

# A · I 笔记

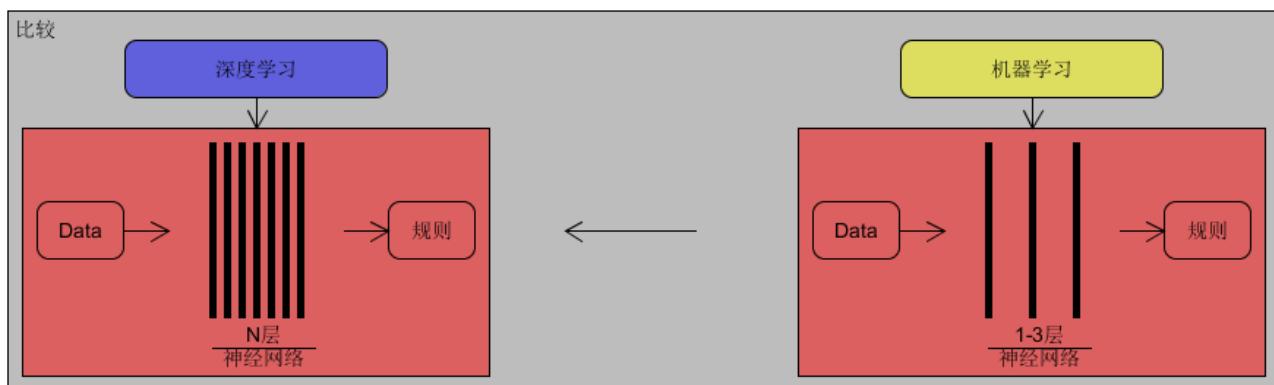
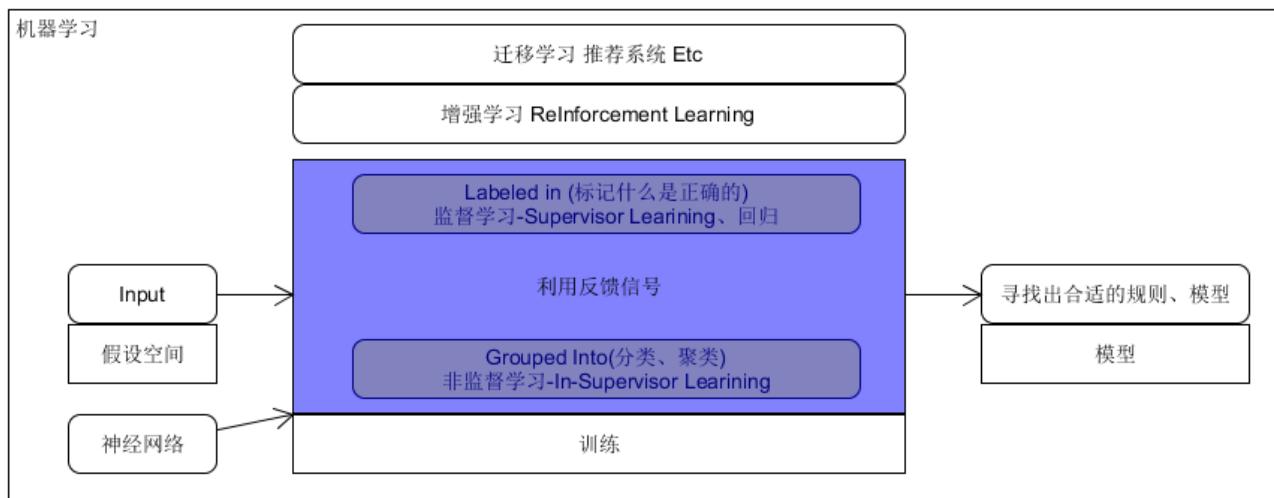
郑华

2020 年 1 月 16 日



# 第一章 基础概念

## 1.1 是什么



利用机器学习，人们输入的是数据和从这些数据中预期得到的答案，系统输出的是规则。这些规则随后可应用于新的数据，并使计算机自主生成答案。

机器学习的技术定义：在预先定义好的可能性空间中（数据不同表示-预处理），利用反馈信号的指引来寻找输入数据的有用表示。

机器学习从学习的种类一般分为3种：无监督学习、监督学习、强化学习。

监督学习：每一个样本都有明确的标签 (Right Answer)，最后总结出这些训练样本向量与标签的映射关系。

无监督学习：在没有标签的情况下尝试找出其内部蕴含关系的一种挖掘工作，常见的如分类、聚合。

强化学习：本质是解决 decision making 问题，即自动进行决策，并且可以做连续决策。

### 1.1.1 无监督学习

聚类 clustering

### 1.1.2 监督学习

分类 classifing

回归 regression 通过 70% 的数据训练出规则，通过 30% 剩下的数据进行回归测试拟合。

加入设计的线性关系类似于  $y = f(x) = wx + b$ ，则训练函数类似于

$$Loss = \sum_{i=1}^n |wx_i + b - y_i|$$

### 1.1.3 迁移学习

专注于存储已有问题的解决模型，并将其利用在其他不同但相关问题上。比如说，用来辨识汽车的知识（或者是模型）也可以被用来提升识别卡车的能力。

### 1.1.4 强化学习

<https://blog.csdn.net/j754379117/article/details/83037799>

<https://www.jianshu.com/p/5ceca53aff0b>

强调如何基于环境而行动，以取得最大化的预期利益。其灵感来源于心理学中的行为主义理论，即有机体如何在环境给予的奖励或惩罚的刺激下，逐步形成对刺激的预期，产生能获得最大利益的习惯性行为。

本质是解决 decision making 问题，即自动进行决策，并且可以做连续决策。

强化学习最早可以追溯到巴甫洛夫的条件反射实验，它从动物行为研究和优化控制两个领域独立发展，最终经 Bellman 之手将其抽象为马尔可夫决策过程 (Markov Decision Process, MDP).

它主要包含四个元素，*agent*，环境状态，行动，奖励，强化学习的目标就是获得最多的累计奖励。

让我们以小孩学习走路来做个形象的例子：

小孩想要走路，但在这之前，他需要先站起来，站起来之后还要保持平衡，接下来还要先迈出一条腿，是左腿还是右腿，迈出一步后还要迈出下一步。

小孩就是 **agent**, 他试图通过采取行动（即行走）来适应环境（行走的表面），并且从一个状态转变到另一个状态（即他走的每一步），当他完成任务的子任务（即走了几步）时，孩子得到奖励（给巧克力吃），并且当他不能走路时，就不会给巧克力。

**要素** 几大元素分别是：

- **Agent** , 输入通常是状态 State, 输出通常是策略 Policy
- **Action** , 就是从一点走到下一点 A -> B, C -> D, etc, 动作空间。比如小人玩游戏，只有上下左右可移动，那 Actions 就是上、下、左、右。
- **States** , 就是节点 A, B, C, D, E, F, 就是 Agent 的输入
- **Reward** , 就是边上的 cost, 进入某个状态时, 能带来正奖励或者负奖励。
- **Policy** , 就是完成任务的整条路径 A -> C -> F
- **Environment** , 接收 action, 返回 state 和 reward。

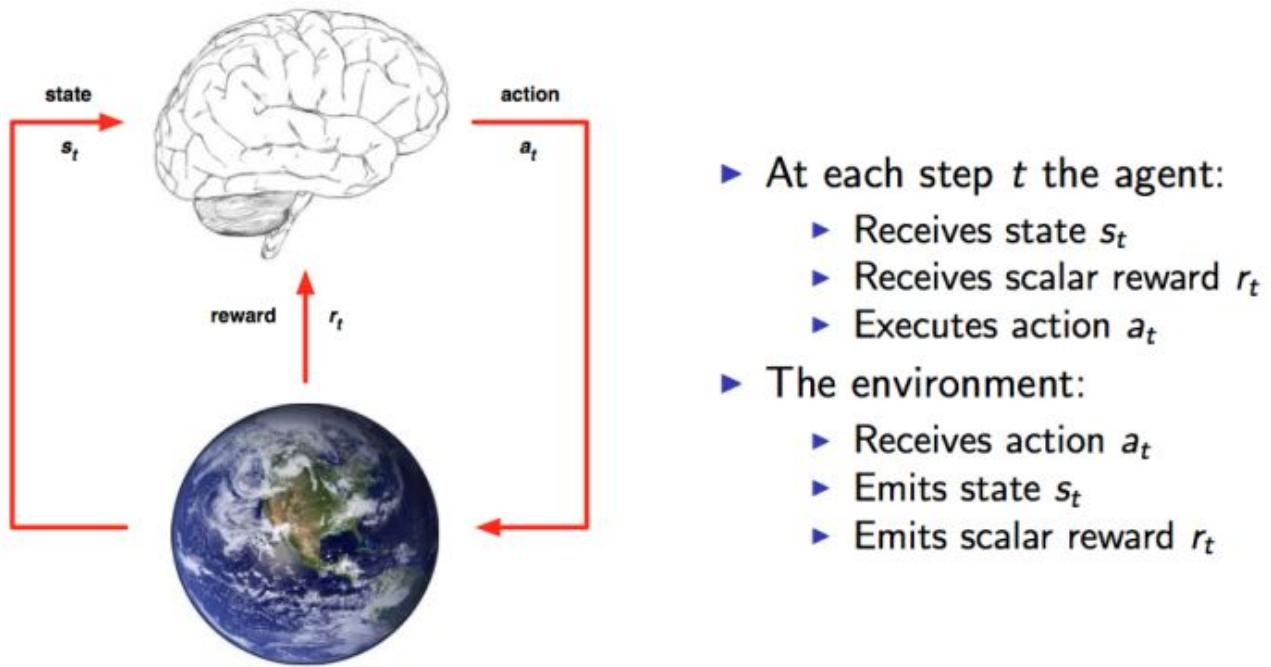


图 1.1: 强化学习示意

首先通过动作  $a_t$  与环境  $env$  进行交互，在动作  $a_t$  和环境  $env$  的作用下，Agent 会产生新的状态  $s_t$ ，同时环境会给出一个立即回报  $r_t$ 。

如此循环下去，智能体与环境进行不断地交互从而产生很多数据。强化学习算法利用产生的数据修改自身的动作策略，再与环境交互，产生新的数据，并利用新的数据进一步改善自身的行为，经过数次迭代学习后，智能体能最终地学到完成相应任务的最优动作（最优策略）。

**分类** 从强化学习的几个元素的角度划分的话，方法主要有下面几类：

- **Policy based**, 关注点是找到最优策略。

- **Value based**, 关注点是找到最优奖励总和。
- **Action based**, 关注点是每一步的最优行动。

**特点** 强化学习所解决的问题的特点:

- 智能体和环境之间不断进行交互
- 搜索和试错
- 延迟奖励 (当前所做的动作可能很多步之后才会产生相应的结果)

收敛条件或目标:

- 获取更多的累积奖励
- 获得更可靠的估计

**与其他机器学习的区别**

**和监督式学习的区别** 监督式学习就好比你在学习的时候，有一个导师在旁边指点，他知道怎么是对的怎么是错的。

强化学习会在没有任何标签的情况下，通过先尝试做出一些行为得到一个结果，通过这个结果是对还是错的反馈，调整之前的行为，就这样不断的调整，算法能够学习到在什么样的情况下选择什么样的行为可以得到最好的结果。

就好比你有一只还没有训练好的小狗，每当它把屋子弄乱后，就减少美味食物的数量（惩罚），每次表现不错时，就加倍美味食物的数量（奖励），那么小狗最终会学到一个知识，就是把客厅弄乱是不好的行为。

**两种学习方式都会学习出输入到输出的一个映射**，监督式学习出的是之间的关系，可以告诉算法什么样的输入对应着什么样的输出，强化学习出的是给机器的反馈 **reward function**，即用来判断这个行为是好是坏。

**强化学习的结果反馈有延时**，有时候可能需要走了很多步以后才知道以前的某一步的选择是好还是坏，而**监督学习**做了比较坏的选择会立刻反馈给算法。

**强化学习**面对的输入总是在变化，每当算法做出一个行为，它影响下一次决策的输入，而**监督学习**的输入是独立同分布的。

通过强化学习，一个 agent 可以在探索和开发 (**exploration and exploitation**) 之间做权衡，并且选择一个最大的回报。**exploration** 会尝试很多不同的事情，看它们是否比以前尝试过的更好。

**exploitation** 会尝试过去经验中最有效的行为。

一般的监督学习算法不考虑这种平衡，就只是是 **exploitative**。

**和非监督式学习的区别** 非监督式不是学习输入到输出的映射，而是**模式**。例如在向用户推荐新闻文章的任务中，非监督式会找到用户先前已经阅读过类似的文章并向他们推荐其一。

强化学习将通过向用户先推荐少量的新闻，并不断获得来自用户的反馈，最后构建用户可能会喜欢的文章的“知识图”。

**DQN:Deep-Q-Network** 深度强化学习全称是 Deep Reinforcement Learning (DRL)，其所带来的推理能力是智能的一个关键特征衡量，真正的让机器有了自我学习、自我思考的能力。

深度强化学习 (Deep Reinforcement Learning, DRL) 本质上属于采用神经网络作为值函数估计器的一类方法，其主要优势在于它能够利用深度神经网络对状态特征进行自动抽取，避免了人工定义状态特征带来的不准确性，使得 *Agent* 能够在更原始的状态上进行学习。

## 1.2 视频游戏的 AI 史

<https://www.gameres.com/853687.html>

### 1.2.1 Fine State Machine

### 1.2.2 Monte Carlo Search Tree

### 1.2.3 Behavioral Decision Trees



## 第二章 练习环境框架

2.1 Keras

2.2 Tensorflow

2.3 PyTorch



## 第三章 监督学习



## 第四章 非监督学习



# 第五章 迁移学习



# 第六章 强化学习-D · Silver

## 6.1 参考

学习途径汇总、Paper 链接: <https://www.cnblogs.com/charlotte77/p/10311255.html>

简书专栏: <https://www.jianshu.com/nb/22672858>

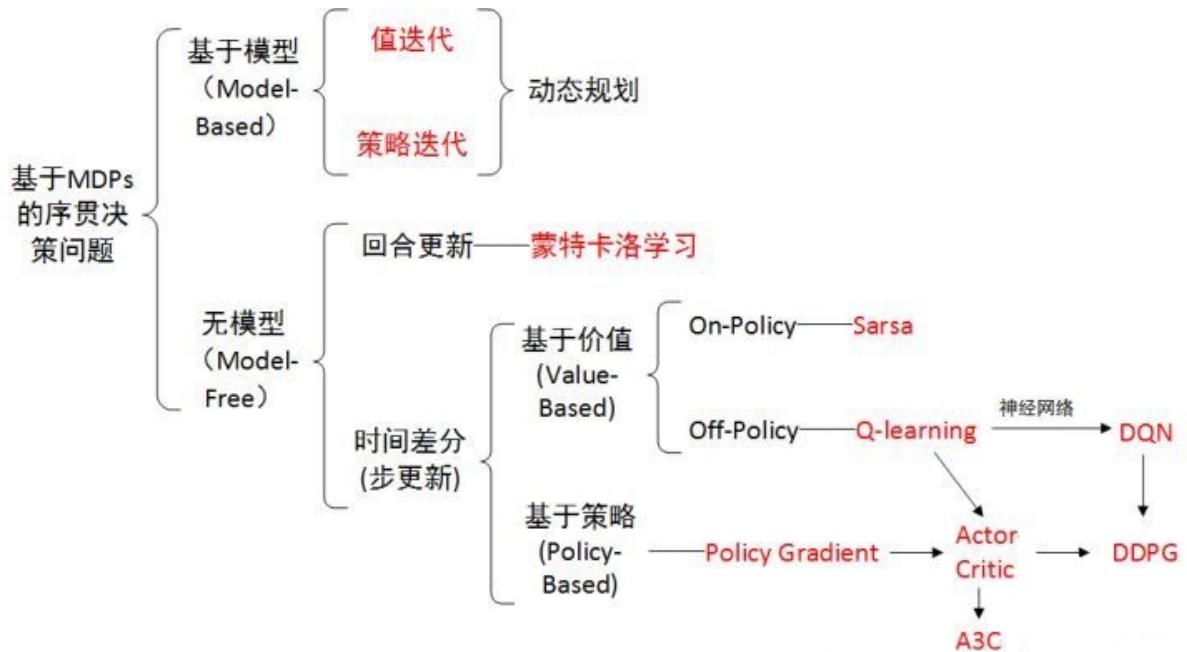
知乎专栏: <https://zhuanlan.zhihu.com/p/25498081>

