家族，它使用一些其他的技巧来使新的策略接近旧的。PPO 方法实现起来要简单得多，而且在经验上似乎至少与 trpo 一样好。

> 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.
>
> Here, we'll focus only on PPO-Clip (the primary variant used at OpenAI).

PPO 有两个主要的变体：PPO-惩罚和 PPO-剪辑。
PPO-惩罚近似地解决了像 trpo 这样的 KL 约束更新，但惩罚了目标函数中的 KL-发散，而不是使其成为一个硬约束，并在训练过程中自动调整惩罚系数，使其适当地缩放。PPO-剪辑在目标中没有 KL-发散项，而且完全没有约束.相反，它依赖于目标函数中的专门剪辑来消除对新政策的激励，使其远离旧的政策。

在这里，我们将只关注 PPO-剪辑(在 OpenAI 中使用的主要变体)。

3.2 Quick Facts

PPO 是一种在策略上的算法。
PPO 可用于具有离散或连续操作空间的环境。
PPO 的旋转实现支持 MPI 的并行化。

3.3 Key Equations

Key Equations

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

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

in which ϵ is a (small) hyperparameter which roughly says how far away the new policy is allowed to go from the old.

This is a pretty complex expression, and it's hard to tell at first glance what it's doing, or how it helps keep the new policy close to the old policy. As it turns out, there's a considerably simplified version [1] of this objective which is a bit easier to grapple with (and is also the version we implement in our code):

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \quad g(\epsilon, A^{\pi_{\theta_k}}(s, a)) \right),$$

where

$$g(\epsilon, A) = \begin{cases} (1 + \epsilon) A & A \geq 0 \\ (1 - \epsilon) A & A < 0. \end{cases}$$

To figure out what intuition to take away from this, let's look at a single state-action pair (s, a), and think of cases.

Advantage is positive: Suppose the advantage for that state-action pair is positive, in which case its contribution to the objective reduces to

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

Because the advantage is positive, the objective will increase if the action becomes more likely—that is, if $\pi_\theta(a|s)$ increases. But the min in this term puts a limit to how *much* the objective can increase. Once $\pi_\theta(a|s) > (1+\epsilon)\pi_{\theta_k}(a|s)$, the min kicks in and this term hits a ceiling of $(1+\epsilon)A^{\pi_{\theta_k}}(s,a)$. Thus: *the new policy does not benefit by going far away from the old policy.*

Advantage is negative: Suppose the advantage for that state-action pair is negative, in which case its contribution to the objective reduces to

$$L(s,a,\theta_k,\theta) = \max\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1-\epsilon)\right) A^{\pi_{\theta_k}}(s,a).$$

Because the advantage is negative, the objective will increase if the action becomes less likely—that is, if $\pi_\theta(a|s)$ decreases. But the max in this term puts a limit to how *much* the objective can increase. Once $\pi_\theta(a|s) < (1-\epsilon)\pi_{\theta_k}(a|s)$, the max kicks in and this term hits a ceiling of $(1-\epsilon)A^{\pi_{\theta_k}}(s,a)$. Thus, again: *the new policy does not benefit by going far away from the old policy.*

What we have seen so far is that clipping serves as a regularizer by removing incentives for the policy to change dramatically, and the hyperparameter ϵ corresponds to how far away the new policy can go from the old while still profiting the objective.

到目前为止，我们所看到的是，剪裁是一个正则化者，它消除了对政策急剧变化的激励，而超参数 epsilon 则对应于新政策与旧政策之间的距离，同时仍在为目标谋利。

虽然这种剪裁对于确保合理的策略更新有很大帮助，但仍然有可能产生一个与旧策略相去甚远的新策略，不同的 PPO 实现使用了许多技巧来避免这种情况。在这里的实现中，我们使用了一个特别简单的方法：早期停止。如果新政策与旧政策的平均 KL-分歧超过一个阈值，我们就停止采取梯度步骤。当您对基本的数学和实现细节感到满意时，值得查看其他实现，看看它们是如何处理这个问题的！

3.4 Exploration vs. Exploitation

PPO 以一种政策上的方式训练随机策略.这意味着它根据其随机政策的最新版本，通过抽样行动进行探索。行动选择的随机性程度取决于初始条件和训练过程。在培训过程中，策略通常变得越来越少随机性，因为更新规则鼓励它利用它已经发现的奖励。这可能导致政策陷入局部最优状态。

3.5 Pseudocode

Algorithm 1 PPO-Clip

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}.
  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.
  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

3.6 Documentation

3.7 Saved Model Contents

3.8 References

3.9 Relevant Papers

- Proximal Policy Optimization Algorithms, Schulman et al. 2017
- High Dimensional Continuous Control Using Generalized Advantage Estimation, Schulman et al. 2016
- Emergence of Locomotion Behaviours in Rich Environments, Heess et al. 2017

3.10 Why These Papers?

舒尔曼 2017 年之所以包括在内，是因为它是描述 PPO 的原始论文。Schulman 2016 之所以包括在内，是因为我们的 PPO 实现使用了广义优势估计来计算策略梯度。Heess 2017 之所以包括在内，是因为它对 PPO 代理在复杂环境中学习到的行为进行了大规模的实证分析(尽

管它使用的是 PPO 惩罚而不是 PPO-剪辑)。

3.11 Other Public Implementations

- Baselines
- ModularRL (Caution: this implements PPO-penalty instead of PPO-clip.)
- rllab (Caution: this implements PPO-penalty instead of PPO-clip.)
- rllib (Ray)

