@[toc]

书籍链接：Reinforcement Learning for Sequential Decision and Optimal Control

本单元作为Reinforcement Learning for Sequential Decision and Optimal Control这本书第一单元的读书笔记，算是一个入门的笔记。介绍了强化学习的历史渊源和面临的挑战。这个单元的内容以介绍性的为主，因此就简单写写。

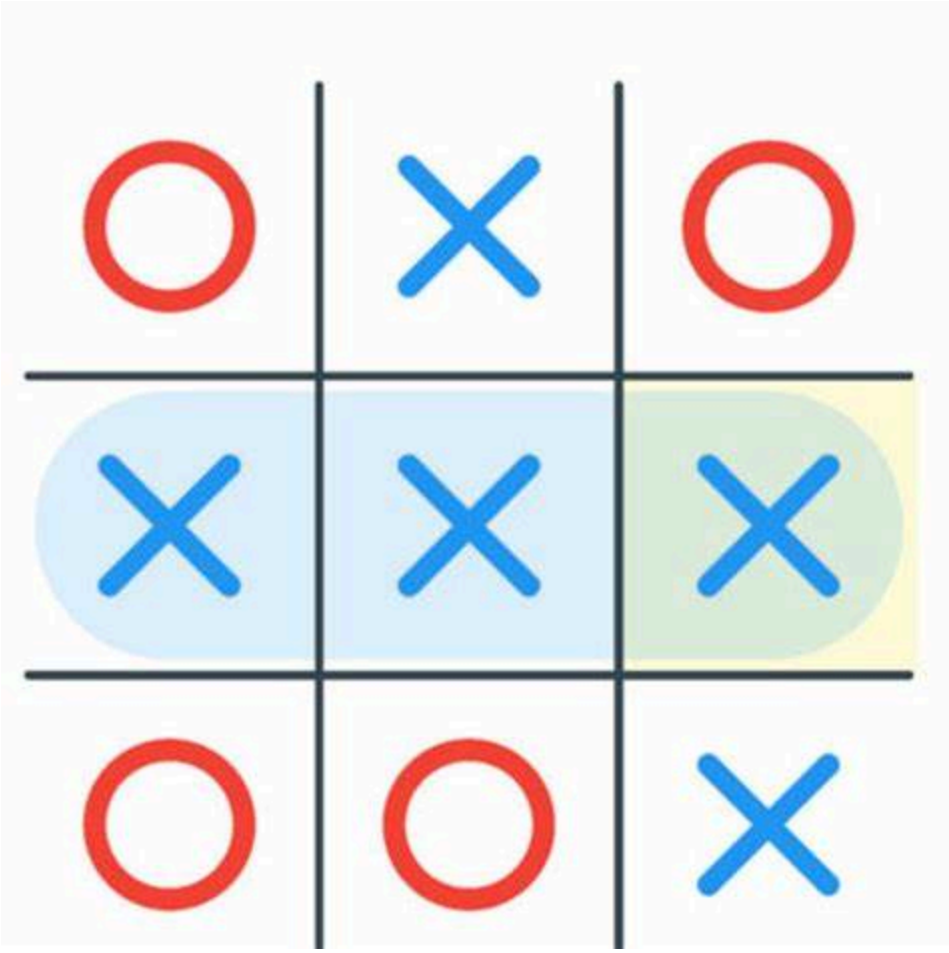
强化学习的历史发展

最优控制理论

其实RL和最优控制理论有很深厚的渊源。最优控制与RL在建模上有诸多相似之处，并且最优控制中的诸多方法也被RL所采用。变分、庞特里亚金极大值原理、动态规划（DP）
是最优控制的三大支柱。其中，DP以及其背后的贝尔曼最优化原理是RL的基础，在后面的博客中会详细介绍。

Trial-and-Error Learning

Trial-and-Error Learning主要是模仿生物体的学习过程，通过尝试和错误来学习。通过这种方式可以使得智能体学会区分“好”和“坏”的行为。实际上，这种学习方式在人工智能发展的早期就被提出，如香农的机械老鼠和Donald Michie的MENACE（用于玩tic-tac-toe）。



从Trial-and-Error Learning的思想衍生出著名的Monte Carlo Learning、TD Learning以及Policy Gradient等方法。这些方法会在后续的博客中详细介绍。

强化学习的最新进展

尽管之前已经提出了一些强化学习方法，但是这些方法在处理高维的问题时很难取得好的效果。将深度学习与强化学习结合而产生的深度强化学习（DRL）是强化学习的又一次里程碑。而这种方法的代表作是AlphaGo。在AlphaGo与李世石的比赛中，AlphaGo以4:1的比分战胜了李世石。这次胜利也使得深度学习/强化学习走入了大众的视野，并在近些年来逐渐成为显学。

强化学习面临的挑战

尽管强化学习在近年来取得了长足的进步，但是仍然面临着一些挑战。这些挑战导致了强化学习还无法在我们日常的生活中的各种应用场景取得广泛的应用。这些挑战主要有以下几个方面。

探索-利用困境

在强化学习中，智能体需要在探索新的行为和利用已有的行为之间取得平衡。如果智能体只是利用已有的行为，那么它就无法发现新的行为，从而无法学习到最优的策略。而如果智能体只是探索新的行为，那么它就无法利用已有的经验，从而无法学习到最优的策略。

不确定性和部分可观测性

传统的强化学习方法主要关注于完全可观测的环境。然而，现实世界中的环境往往是不确定的和部分可观测的，也就是说这里的不是所有的state都是可以观测的，但是state恰恰是强化学习建模的基础。如果存在部分可观测性的话，就需要引入一些对于状态的估计，而这些估计会引入额外的误差。

奖励的延迟

在某些情况下，奖励不是立即给出的，而是具有很大的延迟。这就导致了这些奖励信号经过时间的传递后，会被“稀释”，导致携带不了什么有用的信息，因此会导致学习效果不佳，很难收敛。

安全约束导致的不可行性

安全约束为寻找策略引入了额外的限制条件。而寻找最优策略和确定策略的可行域是两个相互耦合的问题，从而导致求解的困难。

变化的环境

几乎所有学习算法都要求数据独立同分布（IID），强化学习也不例外。然而，强化学习在实际应用中恰恰面临着环境的变化，这就导致了数据的非IID性，从而导致了学习效果的下降。如果环境变化的太快，就可能导致无法学得一个稳定的策略。

泛化能力差

这里的泛化能力查来自于两方面。
一是现有的强化学习方的算法设计与数据来源导致其往往专精于一个特定的任务，而无法泛化到其他的任务。这就导致了强化学习在实际应用中的局限性。二是很多强化学习的任务首先在模拟环境中进行训练，然后再在真实环境中进行测试。然而，模拟环境是对于真实环境的一种简化，可能无法很好的模拟真实环境，从而导致了模型在真实环境中的表现不佳。

@[toc]

书籍链接：Reinforcement Learning for Sequential Decision and Optimal Control

本次笔记对应于原书第二单元的内容，主要介绍了RL的基本要素：环境、state、action、reward、policy以及RL的分类、学习效果的衡量方式，并从更广泛的角度探讨了RL。

2.1 强化学习的基本要素

2.1.1 环境模型

简单来说，环境模型就是给出当前的状态s和动作a，可以给出下一个状态s。为了后面更好的理解，在这里需要介绍一个相关概念——马尔可夫过程。

马尔可夫过程在机器学习、生物、化学等一系列建模种都常常被使用。该过程具有无记忆性，即未来的状态只与当前状态有关，与过去的状态无关。这也就是说，当前状态s已经蕴含了足够的历史信息。马尔可夫过程是随机过程里研究的一个重要领域，这里不做深入讨论，仅就马尔科夫链进行介绍。

一个马尔科夫链的状态空间$\mathcal{S}$和动作空间$\mathcal{A}$具有有限个元素（即离散的）。除了上述动作和状态空间，马尔可夫链还包括了状态转移概率矩阵$\mathcal{P}$，其中$\mathcal{P}(s'|s,a)$表示在状态s下采取动作a后转移到状态s'的概率。这里需要注意的是，状态转移概率矩阵是一个三维矩阵，即$\mathcal{P} \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{S}|\times|\mathcal{A}|}$。马尔可夫过程也可通过一个带权的有向图来表示，其中节点表示状态，边表示状态转移的概率（不同边的权值代表采取不同的动作导致的不同状态转移概率）。当状态部分客观时（即第一篇博客里介绍过的RL的困境之一），可以通过隐马尔可夫模型(HMM)进行建模。这里不仅包含上述状态之间转移的概率，还包含了状态到观测之间的概率。

2.1.2 State-Action Sample

在RL中，由state和action组成的sample蕴含着环境的信息，也是后续一系列算法的基础。这里介绍一下有关的概念。

- **sample**：三元组(s_t, a_t, s_{t+1})，其中s_t表示当前状态，a_t表示当前动作，s_{t+1}表示下一个状态。
- **trajectory**：由一系列sample按照时间顺序排列组成，即$s_0, a_0, s_1, a_1, ..., s_T$。
- **dataset**：可以包含一条或多条trajectory，即

$$D \overset{\text{def}}{=} \left\{ \underbrace{s_0, a_0, s_1}_{\text{Sample}}, a_1, s_2, a_2, \cdots\cdots, \underbrace{s_t, a_t, s_{t+1}}_{\text{Sample}}, a_{t+1}, \cdots \right\},$$

2.1.3 Policy

所谓策略就是一种从state道action的映射，即$\pi:\mathcal{S}\to\mathcal{A}$。有了策略之后就引入了一个重要的概念：马尔可夫决策过程（**MDP**）。一个具有马尔可夫环境和策略的问题可以被建模为一个MDP。MDP的策略有一个重大特点，即动作只取决于当前的状态，而不是历史状态。这也是MDP的马尔可夫性质。

2.1.3.1 策略的分类

主要分为两类，即确定性策略和随机策略。

- **确定性策略**：$\pi(s):\mathcal{S}\to\mathcal{A}$，即对于每一个状态s，都有一个确定的动作a。
- **随机策略**：$\pi(a|s):\mathcal{S}\times\mathcal{A}\to[0,1]$，即对于每一个状态s和动作a，都有一个概率。也就是说，随机策略是一个分布。

2.1.3.2 策略的表示

- **表格表示**：对于状态空间和动作空间较小且离散的情况（有限集、低维），可以使用表格来表示策略。在早期的RL算法中，常常使用表格来表示策略。
- **参数表示**：对于状态空间和动作空间较大且连续的情况，可以使用函数来表示策略。而函数中包含一些参数。常用的函数为多项式函数、神经网络等。使用参数表示可以节省计算和存储空间，也可以纳入一些先验知识。对于确定和随即策略，参数表示分别为：
  - **确定性策略**：$\pi_\theta(s)$，其中θ是参数。
  - **随机策略**：$\pi_\theta(a|s)$，其中θ是参数。

2.1.4 Reward

接下来就到了RL里的一系列核心概念——奖励（reward）、回报（return）、值函数（value function）等。它们的关系我这里画了一张图来更形象的说明：

2.1.4.1 Reward

$$r_{ss'}^{a} \overset{\text{def}}{=} r_t = r(s_t, a_t, s_{t+1}).$$

reward可以看作从sample三元组到实数的映射，即$r:\mathcal{S}\times\mathcal{A}\times\mathcal{S}\to\mathbb{R}$。上面这种形式的reward保证了reward只与当前状态、动作和下一个状态有关，而与历史状态无关。这种reward的构建方式与一个马尔可夫环境结合，就保证了我们可以把一个序列决策问题拆分为若干小的子问题进行求解。这也是后续可以利用Bellman最优性原理构建Bellman方程的基础。

2.1.4.2 （Long-term）Return

return是一个与reward紧密相连却又有所区别的概念，两者很容易在刚学的时候搞混，因此需要着重区分。return是一个序列的reward的和的函数，主要有两种形式：

- **Discounted return**：采用折扣因子γ进行加权求和，γ是一个0到1的数来保证序列的收敛性。即

$$G_t^{\gamma} = \sum_{k=0}^{\infty} \gamma^k r_{t+k}.$$

- **Average return**：从起始到终止的reward的平均值，即

$$G_t^{avg} = \lim_{T\to\infty} \frac{1}{T} \sum_{i=0}^{T-1} r_{t+i}$$

Discounted return更适用于infinite horizon的、stochastic问题，而average return更适用于finite horizon的、deterministic（这样的话能收敛，自然为有限的）问题。总的来说，Discouted return更常用。在以后的笔记中若无特殊说明，记号G_t表示Discounted return。

2.1.4.3 值函数

想必对于RL有一些了解的话都听说过值函数。鉴于之前对于这个概念的认识比较模糊，我在这里根据原书的内容再系统梳理一遍。

值函数是return的期望，因此也叫expected return，它常常被用于衡量一个策略的好坏。值函数有两种形式：状态值函数（state value function）和动作值函数（action value function）：

- **状态值函数**：

$$v^{\pi}(s) \overset{\text{def}}{=} \mathbb{E}_{\pi}\{G_t|s\} = \mathbb{E}_{\pi}\left\{\sum_{i=0}^{+\infty}\gamma^i r_{t+i}|s_t = s\right\}$$

值函数的定义也暗示了这里的策略π是一个随机策略。**因为如果是确定策略的话，就不存在期望了。上式直接就退化为return的定义式了。**

- **动作值函数**：

$$
\begin{aligned}
q^{\pi}(s,a) &\overset{\text{def}}{=} \mathbb{E}_{\pi}\{G_t|s,a\} \\
&= \mathbb{E}_{\pi}\left\{\sum_{i=0}^{+\infty}\gamma^i r_{t+i} \mid s_t=s, a_t=a\right\}
\end{aligned}
$$

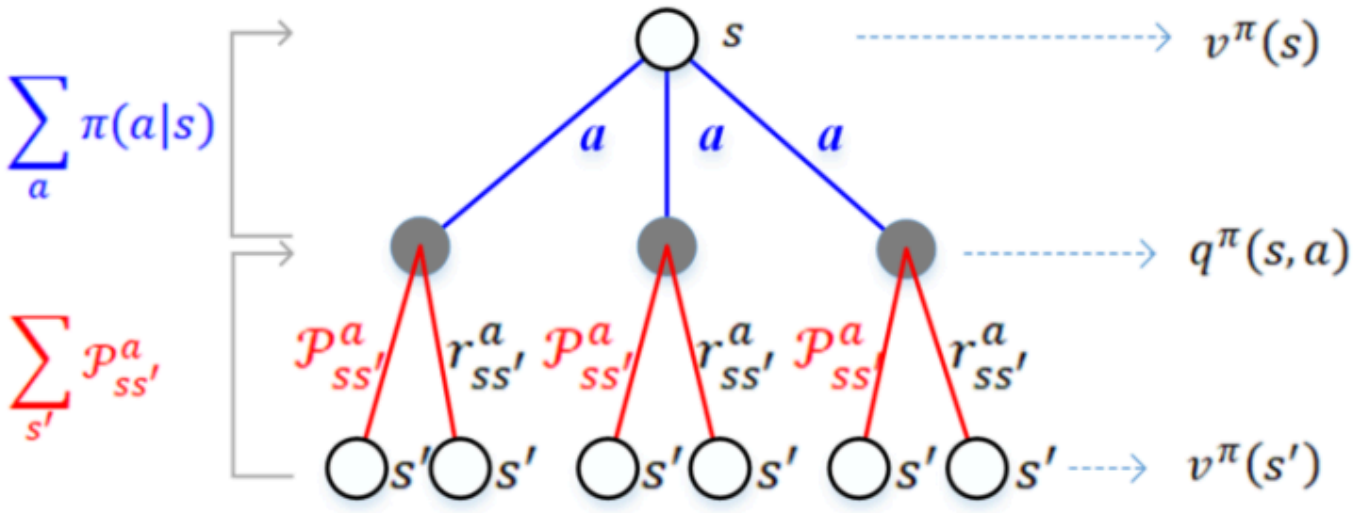
两种值函数之间以及内部均有关系，这也是值函数的一个重要性质。首先是用动作值函数来表示状态值函数：

$$
v^{\pi}(s)=\sum_{a\in\mathcal{A}}\pi(a|s)q^{\pi}(s,a)
$$

这个式子的意义是，一个状态的值函数等于在这个状态下所有动作的值函数的期望。$\pi(a|s)$是一个概率值（即随即策略的分布），因此该式子的含义也可解释为固定某一个状态s，枚举所有可能的动作a后对于所有的动作值函数$q^{\pi}(s,a)$进行加权求和。接下来再看看如何使用状态值函数来表示动作值函数：

$$
q^{\pi}(s,a)=\sum_{s'\in\mathcal{S}}\mathcal{P}(s'|s,a)\left(r_{ss'}^{a}+\gamma v^{\pi}(s')\right)
$$