博客专栏: <https://www.cnblogs.com/jinxulin/tag/>

D · Silver 笔记专栏: <https://zhuanlan.zhihu.com/p/50478310>

Towards Data: <https://towardsdatascience.com/understanding-markov-decision-processes-b5>

通过例子了解强化学习: [http://www.sohu.com/a/228536039\\_129720](http://www.sohu.com/a/228536039_129720) +  
<https://www.freecodecamp.org/news/diving-deeper-into-reinforcement-learning-with-q-learning/>

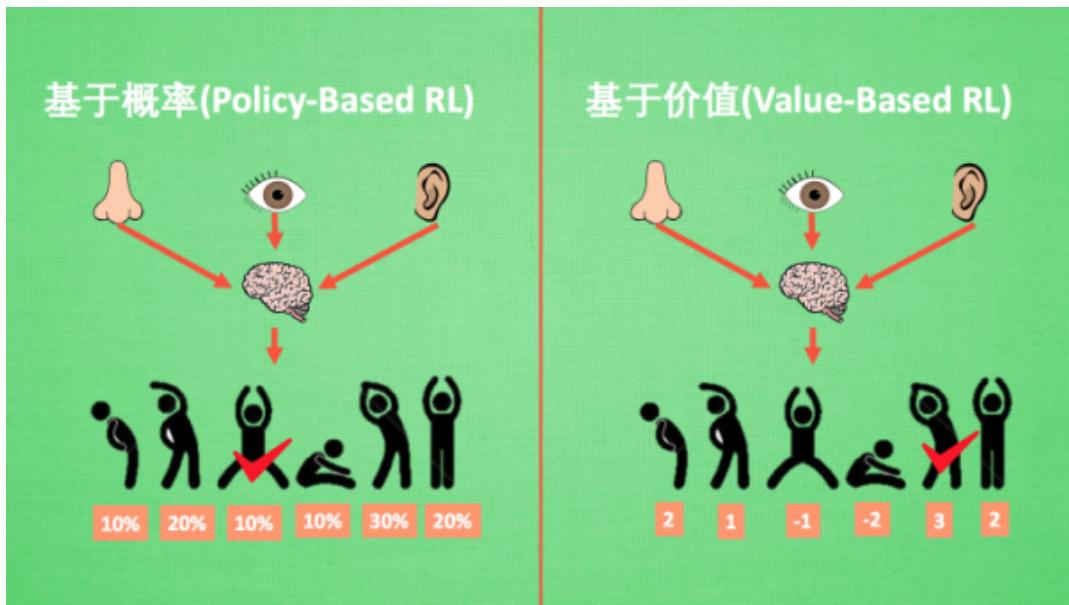


## 6.2 基本概念

### 6.2.1 ModelFree 与 ModelBased

model-free 是指 agent 对环境不了解， model-based 指 agent 对环境了解。

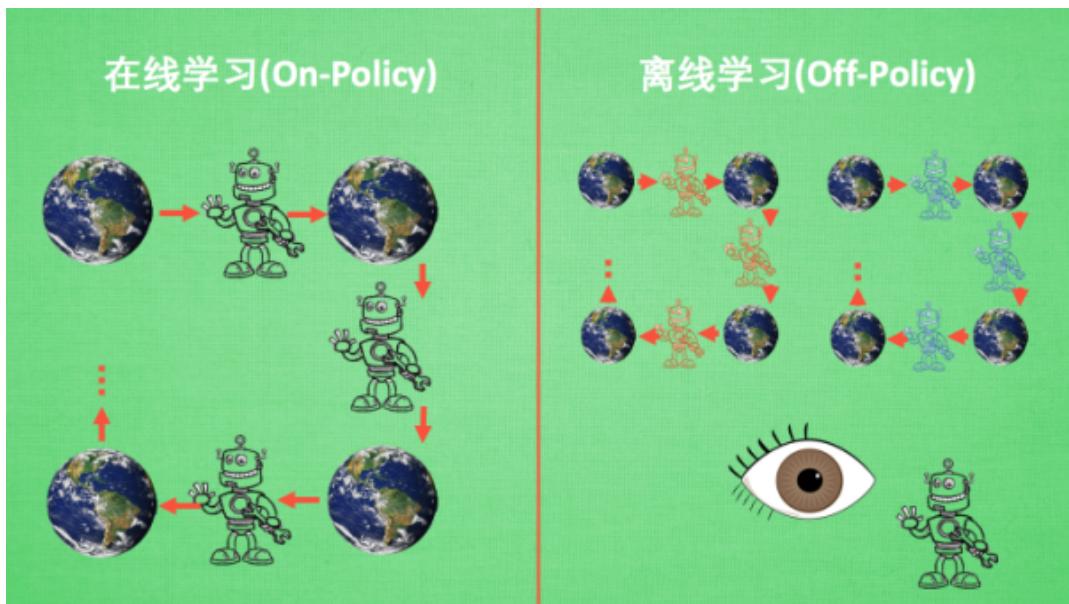
## 6.2.2 Policy-Based 与 Value-Based



基于概率的话，有几率选到概率比较小的 action. 基于价值的话，永远选 value 最大的动作。另外基于价值的无法在连续动作过程中实现。

在基于概率这边，有 Policy Gradients，在基于价值这边有 Q learning, Sarsa 等. 而且我们还能结合这两类方法的优势之处，创造更牛逼的一种方法，叫做 Actor-Critic, actor 会基于概率做出动作，而 critic 会对做出的动作给出动作的价值，这样就在原有的 policy gradients 上加速了学习过程.

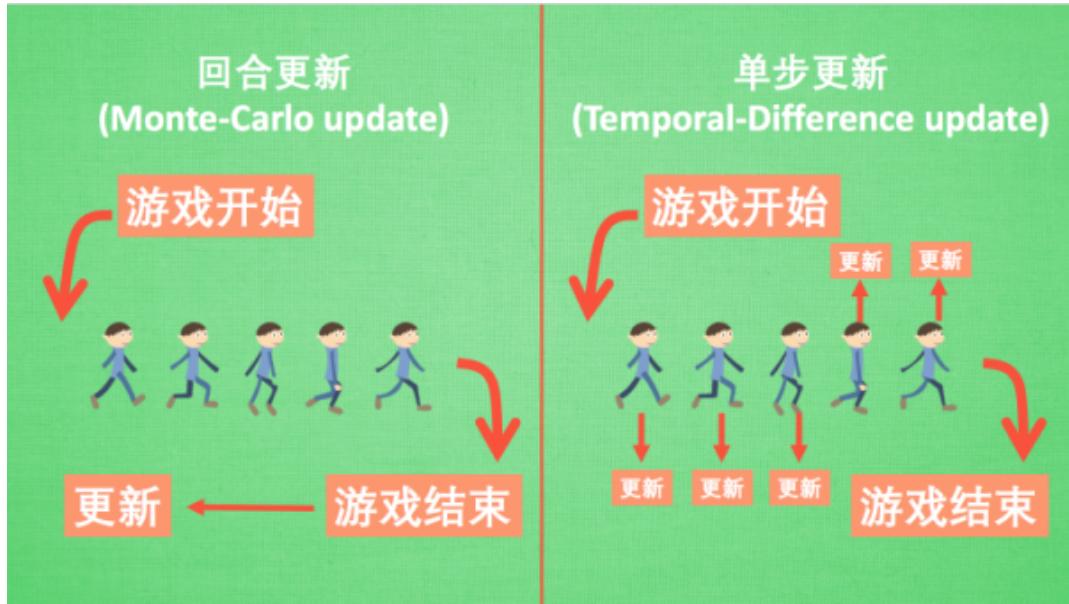
## 6.2.3 OnPolicy 与 OffPolicy



在线学习，就是指我必须本人在场，并且一定是本人边玩边学习。而离线学习是你可以选择自己玩，也可以选择看着别人玩，通过看别人玩来学习别人的行为准则。

在线学习有 Sarsa, Sarsa lambda, 最典型的离线学习就是 Q learning, Deep-Q-Network.

#### 6.2.4 单步更新与回合更新



回合更新指的是游戏开始后，我们要等待游戏结束，然后再总结这一回合中的所有转折点，再更新我们的行为准则。而单步更新则是在游戏进行中每一步都在更新，不用等待游戏的结束，这样我们就能边玩边学习了。

再来说说方法，Monte-carlo learning 和基础版的 policy gradients 等都是回合更新制，Qlearning, Sarsa, 升级版的 policy gradients 等都是单步更新制。因为单步更新更有效率，所以现在大多方法都是基于单步更新。

#### 6.2.5 强化学习的解法的通用框架

有限马尔科夫状态（finite MDP）包含有限的状态集和动作集。大多数强化学习的理论或算法都需要马尔科夫过程的前提限制，但是其思想具有广泛通用性。

### 6.3 MDP-马尔可夫决策

#### 6.3.1 马尔科夫性质

马尔科夫性是指系统的下一个状态  $s_{t+1}$  仅与当前状态  $s_t$  有关，而与以前的状态无关。

定义：状态  $s_t$  是马尔科夫的，当且仅当  $P[S_{t+1}|s_t] = P[s_{t+1}|s_1, \dots, s_t]$ 。

定义中可以看到，当前状态  $s_t$  其实是蕴含了所有相关的历史信息  $s_1, \dots, s_t$ ，一旦当前状态已知，历史信息将会被抛弃。

#### 6.3.2 马尔科夫过程

马尔科夫性描述的是每个状态的性质，但真正有用的是如何描述一个状态序列。数学中用来描述随机变量序列的学科叫随机过程。所谓随机过程就是指随机变量序列。

若随机变量序列中的每个状态都是马尔科夫的则称此随机过程为马尔科夫随机过程。

马尔科夫过程的定义：马尔科夫过程是一个二元组  $(S, P)$ ，且满足： $S$  是有限状态集合， $P$  是状态转移概率。状态转移概率矩阵为：

$$P = \begin{bmatrix} P_{11} & \cdots & P_{1n} \\ \vdots & \ddots & \vdots \\ P_{n1} & \cdots & P_{nn} \end{bmatrix}$$

**示例** 一个学生的 7 种状态娱乐，课程 1，课程 2，课程 3，考过，睡觉，论文，每种状态之间有一定的转换概率。具体如下图所示。

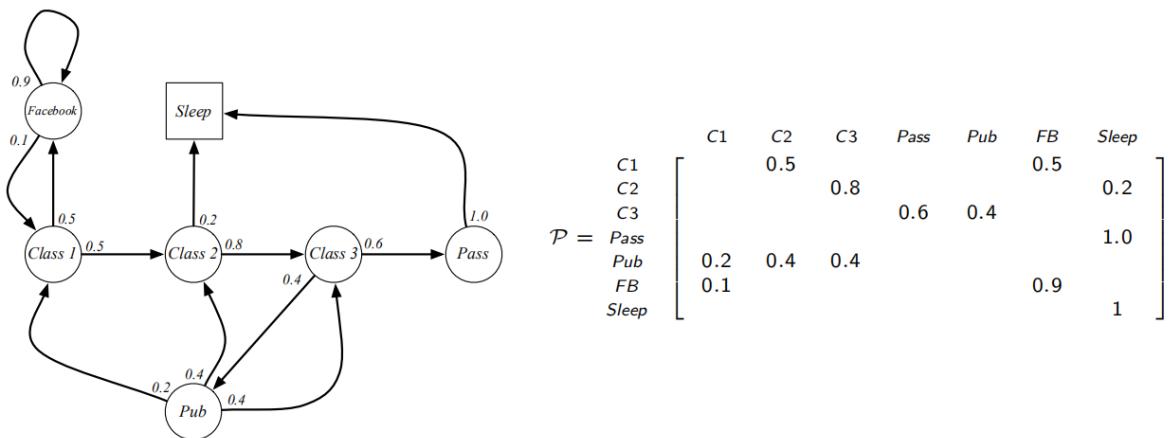


图 6.1: 马尔可夫决策示例

以上状态序列称为马尔科夫链。当给定状态转移概率时，从某个状态出发存在多条马尔科夫链。对于游戏或者机器人，马尔科夫过程不足以描述其特点，因为不管是游戏还是机器人，他们都是通过动作与环境进行交互，并从环境中获得奖励，而马尔科夫过程中不存在动作和奖励。将动作（策略）和回报考虑在内的马尔科夫过程称为马尔科夫决策过程。

### 6.3.3 马尔科夫决策过程

**基本组成** 基本组成：五元组  $M = (S, A, P, \gamma, R)$ .

- **S:** 表示状态集 (states)，有  $s \in S$ ， $s_i$  表示第  $i$  步的状态。
- **A:** 表示一组动作 (actions)，有  $a \in A$ ， $a_i$  表示第  $i$  步的动作，由状态与策略函数  $\pi$  决定。
- **P:** 表示状态转移概率。 $s$  表示的是在当前  $s \in S$  状态下，经过  $a \in A$  作用后，会转移到的其他状态的概率分布情况。比如，在状态  $s$  下执行动作  $a$ ，转移到  $s'$  的概率可以表示为  $p(s'|s, a)$ 。
- **R:**  $S \times A \rightarrow \mathbb{R}$ ， $R$  是回报函数 (reward function)。有些回报函数状态  $S$  的函数，可以简化为  $R : S \rightarrow \mathbb{R}$ 。如果一组  $(s, a)$  转移到了下个状态  $s'$ ，那么回报函数可记为  $r(s'|s, a)$ 。如果  $(s, a)$  对应的下个状态  $s'$  是唯一的，那么回报函数也可以记为  $r(s, a)$ 。

回报函数可以看成是一个映射，关于当前的动作，或者当前环境和当前动作的 pair 的好不好的一个评价。属于立即评价，只考虑当前这一步的好坏。

- $\gamma$ : 折现因子: 对未来的价值抱有的希望参数, 取值在  $[0, 1]$

**状态转移策略**  $\pi(s)$  policy – 在当前环境的基础上, 引入动作 Action, 并将环境和动作之间添加一种映射, 某种环境下最应该做什么动作呢? 这个是由 policy 决定的。policy 的所有可能组成一个 policy 空间, 强化学习的目的, 就是在这个巨大的空间中, 学习到某一种最优的 policy。

马尔科夫决策过程的状态转移概率是包含动作的:

$$P_{ss1}^a = P[S_{t+1} = s1 | S_t = s, A_t = a]$$

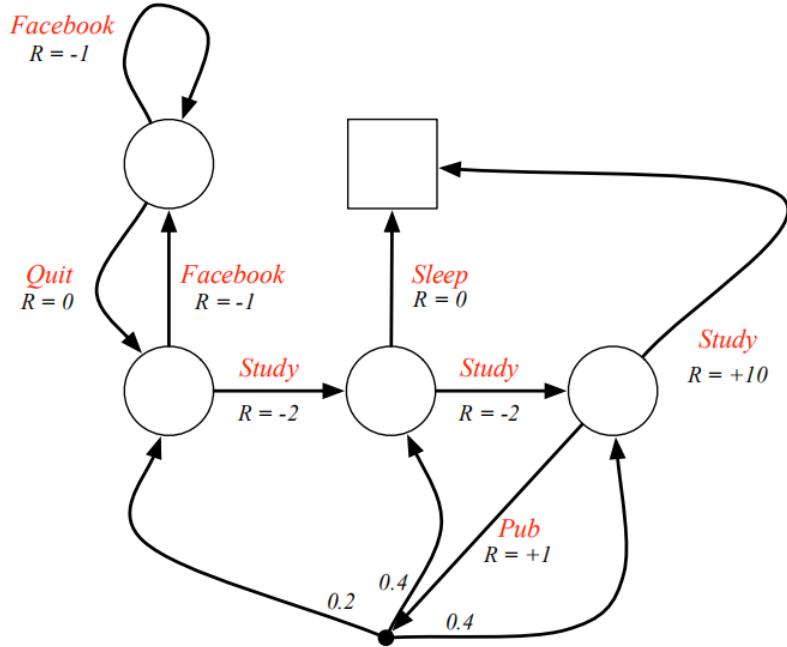


图 6.2: 马尔可夫决策过程示例

学生有五个状态, 状态集为  $S = s_1, s_2, s_3, s_4, s_5$ , 动作集为  $A = \text{玩 退出 学习 发论文 睡觉}$ , 在上图中  $R$  表示立即回报。

强化学习的目标是给定一个马尔科夫决策过程, 寻找最优策略。所谓策略是指状态到动作的映射, 策略常用符号  $\pi$  表示, 它是指给定状态  $s$  时, 动作集上的一个分布, 即

$$\pi(a|s) = p[A_t = a_t | S_t = s] \quad (6.1)$$

策略的定义是用条件概率分布给出的。策略  $\pi$  在每个状态  $s$  指定一个动作概率。如果给出的策略  $\pi$  是确定性的, 那么策略  $\pi$  在每个状态  $s$  指定一个确定的动作。

