Reinforcement Learning总结

# 背景

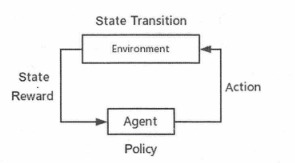
目前行走样机在使用过程中，自动化智能化程度还不够高，许多参数还需要手动调整，所以希望使用强化学习的方式找到最优参数，或者训练模型去在线优化参数，达到快速智能匹配的效果。在行走设备实际运行中，可能需要对患者的每一步进行打分，然后强化学习基于环境反馈的分数reward来寻找最优参数或者训练优化模型。

为实现该目标，需要结合我司的具体业务需求，对RL（Reinforcement Learning）进行相关学习探索。本文档后续章节会依次介绍RL思想、不同的算法分支的特点及着重解决的问题，以及相关实例展示，最后会对RL在机器人领域的应用进行展望。

因为我们的应用于机器人，获取数据和训练RL模型的成本会很高，所以在实现RL的过程中需要重点关注强化学习所需要资源、计算量等指标。

# RL简介

RL是机器学习中的一个领域，强调如何基于环境而行动，以取得最大化的预期利益。从本质上看，强化学习是一个通用的问题解决框架，其核心思想是 Trial & Error，让被实验者通过与环境的不断交互和试错来学习，通过reward影响被实验者的行为。为了实现这个目标，实验者需要构建一个完整的实验环境，通过给予被实验者一定的观测和回报，让其产生实验者想要的结果。实验流程如下图所示。



图表 1 RL过程表示

|  |
| --- |
| **Agent**: 实验的主角，一般翻译为智能体；  **Environment**：实验的操控者，一般翻译为系统环境；  **Observation/State**: 环境的状态或者对环境的观测  **Action**： Agent采取的行动  **State Transition** : Agent 对Env采取行动后会引起Env的状态state变化，状态变化的过程就是State Transition；  **Reward**： 代表回报或奖励，代表环境(Env)对Agent所采取Action的回报；  **Policy**： 代表策略，RL算法为了最大化整体的回报，需要在不同的state下采取不同的action，如何采取action由Policy决定，Policy是RL算法要学习的目标。 |

为了最大化整体回报，通常Agent需要学习不断试错，因为根据环境状态给出行动的Agent 有时会收到较多回报，有时回报较少，还可能收到负的回报，Agent 自己并不知道究竟怎样才能获得最多的回报，“实验者”也不会告诉Agent。所以Agent 需要根据回报的多少不断地调整自己的策略，从而尽可能多地获得回报。这个过程中Agent 需要不断尝试，尝试应对状态的各种可能的行动，并收集对应的回报，只有收集到各种反馈信息，才能更好地完成学习任务。因此这是一个不断试错（Trial and Error ）的过程，只有经过尝试、遇到失败，才能获得最终的成功。

## 评价指标：

除了一些常见的衡量指标（**算法的效果、计算时间、稳定性**和**泛化性**等），我们还要重点考虑**“学习时间”**。由于学习和尝试相关，所以这个指标一般也看作尝试和探索的次数。如果一个算法需要尝试的次数比较多，我们一般认为算法要花费的时间比较长；如果一个算法需要尝试的次数比较少，那么相对来说花费的时间比较短。站在机器学习的角度，我们可以认为尝试的样本本身会影响学习的时间，例如样本的代表性、重合度等。

对强化学习来说，由于学习本身的特点，我们需要考虑训练**样本的使用率(Sample Efficiency)**。不同算法对样本的重复使用次数不同，有的算法对于尝试的样本只能使用一次，而有的算法可以反复使用同样的样本。训练样本的使用率会直接影响学习时间。前面提到Agent 的学习样本要通过自身与环境的交互得到，而这个过程是要花费时间的。需要的样本量少，学习时间就可以缩短；反之学习时间会比较长。对计算机模拟的学习问题来说，样本量并不算个大问题，因为计算机可以在短时间内快速模拟出大量的样本；但是对于在真实场景进行训练的问题来说，产生样本意味着要在真实世界的时间尺度下进行交互，花费的时间会很长。为了一点效果的提升花费大量的时间，对我们来说有点得不偿失。因此，很多研究人员都在思考如何提高真实世界学习的速度，这就涉及提高样本利用率、迁移学习等内容。这样我们就了解了强化学习关注的两个目标：学习效果和学习时间。学习时间也成了算法十分看重的一个目标。

## 强化学习于有监督学习之间的区别：

监督学习的目标更明确，输入对应的是确定的输出，而且理论上一个输入只对应一个输出；而强化学习的目标没有这么明确，使当前状态获得最大回报的行动可能有很多。

总的来说，强化学习相比监督学习有两个优点：

( 1 ）定义模型需要的约束更少，影响行动的反馈虽然不及监督学习直接，却降低了定义问题的难度；

( 2 ）更看重行动序列带来的整体回报，而不是单步行动的一致性；

# 环境要求

python=3.7.7

gym==0.17.2

opencv-python==4.2.0.34

numpy=1.18.1

tensorflow=1.15.0

pygame==1.9.6

# RL算法分支

## 4.1 RL基本算法

### 4.1.1 马尔可夫决策过程

马尔可夫决策过程（Markov Decision Process , MDP ）是当前强化学习理论推导的基石，通过这套框架，强化学习的交互流程可以很好地以概率论的形式表示出来，解决强化学习问题的关键定理也可以依此表示出来。

我们用St 表示t 时刻游戏状态的观测值，用at 表示t 时刻选择的手法，那么过程就可以用一条状态－行动链条表示：

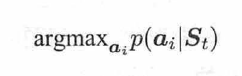


这个链条包含了两种状态转换，一种是从状态到行动的转换，由 agent 的策略决定的；另一种是从行动到状态的转换，由环境决定的。

第一种转换一一**策略**，指的是Agent 根据当前的状态选择“自认为”最好的行动方式。如果用严谨的方式表述，策略是一种映射，它将环境的状态值St 映射到一个行动集合的概率分布或概率密度函数上。可以认为Agent 会对每一个行动进行衡量，并最终选择评价最高的行动；对于每一个行动，Agent 都有－定的概率去执行，且行动的评价越高，行动产生的概率越大，形式化来说就是在时间t 的时刻，选择公式：



的结果作为行动。对其做一定的化简，假定状态之间无依赖，即当前时刻选择什么行动只和当前的状态有关，和前面的状态与行动无关，用到了序列的马尔可夫性，化简后的公式如下：



第二种转换--**环境的状态转换**。当Agent 完成行动后，环境会受到影响并完成状态的转换。下一步的状态只受前一步状态影响，不受更前面的状态影响，于是这里的状态转换就能以概率的形式表现为：



了解了这两个过程，就可以重新研究MDP 这个词了。马尔可夫决策过程包含以下

三层含义：

( I ）“**马尔可夫**”表示了状态间的依赖性。当前状态的取值只和前一个状态产生依赖，不和更早的状态产生联系。虽然这个条件在有些问题上有些理想，但是由于它极大地简化了问题，所以人们通常会选择使用它。

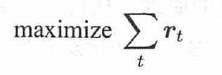
( 2 ）“**决策**”表示了其中的策略部分将由Agent 决定。Agent 可以通过自己的行动改变状态序列，和环境中存在的随机性共同决定未来的状态。

( 3 ）“**过程**”表示了时间的属性。如果把Agent 和环境的交互按时间维度展开，那么Agent 行动后，环境的状态将发生改变，同时时间向前推进，新的状态产生，Agent将获得观测值，于是新的行动产生，然后状态再更新。

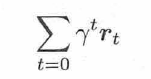
* **长期回报**

游戏的关键在于策略，也就是如何做出决策与执行行动。在理想状态下，每一个行动都要为最终的目标一一最大化长期回报努力，那么理论上只要能够找到一种方法，量化每一个行动对实现最终目标贡献的价值，Agent 就可以根据这些量化指标做出明智的判断。所以接下来一个重要的工作就是量化这些价值。

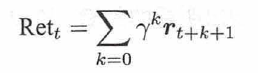
幸运的是，环境反馈的**reward**提供了局部的回报信息，将其扩展成为我们的目标，如下：



这个公式并不容易计算，它的困难反映在两方面，其中一个在计算的时间上。如果们可以在有限步内完成游戏，那么这个公式虽然复杂，但至少是可以计算的；为了解决这个问题，使这个无穷数列的和收敛，我们要降低未来回报对当期的影响，也就是对未来的回报乘以一个打折率，使长期回报变得更有意义。所以修正后的公式变为：

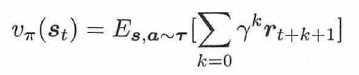


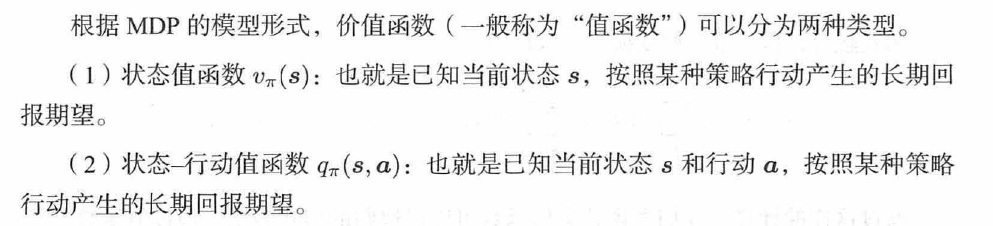
这个打折率一般来说小于1 ，这样一来长期回报的这个数列的和就变得有界了，也就可以计算出具体的值了。于是我们正式定义另外一个变量一一**长期回报**。所谓长期回报，是将当前状态之后所有的回报取出，分别乘以对应的打折率，然后加起来得到一个汇总的值。

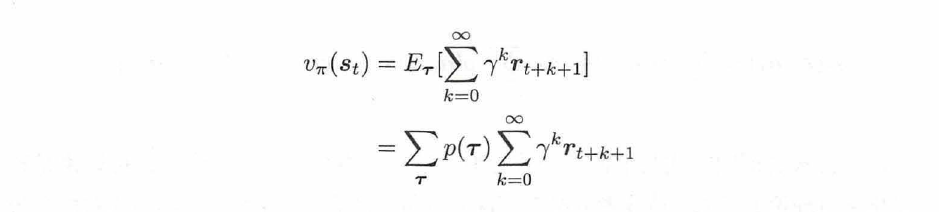


* **值函数**

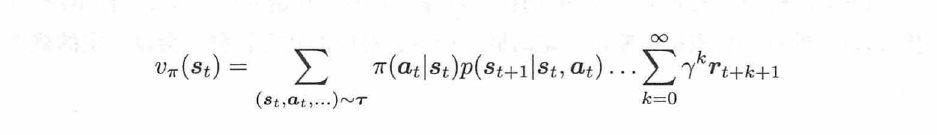
基于MDP 形成的交互序列，从状态到行动的转换可以通过策略确定，而由于环境的原因，从行动到下一个状态的转换有时并不能确定。因此在衡量价值时，我们需要考虑每一种状态转换的影响，这就需要基于状态转换求解长期回报的期望。令τ为根据策略和状态转换采样得到的序列，那么价值的公式定义就可以写成：



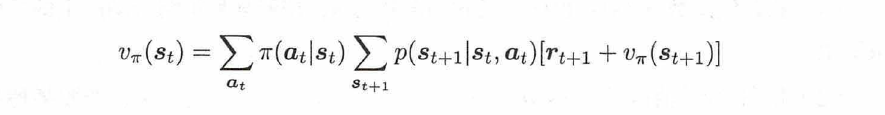




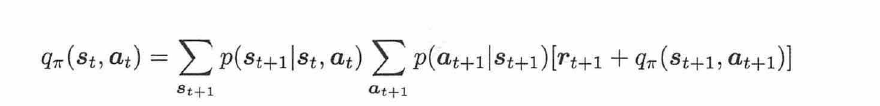
其中T 表示从状态St 出发的某条路径。由于我们将强化学习的过程表示成马尔可夫决策过程，于是路径部分可以展开为：