这个式子其实也很好理解。这里我们已知了当前的状态s和动作a，那么为了使用状态值函数来表示动作值函数，我们需要考虑到下一个状态s'并利用下一个状态的状态值函数$v^{\pi}(s')$。考虑到我们的策略为随即策略，因此下一个状态s'并不是固定的，而是需要对于状态空间中所有可能的状态进行加权求和。这里的加重就是状态转移概率$\mathcal{P}(s'|s,a)$。这里主要部分就讲解完了。那么为什么还要加上一个reward $r_{ss'}^{a}$呢？这是由于状态值函数和动作值函数处于不同的“位置”导致的。放一张图看得更清楚一点：



可以看出，动作值函数$q^\pi(s,a)$介于状态值函数$v^\pi(s)$和$v^\pi(s')$之间，因此需要加上一个reward来进行修正。有了上述两个描述两种值函数之间关系的式子，就可以通过代入消元得到两种值函数内部的递归关系：

$$v^\pi(s) = \mathbb{E}_\pi\{r + \gamma v^\pi(s')|s\} = \sum_{a\in s^a} \pi(a|s) \left\{ \sum_{s'\in\mathcal{S}} \mathcal{P}(s'|s,a) \left(r^a_{ss'} + \gamma v^\pi(s')\right) \right\}$$

$$\begin{aligned} q^\pi(s,a) &= \mathbb{E}_\pi\{r + \gamma v^\pi(s')|s,a\} \\ &= \sum_{s'\in\mathcal{S}} \mathcal{P}(s'|s,a) \left(r^a_{ss'} + \gamma \sum_{a'\in\mathcal{A}} \pi(a'|s') q^\pi(s',a') \right) \end{aligned}$$

这里两个式子的得到无需多言，就是代入就行了。这两个式子的最重要的贡献再与在值函数与策略之间建立了联系，这也给出了如何评估一个策略好坏的方法。这也是后续RL算法的基础。评估方法可分为sample-based和value-based，在后续博客会有介绍。以上两式也被称为self-consistency condition。

2.2 强化学习的定义

这里给出了强化学习算法的定义：

$$\max_\pi / \min_\pi J(\pi) = \mathbb{E}_{s\sim d_{\text{init}}(s)}\{v^\pi(s)\},$$

$$\text{subject to}$$
$$(1)\Pr\{s_{t+1}=s'|s_t=s,a_t=a\}=\mathcal{P}_{ss'}^a,$$
$$or$$
$$(2)\mathcal{D}=\{s_0,a_0,s_1,a_1,s_2,a_2,\cdots\cdots,s_t,a_t,s_{t+1},a_{t+1},\cdots\},$$
$$a\in\mathcal{A},s\in\mathcal{S},$$

这里的$J(\pi)$是一个关于策略π的函数，表示策略π的好坏，叫做 overall objective function （或者 performance index）；$d_{\text{init}}(s)$是一个初始状态的分布；$v^{\pi}(s)$是策略π的状态值函数。约束条件为各个状态之间的转移概率或者是一个数据集$\mathcal{D}$。

2.3 强化学习的分类

强化学习的分类有很多种，这里介绍几种常用的分类方法：

- **按环境模型是否已知**：
  - **Model-based**：环境模型已知。这类问题很类似于最优控制，所用到的知识很多也来自于最优控制。
  - **Model-free**：环境模型未知。这类问题中我们不知道环境的具体模型，只能通过与环境的交互来学习。这实际上就是Trial-and-error的过程。描述此类算法的很多概念来自于统计学习。
- **按是否使用了最优性条件**：
  - **Indirect RL**：使用最优性条件。这类算法使用、Bellman方程或者 Hamilton-Jacobi-Bellman的解；来作为最优策略。
  - **Direct RL**：不使用最优性条件。这类算法直接学习策略，在整个策略空间中进行搜索。

这里以书中的的表格做一个总结：

	Model-based	Model-free
Indirect	DP, ADP, HDP, ADHDP, DHP, GDHP, ADGDHP, CDADP, DGPI	MC, SARSA, Q-learning, A2C*, A3C*, DQN, GAE, DDQN, Dueling DQN, C51, Rainbow, NAF, R2D2, HRA
Direct	PILCO, GPS, I2A, MVE, STEVE, ME-TRPO, SLBO, MBPO, DMVE, MBMF	TRPO, PPO, DPG, DDPG, Off-PAC, ACER, REACTOR, IPG, TD3, SAC, DSAC, Trust-PCL, SIL, APE-X, IMPALA, PPG

对这里表格中具体名词的解释参考原书或者后续博客内容。下面重点讲一讲上面提到的Indirect RL和

Direct RL。这两种方法与贝尔曼最优性原理有着密切的联系，详见贝尔曼最优性原理。注意，这里只是先介绍一下Indirect RL和Direct RL的思想，具体的算法会在后续博客中专门介绍。

2.3.1 Indirect RL

这里的核心思想是：既然Bellman方程是最优策略的充要条件，那么就可以通过求解Bellman方程来得到最优策略。求解的方法又可进一步分为策略迭代（ policy iteration）和值迭代（ value iteration）。

2.3.1.1 策略迭代

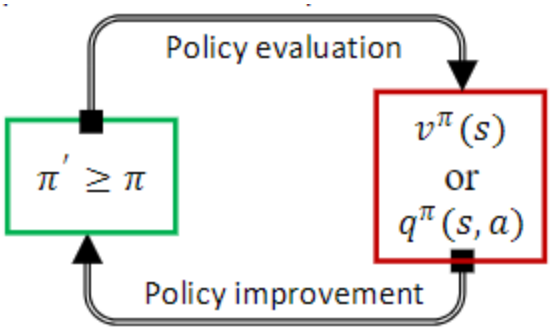
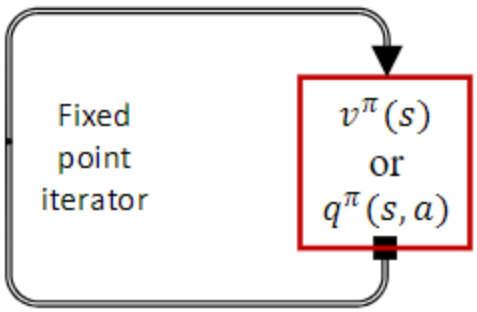


Figure 2.5 Policy iteration

策略迭代分为两步：策略评估和策略改进。首先先根据策略对应的值函数来评估策略的好坏，然后再根据评估的结果来改进策略。这个过程是一个迭代的过程，直到策略不再发生变化为止。

2.3.1.2 值迭代



把值函数当作需要迭代的变量，使用ixed-point iteration计数进行迭代。当找到最优值函数后，最优策略就很容易得到了（采用greedy策略即可）。

2.3.2 Direct RL

Direct RL的核心思想是直接学习策略，而不是通过求解Bellman方程来得到最优策略。这里求解的核心思想就是直接优化一个performance index $J(\pi)$。

$$\theta^* = \arg\max_{\theta} J(\theta) = \arg\max_{\theta} \mathbb{E}_{s\sim d_{\text{init}}(s)}\{v^{\pi_\theta}(s)\},$$
$$\theta \leftarrow \theta + \alpha \cdot \nabla_\theta J(\theta)$$

注意，这里的方法就很类似于深度学习里面那一套了。
最常用的就是一阶优化，如梯度下降。其实这里就相当于求解一个优化问题，有很多方法可以用。为什么不用二阶优化，如牛顿法等？其实更多还是基于计算的考虑，因为RL的问题往往是高维的，计算复杂度很高。因此，直接优化方法更加常用。

从这里也可以看出，Direct RL的关键点有两个，一是指标$J(\theta)$的选择，二是优化方法的选择。这两个问题的解决是Direct RL的核心。

2.4 贝尔曼最优性原理

后续内容的核心，这里有必要重点讲一讲。

2.4.1 贝尔曼最优性原理

原始的贝尔曼最优性原理是这样的：无论初始的状态和决策是什么，剩下的决策仍然构成一个最优策略。

2.4.2 最优值函数和最优策略

最优策略比其他任何策略都要好，至少不会更差。而最优值函数是最优策略对应的值函数。它们的定义如下：

$$v^*(s) \overset{\text{def}}{=} v^{\pi^*}(s) = \max_{\pi} v^{\pi}(s), \forall s \in \mathcal{S}.$$

其中，π^*是最优策略。同样的，最优动作值函数也有类似的定义：

$$q^*(s,a) \overset{\text{def}}{=} q^{\pi^*}(s,a) = \max_{\pi} q^{\pi}(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}.$$

二者的关系可由类似于上面推导两种值函数的关系那里的方法得到：

$$\begin{aligned} v^*(s) &= \max_{a \in \mathcal{A}} q^*(s,a), \forall s \in \mathcal{S}, \\ q^*(s,a) &= \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a) \left(r^a_{ss'} + \gamma v^*(s') \right), \forall s \in \mathcal{S}, \forall a \in \mathcal{A}. \end{aligned}$$

值得注意的是，为什么第一个式子是一个max而下面的第二个是一个求和呢？第一个式子是max是因为

在状态s下对应于最优策略的最优动作a*只有一个（或者说有多个但是它们的动作值函数值相同）。而第二个式子是求和是因为在状态s下对应于最优策略的最优动作a*只有一个，但是即使对于同样的状态s和动作a，下一个状态s'也是不确定的，因此需要对所有可能的下一个状态进行加权求和。当获得了最优的动作值函数后，最优策略就很容易得到了：

$$a^* = \arg\max_a q^*(s,a).$$

2.4.3 贝尔曼方程

2.4.3.1 状态值函数的贝尔曼方程（第一类贝尔曼方程）

把最优值函数那里的二式代入一式，可得第一类贝尔曼方程：

$$v^*(s) = \max_{a\in\mathcal{A}} \sum_{s'\in\mathcal{S}} \mathcal{P}(s'|s,a)\left(r^a_{ss'} + \gamma v^*(s')\right), \forall s \in \mathcal{S}.$$

这个式子的含义可以这样理解：在状态s下，最优的状态值函数$v^*(s)$等于在所有可能的动作a中选择最优的动作a*后对于所有可能的下一个状态s'进行加权求和（期望）。

2.4.3.2 动作值函数的贝尔曼方程（第二类贝尔曼方程）

把最优值函数那里的一式代入二式，可得第二类贝尔曼方程：

$$q^*(s,a) = \sum_{s'\in\delta} \mathcal{P}(s'|s,a)\left(r^a_{ss'} + \gamma \max_{a'\in\mathcal{A}} q^*(s',a')\right), \forall s \in \mathcal{S}, \forall a \in \mathcal{A}.$$

2.5 从更广阔的角度看强化学习

2.5.1 初始状态的分布的影响

这里要讨论一个有意思的问题，即初始状态对于RL算法求出的策略到底有没有影响。为了回答这个问题，首先需要定义两种由不同方法求出的最优策略。

- **由最大化指标$J(\pi)$求出的最优策略π^o**：这种方法是直接优化$J(\pi)$，即直接优化performance index。这种方法的最优策略记为π^o。这种求解方法我的理解就是上面见过的Direct RL。

$$\pi^\circ(s) = \arg\max_{\pi(s)\in\Pi} J\big(\pi(s)\big)$$

- **由贝尔曼方程求出的最优策略π^***：这种方法是通过求解贝尔曼方程得到的。这种方法得出的最优策略因为在计算的时候已经枚举了所有状态，因此不受初始状态的影响。

$$\pi^*(s) = \arg\max_{a\in\mathcal{A}} \sum_{s'\in\mathcal{S}} \mathcal{P}(s'|s,a)\Big(r + \gamma v^*(s')\Big), \forall s \in \mathcal{S},$$

该式可通过把最优值函数那里的第一个式子代入第二个式子得到（或者从第一类贝尔曼方程也可得到）。

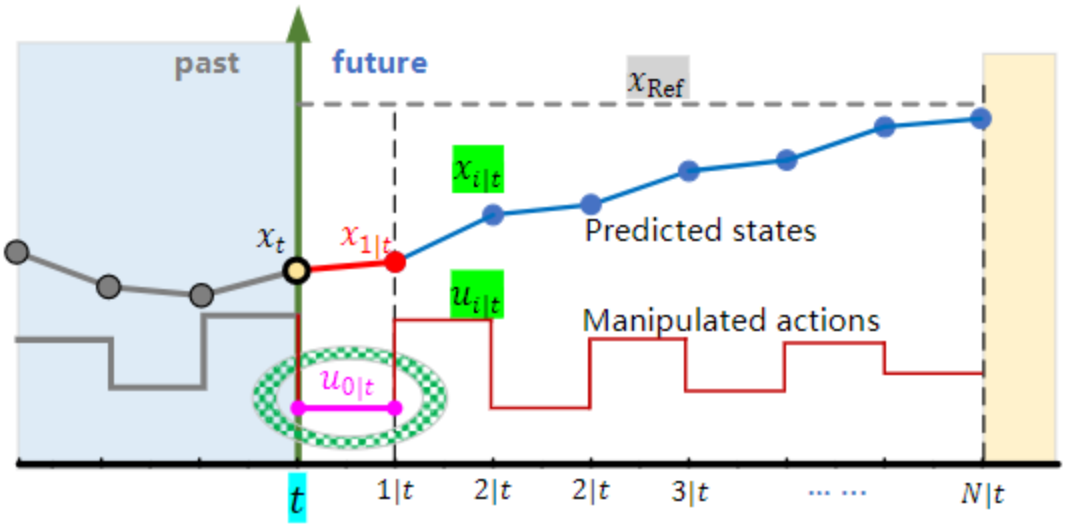
下面要讨论两种情况，即由贝尔曼方程求出的最优策略π^*在不在策略集合Π中。这里的策略集合指的是所有可行的策略的集合，主要是由策略的表现形式而非策略本身来决定的。比如当我们采用线性函数来参数化的表达策略时，自然有些策略是不可行的。

- **情况一：**$\pi^* \in \Pi$：这种情况下，由贝尔曼方程求出的最优策略π^*与由最大化指标$J(\pi)$求出的最优策略π^o是一样的。这种情况下，初始状态的分布对于最优策略是没有影响的。此时$J(\pi^*) = J(\pi^\circ)$。
- **情况二：**$\pi^* \notin \Pi$：这种情况下，由贝尔曼方程求出的最优策略π^*与由最大化指标$J(\pi)$求出的最优策略π^o是不一样的。此时$J(\pi^o) \leq J(\pi^*)$。此时，初始状态的分布对于**由最大化指标$J(\pi)$求出的最优策略π^o**是有影响的。如果使用线性函数这种本身对于策略空间有较大限制的方法，初始状态会对于求解有很大的影响。但是值得指出的是，因为目前使用的参数化策略表示方法通常是神经网络，而神经网络的表达能力很强，其能表达的策略空间接近于整个策略空间，因此两种方法求出的最优策略是基本一样的。

关于以上两种情况的详细数学推导参看原书即可。

2.5.2 RL和MPC的区别

MPC是一种控制方法，即model predictive control。



上图展示了一种追踪一个固定参考信号（图上灰色虚线）的过程。在时间t时，MPC会根据当前的状态s

和参考信号r，通过最小化一个cost function来得到一个未来若干步的action序列，然后把序列的第一个作为input给系统。

MPC和RL的区别如下：

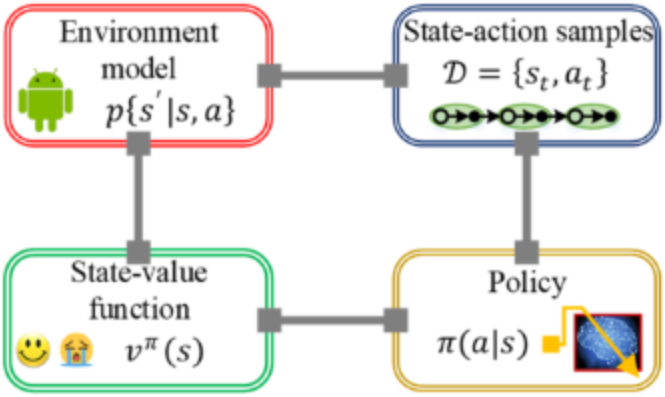
	RL (Discounted cost)	MPC (Infinite horizon)
•	Stochastic system (e.g., environment)	Deterministic system (e.g., plant and process)
•	State & Action (s, a)	State & Control input (x, u)
•	Probabilistic model $\Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\} = \mathcal{P}_{ss'}^{a}$	State-space model $x_{t+1} = f(x_t, u_t)$
•	Policy $\pi(a \mid s)$	Controller $u = \pi(x)$
•	Reward signal $r(s, a, s')$	Utility function $l(x, u)$
•	State-value function $v^{\pi}(s) = \mathbb{E}_{\pi}\left\{\sum_{i=0}^{+\infty} \gamma^i r_{t+i} \mid s_t = s\right\}$	Infinite-horizon cost function $V(x) = \sum_{i=0}^{+\infty} l(x_{t+i}, u_{t+i})\big\vert_{x_t = x}$
•	Maximize a weighted state-value function $\max_{\pi} \mathbb{E}_{s \sim d(s)}\{v^{\pi}(s)\}$	Minimize a cost function $\min_{u} V(x)$
•	Self-consistency condition $v^{\pi}(s) = \sum_{a} \pi(a \mid s)\left\{\sum_{s'} \mathcal{P}\left(r + \gamma v^{\pi}(s')\right)\right\}$	Self-consistency condition $V(x) = l(x, u) + V(x')$
•	Bellman equation $v^{*}(s) = \max_{a} \sum_{s'} \mathcal{P}\left(r + \gamma v^{*}(s')\right)$	Bellman equation $V^{*}(x) = \min_{u}\{l(x, u) + V^{*}(x')\}$

