# 如何直观理解PPO算法[实战篇]（附代码及代码解释）

[](https://www.zhihu.com/people/zhang-si-jun-52)

[张斯俊](https://www.zhihu.com/people/zhang-si-jun-52" \t "/home/admin/文档\\x/_blank)

愿成为一把梯子，助你跨过无数的坑。

在上一篇文章中，已经说了PPO三个重点：

* 用网络求解连续动作型问题；
* 进行N步更新；
* 重要性采样及PPO网络的更新学习。

本篇将会解释的示例代码，同样会以这三个为重点。

如果对原理有所疑问，可以翻看上一篇专栏：

[张斯俊：如何直观理解PPO算法?[理论篇]​](https://zhuanlan.zhihu.com/p/111049450" \t "/home/admin/文档\\x/_blank)

这一篇，以tensorflow给出的强化学习算法示例代码为例子，看看PPO应该如何实现。

[https://github.com/tensorlayer/tensorlayer/blob/master/examples/reinforcement\_learning/tutorial\_PPO.py​github.com](https://link.zhihu.com/?target=https://github.com/tensorlayer/tensorlayer/blob/master/examples/reinforcement_learning/tutorial_PPO.py" \t "/home/admin/文档\\x/_blank)

如果看代码有困难，可以看我的带注释版本。希望能帮助到你。

[https://github.com/louisnino/RLcode/blob/master/tutorial\_PPO.py​github.com](https://link.zhihu.com/?target=https://github.com/louisnino/RLcode/blob/master/tutorial_PPO.py" \t "/home/admin/文档\\x/_blank)

### Pendulum-v0 说明

在开始解释算法前，先对这次要征服的游戏做一个简单的说明。

Pendulum-v0是一个简单的环境，如图。



图中的杆围绕A点转动的任务是，让智能体学会给杆子施加一定的“力”，让杆子立起来。

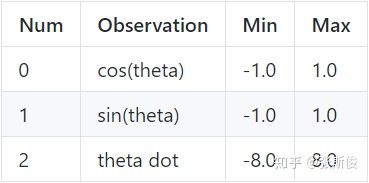
这里的“力”用了带箭头的圆环表示，圆环越大，表示用的“力”越大。

这里的“力”之所以用双引号括起来，是这个游戏并不完全遵从现实世界的物理规则。它只是一个游戏而已。

正如之前说到，这个“力”并不是开关，是有大小之别；因此，这是一个连续型控制的问题。

主要关心还是三要素，state，action，reward

输入的state特征有三个



输入的action是[-2.0,2.0]范围内的数值

奖励的计算： -(theta^2 + 0.1 *theta\_dt^2 + 0.001* action^2)

由此，可以知道，reward的最小值是-16.273 最大值是0。获得最大值时，就是杆子正立的时候。任务就是调整输入的“力”的大小，让杆子一直树立起来。

### 求解连续动作问题

一起看看\_build\_anet()函数如何实现这一点。

def \_build\_anet(self, name, trainable):

tfs = tl.layers.Input([None, S\_DIM], tf.float32, name + '\_state')

l1 = tl.layers.Dense(100, tf.nn.relu, name=name + '\_l1')(tfs)

a = tl.layers.Dense(A\_DIM, tf.nn.tanh, name=name + '\_a')(l1)

mu = tl.layers.Lambda(lambda x: x \* 2, name=name + '\_lambda')(a)

sigma = tl.layers.Dense(A\_DIM, tf.nn.softplus, name=name + '\_sigma')(l1)

model = tl.models.Model(tfs, [mu, sigma], name)

if trainable:

model.train()

else:

