

# 近端策略优化算法 PPO

2016 年 10 月 OpenAI 公司发布了《Dota 2》强化学习游戏智能体 [OpenAI Five](#)，并且经过多年的开发训练后于 2019 年 4 月战胜了世界冠军团队 OG，技术细节可以看[发表的论文](#)。OpenAI Five 的核心算法是 PPO，一种称为近端策略优化的算法，属于策略梯度算法的一种。OpenAI Five 和 PPO 的成功大大增加了 AI 研究者对强化学习解决复杂问题的信心，PPO 也成为使用强化学习解决各类问题的一个基准算法。



本教程主要内容：

- 讲解 PPO 提出的背景
- 讲解 PPO 算法的理论和技巧
- 基于 Pytorch 实现 PPO 算法

## 一、REINFORCE 存在的问题

回顾 REINFORCE 算法，它属于策略梯度算法，所谓策略梯度算法就是通过求解强化学习问题中目标函数的梯度，并利用梯度提升方法训练强化学习智能体的一类算法。这类算法的鼻祖应该要算 REINFORCE 算法，[上一节](#)我们详细介绍并实现了该算法。REINFORCE 算法总结起来，包含以下几个步骤：

- 第一步，初始化一个随机策略  $\pi_\theta$ ，并用其收集  $N$  条轨迹

$$\{\tau^i : s_0^i, a_0^i, r_1^i, s_1^i, a_1^i, r_2^i, \dots\}_{i=0}^{N-1} \quad (1)$$

- 第二步，计算轨迹的收益  $G(\tau^i) = r_1 + r_2 + r_3 + \dots$ ，计算目标函数梯度

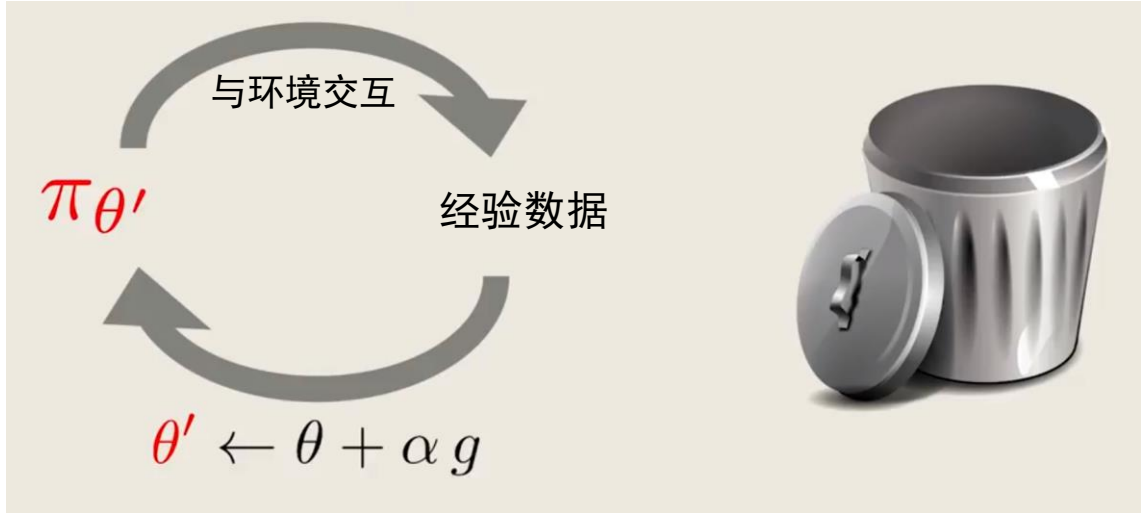
$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=0}^{N-1} G(\tau^i) \sum_t \nabla_\theta \log \pi_\theta(s_t^i, a_t^i) \quad (2)$$

- 第三步，按照梯度提升方式更新策略参数(更新率为  $\alpha$ )