除此之外，它们的区别还有：

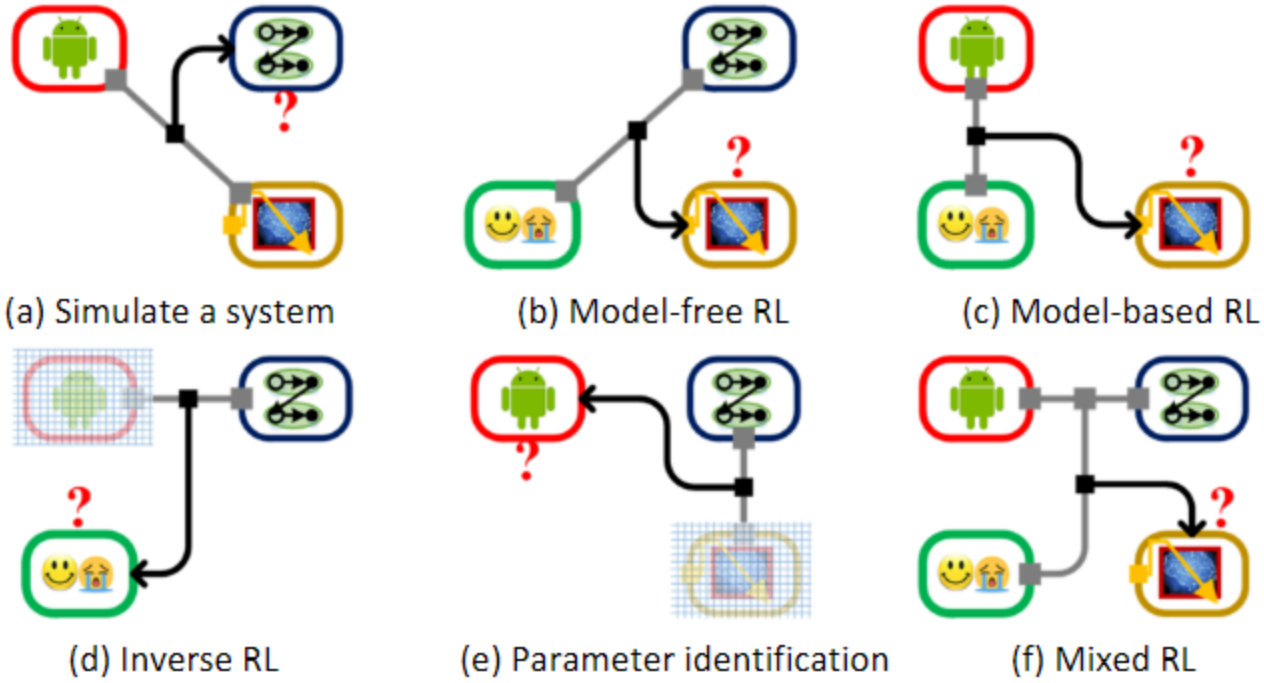
- **需不需要知道环境模型**：MPC需要知道环境模型，而RL不需要。
- **是否在线优化**：MPC是在线优化，即在每一个时间步都会进行优化；而RL是offline优化，可以离线的求出策略之后在线运行。
- **对于不可行state的tolerance**：因为RL是在整个state space内进行搜索，因此不可避免地会遇到一些不可行的state。而MPC则是在state与state之间进行转移，对于不可行state的容忍度更高。

2.5.3 RL的四个要素的不同组合

RL由四个要素构成：$\{\mathcal{D}, \pi, v^{\pi}, \mathcal{P}\}$，分别是数据集、策略、值函数和环境模型。



这四个要素的不同组合对应于不同的问题：



由图可得具体的问题形式，图中阴影部分表示可选的部分。

2.6 RL的学习效果的衡量

2.6.1 模型表现

$$R_{\text{Avg}} = \sum d_{\text{init}}(s)\Big\{\frac{1}{N}\sum_{i=1}^{N} G_i(s)\Big\},$$

$$G_i(s) = \sum_{t=0}^{L_i-1} r_t,$$

total average return (TAR)是一种常见的衡量RL算法性能的指标。这个指标的计算方法是：首先对于每一个初始状态s，计算出它的平均回报$G_i(s)$，然后再对所有的初始状态s求和。对于每个初始状态s，我们都会进行N次实验，每次实验的长度为L_i。注意，对于不同的初始状态对应的实验长度L_i可能是不同的，而且这个长度不是必须的。当任务为episodic时，会自动达到终止状态而停止；当任务为continuing时，才需要显式指定长度。

另外注意，这里在计算$G_i(s)$的时候并没有像之前的return那样加上折扣因子γ，这是因为这里的目的是衡量算法的性能，而不是衡量长期的回报。这样也能为采取不同的折扣因子的算法提供一个公平的比较，防止折扣因子的选择对于算法性能的影响（比如恰好训练和评估的折扣因子选为一样的）。

最后，对于某些随机任务，我们只评估其确定性的部分。因为对于最优策略来说，其确定性部分应是一样的。

2.6.2 （学习）准确率

$$R_{\text{Avg}} = \sum d_{\text{init}}(s)\Big\{\frac{1}{N}\sum_{i=1}^{N} G_i(s)\Big\},$$
$$G_i(s) = \sum_{t=0}^{L_i-1} r_t,$$

准确率就是对于策略$\pi(s)$或者状态值函数$v^{\pi}(s)$的估计与真实值之间差距的平方按照每个状态s的分布的加权平均后的均方根。但是这里就有个很大的问题，在求出最优策略之前我们怎么知道真实值呢？这本身看上去就像一个悖论，因此这个指标一般只适用于一些极其简单的问题。

2.6.3 学习时间

学习时间通常有两种表现形式：

- **时钟时间**：即算法运行的实际时间。
- **迭代次数**：即算法运行的迭代次数。

后者常常更常用，因为不同的电脑运行速度不同，时钟时间并不具有普适性。通常取算法到达某个点的时间作为学习时间。

2.6.4 Sample Efficiency

这个指标是指算法需要多少**新**样本才能达到某个性能水平。这个指标是一个很重要的指标，因为在很多实际问题中，样本是很宝贵的。这个指标的计算方法是：在达到某个性能水平时，算法所用的**新**样本数。对于重复使用的历史样本不计入其中。

2.6.5 Approximation准确率

$$\mathrm{RMSE}_{v\mathrm{Appr}} = \sqrt{\sum_{s\in\mathcal{S}} d_\pi(s)\left(V^\pi(s;w) - v^\pi(s)\right)^2}$$

不同于学习准确率关注当前策略与最优策略之间的差距，Approximation准确率关注的是使用当前的参数来逼近真实的值函数的准确率。同样的，真实的值函数是未知的，因此这个指标也只适用于一些简单的问题。或者可以通过在当前策略下与环境多次交互收集数据来近似得到真实值函数。

@[toc]

书籍链接：Reinforcement Learning for Sequential Decision and Optimal Control

这篇博客主要介绍一种Model-free的Indirect RL方法——Monte Carlo Learning。这里的Model-free指的是Monte Carlo Learning不需要环境的模型，而是通过和环境的交互来学习。Indirect RL指的是Monte Carlo Learning不直接优化策略，而是通过求解Bellman方程来学习策略。更具体地来说，Monte Carlo Learning采用的是策略迭代（Policy Iteration）的方法。注意到上一篇博客讲到策略迭代是并没有具体说Policy Evaluation和Policy Improvement是如何实现的，这篇博客将会详细介绍在Monte Carlo Learning中如何实现Policy Evaluation和Policy Improvement。在下文中Monte Carlo将简称为MC。

Monte Carlo Learning的核心思想是通过采样的方法来估计状态值函数和动作值函数。通过与环境的交互可以获得很多轨迹（Trajectory），根据这些轨迹我们可以计算出值函数的平均值。随着采样数目的增加，值函数的估计值逐渐收敛到其真实值。这就是MC的基本思想。

3.1 MC Policy Evaluation

MC Policy Evaluation最适合那些有明确的终止时间或易于终止的任务。在下面的讨论中，我们来关注上述的那种任务——episodic task，且这些任务的状态和动作空间都是离散且有限的。

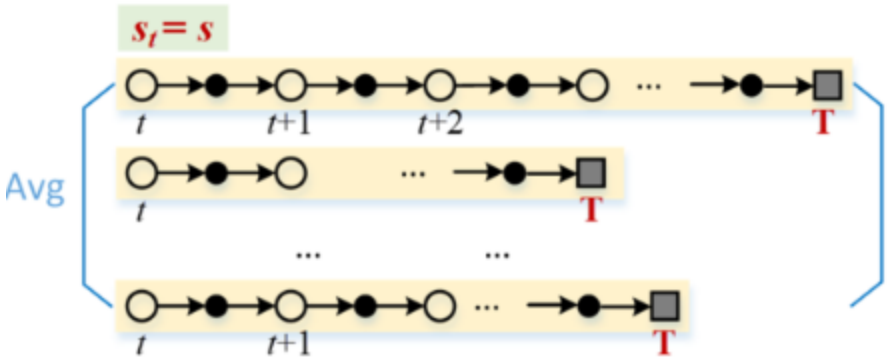
下面我们来看看如何估计两种值函数

- State Value Function: 估计某个状态s的值函数$V(s)$的方法是收集多次从状态s开始得到的return，然后求这些return的平均值。

$$V^{\pi}(s) = \mathrm{Avg}\{G_{t:T}|s_t = s\},$$
$$G_{t:T} = \sum_{i=0}^{T-t} \gamma^i r_{t+i}.$$

  这里的$G_{t:T}$是从时刻t开始，在T时刻终止的return，γ是折扣因子，r_{t+i}是时刻t+i的reward。注意对于episodic task，T是终止时刻，且不同轨迹的终止时刻可能不同。$V^{\pi}(s)$是从状态s的状态值函数的估计值，而$v_{\pi}(s)$是其真实值。随着采样数目N的增加，$V^{\pi}(s)$逐渐收敛到$v_{\pi}(s)$，即：

$$\lim_{N\to\infty} V^{\pi}(s) = v_{\pi}(s).$$



- Action Value Function: 估计某个状态s下某个动作a的值函数$Q(s,a)$的方法是收集多次从状态s执行动作a得到的return（即从状态-动作对(s, a)出发得到的return），然后求这些return的平均值：

$$Q^{\pi}(s,a) = \mathrm{Avg}\{G_{t:T} | s_t = s, a_t = a\}.$$

当环境模型已知时，估计状态值函数和动作值函数均可；当环境模型未知时，只能估计动作值函数。因为仅仅只通过采样，我们无法得到状态转移概率$p(s'|s,a)$，自然也就无法仅根据状态值函数来选择最佳策略。

MC估计的一个重要property是每个状态或状态-动作对的估计值是独立的。这个property使得MC方法可以并行化，并且当我们只需要估计一个整个状态或状态-动作对集合的一个子集时很有效。

当使用Mc来估计动作值函数时，一个严重的问题是很多状态-动作对可能永远不会被访问到。这个问题被称为Exploration Problem。为了解决这个问题，我们可以采用ϵ-greedy策略，即以$1-\epsilon$的概率选择当前最优的动作，以ϵ的概率随机选择动作。这样可以保证所有的状态-动作对最终都会被访问到。另一种方法是随机以所有动作-状态对为起点来采样一个episode。

最后还要讨论一个MC中的重要问题——Multiple Visits。在MC中，一个状态或状态-动作对可能被访问多次。为了解决这个问题，我们可以采用First-Visit MC或Every-Visit MC。First-Visit MC只对每个状态或状态-动作对的第一次访问进行估计（即只把从第一次访问得到的return纳入计算），而Every-Visit MC

对每次访问都进行估计（对于每次访问得到的return都纳入计算）。例如上图中，对于状态s(6)，First-Visit MC只把从左边那个s(6)开始，到T终止的return加到状态s(6)的return和中，最后再平均；而Every-Visit MC则把从左边那个s(6)开始，到T终止的return和从右边那个s(6)开始，到T终止的return都加到状态s(6)的return和中，最后再平均。

3.2 MC Policy Improvement

当环境模型已知时，采用状态值函数即可来改进策略；否则，只能采用动作值函数来改进策略。

3.2.1 Greedy Policy

在MC中，我们采取的策略主要是greedy策略及其变式ϵ-greedy策略。Greedy策略是指在某个状态s下，**一定**选择使得**动作**值函数最大的动作a*，而绝不会选择其他策略。

$$a^* = \arg\max_a q^\pi(s,a).$$

但是这种策略会使得算法完全失去随机性。为了更好地平衡Exploration和Exploitation，我们可以采用ϵ-greedy策略。ϵ-greedy策略是指以$1-\epsilon+\frac{\epsilon}{|A(s)|}$的概率选择当前最优的动作a*，以$\frac{\epsilon}{|A(s)|}$的概率随机选择动作。

$$\pi^\epsilon(a|s) = \begin{cases} 1-\epsilon+\frac{\epsilon}{|\mathcal{A}|}, & \text{if } a = a^* \\ \frac{\epsilon}{|\mathcal{A}|}, & \text{if } a \neq a^* \end{cases}$$

注意，这里为什么要除以动作集$\mathcal{A}$的大小呢？因为这样可以保证$\pi^\epsilon(a|s)$是一个概率分布。即所有动作的概率之和为1。可能这里还令人有点疑惑，通常我们看到的ϵ-greedy策略是以$1-\epsilon$的概率选择当前最优的动作，以ϵ的概率随机选择动作，为什么这里的最优策略的概率除了$1-\epsilon$之外还要再加上$\frac{\epsilon}{|\mathcal{A}|}$呢？其实这里没错，只是一般的描述$\epsilon$-greedy策略的表述有些不准确，其实一般的表述的意思是，最优动作本身独占$1-\epsilon$的概率，而剩下的ϵ的概率平均分配给所有动作（包含）最优动作。

在实际的应用中，我们以估计的动作值函数来替代动作值函数的真实值$q^\pi(s,a)$，即：

$$a^* = \arg\max_a Q^\pi(s,a).$$

也可根据状态值函数和动作值函数的关系来使用状态值函数来选择最优动作：

$$a^* = \arg\max_a \sum_{s'\in\mathcal{S}} \mathcal{P}^a_{ss'}\Big(r^a_{ss'} + \gamma V^\pi(s')\Big).$$

注意，使用状态值函数来选择最优动作时，我们需要知道状态转移概率$\mathcal{P}^a_{ss'}$和奖励函数$r^a_{ss'}$，即要求环境模型已知。而使用动作值函数来选择最优动作时，我们只需要知道动作值函数即可，即不需要环境模

型。注意，计算动作值函数远比计算状态值函数要更加消耗计算资源。

3.2.2 策略改进定理（Policy Improvement Theorem）

直观上说，我们说一个策略π优于另一个策略π，是指对于所有的状态s，π在状态s下的状态值函数要大于等于π在状态s下的状态值函数，即：

$$\pi \leq \bar{\pi} \iff v^{\pi}(s) \leq v^{\bar{\pi}}(s), \forall s \in \mathcal{S}$$

上式也可等价的用**策略改进定理**来表述：

$$\pi \leq \bar{\pi} \iff v^{\pi}(s) \leq \sum_{a \in \mathcal{A}} \bar{\pi}(a|s) q^{\pi}(s,a), \forall s \in \mathcal{S}.$$