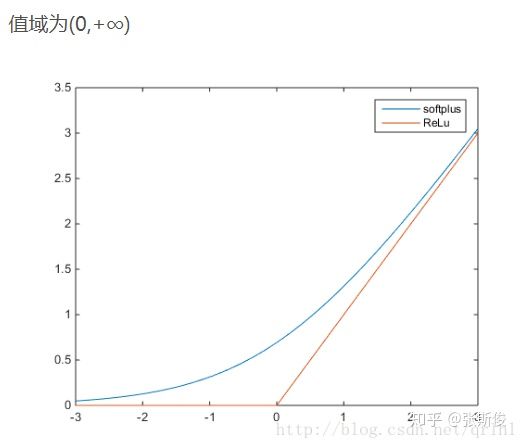
model.eval()

return model

把网络分为三个部分：

1. 输入和计算，示例代码中tfs和l1层，主要是用于输入和计算。这两部分之前已经学过，所以不再赘述。

1. 计算mu，也就是正态分布的均值，也就是最有可能取的action的地方。用tanh激活函数，tanh激活函数函数长这样子，取值范围值[-1.0,1.0]，也就是说，无论l1层输出的数值是多少，a层的tanh激活函数，也会把取值折到[-1.0,1.0]之间。 但由于这个环境，动作的取值是[-2.0,2.0]，所以再加lambda层，把范围映射到[-2.0,2.0]中。
2. 计算方差sigma，softplus激活函数长这样子，可以看出，和Relu很像。是的，其实softplus的作用和relu是一样的。relu有时候会导致神经元“死亡”的问题，softplus可以避免。



4、最后模型的输出，需要把mu和sigma一起输出。

**敲黑板** 那怎么把mu和sigma变成需要的动作呢？需要把mu和sigma代表的正态分布先还原出来，然后在这个分布下再进行抽样。

choose\_action函数：

def choose\_action(self, s):

# 通过actor计算出分布的mu和sigma

# 然后sample动作，并进行裁剪。

s = s[np.newaxis, :].astype(np.float32)

mu, sigma = self.actor(s)

pi = tfp.distributions.Normal(mu, sigma)

a = tf.squeeze(pi.sample(1), axis=0)[0] # choosing action

return np.clip(a, -2, 2)

1. 用mu和sigma建立分布是非常麻烦的，但不怕，tensorflow已经帮准备好函数。只需要调用就可以了。这个分布其实就是的策略pi。
2. 在pi随机抽样出一个动作a。
3. 需要对a的取值范围进行裁剪，因为超过取值范围的数值输入到环境，可能会造成其他问题。这里用clip函数，让a保持在[-2,2]之间。如果大于2，则返回2；如果小于-2，则返回-2。

### 整体流程框架

在这里重点看如何实现N步更新。

if (t + 1) % BATCH == 0 or t == EP\_LEN - 1:

v\_s\_ = ppo.get\_v(s\_)

discounted\_r = []

for r in buffer\_r[::-1]:

v\_s\_ = r + GAMMA \* v\_s\_

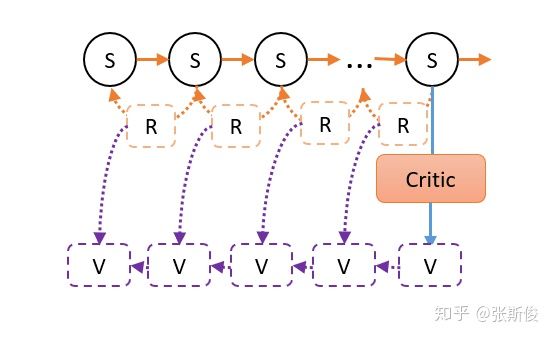
discounted\_r.append(v\_s\_)

discounted\_r.reverse()

bs, ba, br = np.vstack(buffer\_s), np.vstack(buffer\_a), np.array(discounted\_r)[:, np.newaxis]

buffer\_s, buffer\_a, buffer\_r = [], [], []

1. 首先让智能体在环境中探索，并记录每一步的数据。
2. 每N步（在代码中，N用BATCH表示），整理数据，并进行更新。
3. 计算这N步中，最后一步的下一步状态s'的V值。
4. 用公式v\_s\_ = r + GAMMA \* v\_s\_计算，并记录。
5. 把列表倒叙，和s，a对应，并记录。