4 Deep Deterministic Policy Gradient

4.1 Background

4.2 Quick Facts

ddpg 是一种非策略算法.
ddpg 只能用于具有连续操作空间的环境。
对于连续的动作空间，ddpg 可以被认为是深度 Q 学习。
DDPG 的旋转实现不支持并行化。

4.3 Key Equations

这里，我们将解释 ddpg 的两个部分背后的数学：学习 q 函数和学习策略。

4.4 The Q-Learning Side of DDPG

技巧一：重放缓冲区。所有用于训练深度神经网络以逼近 Q^*(s，a)的标准算法都使用了经验回放缓冲器。这是以前经验的集合{mathcal d}。为了使算法具有稳定的行为，重播缓冲区应该足够大，以包含广泛的经验，但它可能并不总是好保持一切。如果你只使用最新的数据，你就会过度适应，事情就会破裂；如果你使用太多的经验，你可能会放慢你的学习速度。这可能需要一些调整才能正确。

4.5 The Policy Learning Side of DDPG

4.6 Exploration vs. Exploitation

ddpg 以一种非政策的方式训练一种确定性的政策。由于策略是确定性的，如果代理要探索 on-策略，那么在一开始它可能不会尝试足够广泛的各种操作来找到有用的学习信号。为了使 ddpg 策略更好地探索，我们在训练时给他们的行为添加了噪音。最初的 ddpg 论文的作者推荐了时间相关的 ou 噪声，但是最近的结果表明，不相关的、平均为零的高斯噪声工作得很好。因为后者比较简单，所以更可取。为了更好地获取高质量的培训数据，你可以减少训练过程中的噪音。(在实施过程中，我们不会这样做，并在整个过程中保持固定的噪音等级。)

4.7 Pseudocode

4.8 Documentation

4.9 Saved Model Contents

4.10 References

4.11 Relevant Papers

4.12 Why These Papers?

4.13 Other Public Implementations

5 Twin Delayed DDPG

5.1 Background

5.2 Quick Facts

5.3 Key Equations

5.4 Exploration vs. Exploitation

5.5 Pseudocode

5.6 Documentation

5.7 Saved Model Contents

5.8 References

5.9 Relevant Papers

5.10 Other Public Implementations

6 Soft Actor-Critic

6.1 Background

软参与者批判性算法(Sac)是一种以非策略的方式对随机策略进行优化的算法，它在随机策略优化和 ddpg 式方法之间架起了一座桥梁。它并不是 td3 的直接继承者(大致是同时发布的)，但它结合了剪裁的双 Q 技巧，而且由于 SAC 中策略的固有随机性，它也最终受益于类似目标策略平滑之类的东西。
Sac 的一个中心特征是熵正则化。该策略被训练成最大限度地在预期收益和熵之间进行权衡，熵是衡量策略中随机性的一种度量。这与勘探开发的权衡有着密切的联系：熵的增加导致了更多的探索，从而加速了以后的学习。它还可以防止策略过早地收敛到坏的局部最优。

6.2 Quick Facts

SAC 是一种非策略算法.
此处实现的 Sac 版本只能用于具有连续操作空间的环境。
可以实现 Sac 的另一个版本，它稍微改变策略更新规则，以处理离散的操作空间。
Sac 的旋转实现不支持并行化。

6.3 Key Equations

为了解释软演员批评，我们首先必须引入熵正则化的强化学习设置.在熵正则化的 rl 中，值函数的方程略有不同.

6.4 Entropy-Regularized Reinforcement Learning

6.5 Soft Actor-Critic

6.6 Exploration vs. Exploitation

SAC 训练具有熵正则化的随机策略，并在策略上进行探索.熵正则化系数α 显着地控制着勘探开发的权衡，较高的α 对应于更多的勘探，较低的α 对应于更多的开采。正确的系数(导致最稳定/最高回报的学习)可能因环境而异，可能需要仔细调整。在测试时，为了了解策略如何利用它所学到的知识，我们消除了随机性，使用了平均值操作，而不是从分布中选择一个样本。这倾向于提高性能比原来的随机策略。

6.7 Pseudocode

Algorithm 1 Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, V-function parameters ψ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\psi_{\text{targ}} \leftarrow \psi$
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 Q and V functions:

$$y_q(r, s', d) = r + \gamma(1 - d)V_{\psi_{\text{targ}}}(s')$$
$$y_v(s) = \min_{i=1,2} Q_{\phi_i}(s, \tilde{a}) - \alpha \log \pi_\theta(\tilde{a}|s), \qquad \tilde{a} \sim \pi_\theta(\cdot|s)$$

      Update Q-functions by one step of gradient descent using

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

      Update V-function by one step of gradient descent using

$$\nabla_\psi \frac{1}{|B|} \sum_{s \in B} \left(V_\psi(s) - y_v(s)\right)^2$$

      Update policy by one step of gradient ascent using

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

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

$$\psi_{\text{targ}} \leftarrow \rho\psi_{\text{targ}} + (1 - \rho)\psi$$

    end for
  end if
until convergence

6.8 Documentation

6.9 Saved Model Contents

6.10 References

6.11 Relevant Papers

6.12 Relevant Papers

- Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor, Haarnoja et al, 2018

6.13 Other Public Implementations

- SAC release repo