这个式子可以这样理解：右端表示动作值函数$q^{\pi}(s,a)$仍使用原策略π下的值，但是在状态s下采取动作条件概率值却是在新的策略$\bar{\pi}$下的值。或者可以看成对于原来状态s下每个动作值函数的值采用新策略下的条件概率作为权重加权平均。这个式子的证明可以参考原书的45到46页（级联的证明还是很有意思的）。由以上证明以及上一篇博客中我们学过的状态值函数和动作值函数之间的关系，可以得到如下不等式：

$$v^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q^{\pi}(s,a) \leq \sum_{a \in \mathcal{A}} \bar{\pi}(a|s) q^{\pi}(s,a) \leq v^{\bar{\pi}}(s) = \sum_{a \in \mathcal{A}} \bar{\pi}(a|s) q^{\bar{\pi}}(s,a).$$

当状态值函数的值不能能再改进时，策略即达到最优。

3.2.3 策略选择

策略选择实际上就是按照策略改进定理，找到一个策略$\bar{\pi}$使得：

$$V^{\pi}(s) \leq \sum_{a \in \mathcal{A}} \bar{\pi}(a|s) Q^{\pi}(s,a), \forall s \in \mathcal{S}.$$

注意这里把真实状态值函数$v^{\pi}(s)$替换成了估计的状态值函数$V^{\pi}(s)$。

3.3 On-Policy v.s. Off-Policy Strategy

在Model-Free的RL中，我们面临exploration-exploitation dilemma。即模型需要得到最优（optimal）策略，但是却需要做出non-optimal的动作来尽可能地探索环境。在on-policy策略中，使用的策略是当前正在学习的策略，即正在学习的策略和指导智能体做出动作的策略是同一个策略。On-policy策略解决上述

困境的方式是采用soft-greedy策略来代替纯粹的greedy策略。Soft-greedy策略中最著名的就是之前说过的ϵ-greedy策略。Off-policy策略则是指学习的策略和指导智能体做出动作的策略是不同的：target policy和behavior policy。其中target policy是我们想要学习的策略，而behavior policy是我们用来采样的策略（指导智能体做出动作与环境交互的策略）。

3.3.1 On-Policy Strategy

首先证明两个定理。
第一，证明greedy policy满足之前讲过的Policy Improvement Theorem：

$$v^{\pi^g}(s) \leq \sum_{a\in\mathcal{A}} \pi^g_{\text{new}}(a|s) q^{\pi^g}(s,a), \forall s \in \mathcal{S},$$

证明如下：

$$\begin{aligned} v^{\pi^g}(s) &= q^{\pi^g}\left(s, \pi^g(s)\right) \leq \max_a q^{\pi^g}(s,a) = q^{\pi^g}\left(s, \arg\max_a q^{\pi^g}(s,a)\right) \\ &= q^{\pi^g}\left(s, \pi^g_{\text{new}}(a|s)\right). \end{aligned}$$

第一个等号成立是因为$v^{\pi}(s) = \sum_{a\in\mathcal{A}} \pi(a|s) q^{\pi}(s,a)$，而greedy策略的分布规律标明了除了对于一个最优策略a*，其他的动作的概率都是0，因此

$$\sum_{a\in\mathcal{A}} \pi^g(a|s) q^{\pi^g}(s,a) = q^{\pi^g}\left(s, a^*\right) = q^{\pi^g}\left(s, \pi^g(s)\right)$$

证明里最后一个等号成立是因为新策略$\pi^g_{\text{new}}(a|s)$是根据选择当前动作值函数的最大值来确定的（正如策略$\pi^g(a|s)$是根据它之前的策略对应的动作值函数来确定的）。
其二，证明ϵ-greedy策略满足Policy Improvement Theorem：

$$v^{\pi^\epsilon}(s) \leq \sum_{a\in\mathcal{A}} \pi^\epsilon_{\text{new}}(a|s) q^{\pi^\epsilon}(s,a), \forall s \in \mathcal{S},$$

证明如下：

$$
\begin{aligned}
v^{\pi^\epsilon}(s) &= \sum_a \pi^{\varepsilon}(a|s) q^{\pi^{\varepsilon}}(s,a) \\
&= \frac{\epsilon}{|\mathcal{A}|} \sum_a q^{\pi^\epsilon}(s,a) + \sum_a \left(\pi^\epsilon(a|s) - \frac{\epsilon}{|\mathcal{A}|} \right) q^{\pi^\epsilon}(s,a) \\
&\leq \frac{\epsilon}{|\mathcal{A}|} \sum_a q^{\pi^\epsilon}(s,a) + \sum_a \left(\pi^\epsilon(a|s) - \frac{\epsilon}{|\mathcal{A}|} \right) \max_a q^{\pi^\epsilon}(s,a) \\
&= \frac{\epsilon}{|\mathcal{A}|} \sum_a q^{\pi^\epsilon}(s,a) + (1-\epsilon) \max_a q^{\pi^\epsilon}(s,a) \\
&= \sum_a \pi^\epsilon_{\text{new}}(s,a) q^{\pi^\epsilon}(s,a).
\end{aligned}
$$

上述等号右边第一式到第二式的意思是把每个动作的概率里面$\frac{\epsilon}{|\mathcal{A}|}$拆出来（第二式第一部分）。第二式第二部分里面$\left(\pi^\epsilon(a|s) - \frac{\epsilon}{|\mathcal{A}|} \right)$对于最优动作等于$1-\epsilon$，对于非最优动作等于0。第三式第二部分到第四式第二部分正如上面所说的，因为$a^* = arg\max_a q^{\pi^\epsilon}(s,a)$，因此其$\left(\pi^\epsilon(a|s) - \frac{\epsilon}{|\mathcal{A}|} \right)$等于1-$\epsilon$。第四式到第五式的理由与第二式到第三式的理由相同。由上述证明可以看出，ϵ-greedy策略由随机部分（第四式第一部分）和贪婪部分（第四式第二部分）组成。这样的策略可以保证探索和利用的平衡。
ϵ-greedy策略的一个缺陷在于它的随机部分（证明的第四式第一部分）对于所有的动作都是相同的，而没有考虑一些动作可能比另一些动作更“好”。为了解决这个问题，可以采用Boltzmann策略：

$$
\pi^B(a|s) = \frac{\exp\left(\tau^{-1} Q(s,a)\right)}{\sum_{a \in c} \exp\left(\tau^{-1} Q(s,a)\right)}, \forall a \in \mathcal{A}, s \in \mathcal{S},
$$

这里的τ是一个温度参数，控制了策略的随机性。当$\tau \to 0$时，Boltzmann策略退化为“exploit-only”地greedy策略；当$\tau \to \infty$时，Boltzmann策略退化为“explore-only”均匀随机策略。Boltzmann策略的一个优点是它可以根据动作值函数的大小来调整随机性，即动作值函数越大的动作被选中的概率越大。触类旁通一下，这里地Boltzmann策略其实是一个Softmax策略，大语言模型中的解码策略也是采用了这个函数（比如调用GPT的API时会看到一个温度系数的参数选项）。由此也可看出，人工智能的各个领域使用的数学工具有很多是相通的。
为了更好的平衡Exploration和Exploitation（前期侧重于Exploration，后期侧重于Exploitation），我们可以动态的调整$\epsilon - greedy$策略中的ϵ或Boltzmann策略中的τ。这种方法被称为Decaying ϵ-greedy策略或Decaying Boltzmann策略。

3.3.2 Off-Policy Strategy

Off-Policy策略通常需要在full-convergence的情况下工作，这样才能保证target policy的所有可能动作都被采样到（至少偶尔会被behavior policy采样到）。Target policy可以是随机策略or确定性策略，而

behavior policy**必须是随机策略**，这样才可保证覆盖到所有可能的动作。下文中我们称target policy为π，behavior policy为b。

3.3.2.1 Importance Sampling（重要性采样,IS）

在off-policy策略中，samples通过behavior policy产生，但是计算值函数却要在target policy下进行。因此，我们**不能直接使用从behavior policy采样得到的return来估计值函数。**为此，我们引入重要性采样（IS）的概念。重要性采样是一种用来估计一个分布下某个随机变量的期望值的方法。在估计时，我们只知道从另一个分布下采样得到的样本（这是自然的，如果我们都有了想求期望的的那个分布下的样本，就不用那么麻烦了，采用简单地估计方法即可得到期望）。
让我们回顾一下概率论里面数学期望的计算方法，对于一个服从分布$d_\lambda(x)$的随机变量x，其数学期望为：

$$\mathbb{E}_{x\sim d(x)}[f(x)]=\int x d_\lambda(x)dx$$

对于随机变量x的函数g(x)，其期望为：

$$\mathbb{E}_{x\sim d(x)}[g(x)]=\int g(x) d_\lambda(x)dx$$

现在回到这里off-policy RL的语境，假设target policy的sample服从的分布为$d_\pi(x)$，behavior policy的sample服从的分布为$d_b(x)$。现在对于一个服从target分布的随机变量x，我们想求其函数h(x)的数学期望，则有：

$$\mathbb{E}_\pi\{h(x)\}=\int d_\pi(x)h(x)\mathrm{d}x$$

现在为了把这个式子和behavior分布$d_b(x)$联系起来，我们引入一个重要性采样比率（**IS Ratio**）$\rho(x)$，定义为：

$$\rho(x)=\frac{d_\pi(x)}{d_b(x)}$$

这个比率的物理意义是，target policy下的采样概率和behavior policy下的采样概率的比值。这个比率的引入是为了把target policy下的期望转化为behavior policy下的期望。注意，尽管在理论上$\rho(x)$是无偏的和恒定的，但是在后续实际根据IS Ratio和从behavior分布得到的samples来更新值函数时，会引入高方差，这可能是由于IS Ratio的商的形式导致的。
这样我们就可以把上面的期望写成：

$$\begin{aligned}\mathbb{E}_\pi\{h(x)\} &= \int d_\pi(x)h(x)\mathrm{d}x \\ &= \int d_b(x)\frac{d_\pi(x)}{d_b(x)}h(x)\mathrm{d}x \\ &= \int d_b(x)\rho(x)h(x)\mathrm{d}x \\ &= \mathbb{E}_b\{\rho(x)h(x)\},\end{aligned}$$

这样就成功把target分布下x的函数h(x)的期望转化为behavior分布下x的函数$\rho(x)h(x)$的期望。这就是重要性采样的核心思想：**维持从不同的分布采样得到的样本有相同的期望（这里即$\mathbb{E}_\pi\{h(x)\} = \mathbb{E}_b\{\rho(x)h(x)\}$）**。这样我们就可以使用一个分布下的样本来估计另一个分布下的期望。但是，注意，这里只是说明两个分布下的两个随机变量函数的期望相等，它们的方差不一定相等。事实上，我们就是要找到一个合适的behavior分布$d_b(x)$，使得target分布的方差尽可能小。总结一下，IS有如下性质：

- **期望相等**：

$$\mu_\pi = \mu_b, \text{where } \mu_\pi = \mathbb{E}_\pi\{h(x)\} \text{ and } \mu_b = \mathbb{E}_b\{\rho(x)h(x)\}.$$

- **方差不相等**：

$$\sigma_b^2 \neq \sigma_\pi^2$$

  关于方差不等的证明如下：

$$\begin{aligned}\sigma_\pi^2 &\overset{\text{def}}{=} \mathbb{D}_\pi\{h(x)\} \\ &= \mathbb{E}_\pi\{(h(x)-\mu_\pi)^2\} \\ &= \mathbb{E}_b\{\rho(x)(h(x)-\mu_\pi)^2\} \\ &= \mathbb{E}_b\{\rho(x)h^2(x)\} - \mu_\pi^2,\end{aligned}$$

  这里第二个等号到第三个等号是因为要在两个分布的期望之间转换需要乘上$\rho(x)$。

$$\begin{aligned}\sigma_b^2 &\overset{\text{def}}{=} \mathbb{D}_b\{\rho(x)h(x)\} \\ &= \mathbb{E}_b\{(\rho(x)h(x)-\mu_b)^2\} \\ &= \mathbb{E}_b\{\rho^2(x)h^2(x)\} - \mu_b^2.\end{aligned}$$

  这里的第二个等号到第三个等号是因为方差的计算性质：

$$\begin{aligned}\mathbb{D}_\lambda\{X\} &= \mathbb{E}_\lambda\{(X-\mu_\lambda)^2\} \\ &= \mathbb{E}_\lambda\{X^2\} - \mu_\lambda^2.\end{aligned}$$

接下来我们关注如何从抽样得到的sample中来估计随机变量的数字特征。现在我们有T个从分布$d_b(x)$数据$x_0, x_1, \cdots, x_{T-1}$，那么$\mu_b$的估计值$\hat{\mu}_b$为：

$$\hat{\mu}_b = \frac{1}{T}\sum_{t=0}^{T-1} h(x_t)\rho(x_t), x_t \sim d_b(x).$$

注意，这里μ_b的估计值也等于μ_π的估计值$\hat{\mu}_\pi$，即：

$$\hat{\mu}_\pi = \hat{\mu}_b$$

σ_b^2的估计值$\hat{\sigma}_b^2$为：

$$\hat{\sigma}_b^2 = \frac{1}{T-1}\sum_{t=0}^{T-1}(h(x_t)\rho(x_t) - \hat{\mu}_b)^2.$$

我们现在希望的是最小方差估计，那么如此选择behavior分布可以使得估计的方差在所有合法（legitimate）的选择中最小：

$$\widetilde{d_b}(x) = \frac{|h(x)|d_\pi(x)}{\int |h(x)|d_\pi(x)\mathrm{d}x}.$$

如此选择后可证σ_b^2的估计值$\tilde{\sigma}_b^2$的上届为σ_b^2，证明如下：

$$
\begin{aligned}
\tilde{\sigma}_b^2 &= \int h(x)^2 \rho(x)^2 \widetilde{d_b}(x) \mathrm{d}x - \mu_b^2 \\
&= \int \frac{h(x)^2 d_\pi(x)^2}{\widetilde{d_b}(x)^2} \widetilde{d_b}(x) \mathrm{d}x - \mu_b^2 \\
&= \int h(x)^2 d_\pi(x)^2 \frac{\int |h(x)| d_\pi(x) \mathrm{d}x}{|h(x)| d_\pi(x)} \mathrm{d}x - \mu_b^2 \\
&= \int |h(x)| d_\pi(x) (\int |h(x)| d_\pi(x) \mathrm{d}x) \mathrm{d}x \\
&= \left(\int |h(x)| d_\pi(x) \mathrm{d}x \right)^2 - \mu_b^2 \\
&= \left(\int |h(x)| \rho(x) d_b(x) \mathrm{d}x \right)^2 - \mu_b^2 \\
&\leq \int h(x)^2 \rho(x)^2 d_b(x) \mathrm{d}x \int d_b(x) \mathrm{d}x - \mu_b^2 \\
&= \int h(x)^2 \rho(x)^2 d_b(x) \mathrm{d}x - \mu_b^2 \\
&= \sigma_b^2.
\end{aligned}
$$

为了理解从第六式到第七式的变化，需要先了解柯西-施瓦茨不等式：

$$
\int [\alpha \bullet \beta] \mathrm{d}\mu \leq \int \alpha \mathrm{d}\mu \times \int \beta \mathrm{d}\mu
$$

有了这个积分不等式之后第六式到第七式的变化就容易理解了。然后第七式到第八式是因为 $\int d_b(x) \mathrm{d}x = 1$（概率密度函数的性质）。

最后来简单谈一谈为什么IS可以降低估计的方差？这是因为随机变量的某些取值对于估计的贡献是不一样的，如果通过对于这些更“重要”的样本更频繁地采样，就可以有效地降低估计的方差。这也是为什么IS的名字叫做“**重要性采样**”的原因。

3.3.2.2 状态值函数的IS Ratio

IS Ratio定义为两个策略下的动作值函数的概率密度的比值，但是实际上我们很难直接计算这个比值。需要根据采样得到的samples来估计。下面就来讲讲如何估计状态值函数的IS Ratio。

假设我们有一个episode，从时刻t开始，到时刻T终止：$a_t, s_{t+1}, \cdots, a_{T-1}, s_T$。那么要估计的状态值函数为：

$$
h(X) \overset{\text{def}}{=} G_{t:T}(a_t, s_{t+1}, \cdots, a_{T-1}, s_T).
$$

现在我们要估计的是多元随机变量$X = \{a_t, s_{t+1}, \cdots, a_{T-1}, s_T\}$的一个概率分布$G_{t:T}(X)$。生成上述episode的概率（这里因为是离散型，所以不是概率密度而是概率）为：

$$d_\pi(a_t, s_{t+1}, \cdots, a_{T-1}, s_T) = \Pr\{a_t, s_{t+1}, \cdots, s_T | \pi, s_t\} = \prod_{i=t}^{T-1} \pi(a_i|s_i) p(s_{i+1}|s_i, a_i),$$

