# 如何理解策略梯度（Policy Gradient）算法？（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。

### 大坑

学到这里，先回顾一下，是怎样从0到DQN的。

一开始，对这个世界一无所知。发现每走一步，都有一个reward，希望能够获得更多的reward，所以在每个状态或者动作都做上记录，希望统计一下，从这个state出发，我能获得多少奖励。这样就可以知道应该走那条路。

于是，从动态规划到蒙地卡罗，到TD到Qleaning再到DQN，一路为计算Q值和V值绞尽脑汁。但大家有没有发现，可能走上一个固定的思维，就是强化学习，一定要计算Q值和V值，往死里算。但算Q值和V值并不是最终目的呀，要找一个策略，能获得最多的奖励。可以抛弃掉Q值和V值么？

答案是，可以，策略梯度(Policy Gradient)算法就是这样一个算法。

### 策略梯度(Policy Gradient)

如果说DQN是一个TD+神经网络的算法，那么PG是一个蒙地卡罗+神经网络的算法。

在神经网络出现之前，当遇到非常复杂的情况时，很难描述，遇到每一种状态应该如何应对。

但现在有了神经网络这么强大的武器，就可以用一个magic函数直接代替想要努力描述的规则。

用P表示策略，也就是动作的分布。那么期望有这么一个magic函数，当我输入state的时候，他能输出pi，告诉智能体这个状态，应该如何应对： pi = magic(state)。

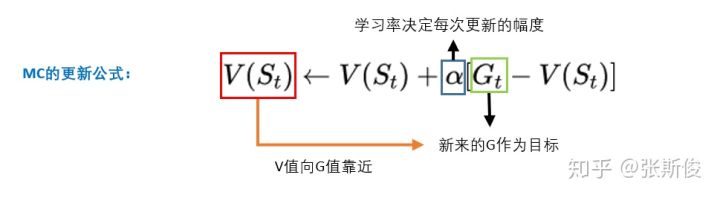
如果智能体的动作是对的，那么就让这个动作获得更多被选择的几率；相反，如果这个动作是错的，那么这个动作被选择的几率将会减少。

问题在于，怎么衡量对和错呢？PG的想法非常简单粗暴：蒙地卡罗的G值！

先来复习一下蒙地卡罗

从某个state出发，然后一直走，直到最终状态。然后从最终状态原路返回，对每个状态评估G值。

所以G值能够表示在策略pi下，智能体选择的这条路径的好坏。



如果你对上面说的关于MC的还有疑惑，可以详细看看这篇文章。

[张斯俊：如何用蒙地卡罗方法（Monte-Carlo）估算V值？​](https://zhuanlan.zhihu.com/p/109755443" \t "/home/admin/文档\\x/_blank)

