# 什么是TD3算法？（附代码及代码分析）

[](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。

TD3是Twin Delayed Deep Deterministic policy gradient algorithm的简称，双延迟深度确定性策略梯度

Deep Deterministic policy gradient 不用解释了，就是DDPG。也就是说TD3是DDPG的一个优化版本。

其中有三个非常重要的优化。明白了这三个优化，就明白了TD3了。

### double network

还是得从DQN说起。DDPG起源于DQN，是DQN解决连续控制问题的一个解决方法。

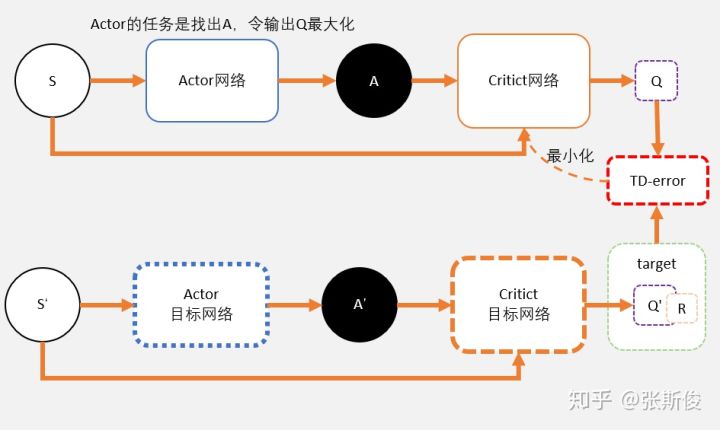
而DQN有一个众所周知的问题，就是Q值会被高估。这是因为用argmaxQ(s')去代替V(s')，去评估Q(s)。当每一步都这样做的时候，很容易就会出现高估Q值的情况。

而这个问题也会出现在DDPG中。而要解决这个问题的思路，也在DQN的优化版本中。相信大家很快就明白，就是double DQN。

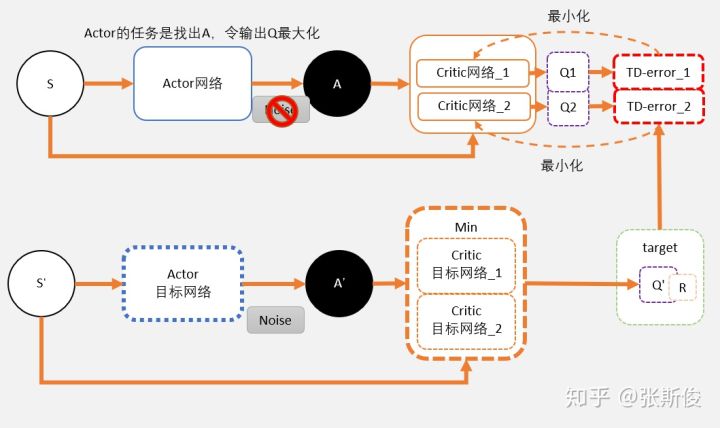
在TD3中，用了两套网络估算Q值，相对较小的那个作为更新的目标。这就是TD3的基本思路。

但要注意，DDPG算法涉及了4个网络，所以TD3需要用到6个网络。所以在实做得时候是比较容易出错的。所以有必要理清楚之间的关系。

先看看DDPG网络架构：



关注上图中，通过Critic网络估算动作的A值。一个Critic的评估可能会较高。所以加一个。



这就相当于把DDPG网络架构图中的Critic方框，一个变为两个。

在目标网络中，估算出来的Q值会用min()函数求出较少值。以这个值作为更新的目标。

这个目标会更新两个网络 Critic网络\_1 和 Critic网络\_2。

你可以理解为这两个网络是完全独立，他们只是都用同一个目标进行更新。

剩余的就和DDPG一样了。过一段时间，把学习好的网络赋值给目标网络。

再仔细分别看Critic部分和Actor部分的学习。

### Critic部分的学习

只有在计算Critic的更新目标时，才用target network。其中就包括了一个Policy network，用于计算A'；两个Q network ,用于计算两个Q值：Q1(A') 和Q2(A')。

Q1(A') 和Q2(A') 取最小值 min(Q1,Q2) 将代替DDPG的 Q(a') 计算更新目标，也就是说： target = min(Q1,Q2) \* gamma + r

target 将会是 Q\_network\_1 和 Q\_network\_2 两个网络的更新目标。

这里可能会有同学问，既然更新目标是一样的，那么为什么还需要两个网络呢?