经过换元法处理得到**贝尔曼公式（Bellman Equation，以下简称Bellman公式）**，是后面章节进行策略求解的基础公式之一：

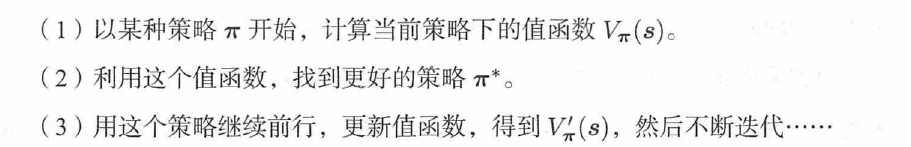


同理**状态-行动值函数**的公式可以变换为：



### 4.1.2 策略迭代

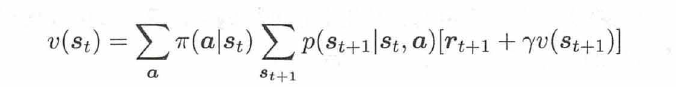
如果想知道最优的策略，就需要能够准确估计值函数。然而想准确估计值函数，又需要知道最优策略，数字才能够估计准确。所以实际上这是一个“鸡生蛋还是蛋生鸡”的问题。碰上这样无解的问题，往往需要采取一些“曲线救国”的方法。那么，能不能把这个问题考虑成一个迭代优化的问题，通过一轮一轮的计算逐渐接近最优的结果呢？答案是可以的。于是我们提出如下计算思路：



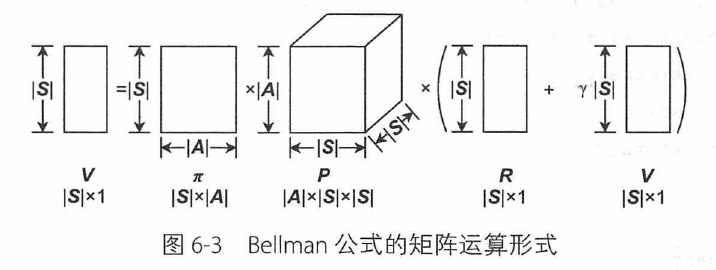
这样经过若干轮计算，如果一切顺利，我们的策略会收敛到最优的策略，如果能够判断出策略已经收敛，问题也就得到了解答。当然，目前我们还没有理论证明，因此这个思路也可能是错的，但我们还是先试着实践这个思路。

* **矩阵求解：**

想要求解值函数，就必须用到Bellman 公式：

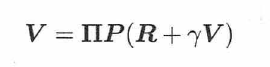


对于这个公式，我们利用了时间不变性这样的假设，于是等号左右两边的值函数值应该是相等的，所以这个问题可以采用解方程的方法求解。上面的公式是采用连加的形式描述的，我们现在将其变成矩阵运算的形式，于是等式变成了图6-3 所示的形式：



如果用V 表示状态值函数的向量，H 表示策略矩阵，P 表示状态转移矩阵，R 表

示回报向量，那么**矩阵版的Bellman 公式**为：



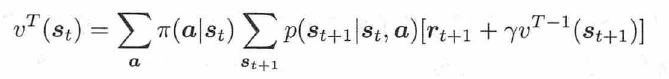
求解，有：



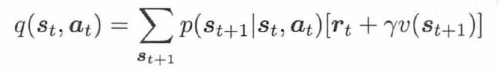
只要（1 一γIIP)-1 的逆矩阵存在，状态价值就可以解出来。实际中由于矩阵数值的限制，逆矩阵是存在的，所以这种方法可以求解。但是对于一些状态和行动比较多的问题，采用这种方法求解的复杂度偏高，因此实际中大家不用这种方法，而是采用迭代的方法进行计算。

* **迭代求解**

那么迭代的方法如何计算呢？在其他数值计算算法中，我们会利用旧参数迭代更新新参数的方法，例如梯度下降法，这里也将采用类似的方法。我们将上面的Bellman公式左右两边的值函数赋予不同的值，右边是当前值，左边是新迭代的值，假设迭代轮数为｛l, 2, 3, .. , T｝，那么迭代更新的公式就变为：



由于有γ的存在，每个状态的价值最终将得到收敛，完成了这一步，下一步就是根据前面的状态值函数计算状态一行动值函数：



完成计算后，就可以根据同一状态下的行动价值更新策略：



每一轮迭代结束，策略都进行了一次更新，当策略没有更新时，迭代结束。前面提到算法的两个部分也分别被称为：**策略评估部分（Policy Evaluation ）**和**策略提升部分（Policy Improvement ）**。

* **迭代求解的收敛性**

从上面的分析中我们发现，策略评估部分在逻辑上没有问题，值函数会随着迭代逐渐收敛，主要的问题出现在策略提升上，我们能不能证明这种改进方案一定得到最优结果呢？当然可以。如果把每一轮迭代生成的策略形成一个策略组，并以迭代轮数进行编号，那么可以得到一个策略列（叫｝，我们不但可以证明随着t 的增大，这个策略列依值函数最终收敛到最优的策略扩，也就是说



而且可以证明这个策略列依值函数一致收敛到最优策略，即对于任意的数ε，我们

都存在一个k ，使得当t > k 时：



换句话说，随着迭代的进行，策略不断趋近于最优，每一轮迭代的策略都不会比前

一轮迭代的策略差。证明过程略。

### 4.1.3价值迭代

#### N 轮策略迭代

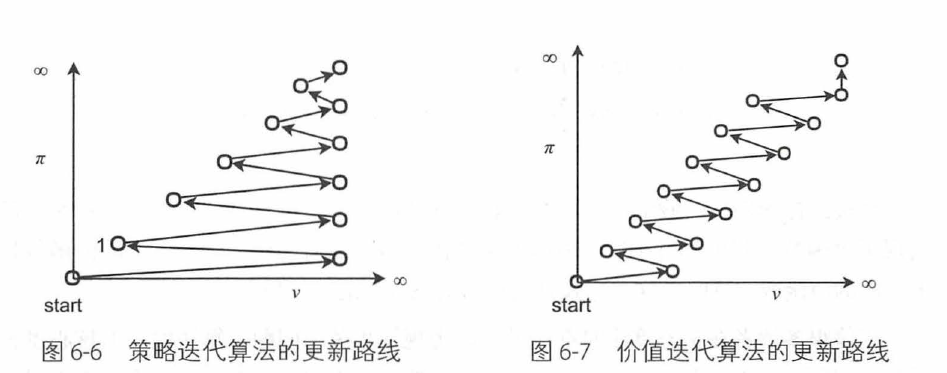
从算法中可以看出，算法的主要时间都花费在策略评估上，对一个简单的问题来说，在策略评估上花费的时间不算长；但对复杂的问题来说，这个步骤的时间实在有些长。一个最直接的想法就是一一我们能不能**缩短在策略评估上花的时间呢**？例如，我们大概估计出当前策略下每个状态的值函数就好，虽然不够精确，但这样也许已经足以帮助我们找出最优的策略了，再做更精细的评估实际上并不必要。这就是本节的主角一一**价值迭代法的思想来源**。

经过蛇棋实验发现，随着策略评估的迭代轮数不断降低，算法的总迭代数在升高。当策略评估的迭代轮数为94 时，算法总迭代数为3 ；而策略评估的迭代轮数为l 时，算法总迭代数变为7 ，但是这样的改变并不会影响最终的结果，反而使它成了这些实验中最快的算法。这个算法就是评估迭代轮数为1 的策略评估算法。

实际上，我们可以认为评估迭代轮数为1 的策略迭代法很像基于价值的迭代法。为什么呢？在策略迭代算法中，**每一轮迭代过后，状态价值得到了收敛，但是当前策略不一定是最优的**，直到策略不再发生变化，训练才会结束。策略迭代的过程如图6-6 所示，其中横轴表示值函数的收敛效果，数值到达∞时完成收敛，纵轴表示策略的优异度，数值到达∞时策略到达最优。

而对于l 轮迭代来说，在**策略达到最优前，值函数都不会收敛**。**如果值函数收敛**，则说明值函数没有发生改变，**策略也就随之收敛**了。对应的迭代过程图如图6-7 所示。

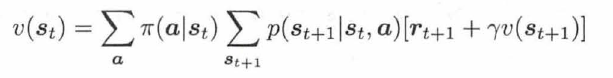
从图6-7 所示的更新路线可以看出，与其说这是更新策略，不如说是更新价值。当价值迭代完成后，策略迭代也就随之完成，所以我们可以将迭代的重点放在价值上。



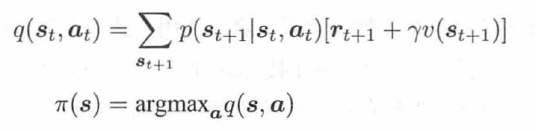
#### 从动态规划看N轮迭代

价值迭代算法看上去似乎是一个“贪心”版的策略迭代算法，它的收敛性质能否保证呢？回答这个问题需要从“贪心”的对立面－一动态规划解释。

前面我们一直站在策略的角度分析，并且把价值迭代法定义为一轮迭代的策略迭代法，那么对于一轮迭代的算法，它的形式能否做一个改变呢？我们知道**策略评估**这一步完成了下面的计算：

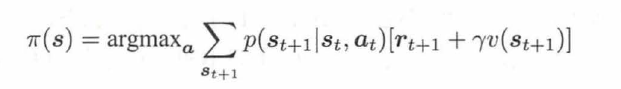


**策略改进**完成了策略改进的计算：

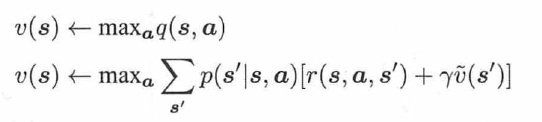


能否将这三个步骤结合，成为一个关于值函数的更新步骤呢？首先**将后两个公**

**式合井，得到：**



由于最优策略的存在，实际上策略最终的选择是单一的。也就是说，对于每一个状态，最优策略会采取某一种行动，这种行动不会比其他行动差，所以我们可以得到**状态值函数的更新方法**：



看到公式的左右两边都出现了值函数，其中等号左边的值函数是迭代更新后的值函数，等号右边的值函数是上一轮的值函数。这时，这个公式就展示出动态规划的“气质”。

**动态规划**的思路也基本相同，它有两个核心特点：**最优子结构**和**重复子结构**。

**最优子结构**：指一个子问题的最优解是可以得到的。对应蛇棋的问题，可以理解为是“从某个位置出发行走一步能够获得的最大奖励”的问题，由于只走一步，这个问题很容易计算。

**重复子结构**，是指一个更大的问题是由一些小问题组成的，而求解不同的大问题时可能会用上同一个子问题，子问题被重复利用，计算量也就减少了。对应蛇棋的问题，可以理解为是“从某个位置出发行走两步能够获得的最大奖励”的大问题，利用前面已经得到的子问题，这个大问题可以用伪代码表示：

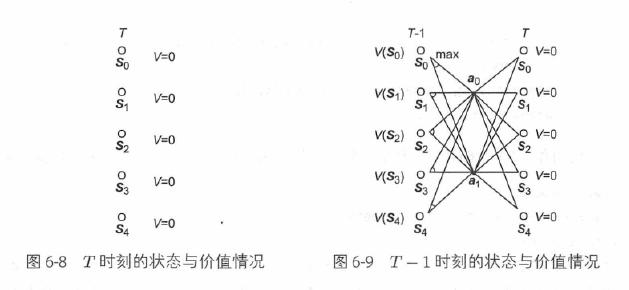
*“某个位直走两步的最大奖励”＝max( ［这一步的奖励＋从到达位直出发走一步获得的最大奖励for 走法in 可能的走法］）*