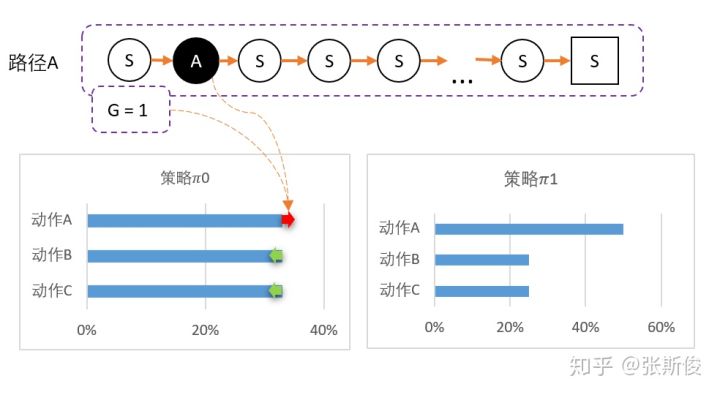
### 直观感受PG算法

先用数字，直观感受一下PG算法。

从某个state出发，可以采取三个动作。

假设当前智能体对这一无所知，那么，可能采取平均策略 Pi = [33%,33%,33%]。

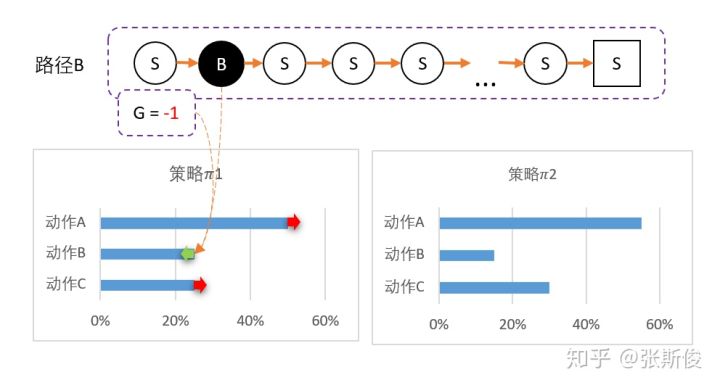
智能体出发，选择动作A，到达最终状态后开始回溯，计算得到 G = 1。



可以更新策略，因为该路径是**选择了A**而产生的，**并获得G = 1**；因此要更新策略：让A的概率提升，相对地，BC的概率就会降低。 计算得新策略为： Pi1 = [50%,25%,25%]

虽然B概率比较低，但仍然有可能被选中。第二轮刚好选中B。

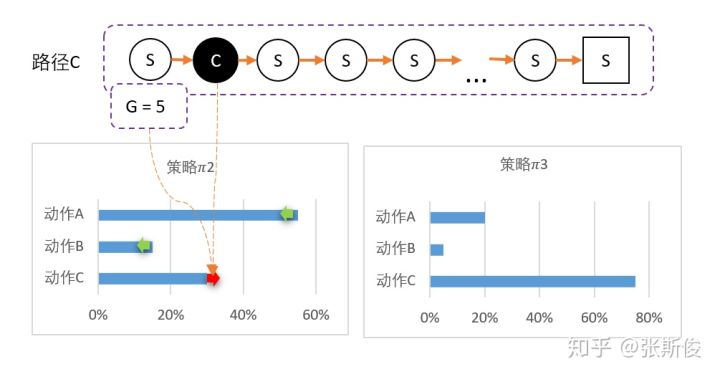
智能体选择了B，到达最终状态后回溯，计算得到 G = -1。



所以对B动作的评价比较低，并且希望以后会少点选择B，因此要降低B选择的概率，而相对地，AC的选择将会提高。

计算得新策略为： Pi2 = [55%,15%,30%]

最后随机到C，回溯计算后，计算得G = 5。



C比A还要多得多。因此这一次更新，C的概率需要大幅提升，相对地，AB概率降低。 Pi3 = [20%,5%,75%]

当然，以上的例子数值上是不严谨的和不精确的。具体怎样调整策略的概率分布，会在代码分析中更详细地说明。

### 示例代码分析

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

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

### 更新框架

for i\_episode in range(num\_episodes): #1

observation = env.reset()

while True:

action = RL.choose\_action(observation) #2

observation\_, reward, done, info = env.step(action) #3

RL.store\_transition(observation, action, reward) #4

if done:

vt = RL.learn() #5

1. 开始一场游戏，并重制state
2. 根据state，选择action
3. 把action代入环境，获得observation\_, reward, done, info
4. 记录数据
5. 计算G值，并开始学习策略。

把打印画图之类的代码先隐藏就能发现，PG的代码结构还是挺简单的。都是熟悉的配方，熟悉的味道。和DQN几乎一样。

但要注意两点：

1. 记录数据: 在DQN，记录data = (s,a,r,s\_,d) 五要素。并且记录在一个队列，需要用到的时候，从队列中随机抽取。 但PG中，记录的data = (s,a,r) 三要素就可以了，数据记录的先后顺序是不能打乱的。因为当要计算G值的时候，需要从后往前回溯计算。

2.清除数据: 这些记录是用完即弃，可以看到，在learn函数最后，智能体学习完数据后，就会清空列表。

### 计算G值

经过一次游戏，到结束状态，计算所有经过的state的G值。

示例代码中，函数\_discount\_and\_norm\_rewards() 就是计算G值，大家可以把函数中的discounted\_ep\_rs看成是G值。

函数分为两部分，一部分计算G值，一部分把G值进行归一化处理。

def \_discount\_and\_norm\_rewards(self):