这里前面的$\pi(a_i|s_i)$是policy，而后面的$p(s_{i+1}|s_i, a_i)$是环境模型。对于behavior policy，生成上述episode的概率为：

$$d_b(a_t, s_{t+1}, \cdots, a_{T-1}, s_T) = \Pr\{a_t, s_{t+1}, \cdots, s_T | b, s_t\} = \prod_{i=t}^{T-1} b(a_i|s_i) p(s_{i+1}|s_i, a_i).$$

则此时我们可以得到一个多步的IS Ratio：

$$\rho_{t:T-1} = \frac{d_\pi(a_t, s_{t+1}, \cdots, a_{T-1}, s_T)}{d_b(a_t, s_{t+1}, \cdots, a_{T-1}, s_T)} = \prod_{i=t}^{T-1} \frac{\pi(a_i|s_i)}{b(a_i|s_i)}.$$

观察上述IS Ratio的形式，可以发现有关环境模型的部分被约掉了，这个ratio只和两个policy有关。
有了IS Ratio之后，我们就可以估计状态值函数的期望了。对于一个episode，我们可以得到：

$$\begin{aligned}
&v^\pi(s) \\
&= \mathbb{E}_\pi\{G_{t:T}|s\} \\
&= \sum_{(a_t, s_{t+1}, \cdots, a_{T-1}, s_T)} d_\pi(a_t, s_{t+1}, \cdots, a_{T-1}, s_T) G_{t:T}(a_t, s_{t+1}, \cdots, a_{T-1}, s_T)|_{s_t=s} \\
&= \sum_{(a_t, s_{t+1}, \cdots, a_{T-1}, s_T)} \frac{d_\pi(*)}{d_b(*)} d_b(*) G_{t:T}(*)|_{s_t=s} \\
&= \sum_{(a_t, s_{t+1}, \cdots, a_{T-1}, s_T)} d_b(*) \rho_{t:T-1} G_{t:T}(*)|_{s_t=s} \\
&= \mathbb{E}_b\{\rho_{t:T-1} G_{t:T}|s\},
\end{aligned}$$

其中，$* = a_t, s_{t+1}, \cdots, a_{T-1}, s_T$。特别的，若T=t+1，即episode从时刻t开始，到时刻t+1终止，则有单步IS Ratio：

$$\rho_t \overset{\text{def}}{=} \rho_{t:t} = \frac{\pi(a_t|s_t)}{b(a_t|s_t)}.$$

与后文的动作值函数的单步IS Ratio相比，这里比值不是一个常数，因此是非平凡的（non-trivial）的。当两个policy相远离时，IS Ratio的方差会变得非常大，尤其是有一些发生概率很小的动作时。

3.3.2.3 动作值函数的IS Ratio

动作值函数与状态值函数在计算时的唯一区别就是初始时是知道(s_t, a_t)还是只知道s_t。因此，这里视$x = \{s_{T+1}, a_{T+1}, s_{T+2}, a_{T+2}, \cdots, s_{T'}, a_{T'}\}$为一个多元随机变量。这里先给出$d_\pi(x)$和$d_b(x)$：

$$
\begin{aligned}
d_\pi(s_{t+1},\cdots,a_{T-1},s_T) &= \Pr\{s_{t+1},\cdots,s_T|\pi,s_t\} \\
&= p(s_{t+1}|s_t,a_t)\prod_{i=t+1}^{T-1}\pi(a_i|s_i)p(s_{i+1}|s_i,a_i),
\end{aligned}
$$

$$
\begin{aligned}
d_b(s_{t+1},\cdots,a_{T-1},s_T) &= \Pr\{s_{t+1},\cdots,s_T|b,s_t\} \\
&= p(s_{t+1}|s_t,a_t)\prod_{i=t+1}^{T-1}b(a_i|s_i)p(s_{i+1}|s_i,a_i).
\end{aligned}
$$

因此，我们可以得到动作值函数的IS Ratio：

$$
\rho_{t+1:T-1} = \frac{d_\pi(s_{t+1},\cdots,a_{T-1},s_T)}{d_b(s_{t+1},\cdots,a_{T-1},s_T)} = \frac{p(s_{t+1}|s_t,a_t)}{p(s_{t+1}|s_t,a_t)}\prod_{i=t+1}^{T-1}\frac{\pi(a_i|s_i)}{b(a_i|s_i)} = \prod_{i=t+1}^{T-1}\frac{\pi(a_i|s_i)}{b(a_i|s_i)}.
$$

现在我们可以给出动作值函数的估计值：

$$
\begin{aligned}
&q^\pi(s,a) \\
&= \mathbb{E}_\pi\{G_{t:T}|s,a\} \\
&= \sum_{(s_{t+1},\cdots,a_{T-1},s_T)} d_\pi(s_{t+1},\cdots,a_{T-1},s_T)G_{t:T}(s_{t+1},\cdots,a_{T-1},s_T)|_{s_t=s,a_t=a} \\
&= \sum_{(s_{t+1},\cdots,a_{T-1},s_T)} \frac{d_\pi(\#)}{d_b(\#)}d_b(\#)G_{t:T}(\#)|_{s_t=s,a_t=a} \\
&= \sum_{(s_{t+1},\cdots,a_{T-1},s_T)} d_b(\#)\rho_{t+1:T-1}G_{t:T}(\#)|_{s_t=s,a_t=a} \\
&= \mathbb{E}_b\{\rho_{t+1:T-1}G_{t:T}|s,a\},
\end{aligned}
$$

其中，$\# = s_{t+1},\cdots,a_{T-1},s_T$。特别的，若T=t+1，即episode从时刻t开始，到时刻t+1终止，则有单步IS Ratio：

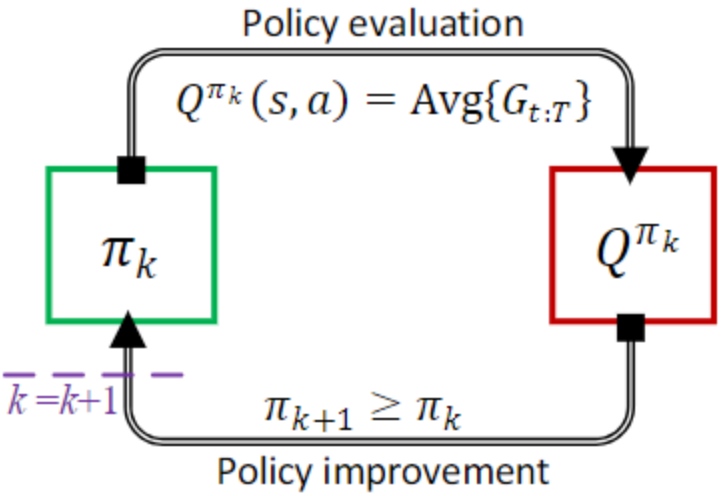
$$
\rho_{t+1:t} = \frac{d_\pi(s_{t+1})}{d_b(s_{t+1})} = \frac{\Pr\{s_{t+1}|\pi,s_t,a_t\}}{\Pr\{s_{t+1}|b,s_t,a_t\}} = \frac{p(s_{t+1}|s_t,a_t)}{p(s_{t+1}|s_t,a_t)} = 1.
$$

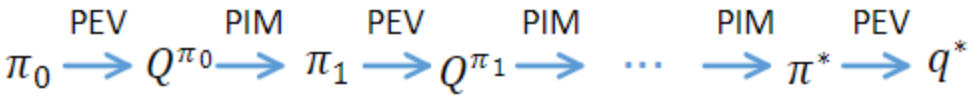
单步IS Ratio的终止时刻T=t+1，则可得上式。至于为什么此时的比值是两个转移概率p的比值而非策略概率的比值，观察求IS Ratio的过程即可。

这里的单步IS Ratio带来了一个极为有趣的性质：当执行单步的bootstrapping来估计动作值函数时，IS Ratio不显含target和behavior的信息，因此**采用从任意的behavior分布中采样得到的sample都可以用来估计动作值函数。** 这个性质是Q-learning的核心，也是后续的DRL中experience replay的基础，如DQN、DDQN、DDPG等！！！

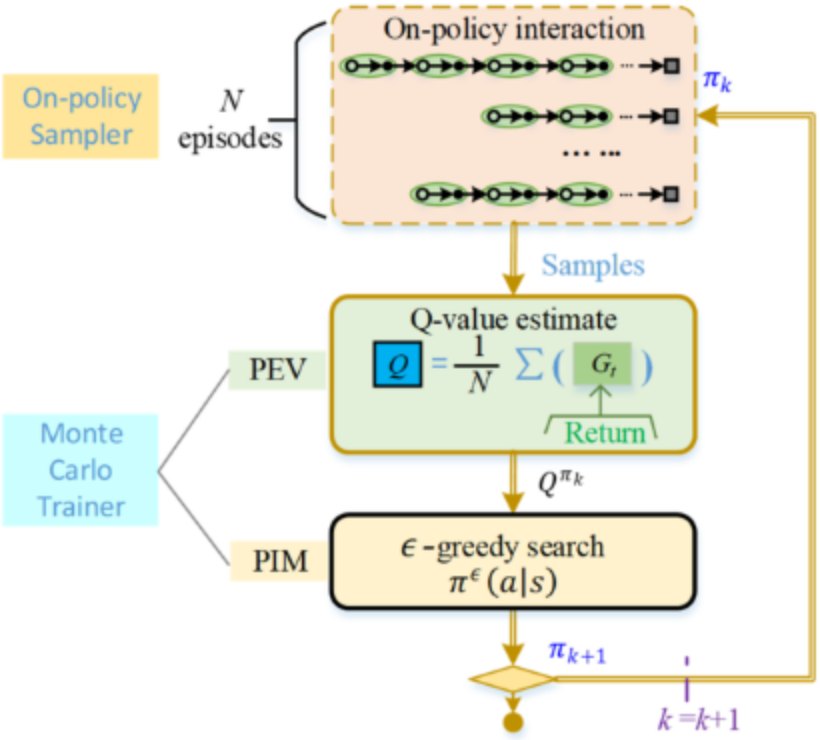
3.4 Monte Carlo学习算法

Monte Carlo方法是一种采取策略迭代（Policy Iteration）的indirect RL方法。一般来说，MC类算法估计的都是动作值函数，因为环境模型通常是未知的。学习过程通常是一个episode一个episode地学习的，迭代时策略评估（Policy Evaluation）和策略改进（Policy Improvement）是交替进行的。





3.4.1 On-Policy MC学习算法



On-Policy MC学习算法由两个部分构成：on-policy sampler以及Monte Carlo trainer。

- **On-policy sampler**：On-policy sampler是用来和环境交互的，它根据当前的策略（记得on-policy的算法里面只有一个策略）来与环境交互生成一个episode的batch。首先，它随机选取一个初始的状态-动作对(s, a)，然后生成N个episode：

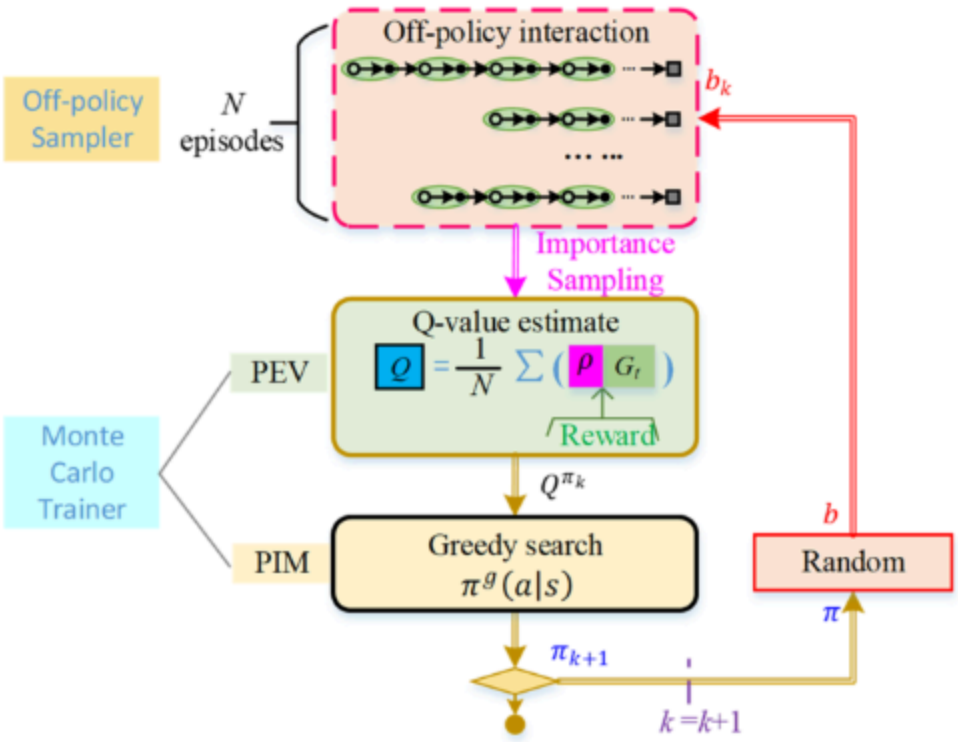
$$G_{t_1:T}^{(1)}, G_{t_2:T}^{(2)}, \cdots, G_{t_i:T}^{(i)}, \cdots G_{t_N:T}^{(N)},$$

  注意，这N个episode都是**从同一个初始状态开始**且各个episode的长度不一定一样（终止时间不一定一样）。
- **Policy Evaluation**：，每次采样得到从(s, a)开始的N个episode后，进行PEV来更新动作值函数的值：

$$Q^{\pi}(s,a) = \frac{1}{N}\sum_{i=1}^{N} G_{t_i:T}^{(i)},$$
$$\forall s_{t_i} = s, a_{t_i} = a.$$

- **Policy Improvement**：在PEV之后，动作值函数的值发生了变化，从而策略也可能发生变化，需要进行更新。可以采用greedy或者ϵ-greedy等策略来更新策略。具体选择参前。因为on-policy算法总共只有一个policy，为了在探索和利用之间达到平衡，通常使用ϵ-greedy策略。也可采取更复杂的如decay ϵ-greedy策略。

3.4.2 Off-Policy MC学习算法



Off-Policy MC学习算法由两个部分构成：off-policy sampler以及Monte Carlo trainer。

- **Off-policy sampler**：使用behavior policy来与环境交互。这里的策略**常常采用ϵ-greedy策略**，以便对环境进行较好的探索。交互后可得到一个batch：

$$G_{t_1:T}^{(1)}, G_{t_2:T}^{(2)}, \cdots, G_{t_i:T}^{(i)}, \cdots G_{t_N:T}^{(N)},$$

Off-policy sampler的一个重要性质是**它可以从不同的behavior policy中采样得到的样本来估计动作值函数**，可以重复利用历史的样本来提高data efficiency。只需要为每个样本使用不同的IS Ratio：

$$\rho_{t_1+1:T-1}^{(1)}, \rho_{t_2+1:T-1}^{(2)}, \cdots, \rho_{t_i+1:T-1}^{(i)}, \cdots \rho_{t_N+1:T-1}^{(N)}.$$

之后按照这些IS Ratio加权可以得到动作值函数的估计值。

- **Policy Evaluation**：得到值函数的估计值有两种方法：
  - **ordinary batch average（除以batch size）**：

$$Q^{\pi}(s,a) = \frac{1}{N}\sum_{i=1}^{N}\rho_{t_i+1:T-1}^{(i)}G_{t_i:T}^{(i)},$$
$$\forall s_{t_i} = s, a_{t_i} = a.$$

  - **weighted batch average（除以ISRatio的和）**：

$$Q^{\pi}(s,a) = \frac{1}{\sum_{i=1}^{N}\rho_{t_i+1:T-1}^{(i)}}\sum_{i=1}^{N}\rho_{t_i+1:T-1}^{(i)}G_{t_i;T}^{(i)},$$
$$\forall s_{t_i} = s, a_{t_i} = a.$$

这两种方法的区别在于ordinary batch average是无偏的，而weighted batch average是有偏的（尽管bias可以渐进的收敛到0）。另一点区别在于ordinary batch average的方差较大，而weighted batch average的方差较小。这是因为IS Ratio可能有较大的error，而在weighted batch average中做了归一化。
另外，当episode足够长，weighted batch average的公式可以简化为：

$$Q^{\pi}(s,a) = \frac{1}{N}\sum_{i=1}^{N}G_{t_i:T}^{(i)},$$
$$\forall s_{t_i} = s, a_{t_i} = a.$$