可以看出，这个公式用到了最优子结构的性质。由于打折率的存在，值函数的绝对值存在上界，值函数在这样的更新中最终会得到收敛，所以这个方法是可行的。

想要理解，**价值迭代和动态规划的关系**，除了使用公式表示外，还可以使用图像。这里需要对问题做一个假设：假设所有的问题都存在一个“终点时刻”，对有明确终止条件的问题来说，这样的终点当然存在；但对没有终点状态的问题来说，可以假设某一个时刻为终点，例如时间的终点、外物灭亡时刻的终点……那么我们可以记录这一时刻为T ，所有状态的价值为0 ，由于没有后面的行动，此刻价值设为0 也是正常的。T 时刻的状态与价值情况如图6-8 所示。知道了时刻T 的情况，我们就能得到T-1 时刻从每个状态出发能获得的最优价

值。对于每一个状态，我们遍历所有的行动，通过Bellman 公式，可以得到这一时刻的

最优长期回报，如图6-9 所示。再向前一步，来到T-2 时刻，我们同样可以利用T - l 时刻的结果计算出T-2时刻的最优价值。依此类推，我们可以向前推进无数步，直到价值收敛，此时的收敛值就是从当前位置到时间终结点的最优价值，也就是价值迭代法要求出的结果。



价值迭代和策略迭代得到了相同的结果，但它比策略迭代快一些。价值迭代的速度没有中的一轮策略评估迭代速度快，一轮策略评估迭代的时间为0.023 s 。这说明两个算法仍然存在一定的区别，所以笔者提到两者的思想相近，但并不相同。**那么它们的区别是什么呢？我们还能不能让训练速度更快呢？**

### 4.1.4 泛化迭代

* **两个极端**

在上一小节我们留了一个悬念：为什么价值迭代法的运算速度不如一轮策略迭代法的速度快呢？本节将解答这个问题。在解答问题前，我们先对比策略迭代法和价值迭代法的特点。

**策略迭代法**的中心是策略函数，它通过反复执行“策略评估＋策略提升”两个步骤使策略变得越来越好；**价值迭代法**的中心是值函数，它通过利用动态规划的方法迭代更新值函数，并最终求出策略函数。由此可以看出两者有如下特点。

( 1 ）两个方法最终都求出策略函数和值函数。

( 2 ）最优的策略函数都是由收敛的值函数得到的。

( 3 ）值函数通过Bellman 公式收敛。

由此发现一个关键：两者**都需要训练和更新策略函数和值函数**，只是**侧重点不同**。**策略迭代**的核心是策略，为了提升策略，值函数可以求解得准确，也可以求解得不那么准确；**价值迭代**的核心是价值，算法的核心部分根本没有出现与策略有关的内容，直到算法最后通过值函数求出策略。

两种方法都十分看重自己关心的那部分，可以选择忽略另一部分，因此可以看出两个方法都比较极端。既然找到了两个极端的方法，可不可以找到两种方法的中间地带呢？当然可以，这就是本节要介绍的**广义策略迭代法（Generalized Policy Iteration ）**。

* **广义策略迭代法**

所谓的广义策略迭代法，就是定义一个迭代算法族，其中的算法都是由策略迭代和价值迭代算法组合而成的。组合的方法有很多，可以形成的方法也有很多，而前面提到的两种算法是广义策略迭代法的一种特例。由于其中的算法很多，这些算法中很有可能存在一些比前面两种算法速度更快的方法，例如前面多次提到的一轮策略评估迭代法。

* 分析为什么价值迭代的方法不是最快的

由于蛇棋问题的状态是离散的，对应的行动也是离散的因此策略所在的空间也是离散的，这个问题的策略价值坐标轴实际上应该如图6-11 所示。由于策略空间是离散的，而值函数空间是连续的，同样的策略可能对应着一定范围内的值函数，因此最优策略也一定对应着一定范围的值函数。所以在使用价值迭代法时，需要将值函数更新到最优，所以为了使值函数收敛，它花费了一定的时间；而如果我们不去追求值函数的最优，当值函数接近最优时，使用它进行策略改进，也许就可以得到最优策略了，这样实际上可以节省一部分时间。

### 4.1.5 小结

4.1小节介绍了MDP 、策略迭代法、价值迭代法和广义策略迭代法相关的知识，要点如下：

( 1 ) MDP 包含了状态与行动交替的序列，我们可以将很多问题转化成这样的交互形式。

( 2 ）值函数表示了长期回报的期望值。常见的有状态值函数V(s）和状态一行动值函数q(s, a）。

( 3 ）策略迭代法通过交替执行策略评估和策略提升两个步骤完成最优策略的学习。

( 4 ）价值迭代法通过迭代更新值函数使值函数达到最优，然后更新策略函数。

( 5 ）泛化迭代融合了两种迭代方式。

## 价值迭代

在4.1节中基于马尔可夫决策过程的强化学习框架，我们学习了策略迭代、价值迭代和泛化迭代算法。我们曾提到一个很关键的前提条件：**Agent 知道环境的状态转移概率**。在很多实际问题中，**状态转移的信息P(St+1 |St, at)无法获得**。一般来说，我们将知晓状态转移概率的问题称为“**基于模型**”的问题(Model-based Problem)，将不知晓的称为“**无模型**”问题(Model-free Problem)。本章将基于无模型问题，将介绍新的算法：SARSA, Q-Leaming和基于深层模型的Deep Q Network。

我们不知道状态转移概率，就无法直接执行策略评估，即不能使用Bellman 公式进行值函数的更新。为了解决这个问题，我们需要使用其他方法完成策略评估，这就要追溯回强化学习自身的特点：不断试错，也就是通过尝试与环境交互来解决策略评估的问题。实际上，这个不断尝试的思路和人类的一些学习方式比较接近，我们可以列出新算法的大体思路。

|  |
| --- |
| ( 1 ）确定一个初始策略（这和前面的算法一致）。  ( 2 ）用这个策略进行游戏，得到一些游戏序列（Episode): {s1, a1, s2, a2, …, Sn, an}。  ( 3 ）一旦游戏的轮数达到一定数目，就可以认为这些游戏序列代表了当前策略与环境交互的表现，就可以将这些序列聚合起来，得到状态对应的值函数。  ( 4 ）得到了值函数，就相当于完成了策略评估的过程，这样就可以继续按照策略迭代的方法进行策略改进的操作，得到更新后的策略。如果策略更新完成，则过程结束；否则回到步骤2。 |

上面的流程十分清晰地介绍了学习的过程，此时学习的关键就落在了下面两个问题上：

|  |
| --- |
| ( 1 ）如何得到这些游戏序列。  ( 2 ）如何使用序列进行评估。 |

本章将重点解决这两个问题。接下来介绍两个算法：**蒙特卡罗法（Monte Carlo Method）**和**时序差分法（Temporal Difference Method）**。

### 蒙特卡罗法（Monte Carlo Method）

在前面的章节里，我们曾介绍当环境信息，也就是状态转移概率已知时，可以使用Bellman 公式，通过不断迭代得到 *状态-行动值* 函数：



然后通过值函数进行策略改进。而在无模型问题中，状态转移概率将无法知晓。于是我们需要把公式转变为：



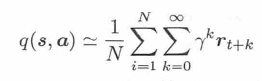
蒙特卡罗法，它是一种通过随机采样估计期望值的方法，假设我们通过一些方法，从状态s 和行动a开始不断地与环境交互，得到了大量的样本序列：



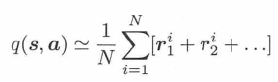
对应的回报序列为：



通过这些样本序列逼近真实的期望值，那么公式就可以近似等于：

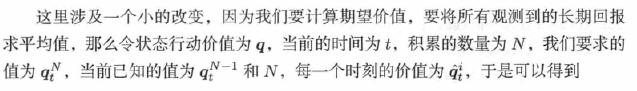


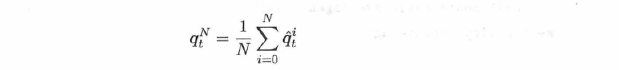
如果这个序列是有限的，我们也可以直接使用序列产生的回报和作为价值：

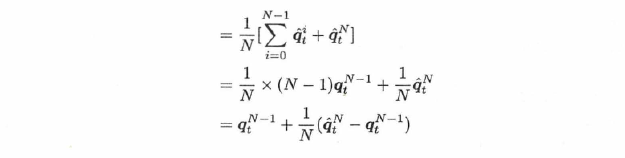


这样就可以得出策略评估的结果。总结算法过程如下：

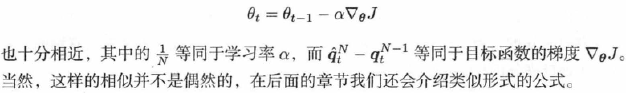
|  |
| --- |
| * 让Agent 和环境交互后得到交互序列。 * 通过序列计算出每一时刻的价值。 * 将这些价值累积到值函数中进行更新。 * 根据更新的值函数更新策略。 |







这样每一时刻我们都可以求出当前所有观测值的平均数，而且这个公式和常见的梯度下降法的更新公式



#### 探索与利用 & ε-greedy算法

当模型未知时，我们只能通过与环境交互得到交互序列。这里就存在一个问题：**交互序列可以真实反映状态转移概率吗**？

在一些极端条件下这是可能的。我们假设通过与环境交互，所有的状态和行动组合全部被Agent 经历过，而且每种情况都经历了足够多次，那么通过统计计算这些序列的长期回报，可以得到接近状态转移概率已知时得到的值函数。

但是对常见的问题来说，实现这个目标比较困难。主要原因在于可行的状态行动序列太多，想要对其进行**一一尝试不太现实**，有很多问题，它们的**状态是近乎无限**的。以Atari 游戏为例，它的游戏界面是指定尺寸的图像，但并不是所有指定尺寸的图像都是Atari 的游戏。因此，我们可以把所有Atari 游戏画面所在的空间想象成指定尺寸内所有图像空间的一个子空间，更麻烦的是，这个子空间的概率分布很难得到，因此遍历所有可能的画面是不可行的，我们必须采取其他方法。

为了达到和基于模型的算法接近的效果，**首先要确保当前的问题有遍历所有状态-行动对的可能**。一般来说，我们认为问题的开始状态是一个集合，我们从中任选一个状态开始，并沿着这个状态序列执行所有可行的行动，那么当这个流程被反复执行到一定次数时，问题的所有状态－行动对一定都会被经历。这段话听上去很拗口，又像是一段废话，但这是**解决问题的基本条件**。如果由于观察信息的损失的状态，导致无模型算法存在无法被感知和经历，那么无模型算法就很难得到最优解了。

既然我们认为这个前提一定满足，那么只要尽可能地让每一个状态下的行动都有概率执行，是不是就可以弥补模型未知的不足呢？这里可以做一个尝试，为了“雨露均沾”，我们必须让其他没有被选为最优策略的行动也参与到与环境交互的过程中，这样才能让每一个q(s, a)都有值，这样做策略改进也就更有意义。

基于这个想法，我们改进了策略模块，这里采用一种叫**ε－greedy** 的算法，首先随机生成一个0～1 的数，然后用这个随机数进行判断，如果随机数小于某个值ε ，就采用完全随机的方式产生行动，此时每个行动产生的概率是一样的；如果随机数不小于某个值ε，就选择当前的最优策略。可以想象，在使用了这个算法后，很多由于策略而无法到达的状态也可以到达了，这样对状态价值的估计也会更准确。

**ε-greedy 算法**实际上是在解决强化学习中的一个经典问题：**探索与利用**。这是两种与环境交互的策略。所谓的**探索**是指不拘泥于当前的表现，选择一些不同于当前策略的行动；**利用**就是持续使用当前的最优策略，尽可能地获得更多的回报。

我们假设总的资源有限，例如时间有限，可以与环境进行交互的轮数有限，**不能无止境地探索**。所以，想要像前面所说等到所有的状态一行动对被多次经历是很难实现的。

如果**只探索不利用**，对每一个行动分配等量的时间，那么最优的策略可能没有经过充分的尝试，得到的值函数不够准确，会使玩家最终丧失选择它的机会，而对非最优行动花费了大量的时间，实际上也是一种浪费。如果**只利用不探索**，就容易满足于当前得到的行动，陷入次优解，与最优解擦肩而过。