例如其中一个学生的策略为  $\pi_1(\text{玩}|s_1) = 0.8$ , 是指该学生在状态  $s_1$  时玩的概率为 0.8, 不玩的概率是 0.2, 显然这个学生更喜欢玩。

另外一个学生的策略为  $\pi_2(\text{玩}|s_1) = 0.3$ , 是指该学生在状态 [公式] 时玩的概率是 0.3, 显然这个学生不爱玩。依此类推, 每学生都有自己的策略。

**累积回报**  $G_t(s)$  和上面的 reward function 对比着看, 这一步考虑的是当前环境状态的长远优势, 也就是以当前状态为起点, 以后的多个时间点之后的各个状态的 reward 之和。如何更好的

估计这个值，是几乎所有增强学习问题的解决重点和难点。这个也是如何评定一个 policy 好坏的标准。

强化学习是找到最优的策略，这里的最优是指得到的总回报最大。

当给定一个策略  $\pi$  时，就可以计算累积回报了。首先定义累积回报：

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (6.2)$$

注意：这里某个时间点上的  $G_t$ ，也就是期望返回值，是从该时间点往后所有获取的 reward 序列和。

discounting rate  $\gamma \in [0, 1]$ ：决定了未来奖励的当前价值。

discounting rate 的含义就是，在当前时间点  $k$  步之后的奖励，在当前的价值是未来的  $\gamma^{k-1}$  倍。当该系数小于 1，最终会收敛，只要奖励序列是有限的。当系数为 0，则执行机构是短视的，只顾眼前利益；如果系数大于 1，执行机构是远视的，也就是重视长远奖励。

在策略  $\pi$  下，可以计算累积回报  $G_1$ ，此时  $G_1$  有多个可能值。由于策略  $\pi$  是随机的，因此累积回报也是随机的。为了评价状态  $s_1$  的价值，我们需要定义一个确定量来描述状态  $s_1$  的价值，很自然的想法是利用累积回报来衡量状态  $s_1$  的价值。然而，累积回报  $G_1$  是个随机变量，不是一个确定值，因此无法进行描述。但其期望是个确定值，可以作为状态值函数的定义。

**状态-价值函数  $V_t(s)$**  累积回报  $G$  是个随机变量，不是一个确定值，因此无法评价状态  $s$  的价值。但其期望是个确定值（状态值  $v_\pi$ ），可以作为评价依据。

当 Agent 采用策略  $\pi$  时，累积回报服从一个分布，累积回报在状态  $s$  处的期望值定义为状态-值函数：

$$\begin{aligned} v_\pi(s) &= E_\pi \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right] \\ &= R_s + \gamma \sum_{s' \in S} P_{ss'} v_\pi(s') \\ &= \text{立即回报} + \text{未来回报} \end{aligned} \quad (6.3)$$

--> 状态值函数是与策略  $\pi$  相对应的，这是因为策略  $\pi$  决定了累积回报  $G$  的状态分布。

**动作：价值函数  $q(a)$**  把每个 action 的真实 value 定义为  $q_*(a)$ ，把每个 action 在第  $t$  个时间步下的估计值定义为  $q_t(a)$ 。

每个 action 的真实 value，是当该动作被选择时所获得的期望 reward，在这里，自然想到用最简洁的历史平均 reward 来表示当前动作的 value 估计值。假定某个 action 在  $t$  时间前总共被选择了  $k_\alpha$  次，于是我们的估计值如下式：

$$q_t(a) = \frac{R_1 + R_2 + \cdots + R_{k_\alpha}}{t}$$

这个方法叫做 sample-average，因为每个动作的 estimated value 都是依据过往 sample reward 的平均值进行计算的。当  $k_\alpha$  等于 0，我们可以把定义为一个默认的初始值，当  $k_\alpha$  趋近无穷，则最终收敛于  $q_*(a)$ 。当然这个 estimate 的方法不一定是最好的，只是为了说明问题而已。

可以发现，为了算在  $t$  时间的 action  $a$  的 value estimation，我们需要  $t$  时间之前所有时间该动作的 reward 记录。随着时间增加，这种计算方式涉及的计算复杂度和存储压力都会增加。

其实这是不必要的，我们完全可以用另一种方式来计算  $q_t(a)$ 。推导如下：

$$\begin{aligned}
Q_{n+1} &= \frac{1}{n} \sum_{i=1}^n R_i \\
&= \frac{1}{n} (R_n + \sum_{i=1}^{n-1} R_i) \\
&= \frac{1}{n} (R_n + (n-1) \frac{1}{n-1} \sum_{i=1}^{n-1} R_i) \\
&= \frac{1}{n} (R_n + (n-1) Q_n) \\
&= \frac{1}{n} (R_n + n Q_n - Q_n) \\
&= Q_n + \frac{1}{n} (R_n - Q_n)
\end{aligned}$$

用这个形式的迭代公式，只需要保存  $q_k(a)$  和  $k$ ，再加上一点简单的加减运算，就可以快速得到  $q_{k+1}(a)$ ，这个形式的写法贯穿于全文。

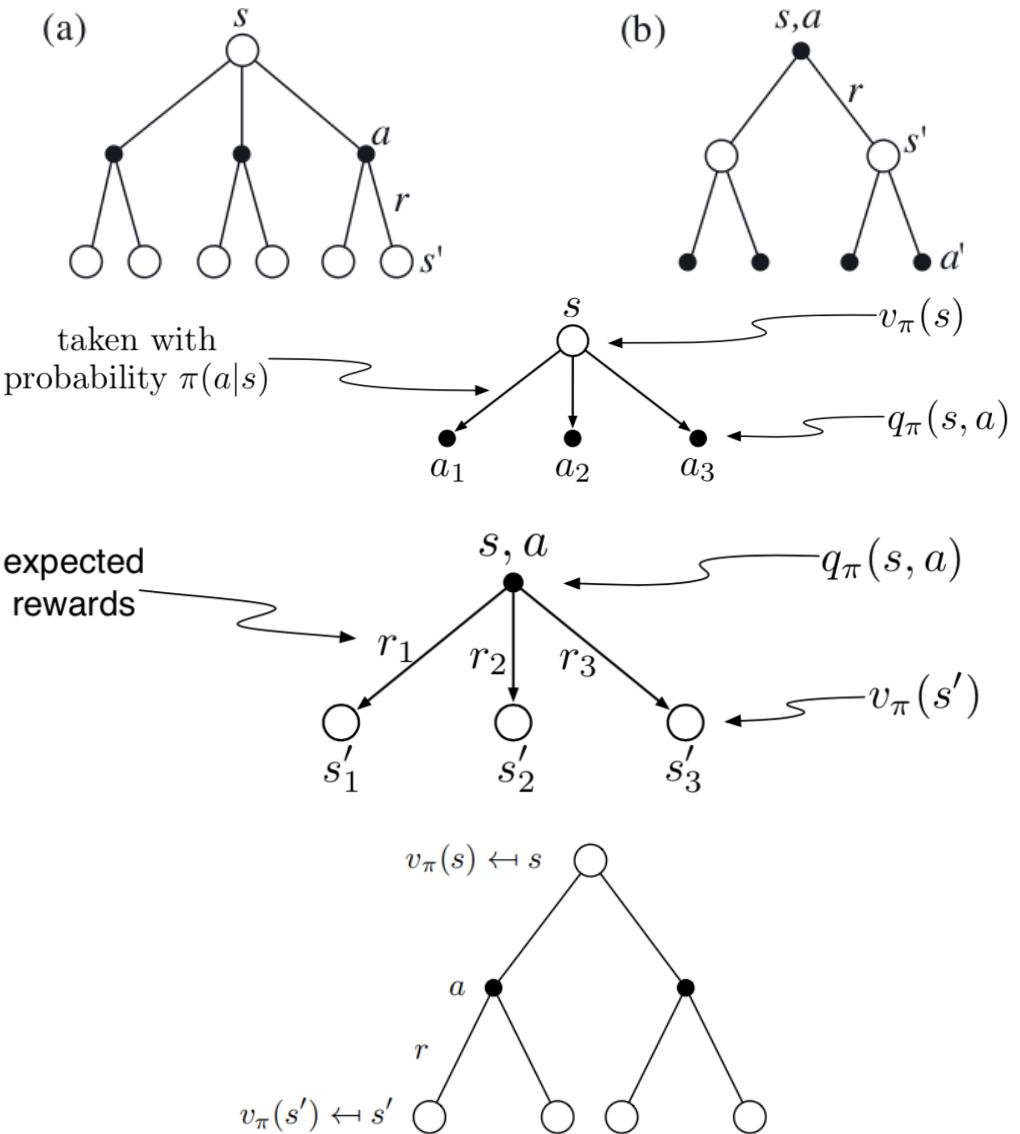
$$newEstimate \leftarrow oldEstimate + stepSize(target - oldEstimate)$$

$target - oldEstimate$  是  $estimate$  的 error，乘以 stepsize 是不断地减小这个 error，逼近 target。

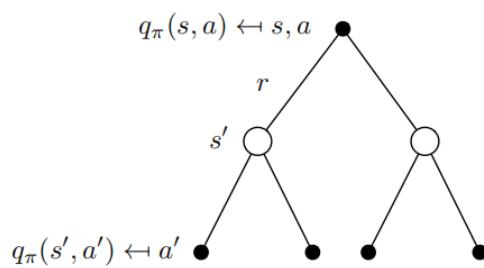
状态-行为：价值函数  $q_\pi(s, a)$  相应地，状态-行为 价值函数为：

$$\begin{aligned}
q_\pi(s, a) &= E_\pi \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right] \\
&= R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a'|s') q_\pi(s', a')
\end{aligned} \tag{6.4}$$

因为引入了动作策略，因而在每一个状态下策略  $\pi$  都会有该状态下会发生动作的概率分布空间，而每种动作都可能导致不同的后续状态。具体关联如下所示：



$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left( \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right)$$



$$q_\pi(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s', a')$$

图 6.3: 状态与动作的关联示例

其中空心表示状态，实心表示某状态下可能产生的动作分布。

从状态  $s$  开始，顶部的根节点，个体可以采取基于其策略  $\pi$  的任何一组动作，图中显示了三个。这些动作中的每一个，环境可以响应下一个状态中的其中一个， $s'$ （图中显示两个），以及奖励  $r$ ，取决于函数  $p$  给出的动态。

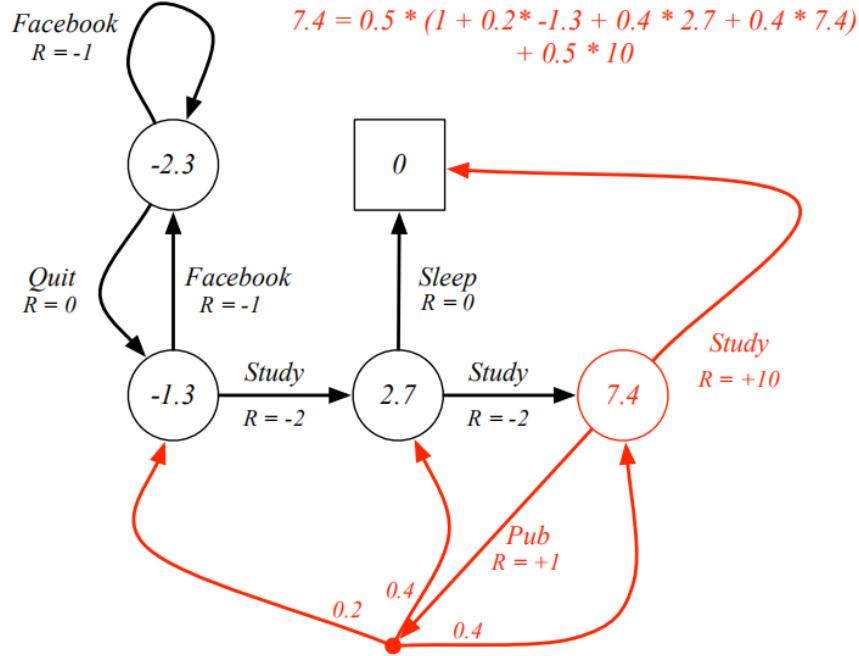


图 6.4: 状态值函数示例

图中白色圆圈中的数值为该状态下的值函数。即： $v_\pi(s_1) = -2.3, v_\pi(s_2) = -1.3, v_\pi(s_3) = 2.7, v_\pi(s_4) = 7.4, v_\pi(s_5) = 0$

-->Notice: 图中，除了 pub 这个动作外，其余各状态也可以看成是执行了某动作，但是其通往某状态的概率为 1. 既  $s \rightarrow a \rightarrow s'$  过程中， $a \rightarrow s'$  的概率始终为 1，所以不存在其他分支，因而没有画中间类似于 pub 的动作与状态的关联图。

**状态: 价值函数与状态-行为: 价值函数的贝尔曼方程** 贝尔曼方程对上述图6.3中  $s \rightarrow s'$  所有可能性进行平均，通过其发生概率对每个可能性进行加权。它指出，开始状态的值必须等于预期的下一个状态的（衰减）值，加上沿途预期的奖励。

由状态值函数的定义式可以得到：

$$\begin{aligned}
 v(s) &= E[G_t | S_t = s] \\
 &= E[R_{t+1} + \gamma R_{t+2} + \dots | S_t = s] \\
 &= E[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) | S_t = s] \\
 &= E[R_{t+1} + \gamma G(t+1) | S_t = s] \\
 &= E[R_{t+1} + \gamma E_{s_{t+1}, \dots}(G(S_{t+1}))] \\
 &= E[R_{t+1} + \gamma v(S_{t+1}) | S_t = s] \\
 &= \text{立即回报} + \text{未来回报}
 \end{aligned} \tag{6.5}$$

需要注意的是对哪些变量求期望。

同样我们可以得到状态-动作值函数的贝尔曼方程：

$$q_\pi(s, a) = E_\pi[R_{t+1} + \gamma q(S_{t+1}, A_{t+1})|S_t = s, A_t = a] \quad (6.6)$$

**最优：价值函数** 可以通过以下方式精确地定义一个最优策略。价值函数对策略进行部分排序。如果策略  $\pi$  所有状态的预期返回值大于或等于策略  $\pi'$  的值，则该策略  $\pi$  被定义为优于或等于策略  $\pi'$ 。换句话说，对所有  $s \in S$ ，当且仅当  $v_\pi(s) \geq v_{\pi'}(s)$  时， $\pi \geq \pi'$  成立。总是至少有一个策略优于或等于所有其他策略。这个策略称为最优策略。虽然可能有不止一个，我们用  $\pi_*$  表示所有最优策略。它们共享同样的状态值函数，称为最优状态价值函数，表示为  $v_*$ ，并定义为

$$v_*(s) \doteq \max_\pi v_\pi(s)$$

**最优动作：价值函数** 最优策略还具有相同的最优动作价值函数，表示为  $q_*$ ，并定义为

$$q_*(s, a) \doteq \max_\pi q_\pi(s, a)$$

对所有  $s \in S$  和  $a \in A(s)$ 。对于状态一动作对  $(s, a)$ ，此函数给出在状态  $s$  中执行动作  $a$  并且此后遵循最优策略的预期返回值。因此，我们可以用  $v_*$  来表示  $q_*$ ，如下所示：

$$q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a]$$

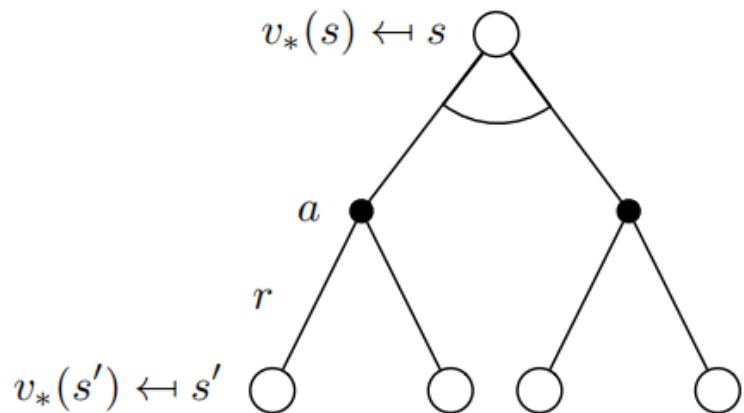
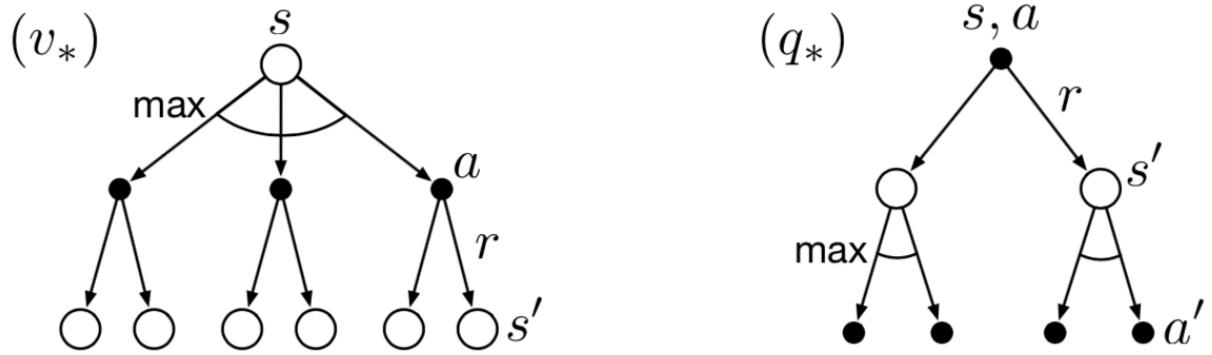
**最优 BellMan 方程** 贝尔曼最优方程式表达了这样一个事实，即最优策略下的状态价值必须等于来自该状态的最佳行动的预期收益：

