# 手把手教你实现Qlearning算法[实战篇]（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。

### Qlearning的基本思路回顾

在上一篇，我们了解了Qlearning和SARSA算法的基本思路和原理。

[张斯俊：[理论篇]怎样直观理解Qlearning算法？​](https://zhuanlan.zhihu.com/p/110338833" \t "/home/admin/文档\\x/_blank)

[](https://zhuanlan.zhihu.com/p/110338833)

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

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

### Qlearning算法流程

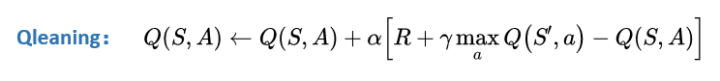
我们现在重新整理下，Qleanring的更新流程。 我们将会在任意的state出发

1. 我们将会用**noisy-greedy的策略**选定动作A

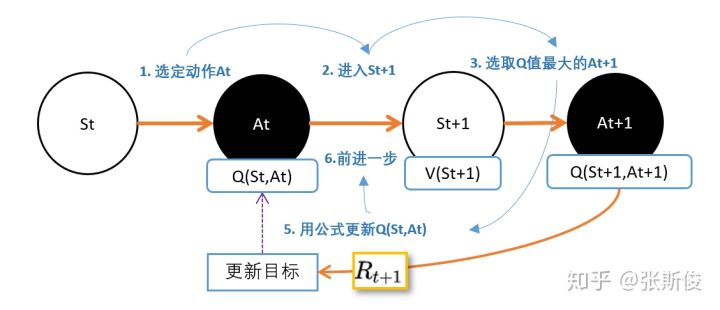
2. 在完成动作后，我们将会进入新状态St+1；

3. 检查St+1中所有动作，看看哪个动作的Q值最大；

4. 用以下的公式更新当前动作A的Q值；



5. 继续从s'出发，进行下一步更新 1-6步我们作为一个EP，进行N个EP的迭代。



如果你已经对之前理论篇比较熟悉，那么理解以上流程应该没有什么难度。

在具体实现的时候，有两个方式需要注意：

### Q-table

Q-table(Q表格) Qlearning算法非常适合用表格的方式进行存储和更新。所以一般我们会在开始时候，先创建一个Q-tabel，也就是Q值表。这个表纵坐标是状态，横坐标是在这个状态下的动作。



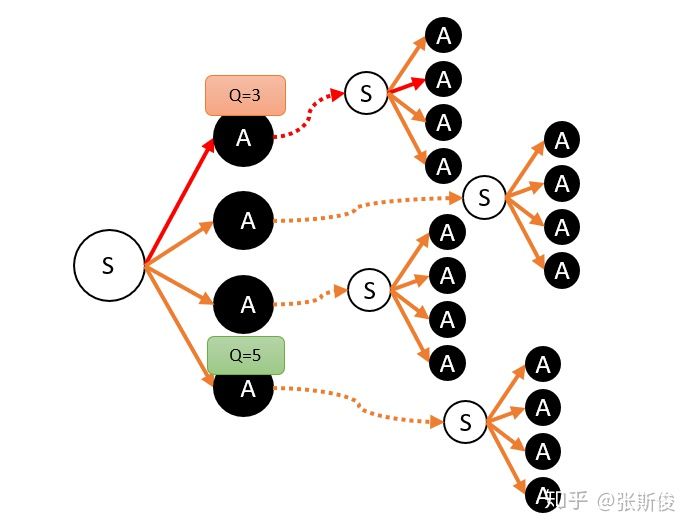
我们会初始化这个表的值为0。我们的任务就是，通过算法更新，把各个状态下的动作的Q值，填到上面去。

### noisy-greedy

我们解说过，在选择动作的时候，理论上每次都会使用当前状态下，Q值最大的动作。这样的选择方式，我们称为“贪婪”(greedy)。

因为我们只选择Q值最大的动作，所以有一些动作没被更新过没有被选择的过的动作，将更新不到。Q值也永远为0。

举个例子：



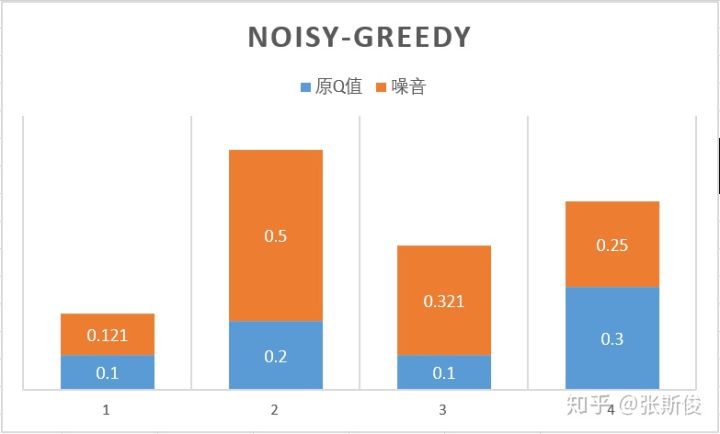
假设某次智能体经过路径(途中的红色线路)，根据Qlearning算法更新公式，我们计算得到某动作Q值为3。