**ε-greedy** 就是这样一种方法，它比较好地在**两种策略间**找到了**平衡**，用较高的性价比完成了交互的过程，并可以使用它求解出接近真实情况的价值。

#### 蒙特卡罗的方差问题

蒙特卡罗法可以比较好地解决无模型场景下的问题，在配合了ε－greedy 策略后，它的效果和策略迭代法十分相近。当然，它也不是完美的，比如：**估计值的方差较大**。

蒙特卡罗通过与环境交互得到序列的回报信息，然后用这些回报信息求平均，就可以得到估计的值函数。这个方法从理论上看是没有问题的，它能够**成立的根本原因**在于**概率论中的大数定理**。大数定理也是没有问题的，可是大数定理**只说明了统计量的收敛性**，并**没有说明收敛的速度**。很显然，如果采样集合的方差比较大，那么想让均值收敛就需要更长的时间；如果采样序列的方差较小，那么收敛的速度也会相应地加快。

但是在数据量足够大时，我们也可以看到它的**优点**：**对期望值的估计是无偏的**，这一点可以很轻易地证明。

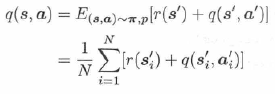
所以最终我们可以得到：蒙特卡罗法在估计价值上具有无偏差但方差较高的特点。当交互的数据量足够大时，蒙特卡罗法是有优势的，但是当交互的数据量不足时，它也会暴露出自己的缺点。

### 时序差分法与SARSA

为了使价值评估的方差进一步降低，本节将介绍另一种方法：**时序差分法（Temporal Difference ，简称TD 法）**。TD 法是一种**结合**了**蒙特卡罗法**和**动态规划法**的方法。从算法的主体结构来看，它同蒙特卡罗法类似，同样通过模拟交互序列的方式进行求解；但是从算法的核心思想来看，它同时用到了强化学习中的经典公式：Bellman 公式进行自迭代更新。前面提到*状态-行动值*函数的Bellman公式的形式为：



利用蒙特卡罗法，可以将公式变为：



TD 法和蒙特卡罗法有一些不同，蒙特卡罗法在估计价值时使用了完整序列的长期回报，而TD 法使用了当前回报和下一时刻的价值估计，但是两个方法的公式在形式上还是有些类似的。如果把TD 法像蒙特卡罗法一样写成迭代更新的形式，就可以得到**TD 法的基本形式**:



前面叙述的算法属于TD 法中的一种，被称为**SARSA 算法**，五个关键因子：S(当前状态）、A(当前行动)、R(模拟得到的奖励）、S(进入的下一个状态)和A(采取的下一个行动）。从上面的公式也可以看出，更新公式中确实只有这5 个变量。

**TD方法和蒙特卡罗法、动态规划法的关系:**

从**形式上**看:

* 蒙特卡罗法会通过大量的采样估计出值函数的统计量;
* TD法也同样需要采样这个过程；

从**更新公式**上看:

* TD 法通过下一时刻的价值更新前一时刻的价值，似乎是承认了下一时刻的回报和价值足够优秀，从而利用了这个最优子结构进行更新。这和动态规划的思想是一致的。

**TD 系列的方法会在方差方面控制得更好：**

* **蒙特卡罗**的计算方法由于使用了完整的采样得到了长期回报值，所以在价值的估计上的偏差更小，但同时它需要收集完整序列的信息，而序列存在一定的波动，所以价值的方差会比较大。
* **TD 方法**只考虑了当前一步的回报值，其余的计算均使用了之前的估计值，所以当整体系统没有达到最优时，这样的估计都是存在偏差的，但是由于它只估计了一步，所以它在估计值方面受到的波动比较小，因此方差也会相应减小许多。
* 蒙特卡罗法和SARSA 算法象征着**两个极端**：一个为了追求极小的误差使方差变大，一个为缩小方差使误差变大。

### QLearning

基于TD 思想的SARSA 算法，它的核心公式为：



接下来，我们要介绍的另一种TD 的算法叫作Q-Learning ，它的基本公式为：



**策略评估的方式**----**On-Policy与Off-Policy：**

* **On-Policy** 对值函数的更新是完全依据交互序列进行的，我们在计算时认为价值可以直接使用采样的序列估计得到；
* **Off-Policy** 在更新值函数时并不完全遵循交互序列，而是选择来自其他策略的交互序列的子部分替换了原本的交互序列。

**Q-Leaming算法分析：**

* 算法的主体框架：和蒙特卡罗法、SARSA 算法基本一致，策略改进部分完全相同。
* SARSA 算法遵从了交互序列，根据下一步的真实行动进行价值估计，属于on-policy算法；
* Q-Leaming 算法没有遵循交互序列，而是在下一时刻选择了使价值最大的行动，属于off-policy算法。
* Q-Leaming 的思想更复杂，它结合了子部分的最优价值，更像是结合了价值迭代的更新算法，希望每一次都使用前面迭代积累的最优结果进行更新。

**Q-Leaming算法问题：**

**对状态行动的“过高估计”。**使用最优价值的行动替代了交互时使用的行动，由于Q­Learning 算法本质上还是采用了采样的方法，所以对于被采样的状态行动，Q-Leaming会对其“过高估计”，而对于没采样到的状态行动，则无法享受“过高估计”的待遇，那么两者之间就可能产生比较大的差距，这些差距会造成算法的波动，在真实的应用场景中更是如此。

**Qlearning收敛性分析：**

Q-Leaming 是一个比较优秀的算法，它具有较好的收敛性质，推导过程略。

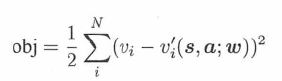
### 从表格形式到价值模型

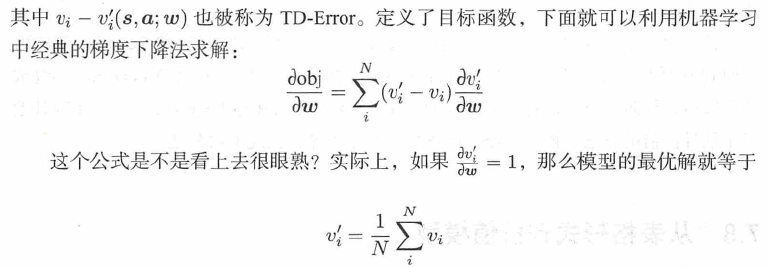
前面的模型都是基于表格形式求解有很大的帮助，但它也有自己的缺点。如果**问题的状态和行动的空间非常大，使用表格表示将难以求解**，因为我们需要将所有的状态行动价值都求解出来，才能保证对于任意一个状态和行动，我们都能够得到对应的价值。于是研究人员就开始研究一些**更合适的表达方式**，这时机器学习的方法终于登场，我们可以建立一些模型，并用一个**模型**来表示**状态、行动到值函数的关系**。我们令状态为S ，行动为A ，最终的值函数为R，那么我们要建立这样一个映射：



这样我们就把强化学习中**求解值函数的问题**转换成了**监督学习的问题**，可以把这个问题看作回归问题，于是所有可以用于回归问题的模型都可以被我们用上，例如线性回归（Linear Regression ）、支持向量回归（Support Vector Regression ）、决策树（Decision Tree ），以及神经网络(Neural Network ）等。在后面的章节中，我们将经常使用一个深层的神经网络完成这个映射的构建。

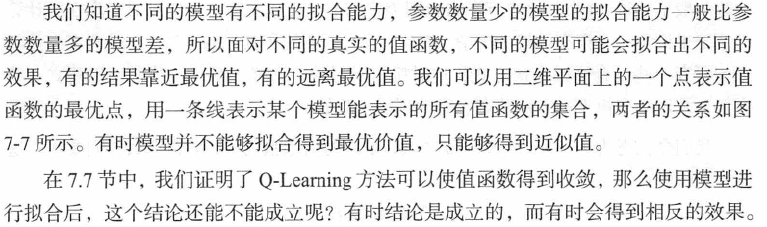
**目标函数**为平方损失函数，通过Q-Learning 算法得到的状态行动价值为目标价值vi，而模型计算得到的价值为估计价值叫vi`(s,a;w)，那么目标值为:

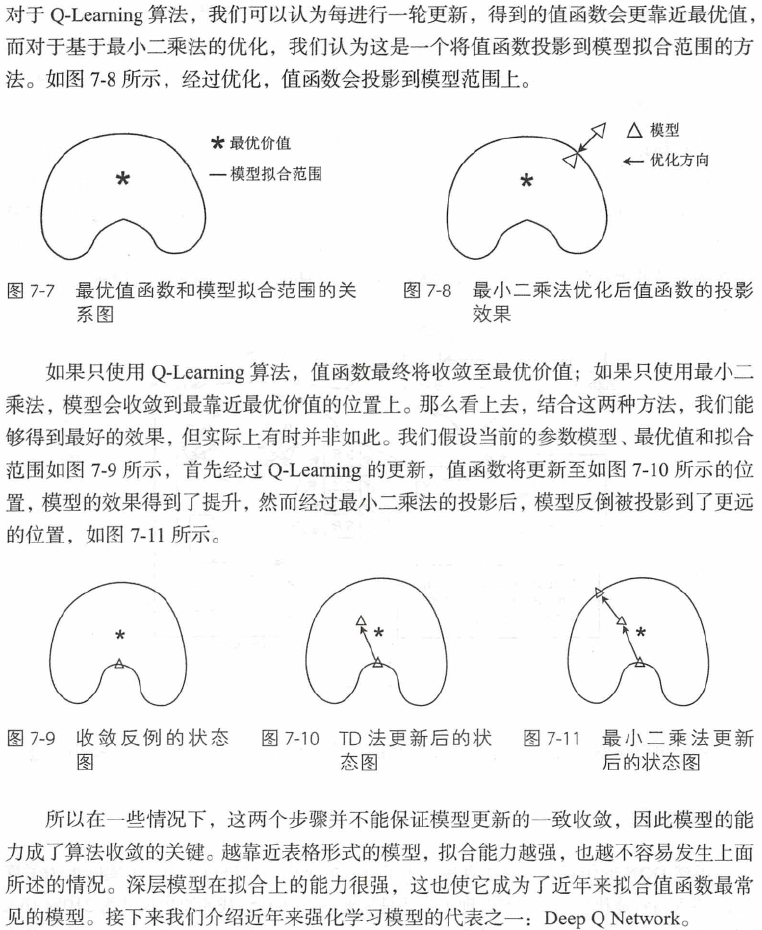




也就是所有训练数据的平均值，这和表格版的计算公式一致，也就是说表格版的算法实际上也是一种模型，只不过它的梯度处处为1 。这样，**机器学习方法和表格法得到了统一**，我们可以认为表格法是一种参数量为|S| × |A| 的模型，它的优点是可以完美拟合任意的模型（因为每一点可以独立收敛），缺点是它需要太多的参数。

在实践中，表格法有时是无法进行训练的；而对于拟合得到的模型来说，它的参数数量相对可控，我们可以确保模型训练是可行的但是它的收敛性就不再有保证。





### Deep Q Network

Deep Q Network (DQN) 方法来自论文*Human-level control through deep reinforcement learning*。论文主要介绍了如何使用DQN网络训练Agent在Atari游戏平台上尽可能获得更多的分数。DQN整体的算法思路不那么复杂：在使用Q-Leaming作为优化算法的前提下，使用深层神经网络表示值函数，直接使用游戏图像和得分进行训练，不再使用其他任何领域的知识。

在这篇论文之前，大多数的强化学习问题都需要研究任务的领域知识，同时需要解决同一序列内样本之间相互关联的问题，以及Q-Leaming算法中价值估计值与更新值相互关联的问题。这篇论文提出了一系列的解决方案，这些方案虽然并非首创，但是通过将这一系列的设计和改进融入其中，使模型在Atari 游戏上获得了更好的效果，模型在部分游戏上达到了媲美专家级玩家的效果。