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s, a) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t|S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1}|S_t = s, A_t = a] \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a] \\ &= \max_{a \in \mathcal{A}(s)} \sum_{s', r} p(s', r|s, a)[r + \gamma v_*(s')] \end{aligned}$$

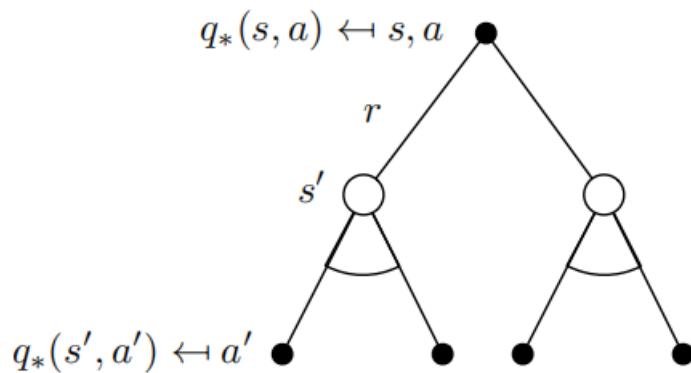
最后两个方程是  $v_*$  的贝尔曼最优方程的两种形式， $q_*$  的贝尔曼最优方程为

$$\begin{aligned} q_*(s, a) &= \mathbb{E}\left[R_{t+1} + \gamma \sum_{a'} q_*(S_{t+1, a'})|S_t = s, A_t = a\right] \\ &= \sum_{s', r} p(s', r|s, a)[r + \gamma \sum_{a'} q_*(s', a')] \end{aligned}$$

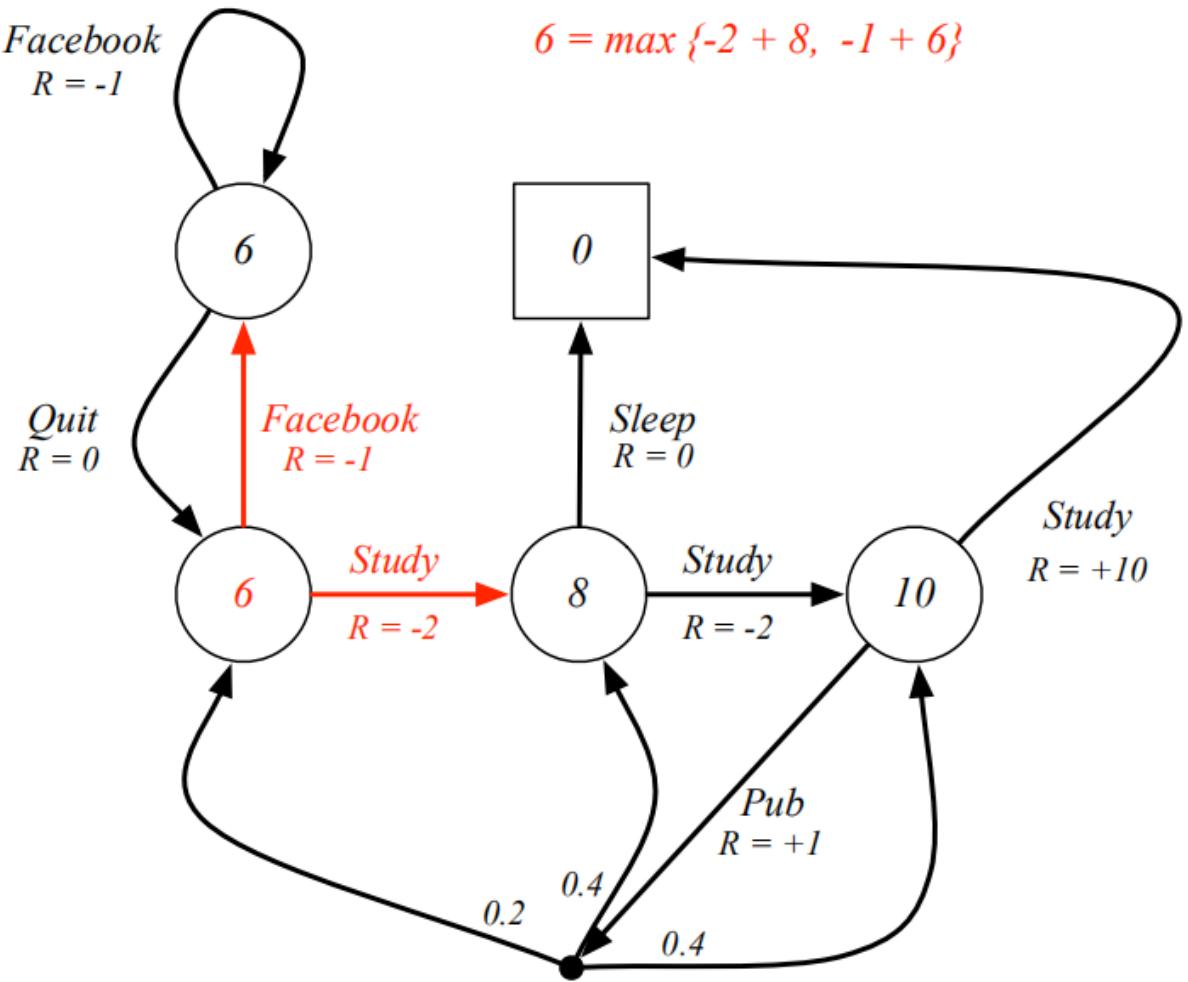
与上述状态-行为价值函数相似的关联如下所示：



$$v_*(s) = \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$



$$q_*(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s', a')$$



### 6.3.4 总结

表 6.1: MDP 相关概念

关键点	定义
状态转移矩阵	$P_{ss'}^a = P[S_{t+1} = s   S_t = s, A_t = a]$ 分布
Reward Function	$R_s^a = E[R_{t+1}   S_t = s, A_t = a]$ 固定, 不是随机变量
Policy	$\pi(a s) = P[A_t = a   S_t = s]$ 分布
Return	$G_t = R_{t+1} + \gamma R_{t+2} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$
state-value function	$V_{\pi}(s) = E_{\pi}[G_t   S_t = s]$
action-value function	$Q_{\pi}(s, a) = E_{\pi}[G_t   S_t = s, A_t = a]$
Expectation Equation For Predict	$v_{\pi}(s) = R_s + \gamma \sum_{s' \in S} P_{ss'} v(s')$ 加权平均-下同 $q_{\pi}(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a' s') q_{\pi}(s', a')$
Optimal Equation For Control	$v_*(s) = \text{Max}(R_s + \gamma \sum_{s' \in S} P_{ss'} v(s'))$ 加权平均取最大值-下同 $q_*(s, a) = \text{Max}(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a' s') q_*(s', a'))$

## 6.4 Model-Based 动态规划

当有了一个可以完美模拟马尔可夫过程的模型之后，如何计算最优 policies?。(注意是 *policies*，表明最优的策略可能不止一个)。

### 6.4.1 prediction-评估一个策略

首先考虑给定任意策略  $\pi$  怎样计算状态值函数  $v_\pi$ 。这在 DP 文献中被称作 策略评估、或者 Prediction。

### 6.4.2 control-找到最优策略

#### - Policy iteration-

等若干步迭代再 greedy 更新一次.

计算某个策略价值函数的目的是找到一个更好的策略。假设我们已经确定了一个任意确定性的策略  $\pi$  价值函数  $v_\pi$ 。对于某些状态  $s$  我们想知道是否应该改变策略来明确的选择一个动作  $a \pi(s)$ 。

我们知道在当前状态  $s$  遵从当前的策略有多好——> 也就是  $v_\pi(s)$ ——但是改变为一个新的状态会更好还是坏呢？一种解决这个问题的方法是考虑从状态  $s$  下选择动作  $a$ ，然后遵从现有的策略  $\pi$ 。

```
Input  $\pi$ , the policy to be evaluated
Initialize an array  $v(s) = 0$ , for all  $s \in \mathcal{S}^+$ 
Repeat
     $\Delta \leftarrow 0$ 
    For each  $s \in \mathcal{S}$ :
         $temp \leftarrow v(s)$ 
         $v(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} p(s'|s, a)[r(s, a, s') + \gamma v(s')]$ 
         $\Delta \leftarrow \max(\Delta, |temp - v(s)|)$ 
    until  $\Delta < \theta$  (a small positive number)
Output  $v \approx v_\pi$ 
```

一旦策略  $\pi$ ，已经用  $v_\pi$  提升为更好的策略  $\pi'$ ，我们可以计算  $v'_\pi$  再次提升策略得到更好的策略  $\pi''$ 。我们可以得到一系列单调提升的策略和价值函数：

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$

```

1. Initialization
 $v(s) \in \mathbb{R}$  and  $\pi(s) \in \mathcal{A}(s)$  arbitrarily for all  $s \in \mathcal{S}$ 

2. Policy Evaluation
Repeat
 $\Delta \leftarrow 0$ 
For each  $s \in \mathcal{S}$ :
 $temp \leftarrow v(s)$ 
 $v(s) \leftarrow \sum_{s'} p(s'|s, \pi(s)) [r(s, \pi(s), s') + \gamma v(s')]$ 
 $\Delta \leftarrow \max(\Delta, |temp - v(s)|)$ 
until  $\Delta < \theta$  (a small positive number)

3. Policy Improvement
 $policy-stable \leftarrow true$ 
For each  $s \in \mathcal{S}$ :
 $temp \leftarrow \pi(s)$ 
 $\pi(s) \leftarrow \arg \max_a \sum_{s'} p(s'|s, a) [r(s, a, s') + \gamma v(s')]$ 
If  $temp \neq \pi(s)$ , then  $policy-stable \leftarrow false$ 
If  $policy-stable$ , then stop and return  $v$  and  $\pi$ ; else go to 2

```

## - Value iteration-

每一步迭代都 greedy 更新一次

1. Initialization $v(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$  2. Policy Evaluation Repeat $\Delta \leftarrow 0$ For each $s \in \mathcal{S}$ : $temp \leftarrow v(s)$ $v(s) \leftarrow \sum_{s'} p(s' s, \pi(s)) [r(s, \pi(s), s') + \gamma v(s')]$ $\Delta \leftarrow \max(\Delta,  temp - v(s) )$ until $\Delta < \theta$ (a small positive number)  3. Policy Improvement $policy-stable \leftarrow true$ For each $s \in \mathcal{S}$ : $temp \leftarrow \pi(s)$ $\pi(s) \leftarrow \arg \max_a \sum_{s'} p(s' s, a) [r(s, a, s') + \gamma v(s')]$ If $temp \neq \pi(s)$ , then $policy-stable \leftarrow false$ If $policy-stable$ , then stop and return $v$ and $\pi$ ; else go to 2	<b>finding optimal value function</b>  Initialize array $v$ arbitrarily (e.g., $v(s) = 0$ for all $s \in \mathcal{S}^+$ )  Repeat $\Delta \leftarrow 0$ For each $s \in \mathcal{S}$ : $temp \leftarrow v(s)$ $v(s) \leftarrow \max_a \sum_{s'} p(s' s, a) [r(s, a, s') + \gamma v(s')]$ $\Delta \leftarrow \max(\Delta,  temp - v(s) )$ until $\Delta < \theta$ (a small positive number)  Output a deterministic policy, $\pi$ , such that $\pi(s) = \arg \max_a \sum_{s'} p(s' s, a) [r(s, a, s') + \gamma v(s')]$
--	--

Figure 4.5: Value iteration.

one policy  
 update (extract  
 policy from the  
 optimal value  
 function

Figure 4.3: Policy iteration (using iterative policy evaluation) for  $v_*$ . This algorithm has a subtle bug, in that it may never terminate if the policy continually switches between two or more policies that are equally good. The bug can be fixed by adding additional flags, but it makes the pseudocode so ugly that it is not worth it. :-)

```

import numpy as np
from gridworld import GridworldEnv

env = GridworldEnv()

def value_iteration(env, theta=0.0001, discount_factor=1.0):
    def one_step_lookahead(state, V):
        A = np.zeros(env.nA)
        for a in range(env.nA):
            for prob, next_state, reward, done in env.P[state][a]:
                A[a] += prob * (reward + discount_factor * V[next_state])
        return A

    V = np.zeros(env.nS)
    while True:
        # Stopping condition
        delta = 0
        # Update each state...
        for s in range(env.nS):
            # Do a one-step lookahead to find the best action
            A = one_step_lookahead(s, V)
            best_action_value = np.max(A)
            # Calculate delta across all states seen so far
            delta = max(delta, np.abs(best_action_value - V[s]))
            # Update the value function
            V[s] = best_action_value
        # Check if we can stop
        if delta < theta:
            break

    # Create a deterministic policy using the optimal value function
    policy = np.zeros([env.nS, env.nA])
    for s in range(env.nS):
        # One step lookahead to find the best action for this state
        A = one_step_lookahead(s, V)
        best_action = np.argmax(A)
        # Always take the best action
        policy[s, best_action] = 1.0

    return policy, V

policy, v = value_iteration(env)

print("Policy\u2225Probability\u2225Distribution:")
print(policy)
print("")

print("Reshaped\u2225Grid\u2225Policy\u2225(0=up,\u22251=right,\u22252=down,\u22253=left):")
print(np.reshape(np.argmax(policy, axis=1), env.shape))
print("")

```

## 6.5 Model-Free 蒙特卡洛

蒙特卡罗强化学习算法通过考虑采样轨迹克服了模型未知给策略估计造成的困难。

“多次”采样”，然后求取平均累积奖赏来作为期望累积奖赏的近似，这称为蒙特卡罗强化学习。

利用 $\epsilon$ 因子，当随机数大于 $\epsilon$ 利用，当随机数小于 $\epsilon$ 进行探索。换句话说，就是以 $1 - \epsilon$ 概

率进行利用，以  $\epsilon$  概率进行均匀探索。

## 6.6 TD-Temporal-Difference 瞬时差分法

<https://www.cnblogs.com/jinxulin/p/5116332.html>

时序差分 (Temporal Difference，简称 TD) 学习则结合了动态规划与蒙特卡罗方法的思想，能做到更高效的、免模型 (不需要提前知道环境模型) 学习。

从而不局限于需要采样整个 Episode，当个状态间就可以实现值迭代。

### 6.6.1 原理

### 6.6.2 典型算法

- Saras
- Q-learning

## 6.7 Saras

在蒙特卡洛的基础上，结合动态规划进行性能提升，使得可以在每一步采样后都可以对 Policy 进行提升，采样的策略与提升的策略相同：OnPolicy。

### 6.7.1 Saras

单步安全迭代更新

### 6.7.2 Saras( $\lambda$ )

$\lambda$  步迭代更新。

## 6.8 Q-Learning

采样的策略与提升的策略不相同：OffPolicy。

1. Initialize Q-values ( $Q(s, a)$ ) arbitrarily for all state-action pairs.
2. For life or until learning is stopped...
3. Choose an action ( $a$ ) in the current world state ( $s$ ) based on current Q-value estimates ( $Q(s, \cdot)$ ).
4. Take the action ( $a$ ) and observe the outcome state ( $s'$ ) and reward ( $r$ ).
5. Update  $Q(s, a) := Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$

```

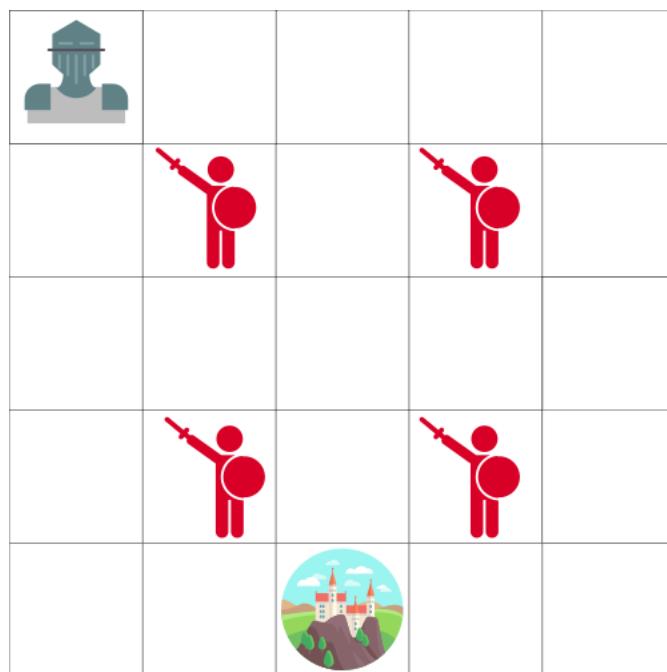
initialize Q[num_states, num_actions] arbitrarily
observe initial state s
repeat
    select and carry out an action a
    observe reward r and new state s'
    Q[s, a] = Q[s, a] + α(r + γ maxa' Q[s', a'] - Q[s, a])
    s = s'
until terminated