要注意，这里的每个state，都是基于最后一步的下一步的状态估算出来的，并不需要每个state都计算一次。

记得把discounted\_r用reverse()函数反转。这样就算出这多步state的V值了。

### PPO2

从update函数说起。

1.在updata函数中，把准备好的数据赋值给s, a, r。

2.执行update\_old\_pi。

* old\_pi的作用其实就是记录的行为策略，现在准备好的数据是从old\_pi这个策略下获得的，也就是说目前的数据的版本都是old\_pi。
* pi是正在更新的目标策略。
* 在开始的时候pi和old\_pi是一样的，但随着更新，pi会被更新到更高版本，而old\_pi记录的是跑数据的额版本。通过old\_pi和pi就能求得重要性权重。

3.计算advantage，这在上一小节说过，这里就不赘述了。

4.Actor更新若干次

5.Critic更新若干次

Critic的更新和AC是一样的，这里就不再赘述了。主要来说说Actor如何用importance sampling更新。

1. 把目标策略和行为策略的分布计算出来，代码如下。

mu, sigma = self.actor(tfs)

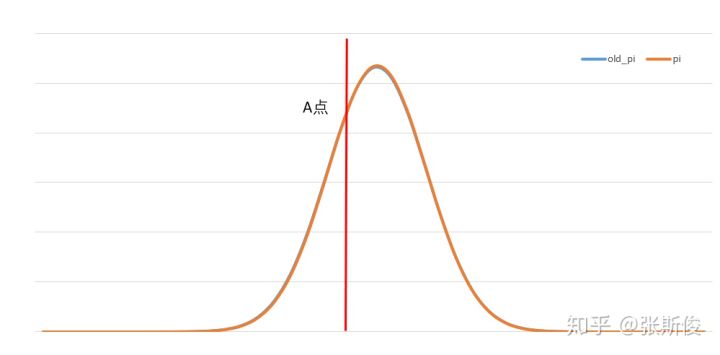
pi = tfp.distributions.Normal(mu, sigma)

mu\_old, sigma\_old = self.actor\_old(tfs)

oldpi = tfp.distributions.Normal(mu\_old, sigma\_old)

可以画这样一张示意图，old-pi和pi，两个分布是一样的，所以他们叠在一起。

而红色竖线表示，当前选择的action，A点表示这个action对应的概率。



当开始更新的时候，pi将会开始调整。

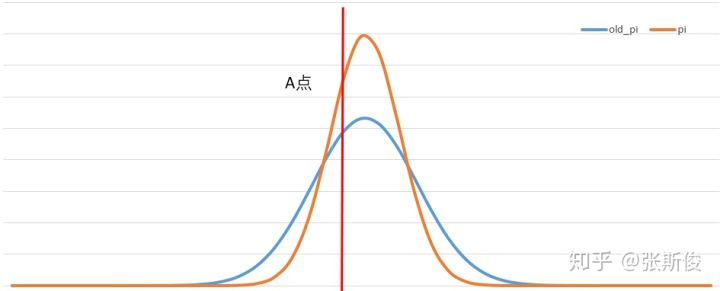
回忆一下AC和PG学到的，A点最终会向上或者向下调整。

决定A点向上或者向下的是advantage，如果advantage是正得，将会向上提，如果advantage是负数，那么将会向下移动。而每次移动的幅度，由advantage的大小决定。