虽然更新目标一样，两个网络会越来越趋近与和实际q值相同。但由于网络参数的初始值不一样，会导致计算出来的值有所不同。所以可以有空间选择较小的值去估算q值，避免q值被高估。

### Actor部分的学习

在DDPG中说过，DDPG网络图像上就可以想象成一张布，覆盖在qtable上。当输入某个状态的时候，相当于这块布上的一个截面，能够看到在这个状态下的一条曲线。

而actor的任务，就是用梯度上升的方法，寻着这条线的最高点。

对于actor来说，其实并不在乎Q值是否会被高估，他的任务只是不断做梯度上升，寻找这条最大的Q值。随着更新的进行Q1和Q2两个网络，将会变得越来越像。所以用Q1还是Q2，还是两者都用，对于actor的问题不大。

### Delayed - 延迟

这里说的Dalayed ，是actor更新的delay。也就是说相对于critic可以更新多次后，actor再进行更新。

为什么要这样做呢？

还是回到qnet拟合出来的那块"布"上。

qnet在学习过程中，的q值是不断变化的，也就是说这块布是不断变形的。所以要寻着最高点的任务有时候就挺难为为的actor了。

可以想象，本来是最高点的，当actor好不容易去到最高点；q值更新了，这并不是最高点。这时候actor只能转头再继续寻找新的最高点。更坏的情况可能是actor被困在次高点，没有找到正确的最高点。

所以可以把Critic的更新频率，调的比Actor要高一点。让critic更加确定，actor再行动。

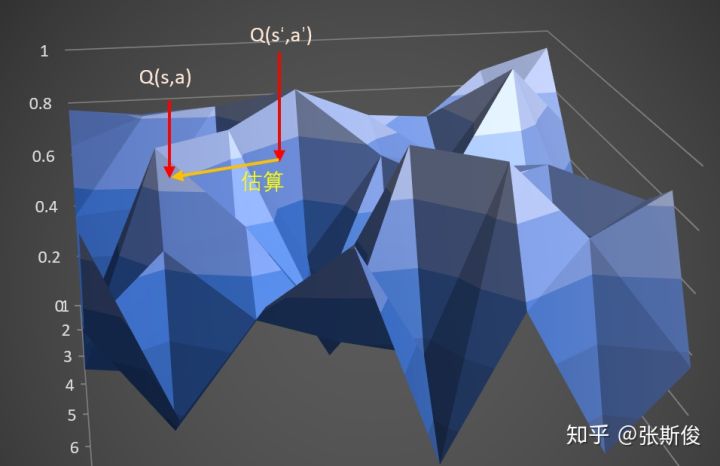
### target policy smoothing regularization