```

### 6.8.1 Q-learning 从案例到原理

[http://www.sohu.com/a/228536039\\_129720](http://www.sohu.com/a/228536039_129720)

Environment: Game *the Knight and the Princess*



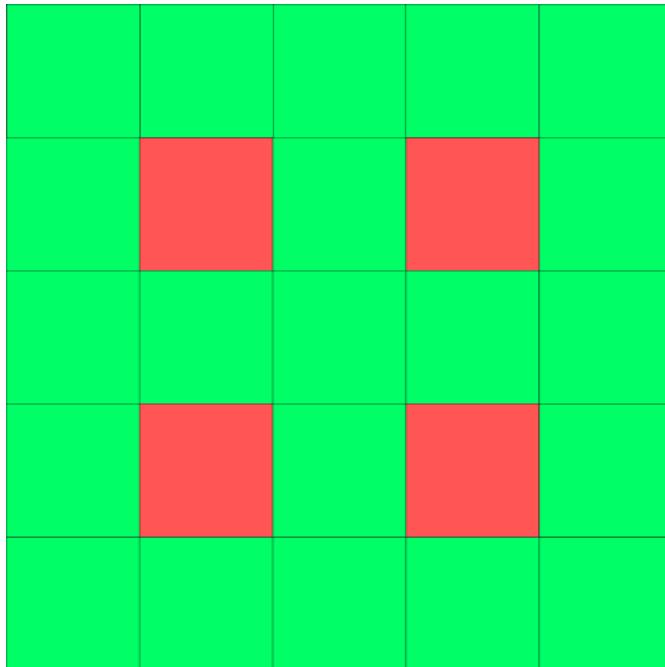
你每次可以移动一个方块的距离。敌人是不能移动的，但是如果你和敌人落在了同一个方块中，你就会死。你的目标是以尽可能快的路线走到城堡去。这可以使用一个「按步积分」系统来评估。

- 你在每一步都会失去 1 分, reward<sub>x</sub> = -1;

- 如果碰到了一个敌人，你会失去 100 分，并且训练 episode 结束。 $rewardEnemy = -100$ ;
- 如果进入到城堡中，你就获胜了，获得 100 分。 $rewardPrincess = 100$ ;

### Q-table:MDP

**Policy  $\pi_1$**  第一个策略。假设智能体试图走遍每一个方块，并且将其着色。绿色代表「安全」，红色代表「不安全」。



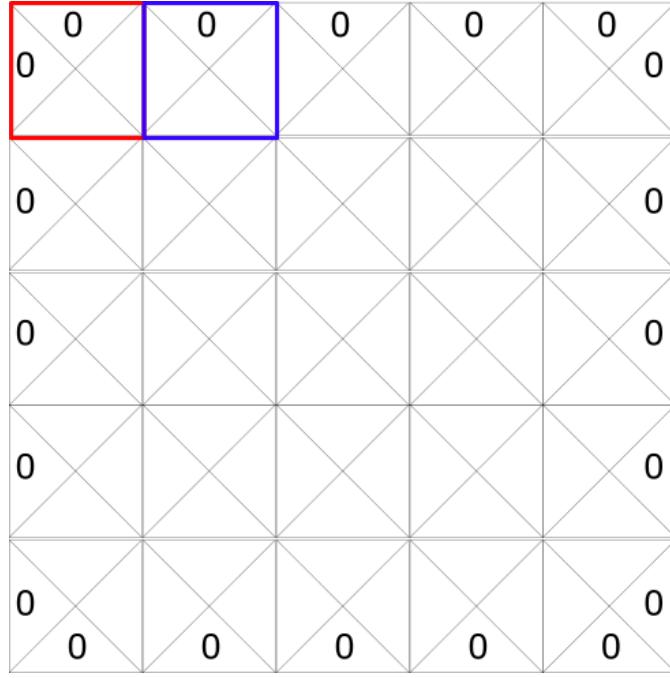
同样的地图，但是被着色了，用于显示哪些方块是可以被安全访问的。接着，我们告诉智能体只能选择绿色的方块。

但问题是，这种策略并不是十分有用。当绿色的方块彼此相邻时，我们不知道选择哪个方块是最好的。所以，Agent 可能会在寻找城堡的过程中陷入无限的循环。

**Policy  $\pi_2$**  第二种策略：创建一个表格。通过它，我们可以为每一个状态（state）上进行的每一个动作（action）计算出最大的未来奖励（reward）的期望。

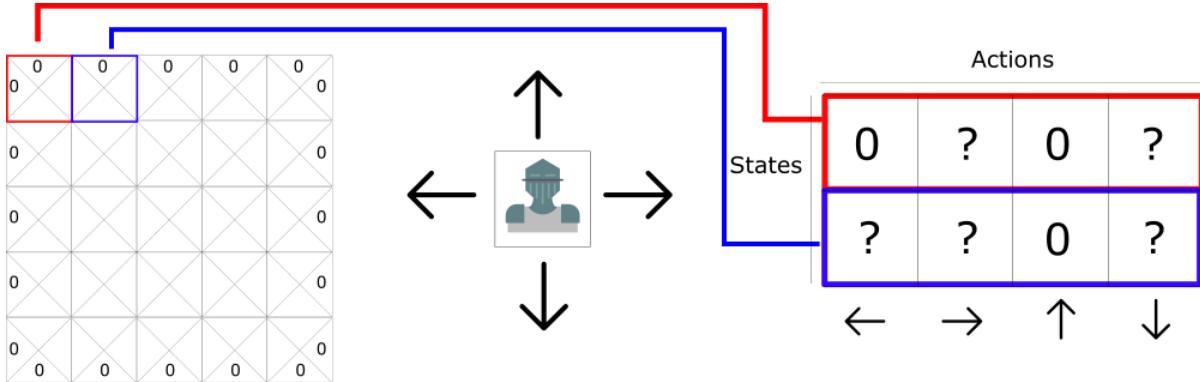
得益于这个表格，我们可以知道为每一个状态采取的最佳动作。

每个状态（方块）允许四种可能的操作：左移、右移、上移、下移。



「0」代表不可能的移动（如果你在左上角，你不可能向左移动或者向上移动！）

在计算过程中，我们可以将这个网格转换成一个表。这种表格被称为 **Q-table**（「Q」代表动作的「未来价值」）。每一列将代表四个操作（左、右、上、下），行代表状态。每个单元格的值代表给定状态和相应动作的最大未来奖励期望。



将这个 Q-table 想象成一个「备忘纸条」游戏。得益于此，我们通过寻找每一行中最高的分数，可以知道对于每一个状态（Q-table 中的每一行）来说，可采取的最佳动作是什么。这样就解决了这个城堡问题！但是，如何计算 Q-table 中每个元素的值呢？也就是 MDP 中提到的 BellManEquation。

**BellMan-Equation** 为了求出 Q-table 中的每个值，将使用 **Q-learning 算法**求解 BellMan Equation.

**Q-learning 算法：学习动作值函数** 动作-值函数（或称「*Q* 函数」）有两个输入：「状态」和「动作」。它将返回在该状态下执行该动作的未来奖励期望。

$$Q^\pi(s_t, a_t) = \underline{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | s_t, a_t]$$

Q value for that state given that action

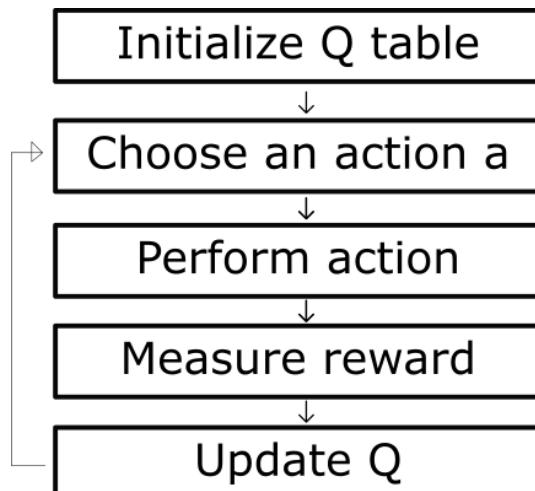
Expected discounted cumulative reward ...

given that state and that action

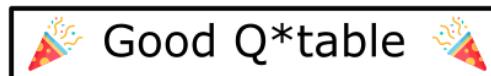
可以把 Q 函数视为一个在 Q-table 上滚动的读取器，用于寻找与当前状态关联的行以及与动作关联的列。它会从相匹配的单元格中返回 Q 值。这就是未来奖励的期望。

在探索环境（environment）之前，Q-table 会给出相同的任意的设定值（大多数情况下是 0）。随着对环境的持续探索，这个 Q-table 会通过迭代地使用 Bellman 方程（动态规划方程）更新  $Q(s,a)$  来给出越来越好的近似。具体求解参考动态规范方法一节。

**Q-learning 流程** 一般计算机模拟流程大概如下：



At the end of the training



- **Step 1: Initialize Q-values**

初始化 Q 值。我们构造了一个  $m$  列 ( $m = \text{动作数}$ )， $n$  行 ( $n = \text{状态数}$ ) 的 Q-table，并将其中的值初始化为 0。

		Actions				
		States	0	0	0	0
			0	0	0	0
			0	0	0	0
			0	0	0	0
			■ ■ ■			
			0	0	0	0

- **Step 2: For life (or until learning is stopped)**

在整个生命周期中（或者直到训练被中止前），步骤 3 到步骤 5 会一直被重复，直到达到了最大的训练次数（由用户指定）或者手动中止训练。

- **Step 3: Choose an action**

选取一个动作。在基于当前的 Q 值估计得出的状态  $s$  下选择一个动作  $a$ 。

但是……如果每个 Q 值都等于零，我们一开始该选择什么动作呢？请转到  $\epsilon$  epsilon 探索/利用比率小节。

- **Step 4-5: Evaluate!**

评价！采用动作  $a$  并且观察输出的状态  $s'$  和奖励  $r$ 。现在我们更新函数  $Q(s, a)$ 。

我们采用在步骤 3 中选择的动作  $a$ ，然后执行这个动作会返回一个新的状态  $s'$  和奖励  $r$ 。接着我们使用 Bellman 方程去更新  $Q(s, a)$ ：

$$NewQ(s, a) = Q(s, a) + \alpha[R(s, a) + \gamma \max Q'(s', a') - Q(s, a)]$$

New Q value for that state and that action  
 Current Q value  
 Reward for taking that action at that state  
 Learning Rate  
 Discount rate  
 Maximum expected future reward given the new  $s'$  and all possible actions at that new state

```

|| New Q value = Current Q value + lr * [Reward + discount_rate * (highest Q
  value between possible actions from the new state s' ) — Current Q
  value]
  
```

$\alpha$  学习率 可以将学习率看作是网络有多快地抛弃旧值、生成新值的度量。如果学习率是 1，新的估计值会成为新的 Q 值，并完全抛弃旧值。

$\epsilon$  epsilon 探索/利用比率 在刚开始初始化后，每个 Q 值都等于零，一开始该选择什么动作呢？在这里，就可以看到探索/利用（exploration/exploitation）的权衡有多重要了。

思路就是，在一开始，将使用 **epsilon** 贪婪策略：

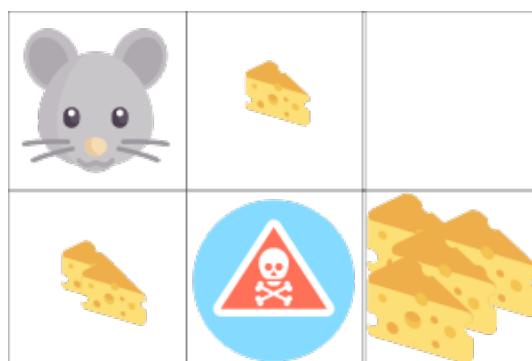
1. 指定一个探索速率「**epsilon**」，一开始将它设定为 1。这个就是我们将随机采用的步长。在一开始，这个速率应该处于最大值，因为我们不知道 Q-table 中任何的值。这意味着，我们需要通过随机选择动作进行大量的探索。
2. 生成一个随机数。如果这个数大于 **epsilon**，那么我们将会进行「利用」（这意味着我们在每一步利用已经知道的信息选择动作）。否则，我们将继续进行探索。
3. 在刚开始训练 Q 函数时，我们必须有一个大的 **epsilon**。随着智能体对估算出的 Q 值更有把握，我们将逐渐减小 **epsilon**。



### 6.8.2 Q-learning 案例示例

案例 2： <https://www.jianshu.com/p/29db50000e3f>

**Environment** 老鼠奶酪游戏。



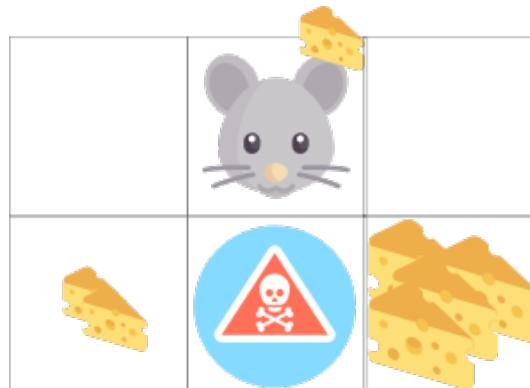
- 一块奶酪 = +1

- 两块奶酪 = +2
- 一大堆奶酪 = +10 (训练结束)
- 吃到了鼠药 = -10 (训练结束)

Step1: 初始化 Q-table 初始化

	←	→	↑	↓
Start	0	0	0	0
Small cheese	0	0	0	0
Nothing	0	0	0	0
2 small cheese	0	0	0	0
Death	0	0	0	0
Big cheese	0	0	0	0

Step2、3: 选择一个动作 从起始点，你可以在向右走和向下走其中选择一个。由于有一个大的 epsilon 速率（因为我们至今对于环境一无所知），我们随机地选择一个。例如向右走。



	←	→	↑	↓
Start	0	0	0	0
Small cheese	0	0	0	0
Nothing	0	0	0	0
2 small cheese	0	0	0	0
Death	0	0	0	0
Big cheese	0	0	0	0

我们随机移动（例如向右走）

我们发现了一块奶酪 (+1)，现在我们可以更新开始时的 Q 值并且向右走，通过 Bellman 方程实现。

#### Step4、5：更新 Q 函数 更新 Q 函数

$$NewQ(s, a) = Q(s, a) + \alpha [R(s, a) + \gamma \max Q'(s', a') - Q(s, a)]$$

$$NewQ(start, right) = Q(start, right) + \alpha [\Delta Q(start, right)]$$

$$\Delta Q(start, right) = R(start, right) + \gamma \max Q'(1cheese, a') - Q(start, right)$$

$$\Delta Q(start, right) = 1 + 0.9 * \max(Q'(1cheese, left), Q'(1cheese, right), Q'(1cheese, down)) - Q(start, right)$$

$$\Delta Q(start, right) = 1 + 0.9 * 0 - 0 = 1$$

$$NewQ(start, right) = 0 + 0.1 * 1 = 0.1$$

- 首先，我们计算 Q 值的改变量  $\Delta Q(start, right)$ 。
- 接着我们将初始的 Q 值与  $\Delta Q(start, right)$  和学习率  $\alpha$  的积相加。

	$\leftarrow$	$\rightarrow$	$\uparrow$	$\downarrow$
Start	0	0.1	0	0
Small cheese	0	0	0	0
Nothing	0	0	0	0
2 small cheese	0	0	0	0
Death	0	0	0	0
Big cheese	0	0	0	0