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \quad (3)$$

REINFORCE 算法存在以下几个方面的问题：

- (1) 梯度的估计值波动很大。每次收集到的轨迹具有很大的随机性，根据轨迹估计出来的目标函数梯度常常变化很大。
- (2) 各个时间步的贡献分配不明确。一条轨迹中既有好的动作，也有差的动作。使用整条轨迹最终的收益去评价每一步的动作，没有将不同时间步动作对总收益的贡献区分出来。
- (3) 更新过程样本利用率非常低。收集到的样本用于一次策略更新后即被丢弃。



于问题(1),通过增加轨迹数量和收益归一化进行解决。当轨迹数量  $N \rightarrow \infty$ ，每一次梯度的估计都等于期望，不存在波动。

此外，随着智能体训练的进行，轨迹的收益分布也在发生变化。可能出现的情况是，训练刚开始智能体的轨迹收益很小，中后期智能体的收益变大。为了减小轨迹之间绝对值对训练造成的影响，对用于更新的同一批(a batch)样本

的收益进行归一化。设这一批样本包含  $N$  条轨迹，第  $i$  条轨迹收益的归一化方式为

$$R_i \leftarrow \frac{R_i - \mu}{\sigma}, \quad \mu = \frac{1}{N} \sum_i R_i, \quad \sigma = \sqrt{\frac{1}{N} \sum_i (R_i - \mu)^2} \quad (4)$$

这样一来，所有的收益都变成零均值了。这种批归一化(batch normalization)方法广泛应用于人工智能的许多子领域。

对于问题 (2)，回到梯度的计算，考虑某一条轨迹  $\tau$ ，有

$$\nabla_{\theta} J(\theta) = (\cdots + r_t + r_{t+1} + \cdots) \sum_t \nabla_{\theta} \log \pi_{\theta}(s_t, a_t) \quad (5)$$

根据该公式，无论我们计算哪个时间步，都是用相同的轨迹收益  $G$ 。事实上，对于  $t$  时刻而言，轨迹的收益包含两部分

$$G = (\overbrace{\cdots + r_{t-1} + r_t}^{G_t^{\text{past}}} + \overbrace{\cdots}^{G_t^{\text{future}}}) \quad (6)$$

其中  $G_t^{\text{past}}$  在动作  $a_t$  做出之前就出现的，不应该算法  $a_t$  的贡献里面。这的理由就是，强化学习的基本假设是马尔可夫决策过程，智能体和环境构成的是一个因果系统，后面发生的事情不能决定前面的结果。去掉  $G_t^{\text{past}}$  这部分后，梯度变为

$$\nabla_{\theta} J(\theta) = \sum_t G_t^{\text{future}} \nabla_{\theta} \log \pi_{\theta}(s_t, a_t) \quad (7)$$

有同学可能会担心修改梯度的计算方式会影响对原有目标函数的优化结果，实际上二者是等价的，但是具有更高的样本效率（即用更少样本即可收敛到有效的策略）。

对于问题 (3)，即解决样本利用率低的问题。一种自然而然的想法就是重复利用已经收集到的轨迹。但是一直使用旧的轨迹训练新的策略，会不会有问题？这部应该算是 PPO 的灵魂部分，下面重点介绍。

## 二、重要性采样

在 REINFORCE 算法中，收集到的轨迹在用于策略的一次更新后就被丢弃了，有点浪费。为了提高样本利用率，要想办法用同一批样本多次更新策略。但是，策略  $\pi_{\theta}$  经过一次更新后变成  $\pi_{\theta'}$ ，与环境的交互方式就会发生变化，要

产生同一条轨迹的概率就会发生变化， $p(\tau; \theta') \neq p(\tau; \theta)$ 。

回到 REINFORCE 中，用  $G_\tau$  表示整条轨迹  $\tau$  的奖励和，梯度的定义

$$\nabla_{\theta} J(\theta) = \overbrace{\sum_{\tau} p(\tau; \theta)}^{\text{所有轨迹的均值}} \underbrace{\left( G_{\tau} \sum_t \nabla_{\theta} \log \pi_{\theta}(s_t^{(\tau)}, a_t^{(\tau)}) \right)}_{\text{其中一条轨迹}} \quad (8)$$