- **Policy Improvement**：同前。

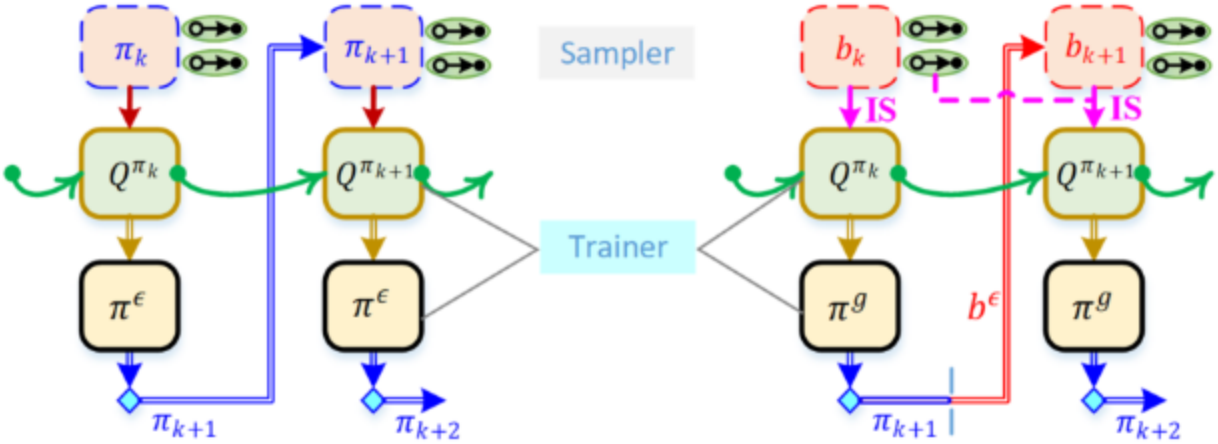
3.4.3 Incremental Monte Carlo学习算法

不知道大家有没有发现，上面每个batch更新(s, a)对应的动作值函数$Q^{\pi}(s,a)$时，都是直接把这个batch的return加起来然后除以batch size。这样做有一个问题，就是忽视了之前的估计值。换句话说，之前的学习等于白学了，这自然会减缓收敛速度。为了解决这个问题，就引入了Incremental Monte

Carlo学习算法。这种算法的核心思想是**每次更新动作值函数的估计值时，都要考虑之前的估计值**。这样可以加速收敛速度。具体来说，Incremental Monte Carlo学习算法的更新公式为：

$$Q^{\pi_{k+1}}(s,a) = (1-\lambda)Q^{\pi_k}(s,a) + \lambda \cdot \mathrm{Avg}\{G|s,a;\pi_{k+1}\}$$

这里的$\lambda \in (0,1]$。调节λ的大小可以控制之前学到的和本次新学的对于整体值函数更新的贡献。结合了incremental的Monte Carlo学习算法如下图所示：



下面来看看一个具体的Monte Carlo学习算法的伪代码的例子。这个伪代码中具体使用了之前讲过的first-visit MC、incremental MC以及on-policy MC。

Hyperparameters: Discount factor γ, Exploration rate ϵ, Incremental rate λ, Episode number per iteration M

Initialization: $Q(s,a) \leftarrow 0, \pi^\epsilon(a|s) \leftarrow 1/|\mathcal{A}|$

Repeat (indexed with k)

Initialize visit counter $\mathcal{N}(s,a) \leftarrow 0$

For j in 1,2, … , M

$s_0 \sim d_{\text{init}}(s)$

//Observe one trajectory

$s_0, a_0, r_0, s_1, \dots, s_{T-1}, a_{T-1}, r_{T-1}, s_T$

//Calculate returns in one trajectory

For t in 0,1,2, … , $T-1$

If (s_t, a_t) is first visited in the trajectory

$$G_j(s_t,a_t) \leftarrow \sum_{i=0}^{T-t} \gamma^i r_{t+i}$$

$$\mathcal{N}(s_t,a_t) \leftarrow \mathcal{N}(s_t,a_t) + 1$$

End

End

End

Sweep (s,a) in $\mathcal{S}\times\mathcal{A}$

//Calculate average return for each visited state-action pair

$$\bar{G}(s,a)\leftarrow\frac{1}{\mathcal{N}(s,a)}\sum_{j=1}^{M}G_j(s,a)\,,\forall(s,a)\in\{\mathcal{N}(s,a)>0\}$$

//Incremental MC estimation

$$Q(s,a)\leftarrow(1-\lambda)Q(s,a)+\lambda\bar{G}(s,a),\forall(s,a)\in\{\mathcal{N}(s,a)>0\}$$

End

//Calculate greedy actions and update policy

$$a^*\leftarrow\arg\max_a Q(s,a)$$

$$\pi^{\epsilon}(a|s)=\begin{cases}1-\epsilon+\epsilon/|\mathcal{A}|, & \text{if } a=a^*\\ \epsilon/|\mathcal{A}|, & \text{if } a\neq a^*\end{cases}$$

End

注意，上述算法里的M就是之前讲解中的N（因为这里的N被用作计数器的意义了）。在原书的第60到65页有一个对于上述伪代码的implementation。里面详细讲了算法的实现以及三个关键参数Ep/PEV（每轮迭代的episode数目，即上文的N或M）、λ（incremental rate）和ϵ（ϵ-greedy策略的参数）的选择对于算法效果的影响。可以参看原书的对应部分。那里写的很明白易懂。

@[toc]

书籍链接：Reinforcement Learning for Sequential Decision and Optimal Control

本博客介绍了TD（Temporal Difference）Learning。TD Learning是一种无模型、Indirect的强化学习方法。TD Learning最大的特点就是bootstrapping，即利用过去的value estimates来更新当前的值函数。TD Learning相较于MC，最大的好处就是不用等到一个episode结束才更新。相反，TD Learning可以通过不完整的episode来更新value function，或者在连续任务中更新（连续任务中没有episode的概念），而不必等待最终的结果（达到终止状态）。TD的缺点在于当与function approximation和off-policy结合时，可能会出现不稳定（unstable）和不收敛（divergent）的情况。另外，由于与生物学习的相似性，TD Learning也被神经科学和心理学领域广泛研究。

4.1 TD Policy Evaluation

这里先讲一讲最简单的TD，one-step：TD(0)的值函数估计。

4.1.1 状态值函数的TD(0)更新

假设我们现在有一个sample：$(s_t = s, a_t = a, s_{t+1} = s')$以及这个三元组对应的reward：$r_{t+1}$。我们可以用这个sample来更新状态值函数$V(s)$：

$$V(s) \leftarrow (1-\alpha)V(s) + \alpha \cdot G_t = V(s) + \alpha\big(G_t - V(s)\big)$$

这里的G_t是从t开始的return。回想我们之前讲过的TD的优点，即不需要完整的episode（即不需要达到终止条件）来更新值函数的估计。因此我们需要一个估计值来代替G_t。回想我们在第二单元的博客那里讲过的关于状态值函数的self-consistency的公式：

$$v^{\pi}(s) = \mathbb{E}_{\pi}\{r + \gamma v^{\pi}(s')|s\}$$

我们可以考虑使用$r + \gamma v^{\pi}(s')$来替代G_t。则TD(0)的更新公式可以写成如下形式：

$$V(s) \leftarrow V(s) + \alpha\big(r + \gamma V(s') - V(s)\big).$$

这个式子也是我们称此算法为one-step TD的原因，因为我们只用了一个step的信息来更新值函数的估计，即one-step look ahead。可以看出，α后面乘的括号里的那一项可以看成是**估计的V(s)与V(s)的实际值之间的差距**。$r + \gamma V(s')$是对于V(s)的近似，而V(s)为实际值。当我们有足够的sample来更新值函数的估计时，该算法即可达到收敛。而实际上，当算法的指标达到一定的阈值后，即可停机。

4.1.2 动作值函数的TD(0)更新

仿照状态值函数的TD(0)更新，我们可以得到动作值函数的TD(0)更新。假设我们现在有一个sample：$(s_t = s, a_t = a, s_{t+1} = s', a_{t+1} = a')$以及这个四元组对应的reward：$r_{t+1}$。我们可以用这个sample来更新动作值函数$Q(s, a)$：

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big(r + \gamma Q(s',a') - Q(s,a)\big).$$

4.1.3 On Policy vs. Off Policy

与Mc那里一样，这里我们也有on-policy和off-policy的TD算法。对于on-policy的TD算法，直接使用上述公式更新即可。而对于off-policy的TD算法，需要使用Importance Sampling技术来更新。

回顾我们在上一篇博客里讲过的单步的状态值函数的IS Ratio：

$$\rho_t \overset{\text{def}}{=} \rho_{t:t} = \frac{\pi(a_t|s_t)}{b(a_t|s_t)}.$$

接着，为了把从behavior policy采样的sample转换成从target policy采样的sample，以致于使用在4.1.1节里的TD(0)更新公式，需要关注公式中三个量如何转化，即r，$V(s')$和$V(s)$。下面给出如下转化公式：

$$\begin{aligned}
\mathbb{E}_\pi\{r(s,a,s')\} &= \mathbb{E}_b\{\rho_{t:t} r(s,a,s')\}, \\
\mathbb{E}_\pi\{V(s')\} &= \mathbb{E}_b\{\rho_{t:t} V(s')\}, \\
\mathbb{E}_\pi\{V(s)\} &= \mathbb{E}_b\{V(s)\}.
\end{aligned}$$

其中，前两式的证明，即上一篇博客里讲过的IS的期望相等的性质：

$$\mathbb{E}_\pi\{h(x)\} = \mathbb{E}_b\{\rho(x)h(x)\}.$$

而最后一个式子是因为当前状态s为一个已知量，因此可以视作一个常数，在不同策略下求期望均相等。因此，我们可以得到off-policy的TD(0)更新公式：

$$V(s) \leftarrow V(s) + \alpha\left(\rho_{t:t}(r + \gamma V(s')) - 1 \cdot V(s)\right), \forall (a, s') \sim b.$$

但是根据这个式子来做off-policy的更新会在实际中引入较大的方差。一种解决方法如下：

$$V(s) \leftarrow V(s) + \alpha\rho_{t:t}\big(r + \gamma V(s') - V(s)\big), \forall (a, s') \sim b.$$

这个形式的更新公式可以很大程度的提高求解的质量。其中，我们称$\delta_t \overset{\text{def}}{=} r + \gamma V(s') - V(s)$为**TD Error**（更准确的来说，这里的是one-step error）。可能看到这里会让人优点疑惑，按之前的分析明明V(s)的期望是不用乘IS Ratio做转换的，这里却把它的前面乘上IS Ratio放进括号里面，难道仅仅是为了

计算方便而没有理论依据吗？其实不然，下面就给出一个证明来说明在behavior policy下，V(s)乘不乘IS Ratio是一样的。证明如下：

$$\begin{aligned}\mathbb{E}_{a\sim b}\{\rho_{t:t}V(s)\} &= \mathbb{E}_{a\sim b}\left\{\frac{\pi(a|s)}{b(a|s)}\right\}\cdot\mathbb{E}_{a\sim b}\{V(s)\}\\ &= \sum_a\left\{b(a|s)\frac{\pi(a|s)}{b(a|s)}\right\}\cdot\mathbb{E}_{a\sim b}\{V(s)\}\\ &= \sum_a\pi(a|s)\cdot\mathbb{E}_{a\sim b}\{V(s)\}\\ &= 1\cdot\mathbb{E}_{a\sim b}\{V(s)\}.\end{aligned}$$

这里第一个等号是因为期望的性质：

$$\begin{gathered}设X,Y是相互独立的两个随机变量,则有\\ E(XY)=E(X)E(Y);\end{gathered}$$

上面因为s是已知量，V(s)是常数，自然与$\rho_{t:t}$独立，所以可以这样写。第二个等号是根据期望的定义式展开。第四个等号是因为$\sum_a\pi(a|s)=1$。至于为什么采取上面的处理后就能减小方差，直观上的解释是乘上IS Ratio后，$r+\gamma V(s')$和$V(s)$可以耦合在一起作为一个整体，从而减小了误差的累积。另外就是在V(s')前乘上IS Ratio也可减少对于误差的放大。

4.2 TD Policy Improvement

因为在上一篇博客里面已经证明了greedy和ϵ-greedy策略都满足Policy Improvement Theorem，因此在PEV更新了值函数之后，可以使用greedy或者ϵ-greedy策略来更新策略。也可采取其它策略。这里不再赘述。

4.3 一些典型的TD算法

算法名称	SARSA	Q-Learning	Expected SARSA
On / Off Policy	On-Policy	Off-Policy	Off-Policy
值函数	动作值函数$Q(s,a)$	动作值函数$Q(s,a)$	动作值函数$Q(s,a)$
策略	greedy、ϵ-greedy 等	**只能采用greedy策略**	greedy、ϵ-greedy 等
Sample形式	五元组(s,a,r,s',a')	四元组(s,a,r,s')	四元组(s,a,r,s')

4.3.1 SARSA

Hyperparameters: Discount factor γ, exploration rate ϵ, learning rate α, pairs per PEV N

Initialization: $Q(s,a) \leftarrow 0, \pi^{\epsilon}(a|s) \leftarrow 1/|\mathcal{A}|$

Repeat (indexed with k)

// Environment initialization

$s_0 \sim d_{\text{init}}(s)$

$a_0 \sim \pi^{\epsilon}(a|s_0)$

$s \leftarrow s_0, a \leftarrow a_0$

Repeat until each episode terminates

Rollout one step with action a, observe next state s' and reward r

Sample action a' under next state s': $a' \sim \pi^{\epsilon}(a|s')$

//Do action-value update

$Q(s,a) \leftarrow Q(s,a) + \alpha\big(r + \gamma Q(s',a') - Q(s,a)\big)$ ⟸ PEV

PIM ⟹ If N steps since last policy update, update $\pi^{\epsilon}(a|s)$ with $Q(s,a)$

$s \leftarrow s', a \leftarrow a'$

End

End

SARSA是一种**on-policy**的TD算法。Sarsa的更新公式如下：

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big(r + \gamma Q(s',a') - Q(s,a)\big).$$

每次拿到一个sample（由五元组(s,a,r,s',a')组成），就可以使用上述更新公式来更新一次动作值函数的值。

4.3.1.1 Policy Evaluation

按照上述公式，在每次PEV中使用N个sample来更新。注意，每轮PEV更新开始前对应的Q函数的初始值应该使用上一轮最后更新结束的Q函数的值。这样可以保证每次PEV的更新都是基于上一轮的Q函数的值。这样的更新方式可以保证Q函数的收敛性。

4.3.1.2 Policy Improvement

当每次通过PEV得到了一个新的Q函数后，我们可以使用greedy或者ϵ-greedy策略来更新策略。这里以ϵ-greedy策略为例：

$$\pi'(a|s) \leftarrow \begin{cases} 1-\epsilon+\epsilon/|\mathcal{A}|, & \text{if } a = a^* \\ \epsilon/|\mathcal{A}|, & \text{if } a \neq a^* \end{cases},$$
$$\text{subject to}$$
$$a^* = \arg\max_a Q^\pi(s,a), \forall s \in \mathcal{S}.$$

4.3.2 Q-Learning

Hyperparameters: Discount factor γ, Learning rate α
Initialization: $Q(s,a) \leftarrow 0$, Behavior policy $b(a|s) \leftarrow 1/|\mathcal{A}|$
// Environment initialization
$s_0 \sim d_{\text{init}}(s)$
$s \leftarrow s_0$
Repeat until episode terminates (indexed with k)
//Sample action a under state s from behavior policy

$a \sim b(a|s)$
Rollout one step with action a, observing s' and r
//Do action-value update 这里不同于SARSA，不需要获取a'

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(r + \gamma \max_a Q(s',a) - Q(s,a)\right)$$

$s \leftarrow s'$
End
Calculate greedy policy $\pi(a|s)$ from $Q(s,a)$ 最终的策略

Q-Learning是一种**off-policy**的TD算法。它是一种很有特点的one-step TD算法。它的最简单的PEV更新公式如下：

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left(r_t + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\right).$$

结合这个公式，具体来说，它有如下的特点：

- **PEV与PIM合二为一**：看其它的RL书籍资料，往往会产生这样的疑惑，就是为什么Q-Learning没有显式的PIM过程？Q-Learning的策略到底是什么？其实答案就蕴含在上述的更新公式中。Q-Learning的策略为greedy策略，因此不涉及复杂的策略更新，因此也就没必要显式的写出来，直接包含在PEV中那里的max操作中了。
- **可任选behavior策略**：Q-Learning有别于其它off-policy的算法的一个最显著特点就是它可以任选behavior策略。这是因为Q-Learning的更新公式中不显含IS Ratio。不显含IS Ratio的好处十分巨大：
  - **可采取任意策略**：因为不显含IS Ratio，因此可以采取任意的behavior策略，而不用担心IS Ratio的问题。甚至可以采取未知分布的策略，而不必知道关于behavior policy的任何结构和参数信息，因为不用在计算中包含有关behavior策略的分布的信息。比如可以采用一个完全随机的uniform的behavior策略来更好地探索环境。
  - **可以使用不同的behavior policy下收集的样本**：因为不显含IS Ratio，因此可以使用不同的behavior policy下收集的样本来更新Q函数。这样可以更好的利用已有的样本，从而提高算法的效率。这也为exprience replay提供了理论支持，从而为后续的DQN等算法提供了理论基础。

