# [A3C]：看我的影分身之术(附代码及代码分析)

A3C: Asynchronous Advantage Actor-Critic

[](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。

### A3C才能解锁更高级的技能

之前说过，强化学习的一个难点，智能体的用于学习的数据，需要智能体和环境不断进行交互。和一般有监督学习的先比，数据数量太少了。

所以，在算法没有更大进步的时候，有人就想出，如果我有多个智能体和环境进行互动，那么每个智能体都能产出数据，这些数据就可以一起给模型进行学习了。

基于这个思路和AC架构，于是便有了A3C。

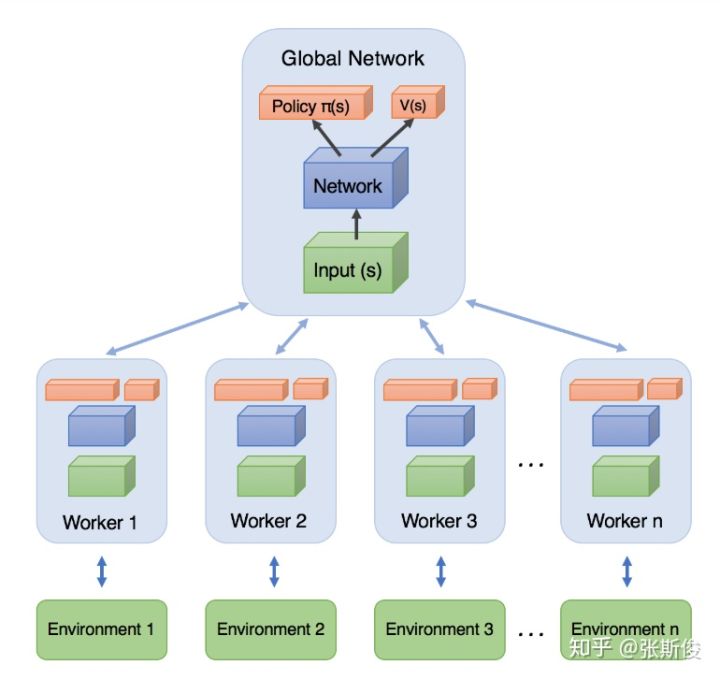
所以A3C在思想上并没有什么突破，原理上的理解也没有什么难度。更多的是实际执行上的理解。

但如果你对AC还不够了解，可以先看看这篇：

[张斯俊：理解Actor-Critic的关键是什么？​](https://zhuanlan.zhihu.com/p/110998399" \t "/home/admin/文档\\x/_blank)

### A3C的架构

先看看经典的A3C架构图。



架构图分为两个主要部分：Global Network（全局网络）和 worker（工人）

其实Global network和 worker都是一模一样的AC结构网络。全局网络并不直接参加和环境的互动，工人与环境有直接的互动，并且把学习到的东西，汇报给全局网络。

这有点像一个班长和同学之间的故事。

邪恶的老师给聪明的班长一个任务，让班长一天之内交出一份十万字的莎士比亚全集的读书心得。

这怎么可能？但班长深得班里同学们喜爱。班长决定发动全体同学来完成这个任务。

但怎么分配任务呢？

聪明的班长给每个同学一套莎士比亚全集，然后公布了任务分配的规则：

1. 同学们就随便看任何一个你们感兴趣的段落就可以了，就算是重复了也没关系；
2. 但同学们需要把读过的提炼成**心得**，每隔一段时间汇报给班长；
3. 班长会负责汇总大家的心得; 但同学提交自己的心得之后，要看一下当前被汇总的最新版本的**心得**，因为这是集体的智慧，有助于大家提高阅读水平。
4. 最后，班长会把最新版本的**心得**提交给老师。

邪恶的老师看到这篇凝聚了大众智慧的读书心得，感动得流下泪水。而这位聪明的班长在毕业后投身AI事业，发明出A3C算法。

...故事我编不下去了...