根据(8)，如果使用旧的轨迹去训练更新后的策略，本质上就是忽略了轨迹生成概率已发生变化这一事实。为减小概率变化带来的影响，在旧的概率上乘以一个重新加权因子(re-weighting factor):  $p(\tau; \theta') / p(\tau; \theta)$ ，这样就变成了新策略下的轨迹生成概率

$$\overbrace{p(\tau; \theta)}^{\text{旧策略下轨迹生成概率}} \cdot \overbrace{\frac{p(\tau; \theta')}{p(\tau; \theta)}}^{\text{重新加权因子}} \quad (9)$$

对于新策略和旧轨迹，加入重新加权因子后，梯度表达式变成

$$\begin{aligned} \nabla_{\theta'} J(\theta') &= \sum_{\tau} p(\tau; \theta) \frac{p(\tau; \theta')}{p(\tau; \theta)} G_{\tau} \sum_t \nabla_{\theta'} \log \pi_{\theta'}(s_t^{(\tau)}, a_t^{(\tau)}) \\ &= \sum_{\tau} p(\tau; \theta) G_{\tau} \sum_t \frac{p(\tau; \theta')}{p(\tau; \theta)} \nabla_{\theta'} \log \pi_{\theta'}(s_t^{(\tau)}, a_t^{(\tau)}) \\ &= \sum_{\tau} p(\tau; \theta) G_{\tau} \sum_t \frac{p(\tau; \theta')}{p(\tau; \theta)} \frac{\nabla_{\theta'} \pi_{\theta'}(s_t^{(\tau)}, a_t^{(\tau)})}{\pi_{\theta'}(s_t^{(\tau)}, a_t^{(\tau)})} \\ &= \sum_{\tau} p(\tau; \theta) G_{\tau} \sum_t \frac{\cancel{\dots \pi_{\theta'}(s_t^{(\tau)}, a_t^{(\tau)}) \dots} \nabla_{\theta'} \pi_{\theta'}(s_t^{(\tau)}, a_t^{(\tau)})}{\cancel{\dots \pi_{\theta'}(s_t^{(\tau)}, a_t^{(\tau)}) \dots} \pi_{\theta'}(s_t^{(\tau)}, a_t^{(\tau)})} \\ &= \sum_{\tau} p(\tau; \theta) G_{\tau} \sum_t \frac{\cancel{\dots} \nabla_{\theta'} \pi_{\theta'}(s_t^{(\tau)}, a_t^{(\tau)})}{\cancel{\dots} \pi_{\theta'}(s_t^{(\tau)}, a_t^{(\tau)})} \end{aligned} \quad (10)$$

最终的表达式里面，还是有“...”这一项。能不能进一步简化？这里就是 PPO 的核心操作了，直接把“...”这部分给扔掉了，梯度表达式变为

$$\nabla_{\theta'} J(\theta') = \sum_{\tau} p(\tau; \theta) G_{\tau} \sum_t \frac{\nabla_{\theta'} \pi_{\theta'}(s_t^{(\tau)}, a_t^{(\tau)})}{\pi_{\theta'}(s_t^{(\tau)}, a_t^{(\tau)})} \quad (11)$$

对于某一条轨迹而言，结合贡献分配原理，梯度可以写为

$$\nabla_{\theta'} J(\theta') = \sum_t \frac{\nabla_{\theta'} \pi_{\theta'}(s_t, a_t)}{\pi_{\theta}(s_t, a_t)} G_t^{\text{future}} \quad (12)$$

