Reinforcement Learning总结

# 背景

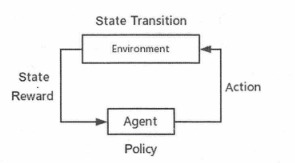
目前行走样机在使用过程中，自动化智能化程度还不够高，许多参数还需要手动调整，所以希望使用强化学习的方式找到最优参数，或者训练模型去在线优化参数，达到快速智能匹配的效果。在行走设备实际运行中，可能需要对患者的每一步进行打分，然后强化学习基于环境反馈的分数reward来寻找最优参数或者训练优化模型。

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：

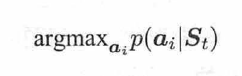


这个链条包含了两种状态转换，一种是从状态到行动的转换，由 agent 的策略决定的；另一种是从行动到状态的转换，由环境决定的。

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。

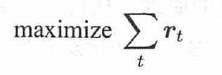
( 2 ）“**决策**”表示了其中的策略部分将由Agent 决定。Agent 可以通过自己的行动改变状态序列，和环境中存在的随机性共同决定未来的状态。

( 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。

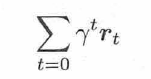
* **长期回报**

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。

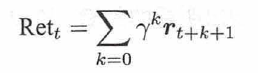
幸运的是，环境反馈的**reward**提供了局部的回报信息，将其扩展成为我们的目标，如下：



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：

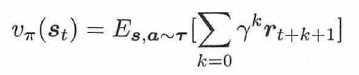


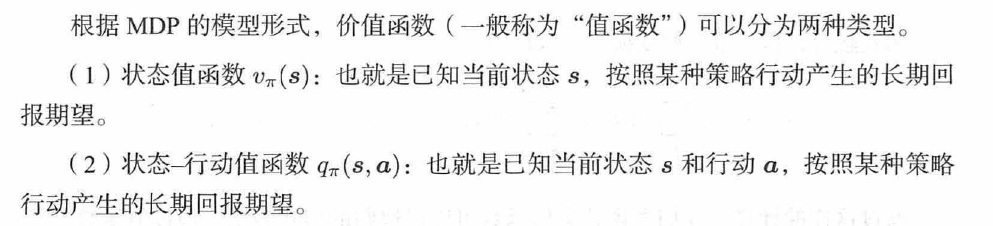
这个打折率一般来说小于1 ，这样一来长期回报的这个数列的和就变得有界了，也就可以计算出具体的值了。于是我们正式定义另外一个变量一一**长期回报**。所谓长期回报，是将当前状态之后所有的回报取出，分别乘以对应的打折率，然后加起来得到一个汇总的值。

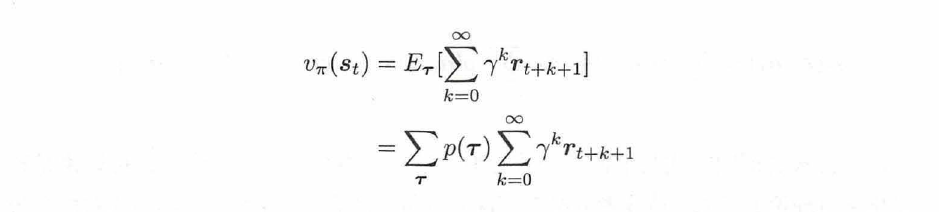


* **值函数**

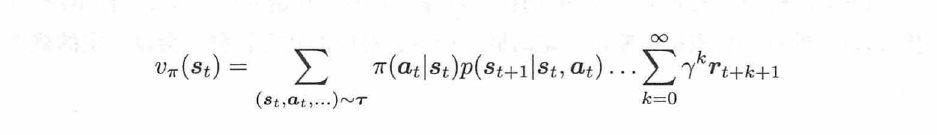
基于MDP 形成的交互序列，从状态到行动的转换可以通过策略确定，而由于环境的原因，从行动到下一个状态的转换有时并不能确定。因此在衡量价值时，我们需要考虑每一种状态转换的影响，这就需要基于状态转换求解长期回报的期望。令τ为根据策略和状态转换采样得到的序列，那么价值的公式定义就可以写成：