* 数据预处理

虽然DQN 的作者表示算法直接使用了游戏的画面信息进行学习，但实际上为了算法能够更高效地执行，一些预处理工作还是需要的。Atari 游戏的原生尺寸为210 × 160,每个像素有128 种颜色，我们首先将其转**换成84 × 84 维度的灰度图**。以Breakout 游戏为例，转换前后的效果如图7- 13 所示，可以看出变换后的图像依然保留了主要的信息，同时减轻了数据处理的负担。

虽然Atari 游戏是一个动态的游戏，但是每一时刻Agent 只能从环境中获得1帧信息，而这种静态的图像信息很难表示游戏的动态信息。以乒乓球游戏为例，当画面静止时我们无法判断球要飞向哪一方。为此，算法将收集从当前时刻起的前N帧画面，并将这些信息结合起来作为模型的输入。获得了一定时间内集合的状态信息，模型可以学习到更准确的行动价值，在实验中N 被设置为4。

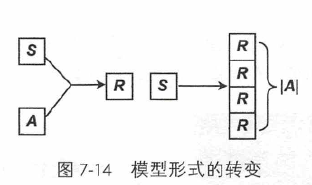
除了游戏的画面，**游戏的得分**也需要做**预处理**。不同的Atari 游戏的得分系统不同，有的得分可以上万，有的得分只有几十。为了模型能够更好地拟合长期回报，我们需要将得分压缩到模型擅长处理的范围中，我们将每一轮得到的回报压缩到［－1, 1 ］。虽然对游戏来说有些不合理，但这样的数值确实更方便模型处理。

* 环境交互

Atari 游戏的可行状态数量非常多，因此如何**更好地探索**更多的状态变得十分关键。DQN 采用了**ε-greedy** 的策略，一开始策略以100% 的概率随机产生行动，随着训练的不断进行，这个概率将不断衰减，最终衰减至10%。也就是说，有90% 的概率执行当前最优策略。这样，从**以探索为主的策略**逐渐**转变**成以**利用为主的策略**，两者得到了很好的结合。

* 模型结构

前面提到价值模型是从到 的映射，采用这种方法构建模型自然是可以的，但是它还有一些缺点。当模型需要通过值函数求解最优策略时，我们需要计算次才能求出，在实际中这样的计算方法效率较低。为了简化计算，我们可以将模型变成这样的形式，模型的输出为长度为的向量，向量中的每一个值表示对应行动的价值估计。这样我们只需要一次计算就可以求出所有行动的价值，无论行动有多少，我们评估价值的时间是一样的。两个模型的表示形式差异如图所示。模型的主体采用了卷积神经网络的结构。



第一层卷积层的卷积核为8 × 8 ，由ide = 4 ，输出的通道数为32 ，此后应用ReLU非线性层。

第二层卷积层的卷积核为4 × 4 ，由ide = 2 ，输出的通道数为64 ，此后应用ReLU非线性层。

第三层卷积层的卷积核为3 × 3 , stride= l ，输出的通道数为64 ，此后应用ReLU非线性层。

第四层是全连接层，输出的维度为512 ，此后应用ReLU 非线性层。

最后一层全连接层得到对应行动的价值估计，输出维度与可行的行动数量相关，一般在4～18 个。

* Random start

有些Atari 游戏的起始画面是完全一样的，例如Space Invader ，如果每次我们都从一个固定的场景开始做决策，那么Agent 总要对这些相同的画面进行决策，这显然不利于我们探索更多的画面进行学习。为了增强探索性同时不至于使模型效果变糟，我们可以设定在游戏开始很短的一段时间内，让Agent 执行随机的行动，这样可以最大程度地获得不同的场景样本。在5.2.3 节我们分析了Baselines 中实现的这部分代码，感兴趣的读者可以回顾相关的实现。

* Frame-Skipping

虽然我们可以根据游戏的每一个状态给出一个输出，但是实际游戏中玩家并不需要如此高频率的操作。模拟器可以达到每秒60 帧的显示速率，但是实际上人类玩家无法实现如此高频率的操作，因此我们可以设定每隔一定帧数执行一次行动，例如每隔41帧进行一次行动选择，那么中间的几帧将重复执行前面选择的行动，这样相当于模仿了人类按下某个按钮并持续一段时间的效果。此外，相邻帧之间的画面存在着极大的相似性，对于十分相似的画面，我们通常可以采用相同的行动，因此这样跳过一定帧数的判断也是合理的。

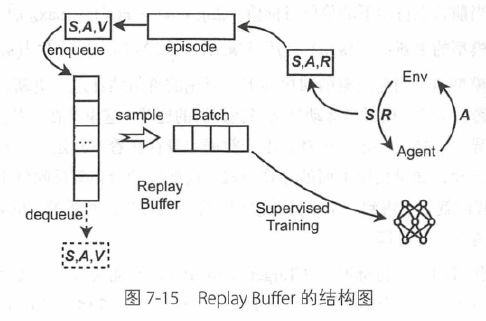
* Replay Buffer

Q-Leaming 方法基于当前策略进行交互和改进，更像是一种在线学习（Online Learning）的方法，每一次模型利用交互生成的数据进行学习，学习后的样本被直接丢弃。但如果使用机器学习模型代替表格式模型后再采用这样的在线学习方法，就有可能遇到两个问题，这两个问题也都和机器学习有关。

( 1 ）**交互得到的序列存在一定的相关性**。交互序列中的状态行动存在着一定的相关性，而对于基于最大似然法的机器学习模型来说，我们有一个很重要的假设：训练样本是独立且来自相同分布的，一旦这个假设不成立，模型的效果就会大打折扣。而上面提到的相关性恰好打破了独立同分布的假设，那么学习得到的值函数模型可能存在很大的波动。

( 2 ）**交互数据的使用效率**。采用梯度下降法进行模型更新时，模型训练往往需要经过多轮迭代才能收敛。每一次迭代都需要使用一定数量的样本计算梯度，如果每次计算的样本在计算一次梯度后就被丢弃，那么我们就需要花费更多的时间与环境交互并收集样本。

为了解决这两个问题，作者使用了Replay Buffer 这个数据结构。这个结构被翻译为“样本回放缓存区”，因为这个名字比较拗口，所以在接下来的内容中我们使用英文表达。Replay Buffer 的结构和交互流程如图所示。



可以看出，Replay Buffer 保存了交互的样本信息，一般来说每一个样本都会保存当前的状态**s**、行动**a**和长期累积回报**v**。对于其他的算法，Replay Buffer还会保存其他的信息。Replay Buffer 的大小通常会设置得比较大，例如，将上限设置为可以存储100 万个样本，这样较长一段时间的样本都可以被保存起来。在训练值函数时，我们就可以从中取出一定数量的样本，根据样本记录的信息进行训练。

总的来说，Replay Buffer 包含了**收集样本**和**采样样本**两个过程。**收集**的样本按照时间先后顺序存入结构中，如果Replay Buffer 已经存满样本，那么新的样本会将时间上最久远的样本覆盖。而对**采样**来说，如果每次都取出最新的样本，那么算法就和在线学习相差不多；一般来说，Replay Buffer 会从缓存中均匀地随机采样一批样本进行学习。**均匀采样**的好处是什么呢？前面提到我们交互得到的序列在时间维度上存在一定的相关性。我们希望学习得到的值函数能够表示在当前状态行动下的长期收益的期望，然而每一次交互得到的序列，只能代表当前状态一行动下的一次采样轨迹，并不能代表所有可能的轨迹，这样估计的结果就和期望的结果存在一定的差距。随着交互时间不断拉长，这个差距的累积会越来越大。如果完全使用序列的估计值进行训练，某一轮训练时模型会朝着一个序列的估计训练，另一轮训练又会朝着另一个序列的估计训练，那么模型很容易产生较大波动。采用均匀采样后，每次训练的样本通常来自多次交互序列，这样单一序列的波动就被减轻很多，训练效果也就稳定了很多。同时，一份样本也可以被多次训练，提高了样本的利用率。

* Target Network

模型不稳定的另外一个原因来自算法本身。从Q-Learning的计算公式可以看出，算法可以分成如下两个步骤。

( 1 ）计算当前状态行动下的价值目标值：



( 2 ）网络模型的更新：



可以看出模型通过当前时刻的回报和下一时刻的价值估计进行更新，这既像一场自导自演的电影，又像一场既当运动员又当裁判员的比赛。这里存在一些隐患，前面提到数据样本差异可能造成一定的波动，由于数据本身存在着不稳定性，每一轮迭代都可能产生一些波动，如果按照上面的计算公式，这些波动会立刻反映到下一个迭代的计算中，这样我们就很难得到一个平稳的模型。为了减轻相关问题带来的影响，我们需要尽可能地将两个部分解耦。

为此论文作者引入了目标网络（Target Network ），它能缓解上面提到的波动性问题。这个方法引入了另一个结构完全一样的模型，这个模型被称为Target Network ，而原本的模型被称为表现模型（Behavior Network ）。两个模型的训练过程如下所示。

( 1 ）在训练开始时，两个模型使用完全相同的参数。

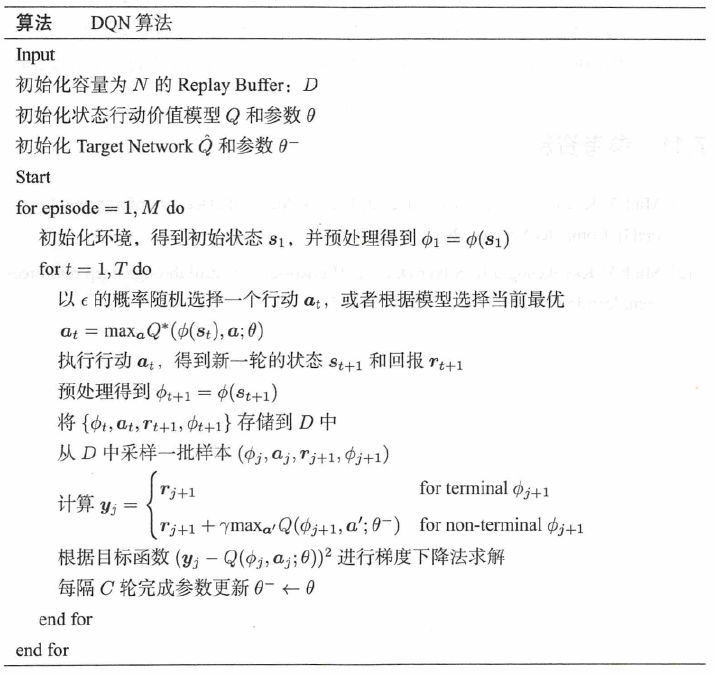
( 2 ）在训练过程中，Behavior Network 负责与环境交互，得到交互样本。

( 3 ）在学习过程中，由Q-Learning 得到的目标价值由Target Network 计算得到；然后用它和Behavior Network 的估计值进行比较得出目标值并更新Behavior Network。

( 4 ）每当训练完成一定轮数的迭代，Behavior Network 模型的参数就会同步给Target Network ，这样就可以进行下一个阶段的学习了。

通过使用Target Network ，计算目标价值的模型在一段时间内将被固定，这样模型可以减轻模型的波动性。

以上就是DQN 模型的主要内容，完整的算法流程如下所示。



### DQN 实验

* 实验概要：

使用DQN玩flappy bird的游戏；

* 代码及运行方式

git clone https://github.com/yenchenlin1994/DeepLearningFlappyBird.git

cd DeepLearningFlappyBird

python deep\_q\_network.py

* 伪代码

Initialize replay memory D to size N

Initialize action-value function Q with random weights

for episode = 1, M do

Initialize state s\_1

for t = 1, T do

With probability ϵ select random action a\_t

otherwise select a\_t=max\_a Q(s\_t,a; θ\_i)