### 三、代理函数及其裁剪

梯度推导出来了，我们可以使用梯度提升算法去更新策略

$$\theta'' \leftarrow \theta' + \alpha \nabla_{\theta'} J(\theta') \quad (13)$$

但这一步其实在编写代码实现的时候，并不会真的去把梯度求解出来，然后显示地去更新策略参数。而是使用下面的目标函数

$$L(\theta') = \sum_t \frac{\pi_{\theta'}(s_t, a_t)}{\pi_{\theta}(s_t, a_t)} G_t^{\text{future}} \quad (14)$$

作为新的代理函数(Surrogate Function)，直接扔给优化器去优化，这和上一节介绍的 REINFORCE 算法的代码实现情况是一样的。但是我们的代理函数的使用具有假设条件：**新策略和旧策略差异很小！**

为此，需要限制新策略  $\pi_{\theta'}$  和旧策略  $\pi_{\theta}$  之间的差异。怎么度量二者的差异呢？智能体的随机策略本质上是一个概率分布，衡量两个概率分布之间的差异，可能有很多方法，有人已经尝试过很多然后发现用散度(LK)去度量比较好。

所以一开始人们想到的办法就是，收集一批轨迹，然后用它们多次更新策略，但是每次更新之前都需要算一下散度，看看差异是不是足够小： $\text{KL}(\pi_{\theta'}, \pi_{\theta}) \leq \sigma$ ， $\sigma \in (0,1)$ 。如果散度太大了，那就把收集到的经验数据扔掉，重新收集后再训练。

其实这样子的 PPO 已经能够比较好地工作了。但是 PPO 的那篇论文里面，对代理函数进行了进一步的裁剪。也就是说，用限制新策略和旧策略的比值的方式强行限制策略更新过程