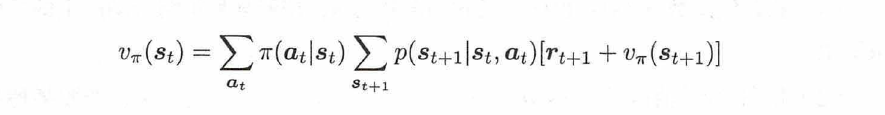




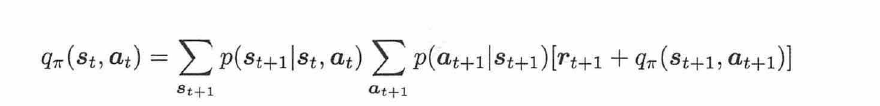
其中T 表示从状态St 出发的某条路径。由于我们将强化学习的过程表示成马尔可夫决策过程，于是路径部分可以展开为：



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：

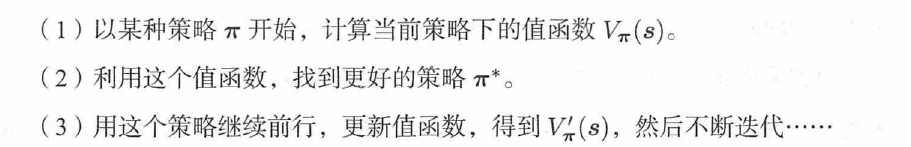


同理**状态-行动值函数**的公式可以变换为：



### 4.1.2 策略迭代

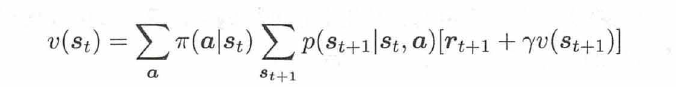
如果想知道最优的策略，就需要能够准确估计值函数。然而想准确估计值函数，又需要知道最优策略，数字才能够估计准确。所以实际上这是一个“鸡生蛋还是蛋生鸡”的问题。碰上这样无解的问题，往往需要采取一些“曲线救国”的方法。那么，能不能把这个问题考虑成一个迭代优化的问题，通过一轮一轮的计算逐渐接近最优的结果呢？答案是可以的。于是我们提出如下计算思路：



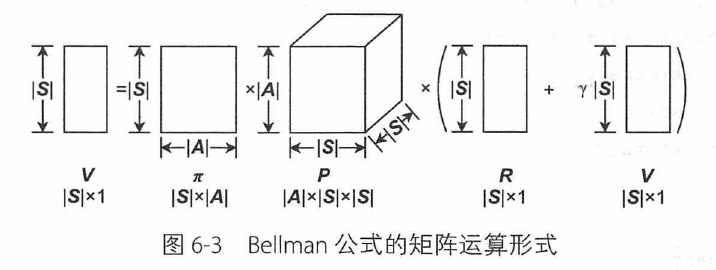
这样经过若干轮计算，如果一切顺利，我们的策略会收敛到最优的策略，如果能够判断出策略已经收敛，问题也就得到了解答。当然，目前我们还没有理论证明，因此这个思路也可能是错的，但我们还是先试着实践这个思路。

* **矩阵求解：**

想要求解值函数，就必须用到Bellman 公式：

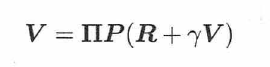


对于这个公式，我们利用了时间不变性这样的假设，于是等号左右两边的值函数值应该是相等的，所以这个问题可以采用解方程的方法求解。上面的公式是采用连加的形式描述的，我们现在将其变成矩阵运算的形式，于是等式变成了图6-3 所示的形式：