相信大家在这个故事中，已经对A3C的思路有个大致的想法。其中有几点需要大家注意的。

1. 在A3C中，worker不仅要和环境互动，产生数据，而且要自己从这些数据里面学习到“心得”。这里的所谓新的，其实就是计算出来的梯度；

需要强调的是，worker向全局网络汇总的是**梯度**，而不是自己探索出来的**数据**。

在这一点上，很容易和后面要学习的DPPO混淆。DPPO和A3C一样，也是一个分布式的架构，但work自己并不学习，而是提交数据让全局网络学习。这有点像同学们并不写“心得”，而是在书里划了重点，让班长自己学习。这会在DPPO中详细介绍。

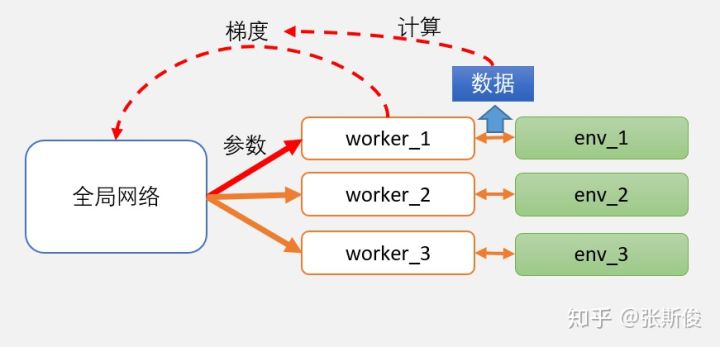
这大家可以想一下，可以直接提供数据让全局网络学习吗？大家可以从on policy 和off policy的角度想一下。

如果想不明白，可以看看专栏中对PPO的介绍:

[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)

1. worker向全局网络汇总**梯度**之后，并应用在全局网络的参数后，全局网络会把当前学习到的最新版本的**参数**，直接给worker。

worker按照最新的网络继续跟环境做互动。互动后，再把梯度提交，获取新的参数...... 如此循环。



现在对A3C的思路已经有所了解了，但应该怎样做呢？这就要用到python的两个概念：全局变量、多线程。

### 全局变量

全局变量，对应是局部变量。大家先来看下面一段代码：

a = 1

def fun():

a = a + 1 #1 local variable 'a' referenced before assignment

print("a=",a)

代码中，fun函数希望打印 变量a + 1 。按照运算，应该输出：2.

但实际执行中，会抛出异常：local variable 'a' referenced before assignment。这是为什么呢。

这是因为第一行代码中，a = 1，定义了全局变量a， a的作用域是整个代码。也就是整个代码都可以去读取它。大家可以可以尝试一下，把第3行代码注释掉。程序可以完整进行，并输出“a=1" 的结果。

但如果尝试修改全局变量a的值，但没有在函数中声明这个a是全局变量。那么python就会默认认为，这个a是一个局部变量，而这个变量还没在之前定义。

所以，如果要修改全局变量，需要在函数内声明。

a = 1

def fun():

global a #声明a是全局变量中的那个a

a = a + 1

print("a=",a)

if \_\_name\_\_=="\_\_main\_\_":

fun()

这样修改之后，程序就正常输出了。

所以应该了解两点： 1. 变量分为全局变量和局部变量。函数中使用的变量默认为局部变量，函数可以对局部变量进行读取和修改。

1. 函数可以读取全局变量，但又没权利去修改。如果需要修改，必须先进行“global声明”。

### 多线程

多线程的对应概念是单线程。所谓线程其实也很好理解，一个线程就相当于一条流水线。多线程，就是多条流水线协同工作。

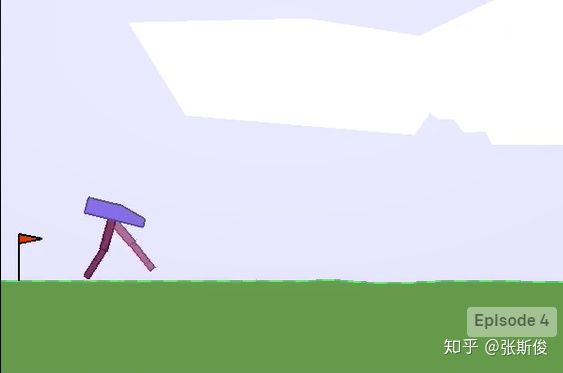
之前说到A3C有若干个worker一起和环境进行互动。利用python的多线程，就可以实现这个功能。