刚刚更新了第一个 Q 值。现在我们要做的就是一次又一次地做这个工作直到学习结束。

## 6.9 Deep-Q-Network

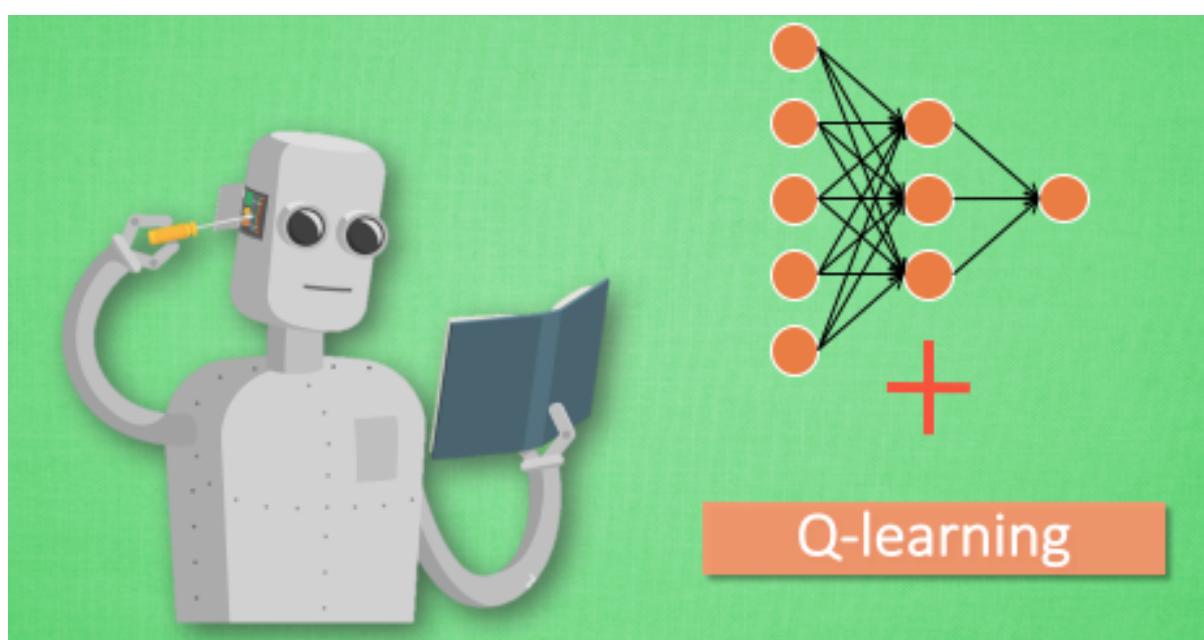
原理概述：<https://www.jianshu.com/p/41659ce20f15>

原理概述：<https://www.jianshu.com/p/10930c371cac>

原理细节(待)：<https://blog.csdn.net/u013236946/article/details/72871858>

DQN 拓展形式汇总: <https://www.jianshu.com/p/1dfd84cd2e69>

这是一种融合了神经网络和 Q learning 的方法, 名字叫做 Deep Q Network.



---

**算法**      : 带经验回放的深度 Q 网络

---

**输入:** 状态空间  $\mathcal{S}$ , 动作空间  $\mathcal{A}$ , 折扣率  $\gamma$ , 学习率  $\alpha$ , 参数更新间隔  $C$ ;

1 初始化经验池  $\mathcal{D}$ , 容量为  $N$ ;

2 随机初始化  $Q$  网络的参数  $\phi$ ;

3 随机初始化目标  $Q$  网络的参数  $\hat{\phi} = \phi$ ;

4 **repeat**

5    初始化起始状态  $s$ ;

6    **repeat**

7      在状态  $s$ , 选择动作  $a = \pi^e$ ;

8      执行动作  $a$ , 观测环境, 得到即时奖励  $r$  和新的状态  $s'$ ;

9      将  $s, a, r, s'$  放入  $\mathcal{D}$  中;

10     从  $\mathcal{D}$  中采样  $ss, aa, rr, ss'$ ;

11      $y = \begin{cases} rr, & ss' \text{ 为终止状态,} \\ rr + \gamma \max_{a'} Q_{\hat{\phi}}(ss', a'), & \text{否则} \end{cases}$ ;

12     以  $(y - Q_{\phi}(ss, aa))^2$  为损失函数来训练  $Q$  网络;

13      $s \leftarrow s'$ ;

14     每隔  $C$  步,  $\hat{\phi} \leftarrow \phi$ ;

15    **until**  $s$  为终止状态;

16 **until**  $\forall s, a, Q_{\phi}(s, a)$  收敛;

**输出:**  $Q$  网络  $Q_{\phi}(s, a)$

---

图 6.5: DQN 算法概述

```

initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
    select an action a
        with probability  $\epsilon$  select a random action
        otherwise select  $a = \operatorname{argmax}_{a'} Q(s, a')$ 
    carry out action a
    observe reward r and new state  $s'$ 
    store experience  $\langle s, a, r, s' \rangle$  in replay memory D

    sample random transitions  $\langle ss, aa, rr, ss' \rangle$  from replay memory D
    calculate target for each minibatch transition
        if  $ss'$  is terminal state then  $tt = rr$ 
        otherwise  $tt = rr + \gamma \max_{a'} Q(ss', aa')$ 
    train the Q network using  $(tt - Q(ss, aa))^2$  as loss

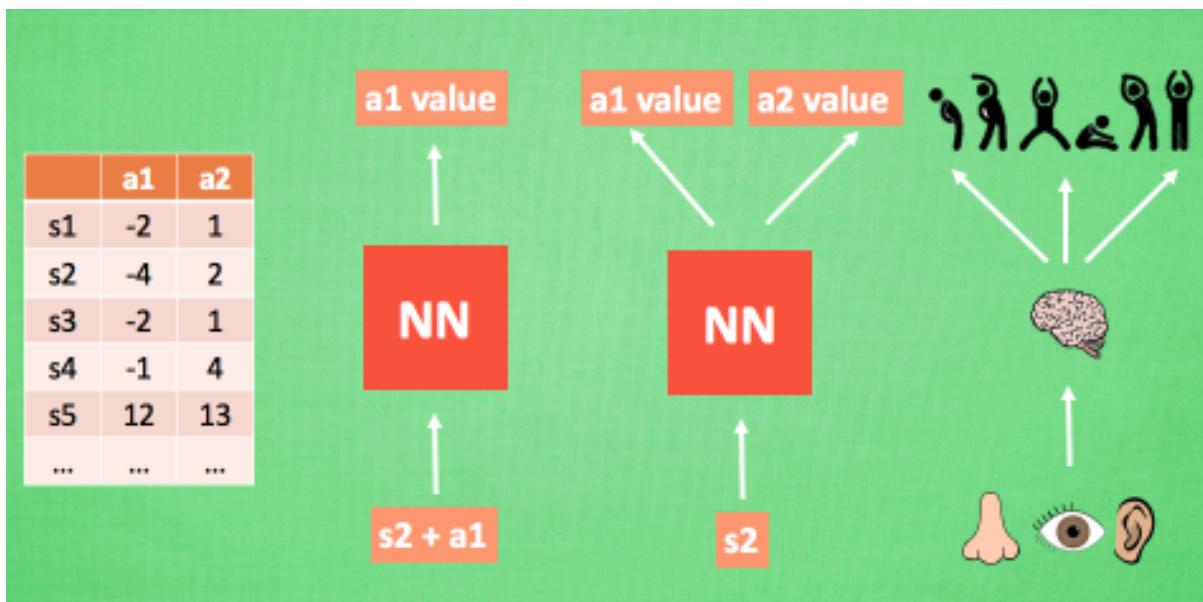
     $s = s'$ 
until terminated

```

图 6.6: DQN 算法概述

### 6.9.1 神经网络的作用

使用表格来存储每一个状态 state, 和在这个 state 每个行为 action 所拥有的 Q 值. 而当今问题是在太复杂, 状态可以多到比天上的星星还多 (比如下围棋). 如果全用表格来存储它们, 恐怕我们的计算机有再大的内存都不够, 而且每次在这么大的表格中搜索对应的状态也是一件很耗时的事. 不过, 在机器学习中, 有一种方法对这种事情很在行, 那就是神经网络. 我们可以将状态和动作当成神经网络的输入, 然后经过神经网络分析后得到动作的 Q 值, 这样我们就没必要在表格中记录 Q 值, 而是直接使用神经网络生成 Q 值. 还有一种形式的是这样, 我们也能只输入状态值, 输出所有的动作值, 然后按照 Q learning 的原则, 直接选择拥有最大值的动作当做下一步要做的动作.



我们可以想象, 神经网络接受外部的信息, 相当于眼睛鼻子耳朵收集信息, 然后通过大脑加工

输出每种动作的值，最后通过强化学习的方式选择动作。

### 6.9.2 DL 和 RL 结合带来的问题

1. DL 需要大量带标签的样本进行监督学习；RL 只有 reward 返回值，而且伴随着噪声，延迟（过了几十毫秒才返回），稀疏（很多 State 的 reward 是 0）等问题；
2. DL 的样本独立；RL 前后 state 状态相关
3. DL 目标分布固定；RL 的分布一直变化，比如你玩一个游戏，一个关卡和下一个关卡的状态分布是不同的，所以训练好了前一个关卡，下一个关卡又要重新训练；
4. 过往的研究表明，使用非线性网络表示值函数时出现不稳定等问题。

### 6.9.3 DQN 解决问题方法

1. 通过 Q-Learning 使用 reward 来构造标签（对应问题 1）
2. 通过 experience replay（经验池）的方法来解决相关性及非静态分布问题（对应问题 2、3）
3. 使用一个神经网络产生当前 Q 值，使用另外一个神经网络产生 Target Q 值（对应问题 4）

构造标签 ->

对于函数优化问题，监督学习的一般方法是先确定 Loss Function，然后求梯度，使用随机梯度下降等方法更新参数。

DQN 则基于 Q-Learning 来确定 Loss Function。我们想要使 q-target 值和 q-eval 值相差越小越好。DQN 中的损失函数是：

$$J_i(\theta_i) = \mathbb{E}_{s,a \sim \pi} [(y_i - Q(s, a; \theta_i))^2]$$

这里  $y_i$  是根据上一个迭代周期或者说 target-net 网络的参数计算出的 q-target 值，跟当前网络结构中的参数无关， $y_i$  的计算如下：

$$y_i = \mathbb{E}_{s' \sim \pi^\epsilon} [r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a]$$

这样，整个目标函数就可以通过随机梯度下降方法来进行优化：

$$\nabla_{\theta_i} J_i(\theta_i) = \mathbb{E}_{s,a \sim \pi; s' \sim \pi^\epsilon} [(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)) \nabla_{\theta_i} Q(s, a; \theta_i)]$$

经验回放 ->

经验池的功能主要是解决相关性及非静态分布问题。具体做法是把每个时间步 agent 与环境交互得到的转移样本  $(st, at, rt, st+1)$  储存到回放记忆单元，要训练时就随机拿出一些（minibatch）来训练。（其实就是将游戏的过程打成碎片存储，训练时随机抽取就避免了相关性问题）

构建一个经验池（Replay Buffer）来去除数据相关性。经验池是由智能体最近的经历组成的数据集。

事实证明，使用非线性函数得到近似的 Q 值是不稳定的。要使它真正收敛还需要加入很多技巧，而且要花费很长的时间，即使使用一个 GPU 也要耗上差不多一周的时间。

最重要的一个技巧就是经验回放。在游戏过程中，所有的经历  $\langle s, a, r, s' \rangle$  都存储在一个可回顾的记忆模块中。训练神经网络的时候，会从记忆中取出随机小批量的记忆片段来训练，而不是直接使用最近期的经历。这样就切断了之后相似训练情况的连续性，这种相似性往往会使整个网络局限于一小块状态区域。此外，经验回放使训练任务更类似于通常的监督学习，这简化了调试和测试算法。而且也可以直接收集人类玩游戏的经验，在此基础上继续训练。

## 双网络结构 ->

在 Nature 2015 版本的 DQN 中提出了这个改进，使用另一个网络（这里称为 `target_net`）产生 Target Q 值。具体地， $Q(s, a; \theta_i)$  表示当前网络 `eval_net` 的输出，用来评估当前状态动作对的值函数； $Q(s, a; \theta_{i-1})$  表示 `target_net` 的输出，代入上面求 Target Q 值的公式中得到目标 Q 值。根据上面的 Loss Function 更新 `eval_net` 的参数，每经过 N 轮迭代，将 MainNet 的参数复制给 `target_net`。

上述的 N 轮迭代既目标网络冻结（Freezing Target Networks），即在一个时间段内固定目标中的参数，来稳定学习目标。

引入 `target_net` 后，在一段时间里目标 Q 值使保持不变的，一定程度降低了当前 Q 值和目标 Q 值的相关性，提高了算法稳定性。

```

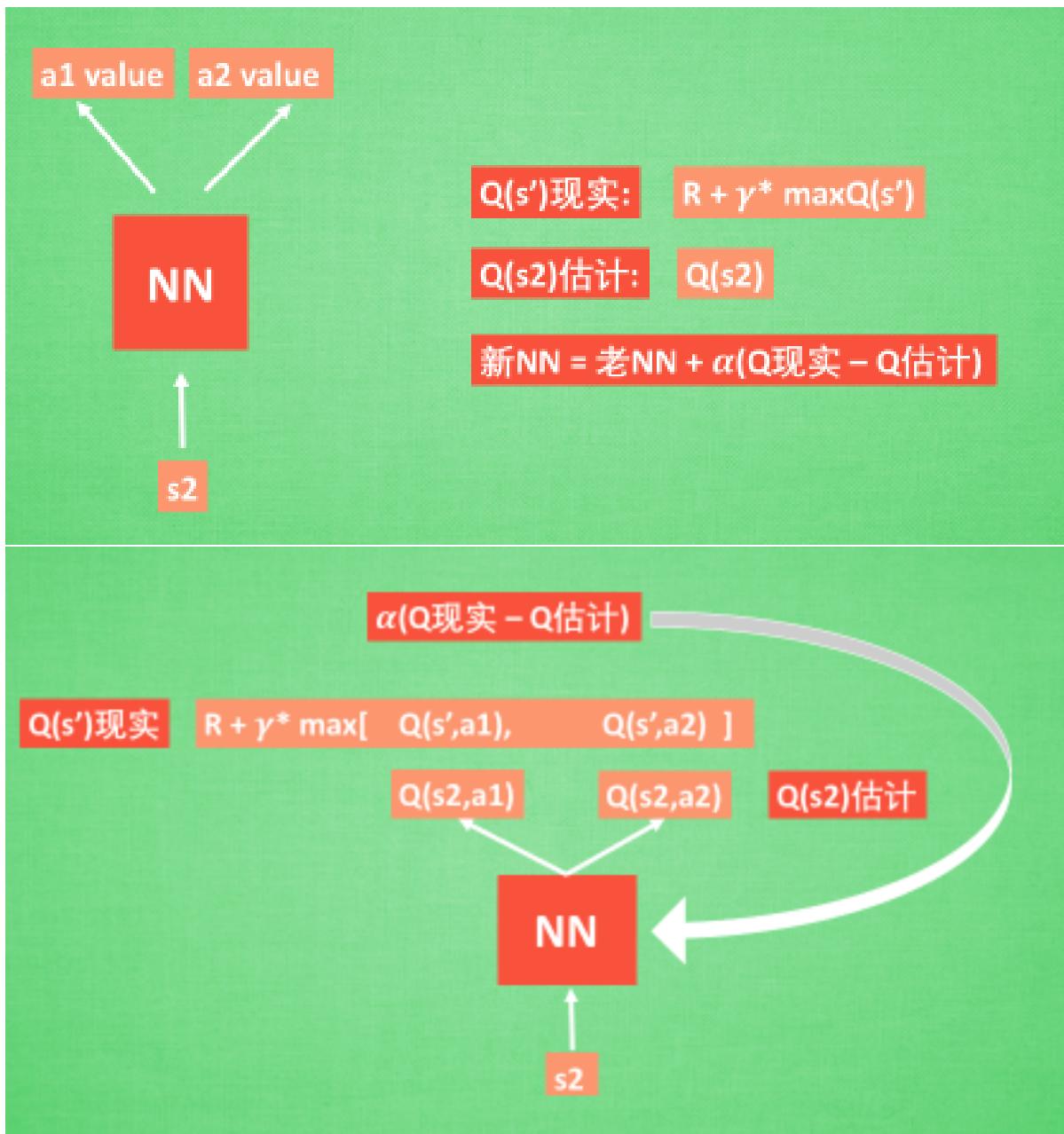
Initialize replay memory  $D$  to capacity  $N$ 
Initialize action-value function  $Q$  with random weights  $\theta$ 
Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$ 
For episode = 1,  $M$  do
    Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$ 
    For  $t = 1, T$  do
        With probability  $\varepsilon$  select a random action  $a_t$ 
        otherwise select  $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$ 
        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $D$ 
        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$ 
        Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$ 
        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the
        network parameters  $\theta$ 
        Every  $C$  steps reset  $\hat{Q} = Q$ 
    End For
End For

```

#### 6.9.4 训练神经网络

基于第二种神经网络来分析

我们知道，神经网络是要被训练才能预测出准确的值。那在强化学习中，神经网络是如何被训练的呢？首先，我们需要  $a_1, a_2$  正确的 Q 值，这个 Q 值我们就用之前在 Q learning 中的 Q 现实来代替。同样我们还需要一个 Q 估计来实现神经网络的更新。所以神经网络的参数就是老的 NN 参数加学习率 alpha 乘以 Q 现实和 Q 估计的差距。我们整理一下。



通过 NN 预测出  $Q(s_2, a1)$  和  $Q(s_2, a2)$  的值，这就是 Q 估计。然后我们选取 Q 估计中最大值的动作来换取环境中的奖励 reward。而 Q 现实中也包含从神经网络分析出来的两个 Q 估计值，不过这个 Q 估计是针对下一步在  $s'$  的估计。最后再通过刚刚所说的算法更新神经网络中的参数。但是这并不是 DQN 会玩电动的根本原因。还有两大因素支撑着 DQN 使得它变得无比强大。这两大因素就是 Experience replay 和 Fixed Q-targets。

### 6.9.5 DQN 两大利器

DQN 有一个记忆库用于学习之前的经历, Q learning 是一种 off-policy 离线学习法, 它能学习当前经历着的, 也能学习过去经历过的, 甚至是学习别人的经验. 所以每次 DQN 更新的时候, 我们都可以随机抽取一些之前的经历进行学习. 随机抽取这种做法打乱了经历之间的相关性, 也使得神经网络更新更有效率.