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：



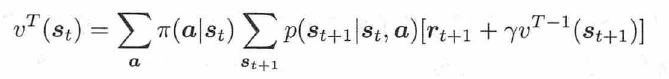
求解，有：



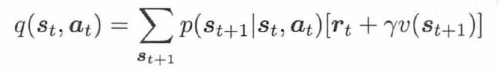
只要（1 一γIIP)-1 的逆矩阵存在，状态价值就可以解出来。实际中由于矩阵数值的限制，逆矩阵是存在的，所以这种方法可以求解。但是对于一些状态和行动比较多的问题，采用这种方法求解的复杂度偏高，因此实际中大家不用这种方法，而是采用迭代的方法进行计算。

* **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**

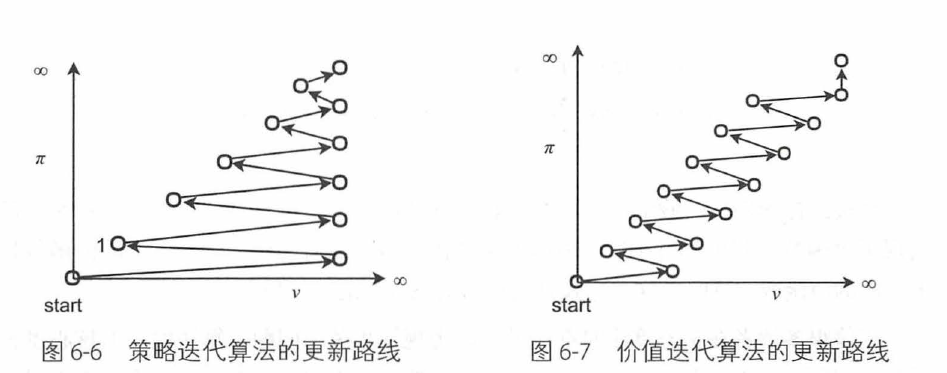
从算法中可以看出，算法的主要时间都花费在策略评估上，对一个简单的问题来说，在策略评估上花费的时间不算长；但对复杂的问题来说，这个步骤的时间实在有些长。一个最直接的想法就是一一我们能不能**缩短在策略评估上花的时间呢**？例如，我们大概估计出当前策略下每个状态的值函数就好，虽然不够精确，但这样也许已经足以帮助我们找出最优的策略了，再做更精细的评估实际上并不必要。这就是本节的主角一一**价值迭代法的思想来源**。

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。

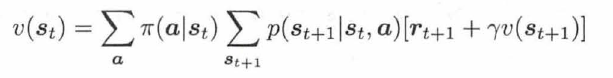
从图6-7 所示的更新路线可以看出，与其说这是更新策略，不如说是更新价值。当价值迭代完成后，策略迭代也就随之完成，所以我们可以将迭代的重点放在价值上。



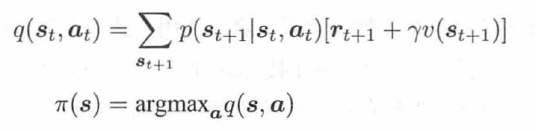
* **从动态规划看N轮迭代**

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：

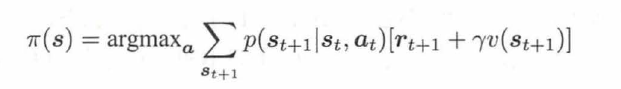


**策略改进**完成了策略改进的计算：

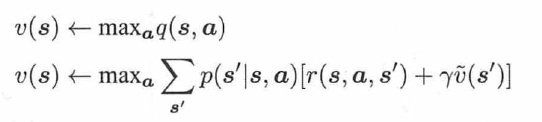


能否将这三个步骤结合，成为一个关于值函数的更新步骤呢？首先**将后两个公**

**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”。

## 价值迭代

SARSA

QLearning

DeepQNetwork

## 策略梯度

Policy Gradient

Actor Critic

# 代码实现及效果展示

# 应用展望