$$\text{clip}(\rho_t(\theta'), 1-\epsilon, 1+\epsilon) \quad (15)$$

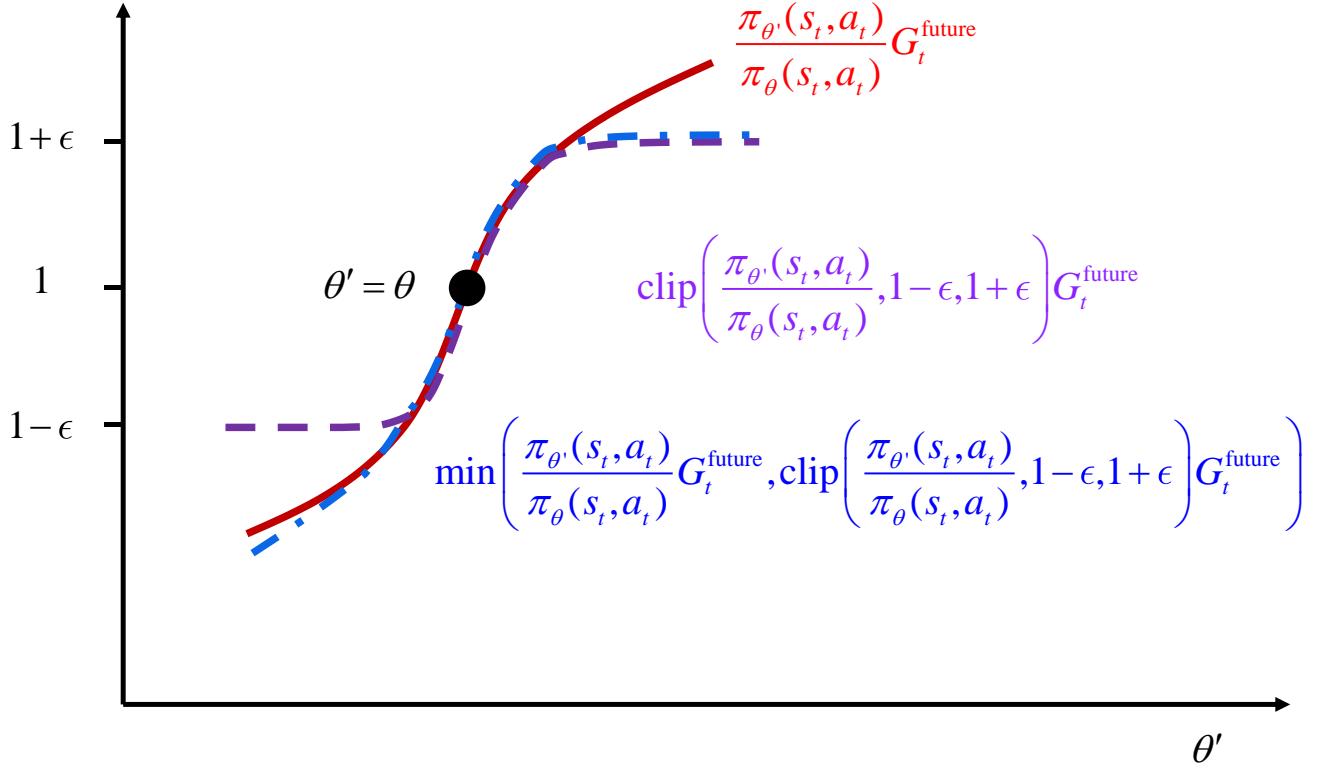
其中

$$\rho_t(\theta') = \frac{\pi_{\theta'}(s_t, a_t)}{\pi_{\theta}(s_t, a_t)} \quad (16)$$

同时，作者认为只需要限制比值不爆炸即可，因此将裁剪后的比值与原来

的又进行了一次取较小者的操作，此时的目标函数为

$$L^{\text{CLIP}}(\theta') = \mathbb{E}_t \left[ \min \left( \rho_t(\theta') G_t^{\text{future}}, \text{clip}(\rho_t(\theta'), 1-\epsilon, 1+\epsilon) G_t^{\text{future}} \right) \right] \quad (17)$$



#### 四、使用优势函数评价动作

在上面推导出来的  $L^{\text{CLIP}}(\theta')$  表达式子中，包含了  $G_t^{\text{future}}$  这一项，回顾它的定义

$$G_t^{\text{future}} = \sum_{t'=t} \gamma^{t'-t} r_{t'} \quad (18)$$

即  $t$  时刻未来奖励和(reward-to-go)。实际上我们会使用多条轨迹求期望，因此它就是未来奖励和的期望。同时注意到价值函数的定义

$$Q(s_t, a_t) = \mathbb{E} \left[ \sum_{t'=t} \gamma^{t'-t} r_{t'} \mid s_t, a_t \right] \quad (19)$$

和  $G_t^{\text{future}}$  有相同的形式，因此可以使用  $Q$  函数评价动作的好坏。进一步，为了稳定训练效果，常常会使用“优势函数(advantage function)”代替  $Q$  函数

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t) \quad (20)$$

其中  $V(s_t) = \mathbb{E}[\sum_{t'=t} \gamma^{t'-t} r_{t'} \mid s_t]$ ，为状态价值函数。因此，PPO 的原文里面的代理函数为



$$L^{\text{clip}}(\theta') = \mathbb{E}_t \left[ \min \left( \rho_t(\theta') A(s_t, a_t), \text{clip}(\rho_t(\theta'), 1-\epsilon, 1+\epsilon) A(s_t, a_t) \right) \right] \quad (21)$$

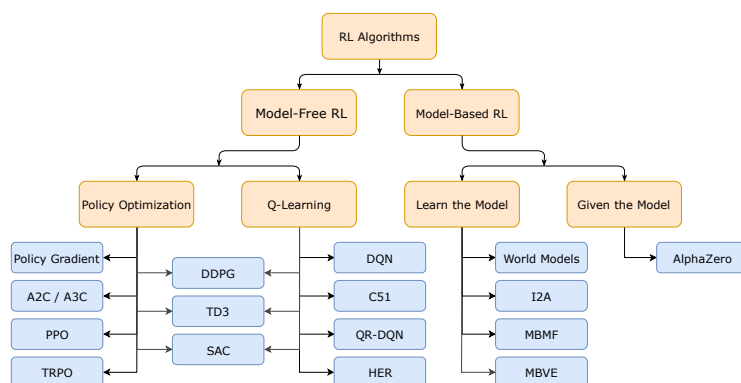
其中  $A(s_t, a_t)$  一般也是用神经网络实现的， $A(s_t, a_t)$  一般直接使用时间差分误差  $\delta = r + \gamma V(s') - V(s)$  进行估计，因此使用神经网络估计  $V(s)$  即可，该神经网络一般称为价值网络(value network)。价值网络通过最小化 TD 误差的方式进行训练(参考 [DQN 那一节的内容](#))。