# 计算G值

discounted\_ep\_rs = np.zeros\_like(self.ep\_rs)

running\_add = 0

for t in reversed(range(0, len(self.ep\_rs))):

running\_add = running\_add \* self.gamma + self.ep\_rs[t]

discounted\_ep\_rs[t] = running\_add

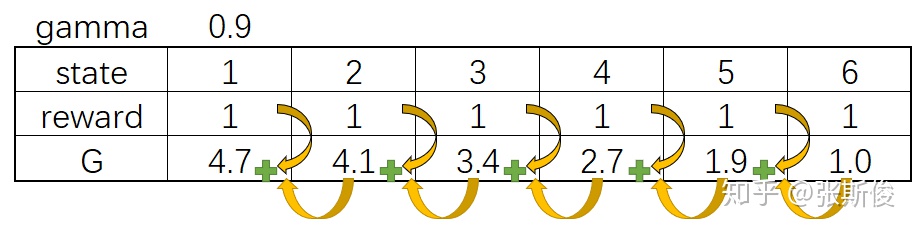
# 归一化处理

discounted\_ep\_rs -= np.mean(discounted\_ep\_rs)

discounted\_ep\_rs /= np.std(discounted\_ep\_rs)

return discounted\_ep\_rs

如图，假设经过6个state，到达最终状态，获得收获如下。



1. 首先创建一个全零向量，其大小和储存reward的列表相同。
2. 从后往前计算，在这个例子中，就是从state6开始。所以用了一个反向循环的方式来计算。
3. 每次循环，把上一个G值乘以折扣(gamma) ，然后加上这个state获得的reward即可。把这个值记录在discounted\_ep\_rs

discounted\_ep\_rs = np.zeros\_like(self.ep\_rs) #1

running\_add = 0

for t in reversed(range(0, len(self.ep\_rs))): #2

running\_add = running\_add \* self.gamma + self.ep\_rs[t] #3

discounted\_ep\_rs[t] = running\_add

可以用G值直接进行学习，但一般来说，对数据进行归一化处理后，训练效果会更好。只需要简单减去平均数，除以方差即可。

# 归一化处理

discounted\_ep\_rs -= np.mean(discounted\_ep\_rs)

discounted\_ep\_rs /= np.std(discounted\_ep\_rs)

### 带权重的梯度下降

def learn(self):

with tf.GradientTape() as tape:

\_logits = self.model(np.vstack(self.ep\_obs))

neg\_log\_prob = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(logits=\_logits, labels=np.array(self.ep\_as))

loss = tf.reduce\_mean(neg\_log\_prob \* discounted\_ep\_rs\_norm) # reward guided loss

grad = tape.gradient(loss, self.model.trainable\_weights)

self.optimizer.apply\_gradients(zip(grad, self.model.trainable\_weights))

经过之前的几个项目，相信大家都对梯度下降比较熟悉。在这里重点说一下，其中的loss函数的意义。

以某一个状态为例，在某个状态下，通过网络预测（\_logits），真实值（ep\_as），G值（discounted\_ep\_rs\_norm）如下图



可以把这个过程想象成一个分类任务。在训练的时候，只有动作2的真实值为1，其他为0。所以动作1,3,4的概率将会向0靠，也就是减少。而动作2的概率将会向1靠，也就是说会有所提升。

sparse\_softmax\_cross\_entropy\_with\_logits函数通过这个过程，不断调整概率分布。

那么G值的意义是什么呢？

可以看看loss = tf.reduce\_mean(neg\_log\_prob \* discounted\_ep\_rs\_norm) ，loss是根据G值进行调整的。当G值调整大小的时候，相当于每次训练幅度进行调整。例如G值为2，那么调整的幅度将会是1的两倍。



还记得我之前的例子吗？就是通过G值，让G值大的动作概率更大。



如果G值是一个负数呢，那么相当于进行反向的调整。如下图，如果G值为-1，那么说明选择动作2并不是一个“明智”的动作。于是让这个动作2的预测值降低，相当于“远离”真实值1。而其他动作的概率有所提升，相当于“远离”真实值0。

结合以上说的，代码就很容易明白了。

\_logits = self.model(np.vstack(self.ep\_obs))

neg\_log\_prob = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(logits=\_logits,

labels=np.array(self.ep\_as))

loss = tf.reduce\_mean(neg\_log\_prob \* discounted\_ep\_rs\_norm)

1. 通过网络，求出预测值pre的分布。
2. 和真实值action进行比较，求得neg\_log\_prob
3. 最终求得neg\_log\_prob乘以G值，求得loss

### 结论

PG用一个全新的思路解决了问题。但实际效果显得不太稳定，在某些环境下学习较为困难。另外由于采用了MC的方式，需要走到最终状态才能进行更新，而且只能进行一次更新，这也是PG算法的效率不高的原因。

后面将会尝试解决这个问题，既然用MC比较慢，那么用TD可以不？答案是，当然可以。下期再说。