Fixed Q-targets 也是一种打乱相关性的机理, 如果使用 fixed Q-targets, 我们就会在 DQN 中使用到两个结构相同但参数不同的神经网络, 预测 Q 估计的神经网络具备最新的参数, 而预测 Q 现实的神经网络使用的参数则是很久以前的.

### 6.9.6 DQN-改进-Double DQN

<https://www.jianshu.com/p/fae51b5fe000>

### 6.9.7 DQN-改进-Prioritised replay

<https://www.jianshu.com/p/db14fdc67d2c>

### 6.9.8 Dueling Network

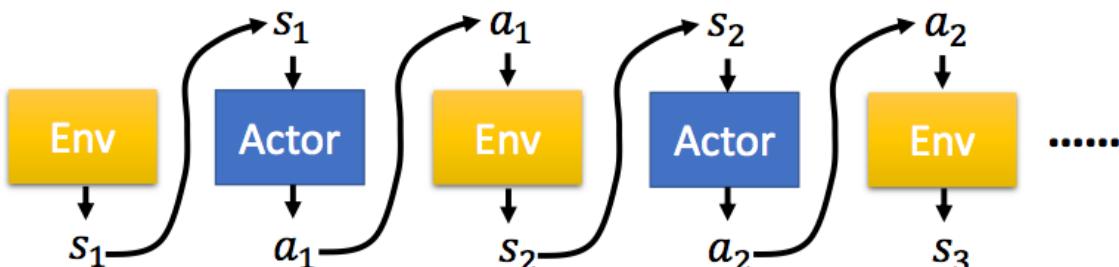
<https://www.jianshu.com/p/b421c85796a2>

## 6.10 Policy Gradient

<https://www.jianshu.com/p/2ccbab48414b>

采样 Trajectory、训练, Loss 函数为奖励最大

在 PG 算法中, 我们的 Agent 又被称为 Actor, Actor 对于一个特定的任务, 都有自己的一个策略  $\pi$ , 策略  $\pi$  通常用一个神经网络表示, 其参数为  $\theta$ . 从一个特定的状态 state  $s$  出发, 一直到任务的结束, 被称为一个完整的 episode, 在每一步, 我们都能获得一个奖励  $r$ , 一个完整的任务所获得的最终奖励被称为  $R$ . 这样, 一个有  $T$  个时刻的 episode, Actor 不断与环境交互, 形成如下的序列  $\tau$ :



**Trajectory**  $\tau = \{s_1, a_1, s_2, a_2, \dots, s_T, a_T\}$

图 6.7: PG 采样结果示例

这样一个序列  $\tau$  是不确定的，因为 Actor 在不同 state 下所采取的 action 可能是不同的，一个序列  $\tau$  发生的概率为：

$$\begin{aligned} p_{\theta}(\tau) &= p(s_1)p_{\theta}(a_1|s_1)p(s_2|s_1, a_1)p_{\theta}(a_2|s_2)p(s_3|s_2, a_2)\dots \\ &= p(s_1) \prod_{t=1}^T p_{\theta}(a_t|s_t)p(s_{t+1}|s_t, a_t) \end{aligned}$$

序列  $\tau$  所获得的奖励为每个阶段所得到的奖励的和，称为  $R(\tau)$ 。因此，在 Actor 的策略为  $\pi$  的情况下，所能获得的期望奖励为：

$$\bar{R}_{\theta} = \sum_{\tau} R(\tau)p_{\theta}(\tau) = E_{\tau \sim p_{\theta}(\tau)}[R(\tau)]$$

而我们的期望是调整 Actor 的策略  $\pi$ ，使得期望奖励最大化，于是我们有了策略梯度的方法，既然我们的期望函数已经有了，我们只要使用梯度提升的方法更新我们的网络参数  $\theta$ （即更新策略  $\pi$ ）就好了，所以问题的重点变为了求参数的梯度。梯度的求解过程如下：

$$\nabla \bar{R}_{\theta} = \sum_{\tau} R(\tau) \nabla p_{\theta}(\tau) = \sum_{\tau} R(\tau) p_{\theta}(\tau) \frac{\nabla p_{\theta}(\tau)}{p_{\theta}(\tau)}$$

$R(\tau)$  do not have to be differentiable

It can even be a black box.

$$= \boxed{\sum_{\tau}} R(\tau) p_{\theta}(\tau) \boxed{\nabla \log p_{\theta}(\tau)}$$

$$\nabla f(x) = f(x) \nabla \log f(x)$$

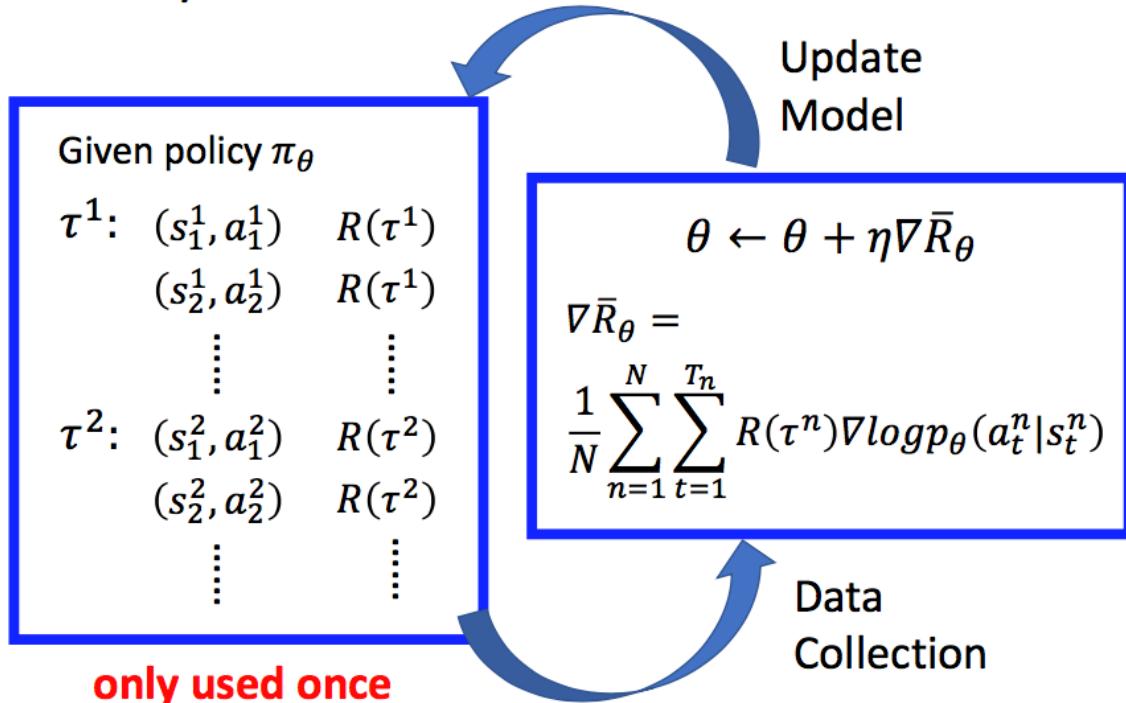
$$\begin{aligned} &= E_{\tau \sim p_{\theta}(\tau)}[R(\tau) \nabla \log p_{\theta}(\tau)] \approx \frac{1}{N} \sum_{n=1}^N R(\tau^n) \nabla \log p_{\theta}(\tau^n) \\ &= \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p_{\theta}(a_t^n | s_t^n) \end{aligned}$$

上面的过程中，我们首先利用  $\log$  函数求导的特点进行转化，随后用  $N$  次采样的平均值来近似期望，最后，我们将  $p_{\theta}$  展开，将与  $\theta$  无关的项去掉，即得到了最终的结果。

所以，一个 PG 方法的完整过程如下：

$$\nabla \bar{R}_\theta = E_{\tau \sim p_\theta(\tau)} [R(\tau) \nabla \log p_\theta(\tau)]$$

# Policy Gradient



我们首先采集数据，然后基于前面得到的梯度提升的式子更新参数，随后再根据更新后的策略再采集数据，再更新参数，如此循环进行。注意到图中的大红字 only used once，因为在更新参数后，我们的策略已经变了，而先前的数据是基于更新参数前的策略得到的。

## 6.10.1 PG ReinForce 算法

策略梯度的基本思想，就是直接根据状态输出动作或者动作的概率。

在训练神经网络中，使用最多的方法就是反向传播算法，其中需要一个误差函数，通过梯度下降来使损失函数的值最小化。但对于强化学习来说，我们不知道动作的正确与否，只能通过奖励值来判断这个动作的相对好坏。基于上面的想法，有个非常简单的想法：

如果一个动作得到的 reward 多，那么我们就使其出现的概率增加，如果一个动作得到的 reward 少，我们就使其出现的概率减小。

根据这个思想，可以构造如下的损失函数： $loss = -\log(prob) * vt$

- $\log(prob)$  表示在状态  $s$  对所选动作  $a$  的吃惊度，如果概率越小，反向的  $\log(prob)$  反而越大
- $vt$  代表的是当前状态  $s$  下采取动作  $a$  所能得到的奖励，这是当前的奖励和未来奖励的贴现值的求和

也就是说，我们的策略梯度算法必须要完成一个完整的 episode 才可以进行参数更新，而不是像值方法那样，每一个  $(s, a, r, s')$  都可以进行参数更新。

Policy Gradient 的核心思想是更新参数时有两个考虑：如果这个回合选择某一动作，下一回合选择该动作的概率大一些，然后再看奖惩值，如果奖惩是正的，那么会放大这个动作的概率，

如果奖惩是负的，就会减小该动作的概率。

策略梯度的过程如下图所示：

```
function REINFORCE
    Initialise  $\theta$  arbitrarily
    for each episode  $\{s_1, a_1, r_2, \dots, s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$  do
        for  $t = 1$  to  $T - 1$  do
             $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$ 
        end for
    end for
    return  $\theta$ 
end function
```

图 6.8: 策略梯度

最后再强调 Policy Gradient 的一些细节：

- 算法输出的是动作的概率，而不是 Q 值。
- 损失函数的形式为： $loss = -\log(prob) * vt$
- 需要一次完整的 episode 才可以进行参数的更新

### 6.10.2 带基准线的 ReInforce 算法

REINFORCE 算法的一个主要缺点是不同路径之间的方差很大，导致训练不稳定，这是在高维空间中使用蒙特卡罗方法的通病。一种减少方差的通用方法是引入一个控制变量。

假设要估计函数  $f$  的期望，为了减少  $f$  的方差，我们引入一个已知期望的函数  $g$ ，令

$$\hat{f} = f - \alpha(g - E[g]).$$

因为  $E[\hat{f}] = E[f]$ ，我们可以用  $\hat{f}$  的期望来估计函数  $f$  的期望，同时利用函数  $g$  来减小  $\hat{f}$  的方差。

## 6.11 Actor-Critic

实现：[https://blog.csdn.net/qq\\_30615903/article/details/80774384](https://blog.csdn.net/qq_30615903/article/details/80774384)

<https://www.jianshu.com/p/25c09ae3d206>

理论：<https://zhuanlan.zhihu.com/p/36494307>

AC 代码的实现地址为：[https://github.com/princewen/tensorflow\\_practice/tree/master/RL/Basic-AC-Demo](https://github.com/princewen/tensorflow_practice/tree/master/RL/Basic-AC-Demo)

---

**算法 14.8: 演员-评论员算法**

---

输入: 状态空间  $\mathcal{S}$ , 动作空间  $\mathcal{A}$ ;  
可微分的策略函数  $\pi_\theta(a|s)$ ;  
可微分的状态值函数  $V_\phi(s)$ ;  
折扣率  $\gamma$ , 学习率  $\alpha > 0, \beta > 0$ ;

1 随机初始化参数  $\theta, \phi$ ;

2 repeat

3    初始化起始状态  $s$ ;

4     $\lambda = 1$ ;

5    repeat

6     在状态  $s$ , 选择动作  $a = \pi_\theta(a|s)$ ;

7     执行动作  $a$ , 得到即时奖励  $r$  和新状态  $s'$ ;

8      $\delta \leftarrow r + \gamma V_\phi(s') - V_\phi(s)$ ;

9      $\phi \leftarrow \phi + \beta \delta \frac{\partial}{\partial \phi} V_\phi(s)$ ;

10     $\theta \leftarrow \theta + \alpha \lambda \delta \frac{\partial}{\partial \theta} \log \pi_\theta(a|s)$ ;

11     $\lambda \leftarrow \gamma \lambda$ ;

12     $s \leftarrow s'$ ;

13 until  $s$  为终止状态;

14 until  $\theta$  收敛;

输出: 策略  $\pi_\theta$

---

图 6.9: 演员-评论员算法

在 PG 策略中, 如果我们用 Q 函数来代替 R, 同时我们创建一个 Critic 网络来计算 Q 函数值, 那么我们就得到了 Actor-Critic 方法。Actor 参数的梯度变为:

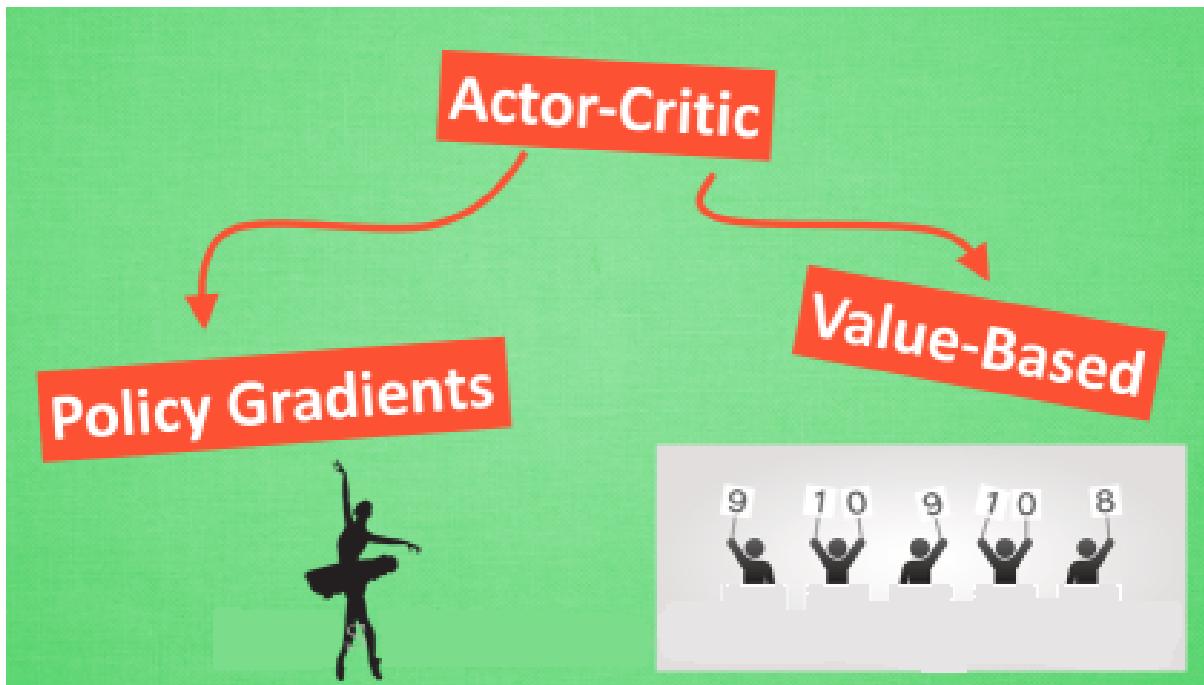
$$\nabla \bar{R}_\theta = \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} Q^{\pi_\theta}(s_t^n, a_t^n) \nabla \log p_\theta(a_t^n | s_t^n)$$

此时的 Critic 根据估计的 Q 值和实际 Q 值的平方误差进行更新, 对 Critic 来说, 其 loss 为:

$$loss = \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} (r_t^n + \max_{a_{t+1}^n} Q^{\pi_\theta}(s_{t+1}^n, a_{t+1}^n) - Q^{\pi_\theta}(s_t^n, a_t^n))^2$$

### 6.11.1 原理概述

Actor-Critic 算法分为两部分，我们分开来看 actor 的前身是 policy gradient 他可以轻松地在连续动作空间内选择合适的动作，value-based 的 Qlearning 做这件事就会因为空间过大而爆炸，但是又因为 Actor 是基于回合更新的所以学习效率比较慢，这时候我们发现可以使用一个 value-based 的算法作为 Critic 就可以实现单步更新。这样两种算法相互补充就形成了我们的 Actor-Critic.