下面给出为什么Q-Learning可以任选策略的证明。首先，给出有关r和Q(s',a')在target和behavior策略下的期望相等的性质：

$$\mathbb{E}_\pi\{r(s,a,s')\} = \mathbb{E}_b\{\rho_{t+1:t} r(s,a,s')\},$$
$$\rho_{t+1:t} = \frac{\Pr\{s'|\pi,s,a\}}{\Pr\{s'|b,s,a\}} = \frac{p(s'|s,a)}{p(s'|s,a)} = 1,$$

$$\mathbb{E}_\pi\{Q(s',a')\} = \mathbb{E}_b\{\rho_{t+1:t+1} Q(s',a')\},$$
$$\rho_{t+1:t+1} = \frac{p(s'|s,a)\pi(a'|s')}{p(s'|s,a)b(a'|s')} = \frac{\pi(a'|s')}{b(a'|s')}.$$

为什么上面两个变换中使用的IS Ratio不同呢？其实不用死记硬背，只需先判断清楚谁是随机变量即可。拿r的IS Ratio那里来说，r是由s,a,s'决定的，而s,a是已知值，可以看成常量，因此随机变量只有一个s'。而IS Ratio就是两个策略下s'的分布的比值，对于离散型随机变量，就是两个策略下得到s'的概率的比值。因此为1。而对于Q(s',a')的IS Ratio，有两个随机变量s'和a'，因此IS Ratio就是两个策略下s'和a'的联合分布的比值。因此为$\frac{\pi(a'|s')}{b(a'|s')}$。为了证明Q-Learning可以任选behavior策略（不显含IS Ratio），就是要证明下式：

$$\mathbb{E}_b\{\rho_{t+1:t} r + \gamma \rho_{t+1:t+1} Q(s',a')\} = \mathbb{E}_b\Big\{r + \gamma \max_a Q(s',a)\Big\}.$$

$$
\begin{aligned}
\mathbb{E}_b\{\rho_{t+1:t} r+\gamma \rho_{t+1:t+1} Q(s', \alpha')\} &= \mathbb{E}_{s' \sim \mathcal{P}}\left\{1 \cdot r+\gamma \mathbb{E}_{a' \sim b}\left\{\frac{\pi(a'|s')}{b(a'|s')} Q(s', \alpha')\right\}\right\}. \\
&= \mathbb{E}_{s' \sim \mathcal{P}}\left\{r+\gamma \sum_{a'} b(a'|s') \frac{\pi(a'|s')}{b(a'|s')} Q(s', a')\right\} \\
&= \mathbb{E}_{s' \sim \mathcal{P}}\left\{r+\gamma \sum_{a'} \pi(a'|s') Q(s', a')\right\} \\
&= \mathbb{E}_{s' \sim \mathcal{P}}\left\{r+\gamma \mathbb{E}_{a' \sim \pi}\{Q(s', a')\}\right\} \\
&= \mathbb{E}_b\left\{r+\gamma \max_a Q(s', a)\right\}.
\end{aligned}
$$

首先从原始式到右边第一式是因为左边在b下求期望实际上是对于(s',a')的联合分布求期望，因此可以分解求内外两层期望（回想期望的定义式，这里又是离散型，因此实际上就是拆成两层求和号）。第一式到第二式是根据期望的定义式展开。第三式到第四式是把求和号再换回期望。第四式到第五式是根据因为target policy π是greedy策略，实际上除了值函数最大的那个动作之外其它的概率都是0，那个动作的概率是1.因此期望可化简为max操作。证毕。

4.3.3 Expected SARSA

Expected SARSA结合了SARSA和Q-Learning的特点，它的PEV更新公式如下：

$$
Q(s_t, a_t) \leftarrow Q(s_t, a_t)+\alpha\big(r_t+\gamma \mathbb{E}_{a_{t+1} \sim \pi}\{Q(s_{t+1}, a_{t+1})\}-Q(s_t, a_t)\big).
$$

该式也可把期望部分按定义展开，得到如下式子：

$$
Q(s_t, a_t) \leftarrow Q(s_t, a_t)+\alpha\left(r_t+\gamma \sum_{a \in \mathcal{A}} \pi(a|s_{t+1}) Q(s_{t+1}, a)-Q(s_t, a_t)\right).
$$

与SARSA相比，Expected SARSA不用获取下一个动作s_{t+1},而是直接对下一个状态s_{t+1}的所有动作的值函数的期望进行更新,可以降低SARSA由于bad sample带来的高方差，提升训练的稳定度；与Q-Learning相比，Expected SARSA算是一种泛化，Expected SARSA不用max操作，而是代之以期望。而当策略π是greedy策略时，Expected SARSA就是Q-Learning。与其它TD Learning算法相比，Expected SARSA通常效果更好，这是因为取期望的操作带来了更稳定的更新。

另外，Expected SARSA和Q-Learning一样，**都不显含IS Ratio**。这可以解释为它们都不包含下一个动作a_{t+1}。Q-Learning不包含下一个动作是因为它直接取max操作，而Expected SARSA不包含下一个动作是因为它取期望操作。**因此，Expected SARSA也可以任选behavior策略，也不用显式的使用IS Ratio进行转换。**

4.3.4 n-step TD Learning

回想一下TD(0)和（incremental）MC的PEV更新公式：

$$V(s) \leftarrow (1-\alpha)V(s) + \alpha \cdot (r + \gamma v^{\pi}(s')) = V(s) + \alpha\big(r + \gamma v^{\pi}(s') - V(s)\big) \quad (\mathrm{TD}(0))$$

$$V(s) \leftarrow (1-\alpha)V(s) + \alpha \cdot Avg(G_t) = V(s) + \alpha\big(Avg(G_t) - V(s)\big) \quad (\mathrm{MC})$$

而MC里的G_t可以写成：

$$G_t = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{T-t-1} r_T + \ldots$$

由此可以看出，MC和TD(0)的区别就在于α后面乘的括号里面有关r的项有几项。TD(0)是一项，MC是无穷项。那么，有没有位于二者之间折中的方案呢？当然有，这就是n-step TD（或者称为TD(n)）。首先定义n步的return：

$$G_{t:t+n-1} \stackrel{\text{def}}{=} \sum_{i=0}^{n-1} \gamma^i r_{t+i}.$$

则n-step TD的更新公式如下：

$$\begin{aligned} V^{\pi}(s_t) &\leftarrow V^{\pi}(s_t) + \alpha\big(v^{\pi}(s) - V^{\pi}(s_t)\big) \\ &\cong V^{\pi}(s_t) + \alpha\big(G_{t:t+n-1} + \gamma^n V^{\pi}(s_{t+n}) - V^{\pi}(s_t)\big), \end{aligned}$$

据此还可以定义n-step TD error：

$$\delta_t^{\mathrm{TD}(n)} \stackrel{\text{def}}{=} G_{t:t+n-1} + \gamma^n V^{\pi}(s_{t+n}) - V^{\pi}(s_t).$$

最后介绍一个关于n-step TD Learning的重要性质：n-step return的期望与s的状态值函数的差距在最坏情况下也小于等于s的状态值函数的·1估计值与实际值的差距。即：

$$\max_s \left|\mathbb{E}_{\pi}\{G_{t:t+n-1} + \gamma^n V^{\pi}(s_{t+n}) | s_t = s\} - v^{\pi}(s)\right| \leq \gamma^n \max_s |V^{\pi}(s) - v^{\pi}(s)|.$$

这个性质也被称为discounted return的error reduction property。可以看出，这个性质说明了当增加n时，n-step return的期望与s的状态值函数的差距会逐渐减小。随着n的增加，n-step return的期望会逐渐收敛到s的状态值函数的真实值。

4.3.5 TD-λ Learning

这里关于TD-λ Learning的介绍仅仅是一个简单地概述，其本身的理论涉及到eligibility trace等理论，比较复杂，这里就不做展开。

TD-λ等更新公式如下：

$$V(s) \leftarrow (1-\alpha)V(s) + \alpha \cdot G_t^{\lambda} = V(s) + \alpha\big(G_t^{\lambda} - V(s)\big)$$

G_t^{λ}的定义式如下：

$$G_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\big(G_{t:t+n-1} + \gamma^n V^{\pi}(s_{t+n})\big),$$

上述公式可以看作对于若干个（无穷多个）TD(n)的加权平均。但是仔细思考就可以发出现，按照上面的定义式计算G_t^{λ}来计算值函数的估计是不现实的。实际上，可以通过forward view和backward view来审视这个计算过程。而使用backward view的能有效的计算。具体计算要结合Sutton等人提出的eligibility trace等理论。这里不做展开。

4.3.6 Recursive Value Initialization

类似上一篇博客里讲过的Incremental MC，这里的Recursive Value Initialization就是每轮在PEV更新值函数时，**使用上一轮的值函数的值来初始化这一轮的值函数的值**。这样可以保证每轮PEV的更新都是基于上一轮的值函数的值。这样的更新方式可以加速收敛，提高算法的效率。

4.4 实验结果

有关SARSA、Q-Learning和MC在一个简单地场景下的实验结果参考原书82到87页。

@[toc]

书籍链接：Reinforcement Learning for Sequential Decision and Optimal Control

本篇博客主要介绍Dynamic Programming（DP），即动态规划，一种Model-Based的Indirect RL方法。DP与动态环境中的随机最优控制问题有紧密的联系。DP通常通过求解Bellman方程（离散时间）或Hamilton-Jacobi-Bellman方程（连续时间）来得到最优策略。对于大部分工程任务来说，以上两个方程是最优解的充要条件。DP的优点在于它的强大的适应性。如线性/非线性系统、离散/连续系统、确定性/随机性系统、可微分/不可微分系统等。但是，DP在实际应用中面临着维度灾难和缺少泛化能力的问题。但是，DP仍然在理论分析最优解的存在以及收敛速度等方面有着重要的作用。

5.1 随机序列决策问题（Stochastic Sequential Decision Problem）

DP的设计包含两个重要的部分： **对于转移的建模**、 **对于奖励的建模** 。因为实际任务的复杂性，对于上述两个问题的回答可能呈现出大相径庭的形式。因此，我们在实际中看到的对于DP的建模可能完全看不出是同一种算法，这对于初学者理解算法是一个很大的困难。因此，书中首先试图给出一种理解DP的统一的视角。

5.1.1 随机环境的建模

有两种方式可以描述随机环境： **状态空间模型**（state space model）和**概率模型**（probabilistic model）。首先来看看状态空间模型。

5.1.1.1 状态空间模型

$$s_{t+1} = f(s_t, a_t, \xi_t)$$

这里f(.)表示环境模型，而ξ_t表示一个分布已知的随机分布。这里，我们需要做两条假设来保证环境的Markov性质：

- ξ_t与t时刻之前的噪声$\xi_{t-1}, \xi_{t-2}, ...$及初始状态$s_0$独立。
- 每个动作a_t的选择是任意的且与历史的状态和随机噪声无关。

有了上述假设后就可证上述状态空间模型具有Markov性质，即下一个状态s_{t+1}只与当前状态s_t、当前动作a_t和随机噪声ξ_t有关。

5.1.1.2 概率模型

上述具有Markov性质的状态空间模型可以转化为概率模型。不失一般性地，可以假设噪声ξ为加性噪声。即：

$$s_{t+1} = f(s_t, a_t) + \xi_t$$

再假设噪声ξ_t的概率密度函数为$p_\xi(\xi_t; \theta)$，其中θ为参数。那么，概率模型可以写为：

$$\mathcal{P}_{ss'}^a \triangleq p_\xi(\xi_t; \theta) = p_\xi(s_{t+1} - f(s_t, a_t); \theta)$$

这里虽然没有直接给出从当前状态s_t采取动作a_t到下一个状态s_{t+1}的概率，但实际上已经使用了随机噪声的概率密度来表示了这个概率。

为了更形象地说明上述两种模型的关系，我们可以通过一个具体的例子来说明。考虑一个线性的状态空间模型以及加性高斯噪声：

$$s_{t+1} = As_t + Ba_t + \xi_t$$

这里假设状态s为n维向量，动作a为m维向量，$A \in \mathbb{R}^{n \times n}, B \in \mathbb{R}^{n \times m}$。且原系统能控（回想一下现代控制理论里面的能控性判据，即$rank([B, AB, A^2B, ..., A^{n-1}B]) = n$）。接下来，写出噪声的概率密度函数：

$$p_\xi(\xi_t) = \frac{1}{(2\pi)^{n/2}|\mathcal{K}|^{1/2}} \exp\left(-\frac{1}{2}(\xi_t - \mu)^{\mathrm{T}}\mathcal{K}^{-1}(\xi_t - \mu)\right)$$

这是一个n维的高斯噪声，其中μ为均值，$\mathcal{K}$为协方差矩阵。那么，接下来把$\xi_t = s_{t+1} - As_t - Ba_t$代入上式，就可以得到概率模型：

$$p(s_{t+1}|s_t, a_t) = \frac{1}{(2\pi)^{n/2}|\mathcal{K}|^{1/2}} \exp\left(-\frac{1}{2}z^{\mathrm{T}}\mathcal{K}^{-1}z\right),$$
$$z = s_{t+1} - (As_t + Ba_t + \mu),$$

这里$z \sim \mathcal{N}(0, \mathcal{K})$。那么，因为$s_{t+1} = As_t + Ba_t + \mu + z$，所以$s_{t+1}$服从高斯分布$s_{t+1} \sim \mathcal{N}\big((As_t + Ba_t + \mu), \mathcal{K}\big)$.

总结来说，概率模型和状态空间模型都可以描述随机的环境。它们之间的区别如下：

- **如何表示随机性**：状态空间模型的随机性是显式表示的，而概率模型的随机性隐含于概率密度之中。
- **是否具有Markov性质**：单步的概率模型天然具有Markov性质，而状态空间模型需要通过一些假设来保证Markov性质。

5.1.2 代价函数的建模

序列决策问题的数学建模可以非常多样化。其中一个重要的原因在于对于目标函数/代价函数有着非常不同的建模方式。对于无限时间的序列决策问题，需要精心设计代价函数来保证该函数只能取有限值。两种流行的代价函数是
折扣代价函数（discounted cost）和平均代价函数（average cost）。

5.1.2.1 平均代价函数

回顾我们在第二章中介绍的average return：

$$G_t^{avg} = \lim_{T\to\infty} \frac{1}{T} \sum_{i=0}^{T-1} r_{t+i}$$

我们可以据此定义average cost函数：

$$G_{\mathrm{avg}}(\pi) = \lim_{T\to\infty} \frac{1}{T} \mathbb{E}\left\{\sum_{i=0}^{T-1} r_{t+i}\right\}$$

可以发现，两式的区别在于第二式在第一式的基础上套了一个期望，该期望表示在策略π下求期望。上式中并没有写出来。通常，我们使用另一种更常见的average cost函数作为优化目标：

$$J_{\mathrm{avg}}(\pi) = \sum d_\pi(s) v_\gamma^\pi(s),$$

$v_\gamma^\pi(s)$是带衰减因子γ的状态值函数（即状态值函数的标准定义：$v^\pi(s) \overset{\text{def}}{=} \mathbb{E}_\pi\left\{G_t \,|s\right\} = \mathbb{E}_\pi\left\{\sum_{i=0}^{+\infty} \gamma^i r_{t+i} \,|s_t = s\right\}$，详见第二单元的博客）。这里状态值函数被使用$d_\pi(s)$进行加权，因为**我们这里的代价函数并不是针对某一个特定的状态来说的，而是针对所有的状态来说的，因此自然要根据哥哥状态出现的概率进行加权**。这里的$d_\pi(s)$表示在策略π下状态s的分布，也被称为weighting function。但是，这里的$J_{\mathrm{avg}}(\pi)$在优化时因为权重$d_\pi(s)$和状态值函数$v_\gamma^\pi(s)$都与当前策略π有关，所以优化起来非常困难。回顾第二单元讲过的RL问题的标准Formulation：

$$\max_\pi / \min_\pi J(\pi) = \mathbb{E}_{s\sim d_{\mathrm{init}}(s)}\{v^\pi(s)\}$$