可以让每个智能体和环境进行互动，产生数据，计算梯度这一过程作为一个线程。这些进行会同时进行，互不干扰。

当一个线程计算好梯度，就可以让全局网络按照这个梯度更新。在更新之后，把全局网络的参数覆盖到本地智能体上，智能体按照这个网络继续和环境进行互动。

这些将会在稍后的代码分析中具体介绍。

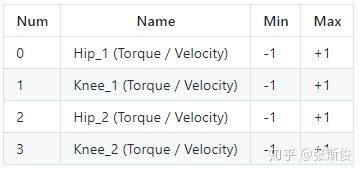
### BipedalWalker-v2



BipedalWalker的任务是让途中的机器人向前行走。

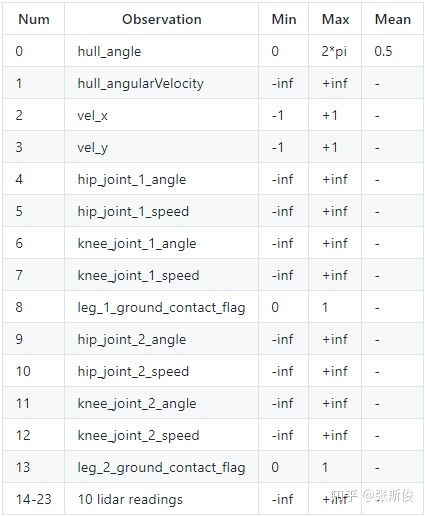
机器人有4个关节可以控制。分别是左腿的根部连接，左腿膝部连接，右腿根部连接，右腿膝部连接。所以这个机器人是在模仿双足动物向前行走的过程。

所以这个机器人动作空间，如图：



BipedalWalker向前走得越远，分数越多。最多可以300分数。如果机器人摔倒，那么扣100分。

机器人的状态空间相对复杂，大家可以仔细研究该表。



### 示例代码解释

这一篇，以tensorflow给出的强化学习算法示例代码为例子，看看A3C应该如何实现。

[https://github.com/tensorlayer/tensorlayer/blob/master/examples/reinforcement\_learning/tutorial\_A3C.py​github.com](https://link.zhihu.com/?target=https://github.com/tensorlayer/tensorlayer/blob/master/examples/reinforcement_learning/tutorial_A3C.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\_A3C.py​github.com](https://link.zhihu.com/?target=https://github.com/louisnino/RLcode/blob/master/tutorial_A3C.py" \t "/home/admin/文档\\x/_blank)

### 创建线程

和平常一样，先来看主流程。但和平常不一样的是，主流程并不是开始游戏跑数据，而是创建线程。在每个线程中，再进行游戏。

with tf.device("/cpu:0"): #以下部分，都在CPU0完成

OPT\_A = tf.optimizers.RMSprop(LR\_A, name='RMSPropA') #创建Actor的优化器

OPT\_C = tf.optimizers.RMSprop(LR\_C, name='RMSPropC') #创建Critic的优化器

GLOBAL\_AC = ACNet(GLOBAL\_NET\_SCOPE) # 创建全局网络GLOBAL\_AC

workers = [] # workers列表

1. with tf.device("/cpu:0"): 这行代码表示，指定在cpu：0进行以下代码。如果安装的是GPU版本，也可以使用/gpu:0。

tensorflow中不同的GPU使用/gpu:0和/gpu:1区分，而CPU不区分设备号，统一使用 /cpu:0

1. GLOBAL\_AC = ACNet(GLOBAL\_NET\_SCOPE)，创建一个GLOBAL\_AC的网络，这个网络并不直接学习，相当于班长的角色。

# 创建worker

# Create worker

for i in range(N\_WORKERS):

i\_name = 'Worker\_%i' % i # worker name

workers.append(Worker(i\_name, GLOBAL\_AC)) # 创建worker，并放在workers列表中，方便统一管理

1. 会创建n个worker，worker的数量最好和cpu的核一致，因为每个线程都是在一个单独的cpu进行。如果超过CPU数量，那么就失去了多线程的意义了。 可以用语句：N\_WORKERS = multiprocessing.cpu\_count() 获取cpu的数量。
2. 按照cpu的核数，创建多个worker对象，每个worker的具体作用会在下面详细说明
3. 把每个worker对象都存放在一个workers列表中，方便使用。