Execute action a\_t in emulator and observe r\_t and s\_(t+1)

Store transition (s\_t,a\_t,r\_t,s\_(t+1)) in D

Sample a minibatch of transitions (s\_j,a\_j,r\_j,s\_(j+1)) from D

Set y\_j:=

r\_j for terminal s\_(j+1)

r\_j+γ\*max\_(a^' ) Q(s\_(j+1),a'; θ\_i) for non-terminal s\_(j+1)

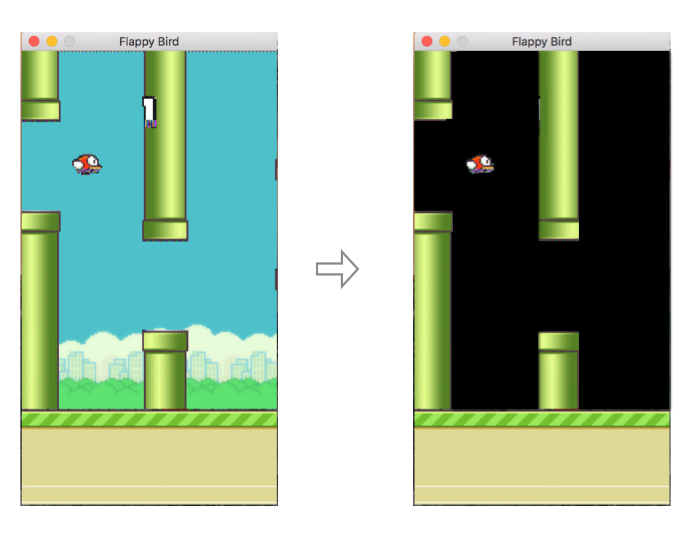
Perform a gradient step on (y\_j-Q(s\_j,a\_j; θ\_i))^2 with respect to θ

end for

end for

* 实验环境

DQN通过游戏界面的图像作为数据输入进行训练，如果将游戏界面的背景去除掉，DQN网络会更快的收敛，去除游戏背景如下所示：

[](https://github.com/yenchenlin/DeepLearningFlappyBird/blob/master/images/preprocess.png)

* DQN网络架构

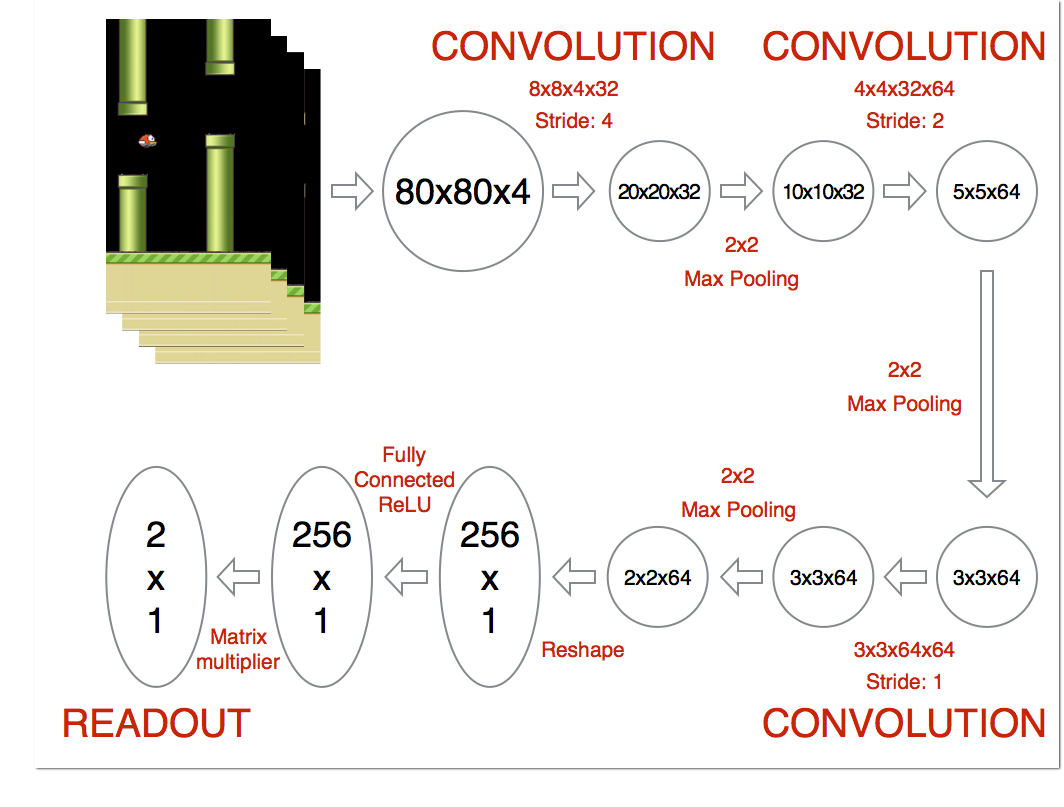
1. 数据预处理：

将游戏界面的图像转为灰度图；

将图像大小resize到80x80；

将4帧图像叠加为80x80x4 的array作为DQN的输入；

1. DQN网络架构示意图：



**第一层**： 对输入的80x80x4的tensor使用8x8x4x32的卷积核以步长4进行卷积操作，对卷积输出进行2x2 pooling，得到10x10x32的tensor；

**第二层**： 使用4x4x32x64的卷积核以步长2进行卷积，然后进行池化；

**第三层**： 使用3x3x64x64的卷积核以步长1进行卷积，然后进行池化；

**第四层**： 将第三层的输出进行展开，进行进行全连接操作，并使用ReLu进行激活，该层含有256个节点；

**输出层**： 输出节点的数目和游戏中Action的数目相同，其中index为0的节点代表的action是“doing nothing”。这一层的输出代表相应Action的Q value，游戏中的每一个step，DQN都会使用ε-greedy策略来计算每个Action的Q value。

1. 训练过程：
   * + 使用标准差为0.01的正态分布来初始化所有的权重，设置Replay Memory的size为50000；
     + 前10000个step随机选择action，期间不更新DQN网络的权重；
     + 在接下来3000,000个train step中逐渐将ε从0.1线性降至0.0001（annealing），这样设置是因为游戏的FPS=30， 高的ε会使得flap过于频繁，bird会停留在屏幕顶端而导致游戏结束。这样设置会使Q function收敛较慢，因为ε设置的比较小。在其他游戏中将ε初始化为1更合适一些。
     + 训练过程中，每个training step，DQN network会从replay memory中采样32个样本来进行训练，使用Adam优化器来优化loss， learning rate 为0.000001。
     + Annealing结束后，将ε固定为0.001，继续训练
2. reproduce：

将如下代码注释掉

if checkpoint and checkpoint.model\_checkpoint\_path:

saver.restore(sess, checkpoint.model\_checkpoint\_path)

print("Successfully loaded:", checkpoint.model\_checkpoint\_path)

else:

print("Could not find old network weights")

将 deep\_q-network.py 中的参数更改如下：

OBSERVE = 10000

EXPLORE = 3000000

FINAL\_EPSILON = 0.0001

INITIAL\_EPSILON = 0.1

* + reference：

<https://github.com/yenchenlin/DeepLearningFlappyBird>

* + 实验结果：

使用保存好的参数模型可以非常的好的玩flappy bird的游戏，使用神经网络拟合状态空间非常大的Q value function是可行的。

具体效果现场演示。

### 小结

本节我们学习了无模型的学习算法，以及其中的经典算法DQN ，下面让我们回顾一下。

( I ）在无模型问题中，我们将无法使用状态转移概率进行策略求解。

( 2 ）蒙特卡罗法通过统计完整序列的长期回报来完成策略评估的过程。

( 3 ）TD通过当前时刻的回报和下一时刻开始的长期回报估计来完成策略评估的过程。

( 4 ) SARSA 法和Q-Learning 法分别代表了On-Policy 和Off-Policy 的TD 法。

( 5 ) DQN 结合了Q-Learning 的价值估计方法和深层模型较强的拟合效果，同时还结合了Replay Buffer 和Target N etwork 这样的结构，在Atari 游戏上获得了不错的结果。

## 策略梯度

基于策略梯度的强化学习算法将直接基于长期回报期望对策略进行优化，使用的也是在机器学习领域常见的基于**梯度的优化**方法。策略梯度法在学习策略模型过程中表现得相对稳定，同时也可以处理**离散和连续行动空间**的计算，基于策略模型的改进方案也使算法有了很大的提高。

### Policy Gradient

**价值迭代**方法的核心在于**对值函数的估计**，如果值函数能够很好地估计得到，我们就可以通过最优值函数得到对应的策略 。 这个思路在实际中得到了很好的应用，但是它也存在一些问题，例如模型训练的稳定性等 。

**策略梯度（Policy Gradient）**法并不采取迂回的方式更新策略，而是**直接计算策略可能更新的方向** 。

**价值迭代**优化的重点都落在了值函数上，无论是策略迭代，还是 Q-Learning，只要能够得到精确的值函数，就可以使用 Bellman 公式求出最优策略，也就是况，我们的最优策略需要满足:



**策略梯度法**则使用了另外一种思路--回到问题的本源 ，强化学习的目标是最大化长期回报期望，于是目标也可以写作:

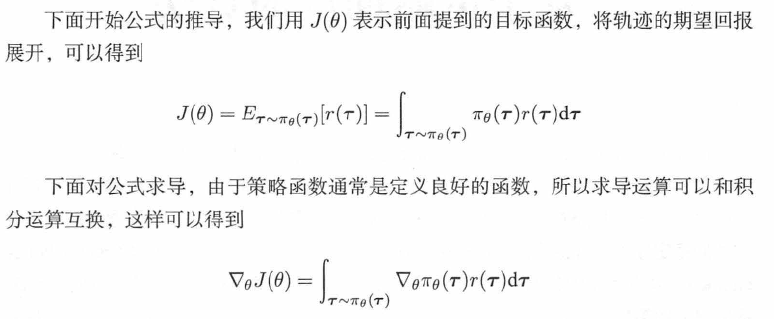


其中表示使用策略进行交互得到的一条轨迹，**r()**表示这条轨迹的**总体回报**。

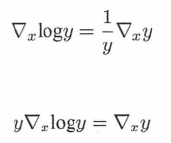
由于值函数实际上也是一个函数，强化学习的目标是最大化这个函数，如果这个函数性质良好，我们是不是可以用求导的方法对其进行优化呢？如果我们可以将**值函数表示为策略参数的某个函数**，就可以**求出值函数关于策略参数的梯度**，并**使参数沿着梯度上升的方向更新**，也就可以**提升策略**了。下面就使用优化的思想介绍这个算法如何对策略进行优化。