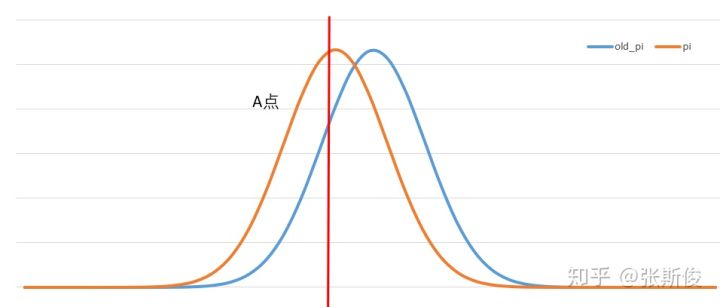
在示例代码中用advantage代表td-error。两者是同一回事。

但A点的移动并不是一个点在移动，而是通过调整输出的分布进行移动。

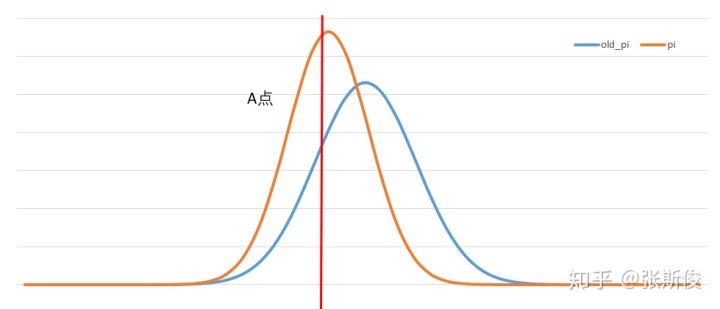
例如网络可能修改sigma，令pi的分布更集中，从而提高A点的概率。如图。



也有可能修改mu的，移来移动正态分布，使得A的概率提升。如图。



也有可能移动和调整分布都有，无论如何就是要让A输出的概率更高一些。



如果能比较形象地理解这个过程，那么以下的代码就很好理解了。

ratio = pi.prob(tfa) / (oldpi.prob(tfa) + EPS)

在第一次更新以后，两个分布的形状就不一样了。之前说过，一个分布B抽样的数据如果要用到分布P上，需要做重要性采样。也就是要乘以一个重要性权重。

pi(a)和oldpi(a)两者相除，就是要的importance weight，重要性权重。

在这里，可以忽略EPS，EPS的功能是为了避免oldpi.prob(tfa) = 0，出现除零错误。

**敲黑板** 在之前说到，加上重要性采样这个技术就可以了。但实际上发现，还没那么简单。人们发现，当两个分布差距太大的时候，就会有问题。

于是，还得限制两个分布差距不能太大！

在PPO1里面，用了是KL散度来衡量两个分布的差距。而PPO2是简单粗暴许多，直接裁剪了更新的范围。但这种简单粗暴却出乎意料地好，真的很神奇。

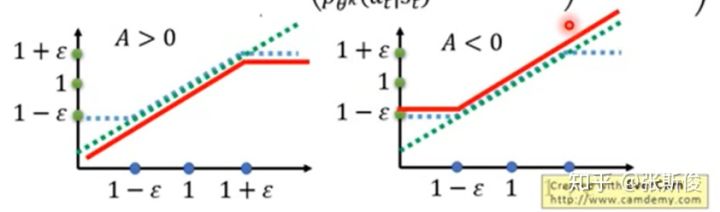
来看一下这几行代码。

aloss = -tf.reduce\_mean(

tf.minimum(surr, #ratio \* tfadv

tf.clip\_by\_value(ratio, 1. - METHOD['epsilon'], 1. + METHOD['epsilon']) \* tfadv)

)

来自李宏毅老师的课程截图

代码看上去很复杂，但要做的事其实很简单，就是希望直接限制重要性权重。

当Advantage正数的时候，做一个封顶，不让重要性参数太大，担心概率提升过头；当Advantage是负数的时候，重要性参数也不要太大，概率下降过了头。

以advantage是正数为例：

* 当ratio大于1. + METHOD['epsilon']的时候，就当 1. + METHOD['epsilon'])好了，不要太猖狂了。
* 当ratio小于1. - METHOD['epsilon']的时候，就用minimum函数，也就是说就选限制的ratio就好了。

就是这样，分布不断在有限度地调整，调整a的输出。

### 总结

PPO是比较复杂的一个算法，主要有三个知识点。

1. 用网络求解连续动作型问题；
2. 进行N步更新；
3. 重要性采样及PPO网络的更新学习。

但PPO的适用范围是很广的，而且比较稳定，值得拥有。

下一篇，将会介绍另外一个解决连续控制型问题的算法——DDPG。