到此，PPO 的理论部分就讲完了。

## 五、实例代码

PPO 这部分的代码参考 OpenAI 开发的一个致力于推广和标准化强化学习算法的项目——[Spinning UP](#)。这个项目包含项强化学习基础、研究和学习资源、算法文档、以及常见算法的实现。强化学习基础部分主要介绍强化学习的关键概念、分类、以及策略梯度思想。研究和学习资源列出了一些经典的文献，给出了一些练习，并给出了常见算法在 MuJoCo 环境中的算法性能基准。算法实现部分，实现的算法包括

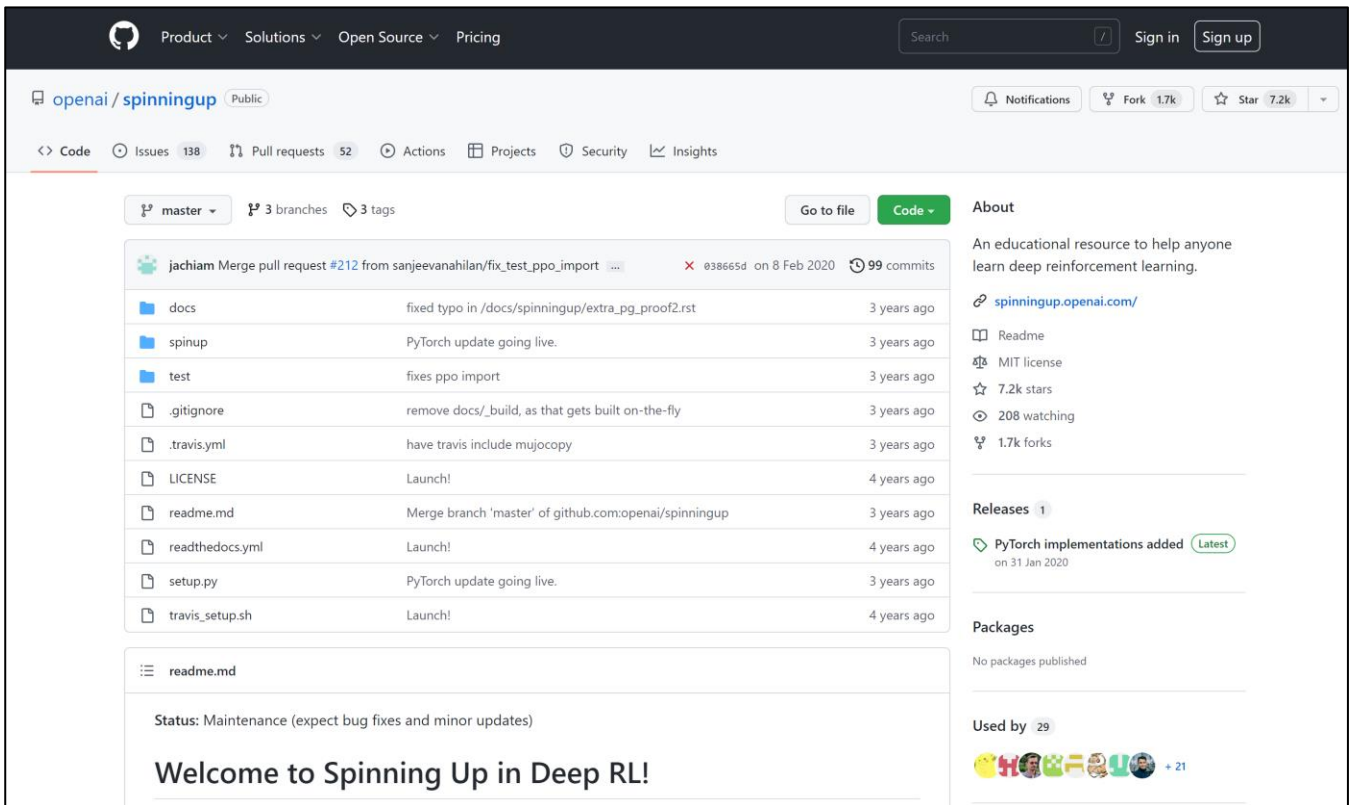
- Vanilla Policy Gradient (VPG)
- Trust Region Policy Optimization (TRPO)
- Proximal Policy Optimization (PPO)
- Deep Deterministic Policy Gradient (DDPG)
- Twin Delayed DDPG (TD3)
- Soft Actor-Critic (SAC)