#### 公式推导

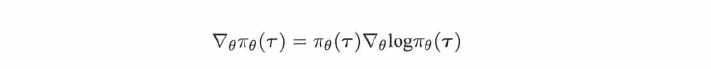
用J(θ）表示前面提到的目标函数，将轨迹的期望回报展开，可以得到：



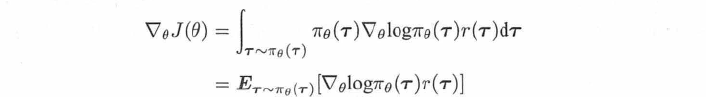
又因：



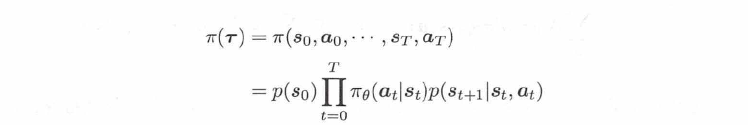
有：



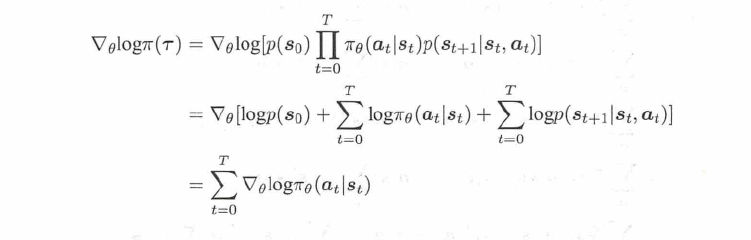
进而有：



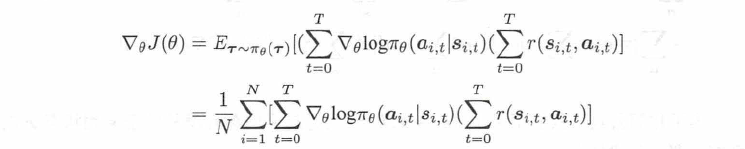
对进行化简，因：



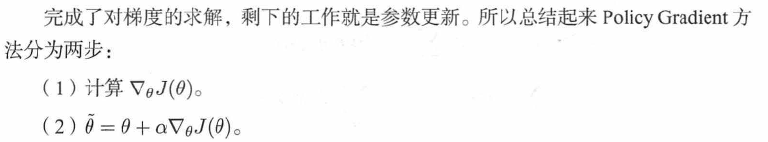
有：



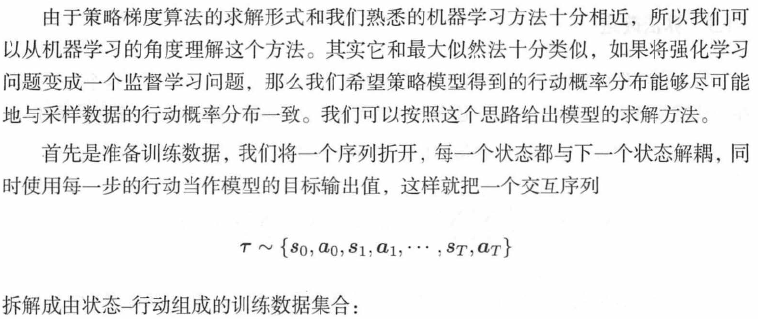
代入，并使用Monte Carlo近似的方式进行替换，可以得到**求解梯度的“最终”形式**：

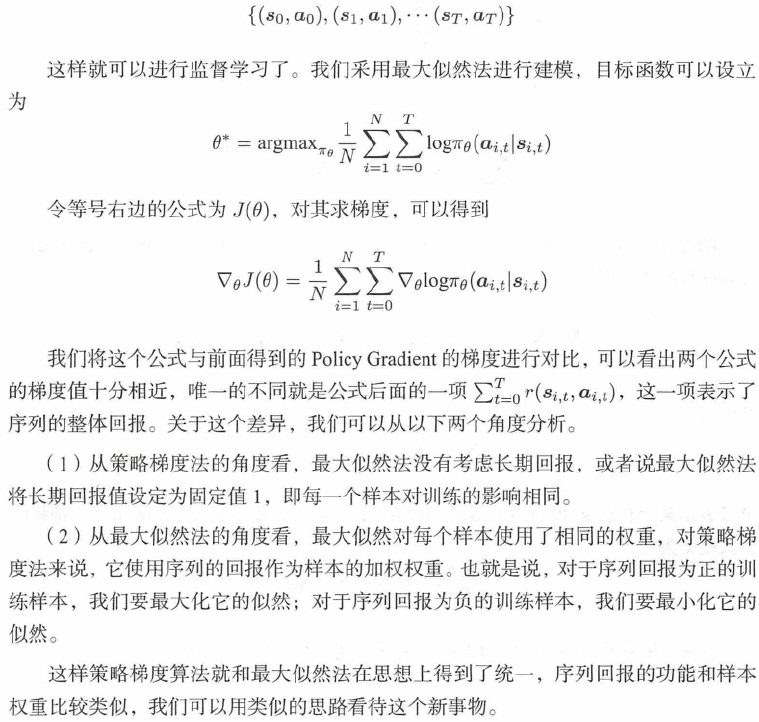


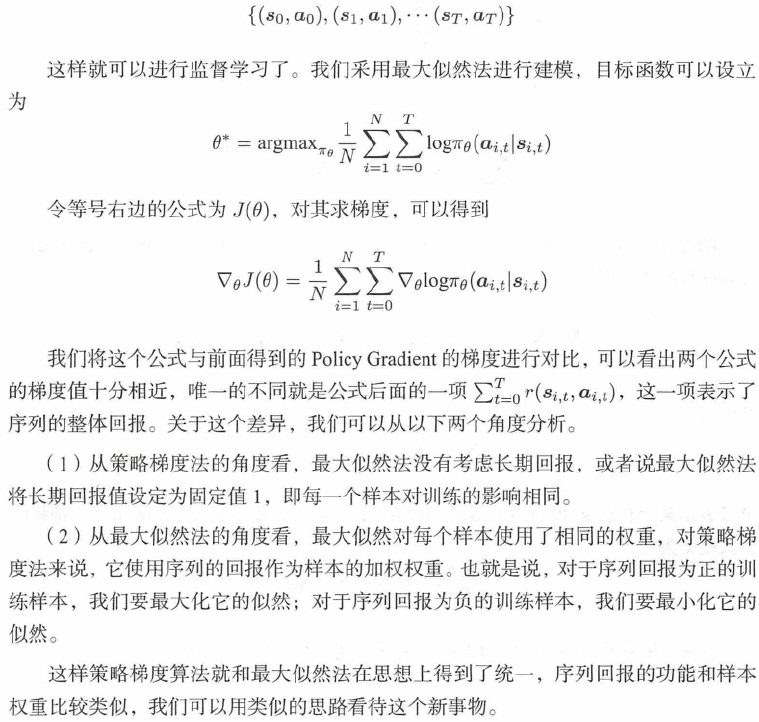
#### 参数更新

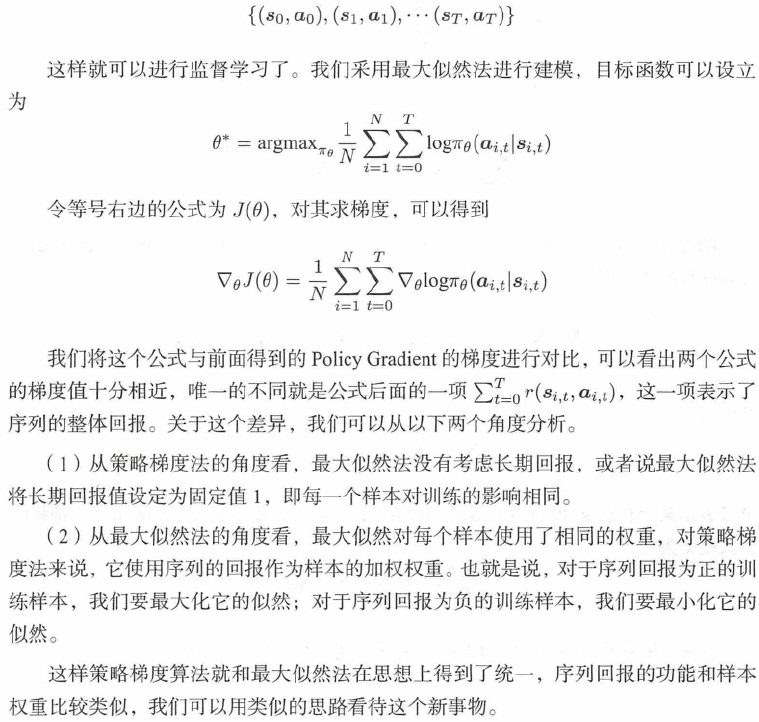


#### 算法分析



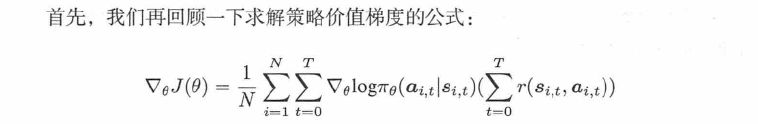




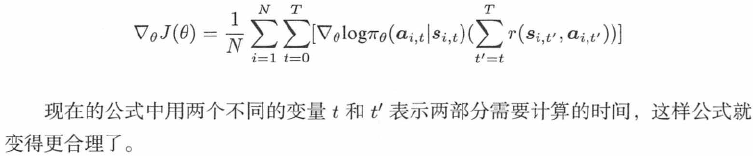


#### 算法优化

* 更新梯度的权重



从这个公式中我们看出了一个问题，不论是哪个时间段，我们都要用策略的梯度乘以所有时刻的回报值总和，这样的设计显然是不合理的。因为理论上，在t 时刻我们完成了决策后，它最多只能影响t 时刻之后的所有回报，并不会影响t 时刻之前的回报，因为我们无法通过现在的决策影响已经发生过的事情，所以这一部分的回报值不应该计算在梯度中。我们的公式应该改写成：



* 更新baseline

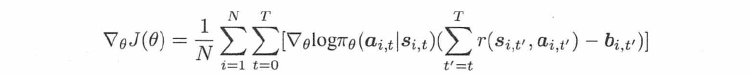
从上面的分析来看，策略梯度法更像是加权版的最大似然优化法。“权重”将直接影响梯度的更新量，这样就会带来以下两个问题：

( I ）如果计算得出的序列回报数值较大，那么对应的参数更新量就会比较大，这样我们的优化就可能出现一定波动，这些波动很可能影响优化效果。

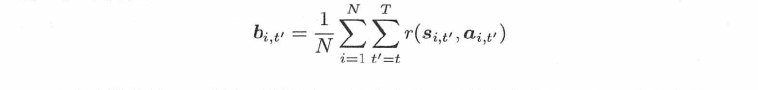
( 2 ）在一些强化学习问题中，环境给予的回报始终为正，那么不论我们的决策如何，最终累积的长期回报值都是一个正数。换句话说，我们会增强所有的策略，只是对于实际效果并不好的策略，我们为其提升的幅度会有所降低。**这样的更新方法和我们的初衷并不一致**，我们降低不好的行动的概率，而不是轻微提升不好的行动的概率。

这两个问题使我们回到了强化学习的目标：**提高能最大化长期回报策略的概率，降低无法最大化长期回报策略的概率**。将上面的思想转化成策略梯度问题的表述形式，就会变成：让能够最大化长期回报策略的“权重”为正且尽可能大，让不能最大化长期回报策略的“权重”为负且尽可能小。

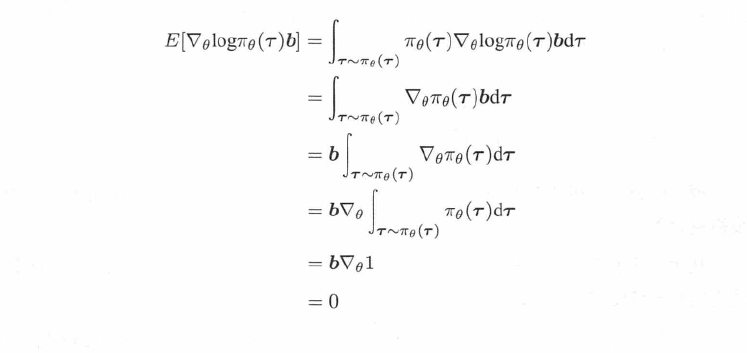
为了实现这个目标，我们可以调整权重的数值和范围，一个简单的方法就是给所有时刻的长期累积回报减去一个偏移量，这个偏移量也被称为**Baseline** ，我们用变量b表示，于是公式就变为：



这个变量可以设计为同一起始点的不同序列在同一时刻的长期回报均值，它的公式形式如下所示：



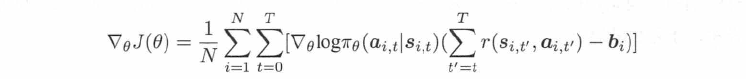
经过这样的处理，所有时刻的**权重均值变为0** ，就会存在权重为正或为负的行动，同时权重的**绝对值也得到了一定的缩小**，这**对于算法的稳定性来说都有一定的帮助**。更重要的是，加入的这个偏移量并**不会使原本的计算值变得有偏**，我们可以得到：