Actor 基于概率选行为，Critic 基于 Actor 的行为评判行为的得分，Actor 根据 Critic 的评分修改选行为的概率。

在演员-评论员算法中的策略函数  $\pi_\theta(s, a)$  和值函数  $V_\phi(s)$  都是待学习的函数，需要在训练过程中同时学习。

一方面，更新参数  $\phi$  使得值函数  $V_\phi(s_t)$  接近于估计的真实回报  $G(r_t : T)$ ，

$$\min_\phi (G(r_t : T) - V_\phi(s_t))^2$$

一方面，将值函数  $V_\phi(s_t)$  作为基线函数来更新参数  $\theta$ ，减少策略梯度的方差。

$$\theta \leftarrow \theta + \alpha \gamma^t (G(r_t : T) - V_\phi(s_t)) \frac{\delta}{\delta \theta} \log \pi_\theta(a_t | s_t)$$

在每步更新中，演员 根据当前的环境状态  $s$  和策略  $\pi_\theta(a|s)$  去执行动作  $a$ ，环境状态变为  $s$ ，并得到即时奖励  $r$ 。评论员（值函数  $V_\phi(s)$ ）根据环境给出的真实奖励和之前标准下的打分

$(r + \gamma V_\phi(s'))$ , 来调整自己的打分标准, 使得自己的评分更接近环境的真实回报。演员则根据评论员的打分, 调整自己的策略  $\pi_\theta$ , 争取下次做得更好。开始训练时, 演员随机表演, 评论员随机打分。通过不断的学习, 评论员的评分越来越准, 演员的动作越来越好。

### 6.11.2 Advantage Actor-Critic(A2C)

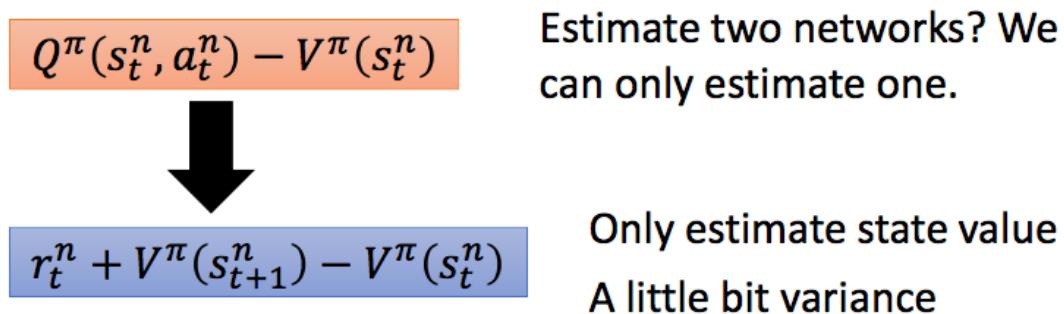
原理: <https://www.jianshu.com/p/428b640046aa>

A2C 代码的实现地址为:[https://github.com/princewen/tensorflow\\_practice/tree/master/RL/Basic-A2C-Demo](https://github.com/princewen/tensorflow_practice/tree/master/RL/Basic-A2C-Demo)

我们常常给 Q 值增加一个基线, 使得反馈有正有负, 这里的基线通常用状态的价值函数来表示, 因此梯度就变为了:

$$\nabla \bar{R}_\theta = \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} (Q^{\pi_\theta}(s_t^n, a_t^n) - V^{\pi_\theta}(s_t^n)) \nabla \log p_\theta(a_t^n | s_t^n)$$

但是, 这样的话我们需要有两个网络分别计算状态-动作价值 Q 和状态价值 V, 因此我们做这样的转换:



$$Q^\pi(s_t^n, a_t^n) = E[r_t^n + V^\pi(s_{t+1}^n)]$$

$$Q^\pi(s_t^n, a_t^n) = r_t^n + V^\pi(s_{t+1}^n)$$

这样会增加一定的方差, 不过可以忽略不计, 这样我们就得到了 Advantage Actor-Critic 方法, 此时的 Critic 变为估计状态价值 V 的网络。因此 Critic 网络的损失变为实际的状态价值和估计的状态价值的平方损失:

$$loss = \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} (r_t^n + V^\pi(s_{t+1}^n) - V^\pi(s_t^n))^2$$

### 6.11.3 Asynchronous Advantage Actor-Critic(A3C)

A3C 代码的实现地址为:[https://github.com/princewen/tensorflow\\_practice/tree/master/RL/Basic-A3C-Demo](https://github.com/princewen/tensorflow_practice/tree/master/RL/Basic-A3C-Demo)

直接更新策略的方法，其迭代速度都是非常慢的，为了充分利用计算资源，又有了 Asynchronous Advantage Actor-Critic 方法，拿火影的例子来说，鸣人想要修炼螺旋手里剑，但是时间紧迫，因此制造了 1000 个影分身，这样它的学习速度也可以提升 500 倍

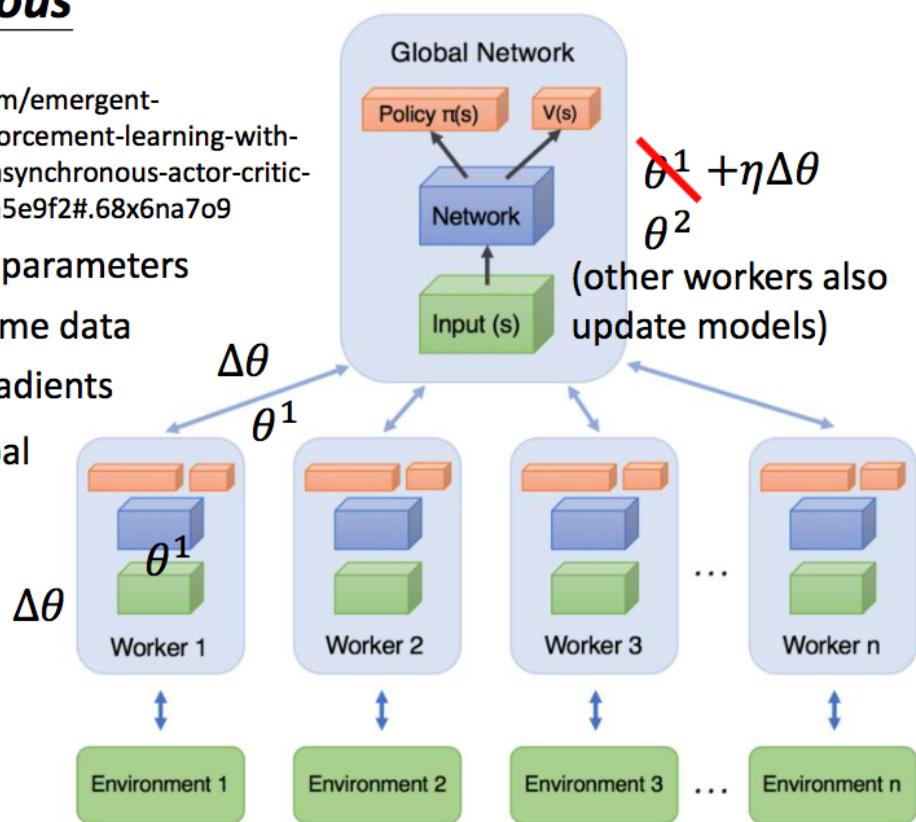
A3C 的模型如下图所示：

### Asynchronous

Source of image:

<https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-8-asynchronous-actor-critic-agents-a3c-c88f72a5e9f2#.68x6na7o9>

1. Copy global parameters
2. Sampling some data
3. Compute gradients
4. Update global models



可以看到，我们有一个主网络，还有许多 Worker，每一个 Worker 也是一个 A2C 的 net，A3C 主要有两个操作，一个是 pull，一个是 push：

- pull：把主网络的参数直接赋予 Worker 中的网络
- push：使用各 Worker 中的梯度，对主网络的参数进行更新

## 6.12 ACER

## 6.13 DPG、DDPG-Deep Deterministic Policy Gradient

是 Actor-Critic 和 DQN 算法的结合体.

连续动作空间 & Actor-Critic (off-policy) & model free

### 6.13.1 基础

<https://www.jianshu.com/p/6fe18d0d8822>

### 6.13.2 多智能体强化

<https://www.jianshu.com/p/4e4e35d80137>

## 6.14 TRPO

## 6.15 PPO-Proximal Policy Optimization

<https://www.jianshu.com/p/9f113adc0c50>

是对 Policy Gradient, 即策略梯度的一种改进算法。Importce Sampling 的方法，将 Policy Gradient 中 On-policy 的训练过程转化为 Off-policy，即从在线学习转化为离线学习。

### 6.15.1 原理

### 6.15.2 实现

## 6.16 模仿学习

在强化学习的经典任务设置中，机器所能获得的反馈信息仅有决策后的累积奖赏，但在现实任务中，往往能得到人类专家的决策过程范例，例如在种瓜任务上能得到农业专家的种植过程范例。从这样的范例中学习，称为“模仿学习”(imitation learning)。

强化学习任务中多步决策的搜索空间巨大，基于累积奖赏来学习很多步之前的合适决策非常困难；而直接模仿，人类专家的“状态-动作对”明显显著缓解这一困难。我们称其为“直接模仿学习”。

### 6.16.1 原理

### 6.16.2 实现

## 6.17 Multi-Agent Reinforcement

<http://wnzhang.net/tutorials/marl2018/index.html>

## 第七章 统计数学方法-李航



# 第八章 神经网络与机器学习-Simon Haykn

## 8.1 什么是神经网络

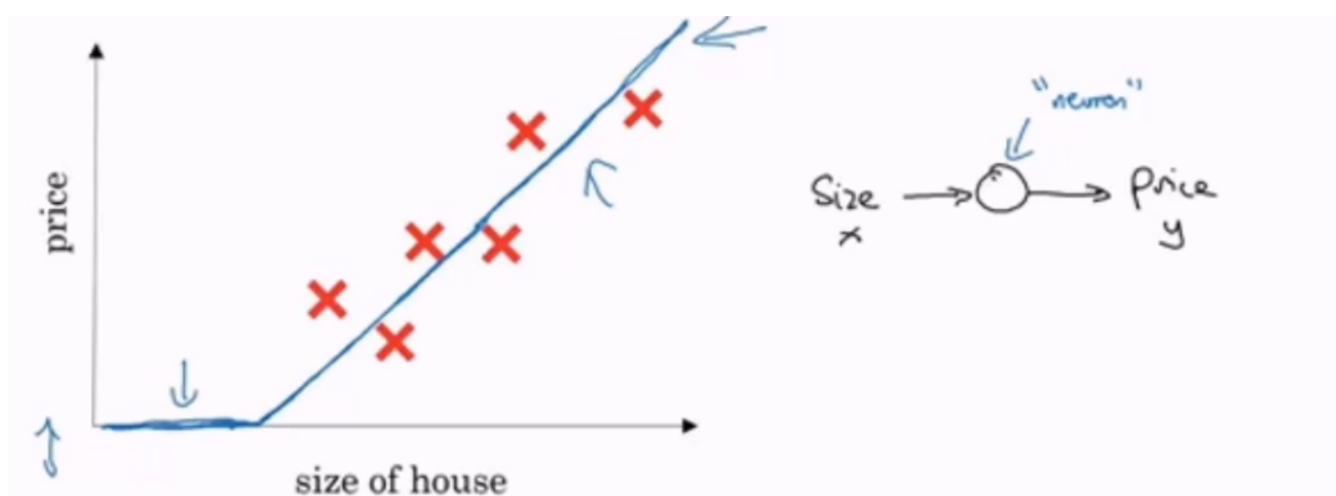


图 8.1: 单神经元

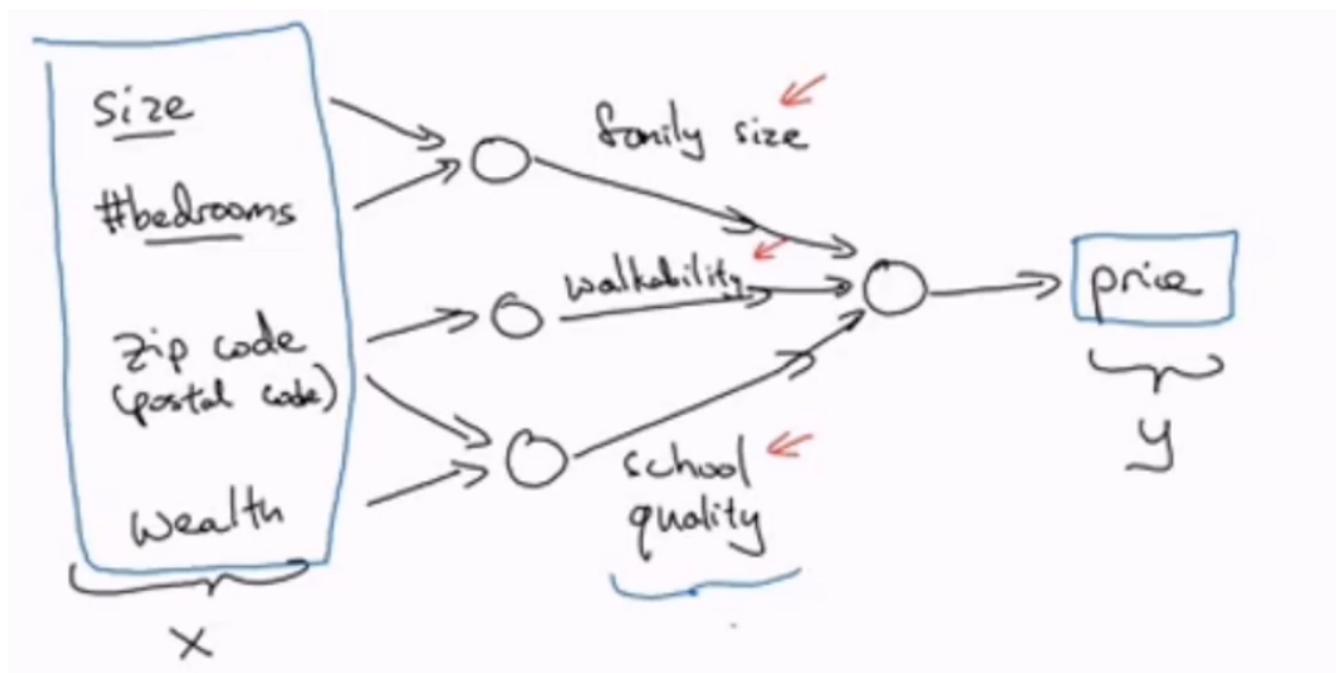


图 8.2: 神经网络



# 第九章 机器学习-周志华、吴恩达



## 第十章 深度学习-花书



# 第十一章 贯 通-O'Reilly.Hands-On.Machine.Learning



## 第十二章 案例分析



## 第十三章 应用