由于其他动作还没执行过，因此他们保持初始值(一般为0)。按照贪婪算法，下一次智能体来到S的时候，会选择Q值最大的动作，也就是Q=3。于是红色路径再次被执行，Q值被更新。然后再一次，智能体仍然只会选红色线路。

但事实上，Q值最大的可能是其他的动作，但其他动作没有Q值，只是因为没有被“探索”出来。事实上我们会希望智能体在开始的时候更多随机行走去探索，而后面更多按照Q值去走动。

在我们的示例代码中，用了一个noisy-greedy的方法：

在每次选择动作的时候，就给我们要选择的动作叠加一个噪音。所谓噪音，就是在原来的值上增加一个随机值。



图中为某状态下，4个可选动作。 其中蓝色部分代表计算出来的Q值，也就是在Qtable上的值。 橙色部分是噪音，通过随机算法随机出来，每一次都会不同。

大家可以看到，原来蓝色部分Q值最大的是**动作4**(0.3)。但在加上噪音之后，Q值最大的是**动作2**(0.2+0.5)。所以最终智能体会选择**动作2**。

所以，我们可以通过噪音来“干扰”智能体的选择，达到让智能体有更多探索的机会。

【敲黑板】 注意！这些噪音只是在选择的时候，临时加上，每次都随机的。只干扰了当前选择，并不会影响真正的Q值。

当我们认为智能体对环境的了解已经足够充分，我们就可以慢慢减少噪音的大小。

在实做中，我们只需要在我们每次游戏后，将会减少产生噪音的方差，这样对干扰仍然有干扰，但这种干扰将会逐渐减少。直到相对于真正的Q值没有影响的程度。最终，agent将会按照自己的策略选择动作。

其实解决随机探索的问题还有好多种方法，也会在后续的文章中介绍给大家。

### Qlearning示例代码分析

在示例代码中，我们的环境是Gym的FrozenLake-v0。关于Gym和FrozenLake-v0的介绍，我们已经在另外一篇番外介绍。有需要的同学可以看一下。

[张斯俊：[番外]如何从师生关系理解环境与智能体的互动？​](https://zhuanlan.zhihu.com/p/110155777" \t "/home/admin/文档\\x/_blank)zhuanlan.zhihu.com[](https://zhuanlan.zhihu.com/p/110155777)

### 建立Qtable

格子世界的状态空间有限，动作空间有限，因此很适合 用Qlearning算法解决。

我们可以做这样的一个表格，我们称为Q值表，如下图所示：当我们预估到相应的Q值后，就直接填上这个表。

## Initialize table with all zeros

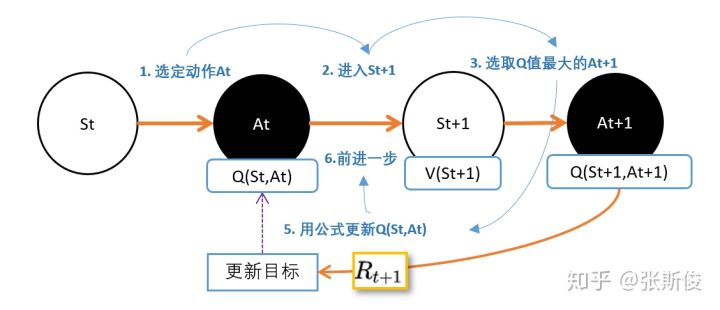
Q = np.zeros([env.observation\_space.n, env.action\_space.n])

np.zeros() 函数，用于生成一个全0矩阵。生成的形状由输入的参数决定。 示例代码中： env.observation\_space.n，表示这个环境中状态的数量。 env.action\_space.n，表示动作空间的数量。

生成出的Q-table我们可视化一下，长这样子：



### 算法主体



为了能更清晰地看出算法的主体，我把示例代码的一些细枝末节先删掉，算法主体大约就是一下几行。

for i in range(num\_episodes):

s = env.reset() #重置开始状态。

num\_episodes，表示进行游戏的次数。每次游戏开始，我们会用env.reset()函数把智能体设置在任意状态出发，一直走到最终状态。

for j in range(99):

....