**不影响期望值**的同时**降低算法的波动性**。这个方法也被广泛应用于Policy Gradient 算法中。当然，在现在常见的方法中，我们已经有了更好的替代方法，这部分内容将在Actor Critic 部分介绍。

### Actor Critic

Baseline的方法还存在着一些问题。我们再来看看下面这个公式



公式的问题仍然在等号右边的最后一项。前面提到，我们用轨迹的回报表示整个序列的价值，这个表示是**准确无偏**的，但是在真实的训练过程中，为了**尽可能地控制学习时间**，往往只会进行有限次数的交互，这些交互有时并不能很好地代表轨迹真实的期望。每一个交互得到的序列都会有一定的差异，对应的回报也会有一定的差异，因此**不充足的交互会给轨迹回报带来较大的方差**，这和前面提到的蒙特卡罗法遇到的问题类似。

由于强化学习中Agent 与环境交互基于MDP ，每一时刻Agent 基于当前的状态可以有很多的选择，而采样得到的序列只是其中之一，所以一条或几条序列的回报和所有路径的回报期望存在一定的差距。

机器学习的一个核心问题是平衡偏差和方差。对策略梯度模型来说，它的方差相对较大，为了模型的稳定，我们可以牺牲一定的偏差来换取方差变小。这其中的一种方法就是采用**Actor-Critic** 算法，其主要特点就是用一个独立的模型估计轨迹的长期回报，而不再直接使用轨迹的真实回报。类似于基于模型的Q-Learning 算法，在估计时使用模型估计轨迹价值，在更新时利用轨迹的回报得到目标价值，然后将模型的估计值和目标值进行比较，从而改进模型。

公式中可以被模型替代的部分有两个，其中代表从t 时刻出发所获得的长期回报，bi 代表待减去的偏移量。Actor Critic使用如下公式来替代策略梯度的权重：



这种方案在计算量和减少方差方面具有一定的优势。由于引入了状态价值模型，算法整体包含了两个模型，一个是策略模型，一个是价值模型，所以这个算法被称为**Actor-Critic** ，其中**Actor** 表示策略模型，**Critic** 表示价值模型。

### A3C

真正将Actor-Critic 应用到实际中并得到优异效果的是A3C (Asynchronous Advantage Actor-Critic ）算法，该算法突出了异步和优势两个概念。另一个优化算法A2C ( Advantage Actor-Critic ），采用了同步的方法进行优化，并同样获得了比较好的效果。

* **On-Policy Learning:**

策略梯度法和Actor-Critic 算法都通过目标函数的梯度进行策略更新，而计算梯度需要基于当前的策略模型，所以每一次计算梯度时，我们需要使用当前最新的策略模型重新进行交互采样，得到相应的序列样本，然后使用这些样本完成梯度计算；当梯度计算完成，模型得到更新后，这些使用过的样本将被丢弃，我们需要采集新的样本。SARSA 算法也是这样的算法。

DQN 方法，由于采用Replay Buffer 存储了一段时间的交互样本，模型学习时使用的样本不一定是由当前模型交互得到的，而且Q-Leaming 在学习时假设在t+l 时刻将采用当前最优的行动，因此它的目标价值并不是由真实的交互序列得到的。这样的学习方式通常被称为Off-Policy Learning。

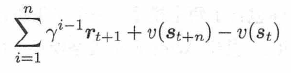
* **Asynchronous（并行的交互采样和训练)：**

由于Off-Policy 算法可以使用非当前策略产生的样本，因此它对样本的重复使用度比较高。而对On-Policy 算法来说，每一次模型更新都需要一定量的“新样本”，所以On-Policy 算法需要花费更多的时间收集样本。所以为了更快地收集样本，我们需要用并行的方法收集。

在A3C 方法中，我们要同时启动N个线程，Agent 将在N 个线程中同时进行环境交互。只要保证每一个线程中的环境设定不同，线程间交互得到的序列就不会完全一样，这样得到的样本就很有意义。收集完样本后，每一个线程将独立完成训练并得到参数更新量，并异步地更新到全局的模型参数中。下一次训练时，线程的模型参数将和全局参数完成同步，再使用新的参数进行新一轮的训练。

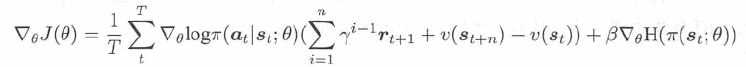
* Advantage

Advantage 表示了算法关于**“权重”**部分的计算方法。前面提到Actor-Critic 算法采用TD-Error 的形式：。这个方法虽然增加了模型的稳定性，但是模型的偏差也相应变大，为了更好地平衡模型的偏差和方差，A3C 方法使用了多步回报估计法，这个方法可以在训练早期更快地提升价值模型。对应的公式变为：



* 增加模型探索性：

在模型的目标函数中加入了策略的熵。由于情可以衡量概率分布的不确定性，所以我们希望模型的情尽可能大一些，这样模型就可以拥有更好的多样性。这样，完整的策略梯度计算公式就变为

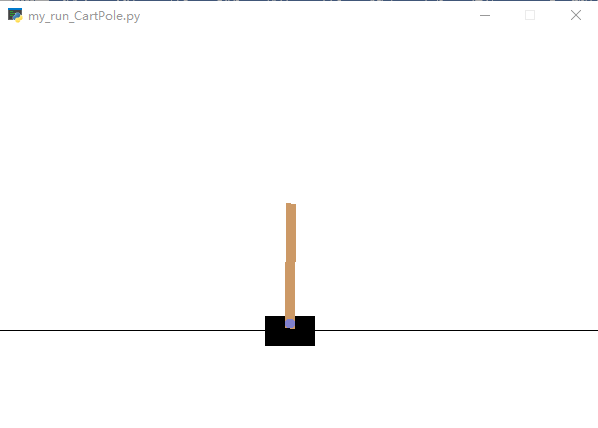


### Actor Critic 实验

实验说明

* 实验概要：

使用Actor Critic玩CartPole-v0游戏，如下图所示：

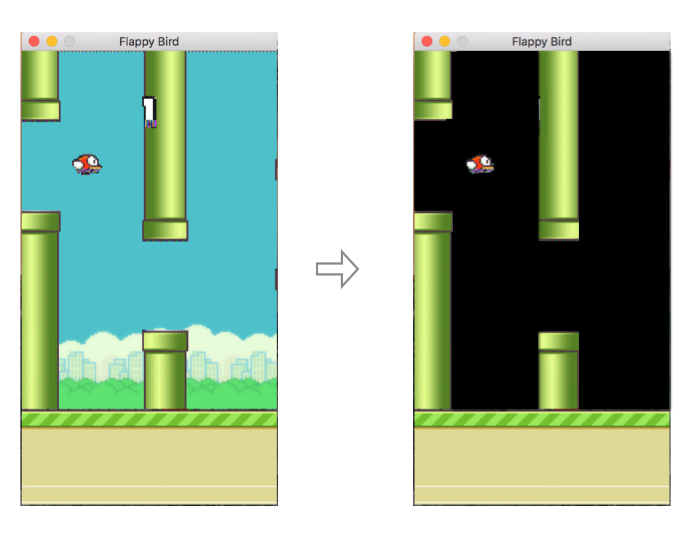


* 代码及运行方式

* 伪代码

* 实验环境

DQN通过游戏界面的图像作为数据输入进行训练，如果将游戏界面的背景去除掉，DQN网络会更快的收敛，去除游戏背景如下所示：

[](https://github.com/yenchenlin/DeepLearningFlappyBird/blob/master/images/preprocess.png)

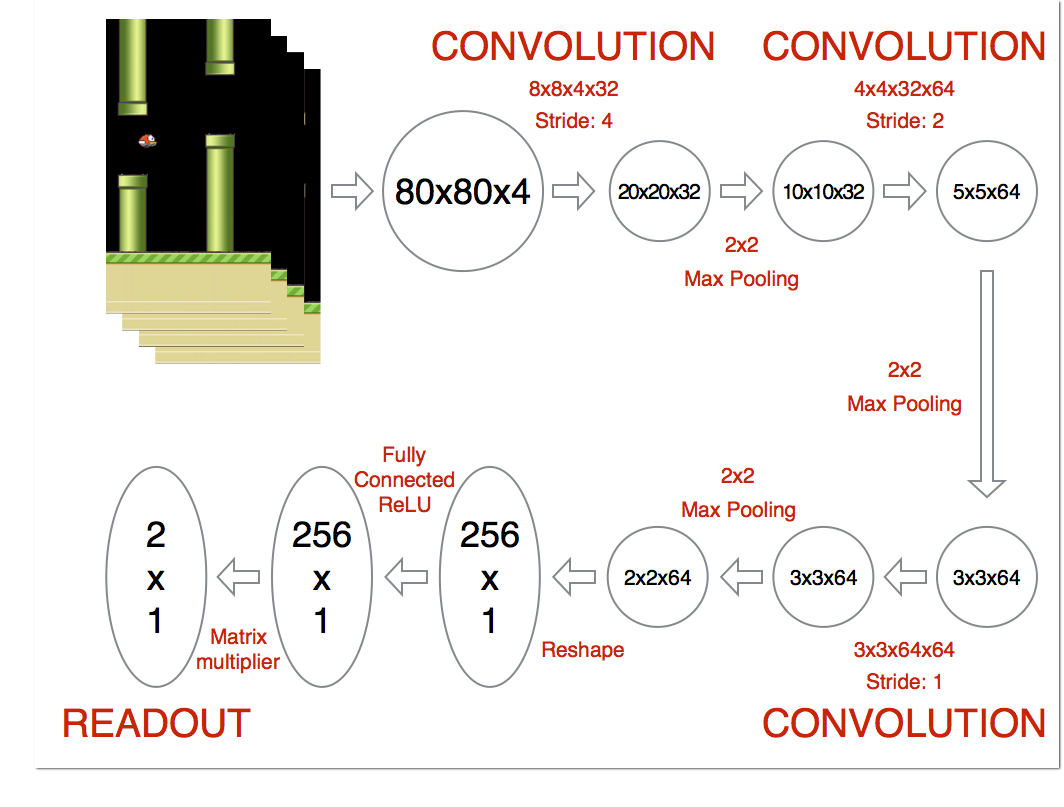
* DQN网络架构

1. 数据预处理：

将游戏

；

1. DQN网络架构示意图：

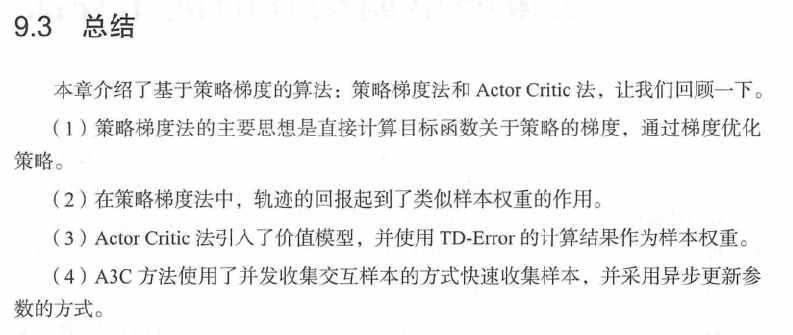


**第一层**：

的每一个step，DQN都会使用ε-greedy策略来计算每个Action的Q value。

1. 训练过程：
2. reproduce：
   * reference：
   * 实验结果：

### 4.3.5 小结：



# 代码实现及效果展示

# 应用展望

# 后续工作规划

主要有回报工程，特征工程，模型选择，虚拟环境的搭建，将虚拟环境中训练的模型迁移到实际环境中进行再学习，搭建实际运行环境（包括不限于控制电机，使用C实现强化学习）

目前工作的不足之处： 仍然有一些算法分支还没有了解到，这个后面根据实际业务需求来进行，如果确实有需要，目前来看Actor Critic 已经可以满足要求了。