TD3中，价值函数的更新目标每次都在action上加一个小扰动，这个操作就是target policy smoothing regularization

为什么要这样呢？

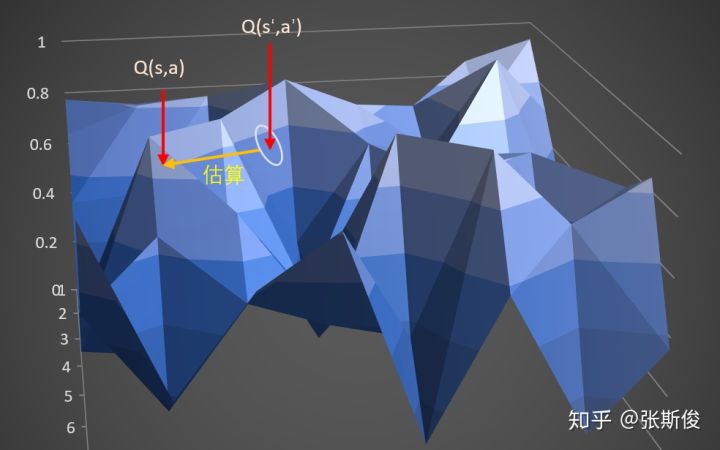
可以再次回到关于“布”的想象。

在DDPG中，计算target的时候，输入时s\_和a\_，获得q，也就是这块布上的一点A。通过估算target估算另外一点s，a，也就是布上的另外一点B的Q值。



在TD3中，计算target时候，输入s\_到actor输出a后，给a加上噪音，让a在一定范围内随机。这又什么好处呢。

好处就是，当更新多次的时候，就相当于用A点附近的一小部分范围（准确来说是在s\_这条线上的一定范围）的去估算B，这样可以让B点的估计更准确，更健壮。



这注意区分三个地方：

1. 在跑游戏的时候，同样加上了了noise。这个时候的noise是为了更充分地开发整个游戏空间。
2. 计算target的时候，actor加上noise，是为了预估更准确，网络更有健壮性。
3. 更新actor的时候，不需要加上noise，这里是希望actor能够寻着最大值。加上noise并没有任何意义。

### 示例代码说明

这一篇，以tensorflow给出的强化学习算法示例代码为例子，看看TD3应该如何实现。

[https://github.com/tensorlayer/tensorlayer/blob/master/examples/reinforcement\_learning/tutorial\_TD3.py​](https://link.zhihu.com/?target=https://github.com/tensorlayer/tensorlayer/blob/master/examples/reinforcement_learning/tutorial_TD3.py" \t "/home/admin/文档\\x/_blank)

[github.com](https://link.zhihu.com/?target=https://github.com/tensorlayer/tensorlayer/blob/master/examples/reinforcement_learning/tutorial_TD3.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\_TD3.py​github.com](https://link.zhihu.com/?target=https://github.com/louisnino/RLcode/blob/master/tutorial_TD3.py" \t "/home/admin/文档\\x/_blank)

以上三点，就是TD3对DDPG的优化。那么现在就跟随着示例代码，看一下这三点应该怎样实现把。

### target policy smoothing regularization

这部分的工作，主要是target\_policy\_net.evaluate函数负责。

和DDPG一样，该函数的主要功能是输入state，期望能够输出一个动作action，这个action能够获得最大的q值。在TD3中，函数允许输入eval\_noise\_scale，用于调节noise的大小。

函数分为两部分，一部分是输入state，输出action，这部分对于来说已经是常规操作，就不说了。来看另一部分，如何加上噪音。

normal = Normal(0, 1)

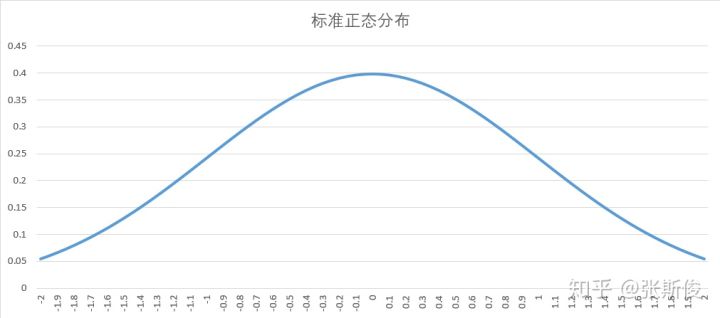
eval\_noise\_clip = 2 \* eval\_noise\_scale

noise = normal.sample(action.shape) \* eval\_noise\_scale

noise = tf.clip\_by\_value(noise, -eval\_noise\_clip, eval\_noise\_clip)