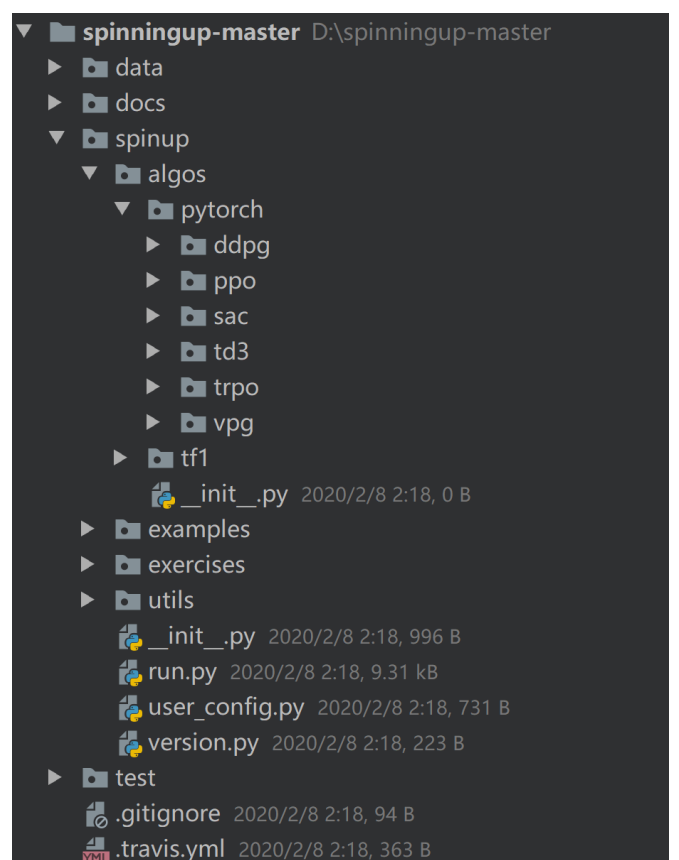
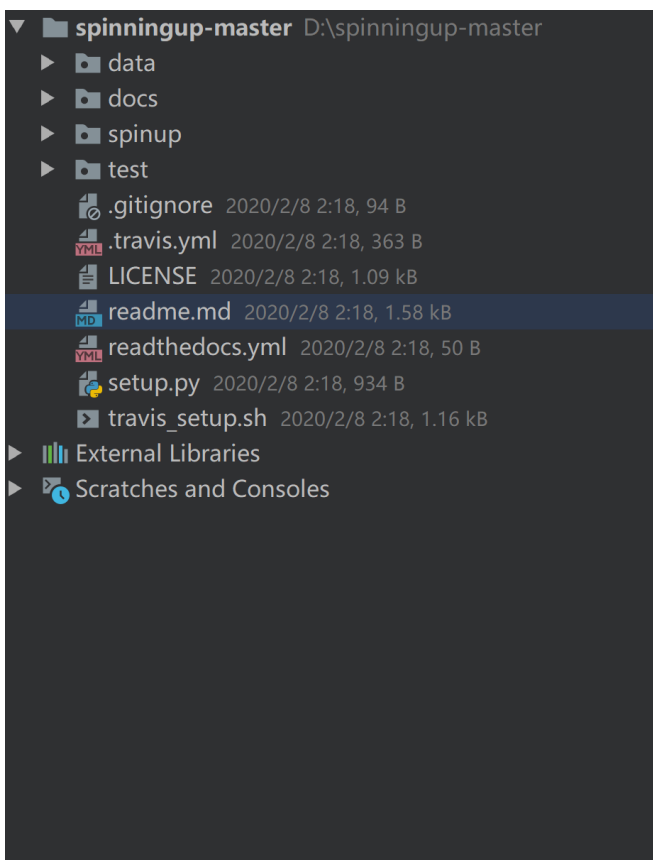
### (一) Spinning Up 代码使用