现在只有多个worker对象，但还没有对应的线程。现在就开始为每个worker开一个线程，让他们同时工作。

worker\_threads = []

for worker in workers: #执行每一个worker

# t = threading.Thread(target=worker.work)

job = lambda: worker.work(GLOBAL\_AC) #worker要执行的工作。

t = threading.Thread(target=job) #创建一个线程，执行工作

t.start() #开始线程，并执行

worker\_threads.append(t) #把线程加入worker\_threads中。

COORD = tf.train.Coordinator() #创建tensorflow中协调器

COORD.join(worker\_threads) #线程由COORD统一管理即可

1. 首先要定义，每个线程执行一系列动作流程，一般来说是一个函数。这个函数包含了整个流程。
2. threading.Thread(target=job)创建线程，其中target要执行的函数。
3. start，开始所有的线程。
4. 把线程放入worker\_threads列表
5. 创建tf的协调器，然后把worker\_threads放入，同一协调。

### worker怎样工作

现在重点来看看，每个worker是怎样工作的。

global GLOBAL\_RUNNING\_R, GLOBAL\_EP

GLOBAL\_RUNNING\_R,GLOBAL\_EP都是全局变量，是所有woker公用的变量，用来记录奖励获得的情况和当前第几个更新的EP。 注意，这里用了global去声明，所以这个变量就可以再这个函数里修改了。

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

buffer是局部的，是每个worker都各自有一个。余下和环境进行互动，保存互动产生的数据，这都是基于AC，所以如果大家有疑问的话，可以翻看AC算法。

现在再看self.AC.update\_global，这是用于更新全局网络的。

### update\_global

update\_global的更新也是基于AC算法，critic会最小化td-error，而actor用td-error作为权重更新。这些也在AC说过，这里可以对比AC算法。

要注意的是两个重点：

1. 更新全局网络的参数。

Critic网络的更新：

self.c\_grads = tape.gradient(self.c\_loss, self.critic.trainable\_weights)

OPT\_C.apply\_gradients(zip(self.c\_grads, globalAC.critic.trainable\_weights))

Actor网络的更新：

self.a\_grads = tape.gradient(self.a\_loss, self.actor.trainable\_weights)

OPT\_A.apply\_gradients(zip(self.a\_grads, globalAC.actor.trainable\_weights))

从buffer拿出数据，计算td-error，求a\_loss,c\_loss，求梯度。这个过程都是在本地进行的，都是对该worker的自身网络参数进行计算的。

当求出梯度后吗，更新的不是本地的网络，而是全局网络(globalAC)的参数。**注意apply\_gradients函数里面的参数是globalAC。**

由于每个线程的梯度不同，计算时间不同，所以不同线程会各自去更新全局网络。更新好之后，就把当前最新的网络拿下来，作为新的网络和环境进行互动。

所以worker的每次更新之间，会有其他的woker更新，所以在比较短的时间内，智能体就能获得巨大的提升。

1. 加入最大熵

如果仔细对比actor的计算，大家可以发现，A3C比AC的loss要多一项,最大熵。

entropy = normal\_dist.entropy()

这里说的熵并不是热力学第二定律的热熵；而是信息熵。

最大熵说起来很复杂，但大家可以先这样理解。信息熵是用来衡量不确定性的，当一件事情的不确定性越大，熵越大。

actor求出mu和std，建立一个服从正态分布的概率分布：normal\_dist。entropy就是这个概率分布的熵。

把熵和原来加权loss加起来，并最大化他们。目的是说，不但要输出的动作能够按照加权调整概率分布，同时希望这个分布的熵更大，也就是说随机性尽量大。从而使得探索能够最大化。

### 总结

其实A3C在思路就是说的“影分身”。只要能够熟练运用全局参数和多线程等工具，A3C是比较容易的。

A3C其实本质上并没有和AC有太大的差别，这种影分身的方法，在工业应用上会更加常用。毕竟，时间就是生命。

下一篇，来到该单元最后一篇DPPO。