action = action + noise

这次是从标准正态分布中获取噪音，也就是说，正态分布的均值是0，标准差是1。长这个样子，随机出来值，大概率从[-2,2]之间。



按照动作输出的形状，从标准正态分布中抽样出noise。然后对noise进行大小缩放。这样当就可以动态调节的噪音大小。

但从图中能看到，仍然有很小的概率，抽样出很大或者很小的值的情况，为了避免这种情况的发生，对噪音大小进行截切。范围相当于两倍的eval\_noise\_scale

最后把噪音加上到action上并进行输出。

总的来说，这部分并没有很复杂的操作。前文已经说到，有三种情况需要用actor评估出action，但只有两种情况需要噪音，这种方法也满足了有时候需要噪音，有时候又不需要噪音的要求，不失为一种好方法。

### double network 的实现。

说过，在TD3中，用到了6个网络，



先对他们进行初始化，并把参数值赋值给target\_net。

# 用两个Qnet来估算，doubleDQN的想法。同时也有两个对应的target\_q\_net

self.q\_net1 = QNetwork(state\_dim, action\_dim, hidden\_dim)

self.q\_net2 = QNetwork(state\_dim, action\_dim, hidden\_dim)

self.target\_q\_net1 = QNetwork(state\_dim, action\_dim, hidden\_dim)

self.target\_q\_net2 = QNetwork(state\_dim, action\_dim, hidden\_dim)

self.policy\_net = PolicyNetwork(state\_dim, action\_dim, hidden\_dim, action\_range)

self.target\_policy\_net = PolicyNetwork(state\_dim, action\_dim, hidden\_dim, action\_range)

# 把net 赋值给target\_network

self.target\_q\_net1 = self.target\_ini(self.q\_net1, self.target\_q\_net1)

self.target\_q\_net2 = self.target\_ini(self.q\_net2, self.target\_q\_net2)

self.target\_policy\_net = self.target\_ini(self.policy\_net, self.target\_policy\_net)

然后回到updata函数。

1、从buffer里面，抽样出一个batch\_size的数据，作为更新的数据。

state, action, reward, next\_state, done = self.replay\_buffer.sample(batch\_size)

2、需要用target\_policy\_net评估new\_next\_action，用来求目标函数。

new\_next\_action = self.target\_policy\_net.evaluate(

next\_state, eval\_noise\_scale=eval\_noise\_scale

)

有两个小细节值得大家注意: 1、这里用的**target**\_policy\_net，不是policy\_net

2、这里需要添加noise。是target policy smoothing regularization应用的地方

3、把new\_next\_action 和 next\_state 分别放到两个network评估q值，并取相对较小的值。

target\_q\_input = tf.concat([next\_state, new\_next\_action], 1)

target\_q\_min = tf.minimum(self.target\_q\_net1(target\_q\_input), self.target\_q\_net2(target\_q\_input))

4、计算target：target\_q\_value = reward + (1 - done) *gamma* target\_q\_min

5、用target更新 q\_net1 和 q\_net2。

以上就是double network的实现过程，大家可以看到，TD3虽然比较复杂，但如果已经明白了DDPG的基本思想，TD3也是很简单的。

最后actor delay更简单。在更新actor更新前做一个判断，critic每更新一定次数(policy\_target\_update\_interval)，actor再进行更新。

if self.update\_cnt % self.policy\_target\_update\_interval == 0

剩下的又是常规操作了。注意两点： 1. 这里评估的new\_action是不需要加上noise的，因为加上是没任何意义的。 2. 在评估new\_action的q值时候，可以用一个q\_net，也可以用两个q\_net。大家可以试试有什么区别。

最后，把更新好的参数，软更新到target\_net去，就完成了

### 总结：

TD3对DDPG的优化，主要包括三个部分：

1. 用类似双Q网络的方式，解决了DDPG中Critic对高估动作Q值的问题；
2. 延迟actor更新，让actor的训练更加稳定；
3. 在target\_actor中加上噪音，增加算法稳定性。

下一篇，将会介绍AC的增强版本，A3C。下一篇见。