去[官网](https://github.com/openai/spinningup)（<https://github.com/openai/spinningup>）下载代码，项目结构如下。

Spinning Up 的代码对新手来说，有一定的难度，但是可靠性高。



The screenshot displays the GitHub repository page for `openai/spinningup`. The repository is public and has 7.2k stars, 208 watchers, and 1.7k forks. The file list includes `docs`, `spinup`, `test`, `.gitignore`, `.travis.yml`, `LICENSE`, `readme.md`, `readthedocs.yml`, `setup.py`, and `travis_setup.sh`. The commit history shows recent updates. The repository statistics show 7.2k stars, 208 watching, and 1.7k forks.



## （二）PPO 代码分析



*# Set up function for computing PPO policy loss*

```
def compute_loss_pi(data):  
    obs, act, adv, logp_old = data['obs'], data['act'], data['adv'], data['logp']
```

*# Policy loss*

```
    pi, logp = ac.pi(obs, act)  
    ratio = torch.exp(logp - logp_old)  
    clip_adv = torch.clamp(ratio, 1-clip_ratio, 1+clip_ratio) * adv  
    loss_pi = -(torch.min(ratio * adv, clip_adv)).mean()
```

*# Useful extra info*

```
    approx_kl = (logp_old - logp).mean().item()  
    ent = pi.entropy().mean().item()  
    clipped = ratio.gt(1+clip_ratio) | ratio.lt(1-clip_ratio)  
    clipfrac = torch.as_tensor(clipped, dtype=torch.float32).mean().item()  
    pi_info = dict(kl=approx_kl, ent=ent, cf=clipfrac)
```

```
    return loss_pi, pi_info
```

*# Set up function for computing value loss*

```
def compute_loss_v(data):  
    obs, ret = data['obs'], data['ret']  
    return ((ac.v(obs) - ret)**2).mean()
```

```

def update():
    data = buf.get()

    pi_l_old, pi_info_old = compute_loss_pi(data)
    pi_l_old = pi_l_old.item()
    v_l_old = compute_loss_v(data).item()

    # Train policy with multiple steps of gradient descent
    for i in range(train_pi_iters):
        pi_optimizer.zero_grad()
        loss_pi, pi_info = compute_loss_pi(data)
        kl = mpi_avg(pi_info['kl'])
        if kl > 1.5 * target_kl:
            logger.log('Early stopping at step %d due to reaching max kl.' % i)
            break
        loss_pi.backward()
        mpi_avg_grads(ac.pi)    # average grads across MPI processes
        pi_optimizer.step()

    logger.store(StopIter=i)

    # Value function learning
    for i in range(train_v_iters):
        vf_optimizer.zero_grad()
        loss_v = compute_loss_v(data)
        loss_v.backward()
        mpi_avg_grads(ac.v)    # average grads across MPI processes
        vf_optimizer.step()

```