....

if d ==True: #d for done。

break

我们设定进行一次游戏，最多进行99步。  
这个和环境状态多少有关系，当前环境状态16个，在99步前，一般都能到达最终状态。

a = np.argmax(Q[s, :] + np.random.randn(1, env.action\_space.n) \* (1. / (i + 1)))

我们之前说过noisy-greedy的原理，现在我们怎样用代码实现。就是上面这一行，也是Qlearning的重点。



这一行代码我们可以切开几个步骤来看一下：

* 首先，Q[s, :] 我们看一下table表的s列，就是我们当前的状态对应各个动作的Q值。
* 其次，np.random.randn(1, env.action\_space.n) 就是我们制造出来的噪音，我们希望噪音随着迭代的进行，将会越来越小。 因此我们乘以 (1. / (i + 1))。当i越来越大的时候，噪音就越来越小了。
* 最后，我们通过np.argmax()获得最大Q值对应的列号，也就是对应的动作。这里要注意，argmax找出最大值后，并不是返回**最大值**，而是返回**最大值的列号**，也就是**动作**。同学在这里要注意理解，我们需要的是动作A，而不是Q值。

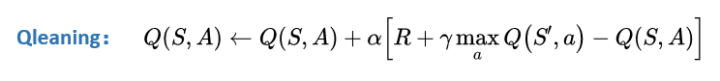
s1, r, d, \_ = env.step(a)

env.step() 我们把动作传入到环境中，环境会给我们返回4个返回值。

* new\_state: 示例代码用s1表示。这个表示我们执行动作后，新的状态。
* reward: 示例代码中用r表示，执行动作a后，获得的收获
* done：一个标志位，表示这个是否最终状态。
* \_ : 其实是info，但我们一般用不到这个值；因此我们把它先忽略。

Q[s, a] = Q[s, a] + lr (r + lambd np.max(Q[s1, :]) - Q[s, a])

我们用newstate的Q值，更新我们现在状态的Q值。我们对应更新公式，就很容易理解了。 注意比较：这里np.max和之前np.argmax函数的区别在于，np.max是返回最大值。而np.argmax返回时最大的行数或者列数。



s = s1

if d ==True:

break

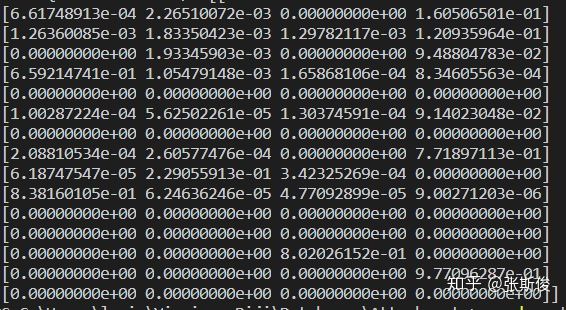
最后，我们更新Q值的任务已经完成，把游戏进行下去。把下一个状态s1赋值给s，重新开始新一步，和新一步的更新。

但在开始之前，我们检查一下，下一个状态是否就是终止状态了，如果是，这一次游戏就算是完成，开始一次迭代。

### 结果

训练过程 Episode [9994/10000] sum reward: 1.000000 running reward: 0.396657 took: 0.00100s Episode [9995/10000] sum reward: 0.000000 running reward: 0.392691 took: 0.00798s Episode [9996/10000] sum reward: 1.000000 running reward: 0.398764 took: 0.00105s Episode [9997/10000] sum reward: 1.000000 running reward: 0.404776 took: 0.00399s Episode [9998/10000] sum reward: 0.000000 running reward: 0.400729 took: 0.00794s Episode [9999/10000] sum reward: 1.000000 running reward: 0.406721 took: 0.00599s

我们把Qtable打印出来。



我们从结果上在理解一次Qtable。因为这和后面DQN等算法有很大关系，希望大家能能够理解。

* Qtable的每一行代表一个状态。
* 在每个状态下，智能体可以做4个动作，每个动作Q值分别对应是1,2,3,4列的数值。
* 智能体选择动作的方式是：

1. 找到代表当前状态S的行数
2. 在该行中找出最大的值，也就是最大Q值
3. 返回这个Q值的列号，这个列号就是动作

### Qlearning总结

Qlearning本质上是TD(0)算法，采用网格方式更新Qtable。

示例代码采noisy-greedy的方法。

但我们也看到Qlearning算法也有很大的局限性，我们看到，无论现实世界还是游戏世界，很多时候状态都是连续的，像表格这种方式，只能解决状态有限且离散的任务。

DQN算法应运而生！用深度网络，解决了连续状态的问题。

最后留给大家一个问题： 我们要从Qlearning改为Sarsa，那么应该怎么改呢？

===========你的支持，就是在下努力的原动力===========

如果专栏对你有用，请点赞并关注在下喔。如果发现有问题，也可以在文章下留言。

你的每一点关注，都是在下的继续努力的动力来源！感激！

[白话强化学习​](https://zhuanlan.zhihu.com/c_1215667894253830144" \t "/home/admin/文档\\x/_blank)zhuanlan.zhihu.com[](https://zhuanlan.zhihu.com/c_1215667894253830144)