标准的RL问题的优化目标函数只与初始状态的分布有关，而不是时刻和当前状态相关联。那么，为什么我们这里要使用如此的定义方式呢？这是因为如此定义的$J_{\mathrm{avg}}(\pi)$与上述$G_{\mathrm{avg}}(\pi)$有着紧密的联系，而后者有一个很好的性质，就是与策略无关，对于任何策略的形式都是一样的。下面给出这个定理及其证明。

$$\text{定理1：}\ J_{\mathrm{avg}}(\pi) = \frac{1}{1-\gamma} G_{\mathrm{avg}}(\pi)$$

定理的证明如下：

$$
\begin{aligned}
J_{\text{avg}}(\pi) &= \sum d_\pi(s) v_\gamma^\pi(s) \\
&= \sum d_\pi(s) \sum_{a\in\mathcal{A}} \pi(a|s) \left\{ \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a \left(r_{ss'}^a + \gamma v_\gamma^\pi(s') \right) \right\} \\
&= G_{\text{avg}}(\pi) + \gamma \sum d_\pi(s) \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\gamma^\pi(s') \\
&= G_{\text{avg}}(\pi) + \gamma \sum_{s'\in\mathcal{S}} d_\pi(s') v_\gamma^\pi(s') \\
&= G_{\text{avg}}(\pi) + \gamma J_{\text{avg}}(\pi) \\
&= G_{\text{avg}}(\pi) + \gamma G_{\text{avg}}(\pi) + \gamma^2 J_{\text{avg}}(\pi) \\
&= G_{\text{avg}}(\pi) + \gamma G_{\text{avg}}(\pi) + \gamma^2 G_{\text{avg}}(\pi) + \cdots \\
&= \frac{1}{1-\gamma} G_{\text{avg}}(\pi).
\end{aligned}
$$

其中，从右边第一式到第二式就是利用值函数内部递归关系来代入。详见本系列博客第二单元第2.1.4.3小节。从第二式怎么化简到第三式的呢？其实就是要证明：

$$
G_{\text{avg}}(\pi) = \sum d_\pi(s) \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a r_{ss'}^a
$$

那么上式右边就是对于reward的期望。而左边的$G_{\text{avg}}(\pi)$是对于一个长度趋近于无限的序列的reward的平均值再求期望。而当序列的长度趋向于无穷时，就也是对于reward的期望。得证。下面再来看看怎么从第三式到第四式。要证三到四式，就是要证明：

$$
\sum d_\pi(s) \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\gamma^\pi(s') = \sum_{s'\in\mathcal{S}} d_\pi(s') v_\gamma^\pi(s')
$$

左边的式子可通过调整求和顺序来等价变形为：

$$
\sum_{s'\in\mathcal{S}} \Big(\sum d_\pi(s) \sum_{a\in\mathcal{A}} \pi(a|s)) \mathcal{P}_{ss'}^a \Big) v_\gamma^\pi(s')
$$

则此时括号里的那一大堆就是$d_\pi(s')$，因为我们先是枚举了所有状态s得分布再通过$\mathcal{P}_{ss'}^a$得到了s'的分布。第四式到第五式就是利用$J_{\text{avg}}(\pi)$的定义。接下来就是递归的套用上述过程并在最后使用等比数列的求和公式即可。得证。

有了上述定理，我们就证明了优化$J_{\text{avg}}(\pi)$与优化$G_{\text{avg}}(\pi)$是等价的：

$$
\max_\pi J_{\text{avg}}(\pi) \Leftrightarrow \max_\pi G_{\text{avg}}(\pi)
$$

也即优化以上两种average cost函数得到的最优策略是相同的：

$$
\pi_{\text{avg}}^* = \arg\max_\pi J_{\text{avg}}(\pi) = \arg\max_\pi G_{\text{avg}}(\pi)
$$

整理一下上面的式子，可得以下结论：

$$
J_{\text{avg}}(\pi_{\text{avg}}^*) = \frac{1}{1-\gamma} G_{\text{avg}}(\pi_{\text{avg}}^*) = \sum d_{\text{avg}}^*(s) v_\gamma^{\pi_{\text{avg}}^*}(s).
$$

两种average cost的等价性也为我们在实际中根据它们各自的优化难度来选择合适的优化目标提供了依据。

5.1.2.2 折扣代价函数

折扣代价函数的定义如下：

$$
J_\gamma(\pi) = \sum d_{\text{init}}(s) v_\gamma^\pi(s)
$$

这里$d_{\text{init}}(s)$表示初始状态的分布，可以取任意一个分布，没有影响。为了后面方便与average cost函数进行比较，我们可以使用average cost下的最优策略对应的分布来作为初始分布，即令$d_{\text{init}}(s)=d^*_{\text{avg}}(s)$。那么，discounted cost函数可以具体化的写为：

$$J_\gamma(\pi)=\sum d^*_{\text{avg}}(s)v^\pi_\gamma(s)$$

定义π^*_γ为discounted cost下的最优策略：

$$\pi^*_\gamma=\arg\max_\pi J_\gamma(\pi)$$

下面来考虑一个有趣的问题：**在各自的cost函数下，代入各自的最优策略π^*_{avg}和π^*_γ，哪个cost值更优**（更大，因为我们优化的目标是max）呢？给出下述结论：

$$\text{定理2：}\ J_{\text{avg}}(\pi^*_{\text{avg}})\le J_\gamma(\pi^*_\gamma)$$

给出这个定理的证明：
首先，观察discounted cost函数的定义式，因为其对于任何的初始分布得到的最优策略都是一样的，因此就等价于把每个状态s对应的值函数$v^{\pi^*_\gamma}_\gamma(s)$都优化到最大值。即：

$$v^\pi_\gamma(s)\le v^{\pi^*_\gamma}_\gamma(s),\text{for all } s\in\mathcal{S}.$$

那么显然有：

$$v^{\pi^*_{\text{avg}}}_\gamma(s)\le v^{\pi^*_\gamma}_\gamma(s),\text{for all } s\in\mathcal{S}.$$

那么就有：

$$\begin{aligned}J_{\text{avg}}(\pi^*_{\text{avg}})&=\sum d_{\pi^*_{\text{avg}}}(s)v^{\pi^*_{\text{avg}}}_\gamma(s)\\&\le\sum d_{\pi^*_{\text{avg}}}(s)v^{\pi^*_\gamma}_\gamma(s)\\&=J_\gamma(\pi^*_\gamma).\end{aligned}$$

下面来看看对于**两种cost函数及其对应的两种最优策略，共有四种组合的情况：$G_{\text{avg}}(\pi^*_{\text{avg}})$、$G_{\text{avg}}(\pi^*_\gamma)$、$J_{\text{avg}}(\pi^*_{\text{avg}})$、$J_\gamma(\pi^*_\gamma)$。它们之间的大小关系如下**：

$$\frac{1}{1-\gamma}G_{\text{avg}}(\pi^*_\gamma)\le\frac{1}{1-\gamma}G_{\text{avg}}(\pi^*_{\text{avg}})=J_\gamma(\pi^*_{\text{avg}})\le J_\gamma(\pi^*_\gamma).$$

证明如下。首先，由π^*_{avg}和π^*_γ分别是各自cost函数下的最优策略，有：

$$\begin{aligned}G_{\text{avg}}(\pi^*_{\text{avg}})&\ge G_{\text{avg}}(\pi^*_\gamma),\\J_\gamma(\pi^*_{\text{avg}})&\le J_\gamma(\pi^*_\gamma).\end{aligned}$$

由这两式的的第一式可证得$\frac{1}{1-\gamma}G_{\text{avg}}(\pi^*_\gamma)\le\frac{1}{1-\gamma}G_{\text{avg}}(\pi^*_{\text{avg}})$。而

$$\frac{1}{1-\gamma}G_{\text{avg}}(\pi^*_{\text{avg}})=J_{\text{avg}}(\pi^*_{\text{avg}})=\sum d^*_{\text{avg}}(s)v^{\pi^*_{\text{avg}}}_\gamma(s)=J_\gamma(\pi^*_{\text{avg}})$$

最后，有这两式的第二式可得$J_\gamma(\pi^*_{\text{avg}})\le J_\gamma(\pi^*_\gamma)$。综上，得证。

在实际中，average cost更受青睐。但是，其分布和值函数的耦合问题导致不易求解。故而常常使用discounted cost函数代替。

5.1.2.3 使用Stochastic Linear Quadratic控制器来说明两种cost函数之间的关系

这部分与最优控制理论里面的LQR控制器有关。有关这部分的基础知识可以参考如下链接：LQR控制器。那么我们就follow5.1.1那里对于随机环境的线性状态空间模型的描述：

$$s_{t+1} = As_t + Ba_t + \xi_t$$

其中的ξ_t是一个服从高斯分布的随机噪声,即$\xi_t \sim \mathcal{N}(0, \sigma^2 I)$。那么，奖励信号也可写成一个关于当前状态$s_t$和动作$a_t$的二次型函数：

$$r(s_t, a_t) = s_t^{\mathrm{T}} Q s_t + a_t^{\mathrm{T}} R a_t$$

这里的Q和R分别是半负定和负定的矩阵。这里与一般的LQR里面的对于Q、R的规定可能恰好相反，这里其实就是考虑一个最大化还是最小化的问题。一般的LQR那里需要最小化cost function，而我们这里需要最大化奖励信号。

- **Average cost函数**
  结合针对于average cost函数的differential Bellman方程以及上述对于环境的状态空间建模，可得下述针对于LQR控制器的differential Bellman方程：

  $$h^*(s) + \rho^* = \max_a \left\{ s^{\mathrm{T}} Q s + a^{\mathrm{T}} R a + \mathbb{E}_\xi \{ h^*(As + Ba + \xi) \} \right\},$$

  关于$h^*(s)$和ρ^*请参看第5.3.2.2节那里的叙述。可以证明，对于上述的二次型形式的代价函数（即reward），h函数也是一个关于状态s的二次型函数：

  $$h(s) = s^{\mathrm{T}} P_{\mathrm{Avg}} s$$

  其中，P_{Avg}可由下式确定：

  $$P_{\mathrm{Avg}} = A^{\mathrm{T}} P_{\mathrm{Avg}} A - A^{\mathrm{T}} P_{\mathrm{Avg}} B \left(B^{\mathrm{T}} P_{\mathrm{Avg}} B + R \right)^{-1} B^{\mathrm{T}} P_{\mathrm{Avg}} A + Q.$$

  那么最优的average cost如下：

  $$G_{\mathrm{avg}}(\pi^*_{\mathrm{avg}}) = \sigma^2 \mathrm{tr}(P_{\mathrm{Avg}}),$$

  最佳的策略（控制率）如下：

  $$\begin{aligned} \pi^*_{\mathrm{avg}}(s) &= -K_{\mathrm{Avg}} s, \\ \text{where} \quad K_{\mathrm{Avg}} &= \left(B^{\mathrm{T}} P_{\mathrm{Avg}} B + R \right)^{-1} B^{\mathrm{T}} P_{\mathrm{Avg}} A. \end{aligned}$$

- **Discounted cost函数**
  写出此时的Bellman方程：

  $$v^*_\gamma(s) = \max_a \mathbb{E}_\xi \left\{ s^{\mathrm{T}} Q s + a^{\mathrm{T}} R a + \gamma v^*_\gamma(As + Ba + \xi) \right\},$$

  最优的值函数也可写成一个关于状态s的二次型函数：

  $$\begin{aligned} v^*_\gamma(s) &= s^{\mathrm{T}} P_\gamma s + M, \\ P_\gamma &= \gamma A^{\mathrm{T}} P_\gamma A - \gamma^2 A^{\mathrm{T}} P_\gamma B \left(\gamma B^{\mathrm{T}} P_\gamma B + R \right)^{-1} B^{\mathrm{T}} P_\gamma A + Q, \\ M &= \frac{\gamma}{1-\gamma} \sigma^2 \mathrm{tr}\left(P_\gamma \right). \end{aligned}$$

  最优的策略仍然是一个线性的反馈控制律：

  $$\begin{aligned} \pi^*_\gamma(s) &= -K_\gamma s, \\ \text{where} \quad K_\gamma &= \gamma \left(\gamma B^{\mathrm{T}} P_\gamma B + R \right)^{-1} B^{\mathrm{T}} P_\gamma A. \end{aligned}$$

还需要说明的是，Linear Quadratic控制问题的一个有意思的性质是，**最优的控制律由Ricaatti方程（Riccati equation）给出**。尽管针对于不同的cost function，Ricatti方程的解可能不同，但是彼此之间都有联系。比如，我们上述的分别使用average cost和discounted cost函数得到的最优cost函数的值有如下关系：

$$\begin{aligned}\lim_{\gamma\to 1}\frac{1}{\sum_{i=0}^{\infty}\gamma^i}v_\gamma^*(s) &= \lim_{\gamma\to 1}\frac{s^{\mathrm{T}}P_\gamma s+\frac{\gamma}{1-\gamma}\sigma^2\mathrm{tr}(P_\gamma)}{\sum_{i=0}^{\infty}\gamma^i}\\ &= \lim_{\gamma\to 1}\frac{s^{\mathrm{T}}P_\gamma s+\frac{\gamma}{1-\gamma}\sigma^2\mathrm{tr}(P_\gamma)}{\frac{1}{1-\gamma}}\\ &= \lim_{\gamma\to 1}\{(1-\gamma)s^{\mathrm{T}}P_\gamma s+\gamma\sigma^2\mathrm{tr}(P_\gamma)\}\\ &= \sigma^2\mathrm{tr}(P_\gamma)\\ &= G_{\mathrm{avg}}(\pi_{\mathrm{avg}}^*).\end{aligned}$$

这里比较有趣的是$v_\gamma^*(s)$是一个与具体状态有关的量，而$G_{\mathrm{avg}}(\pi_{\mathrm{avg}}^*)$与具体状态无关。但是在$\gamma\to 1$的前提下，两者是相等的。这也再次告诉我们，尽管在实际建模中average cost函数更受青睐，但是其不易计算。而当我们采用γ较大的discounted cost函数时，是可以得到较好的近似的。

下面我们用一个实际的例子来比较一下上述两种cost函数下的Linear Quadratic控制器。例子的各个参数如下表所示：

参数	值
状态维数n	1
A	2
B	1
ξ_t	$\mathcal{N}(\mu=0,\sigma^2=1)$
Q	-1
R	-2
γ	0.7

控制律均采用$\pi(s)=-Ks$的形式。初始状态均置为0。我们按以下几个方面来比较：

- **达到稳态时的状态分布**
  将反馈控制律代入，可以将状态空间模型化简为下面的形式：

$$s_{t+1}=(A-BK)s_t+\xi_t,\quad \xi_t\sim\mathcal{N}(0,\sigma^2)$$

  当达到稳态之后，对上式左右两侧取方差，并注意到此时$s_{t+1}=s_t=s$，因此移项整理之后可得状态s的分布：

$$s\sim\mathcal{N}\left(0,\frac{\sigma^2}{1-(A-BK)^2}\right).$$

  带入相应的参数值，我们可以得到两种cost函数下的稳态状态分布如下：

$$\begin{aligned}d_{\pi_{\mathrm{avg}}^*}(s)&=\sqrt{1-\left(A-BK_{\mathrm{Avg}}\right)^2}\frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{s^2}{2\sigma^2}\left(1-\left(A-BK_{\mathrm{Avg}}\right)^2\right)\right),\\ d_{\pi_\gamma^*}(s)&=\sqrt{1-\left(A-BK_\gamma\right)^2}\frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{s^2}{2\sigma^2}\left(1-\left(A-BK_\gamma\right)^2\right)\right).\end{aligned}$$

- **最优cost函数的值**
  我们在第5.1.2.2节中讨论了四种cost函数的值之间的关系。根据那里的公式以及Linear Quadratic控制器的具体情况，可得：

$$J_\gamma(\pi_\gamma^*) = \sum d_{\pi_{\text{avg}}^*}(s) v_\gamma^*(s) = \sigma^2 P_\gamma \left(\frac{\gamma}{1-\gamma} + \frac{1}{1-e^2} \right),$$
$$G_{\text{avg}}(\pi_\gamma^*) = (1-\gamma) \sum d_{\pi_\gamma^*}(s) v_\gamma^*(s) = \sigma^2 P_\gamma \left(\gamma + \frac{1-\gamma}{1-e^2} \right),$$
$$J_\gamma(\pi_{\text{avg}}^*) = \sigma^2 P_{\text{Avg}} \frac{1}{1-\gamma},$$
$$G_{\text{avg}}(\pi_{\text{avg}}^*) = \sigma^2 P_{\text{Avg}}.$$

  其中，前两个式子直接带入公式即可。而第四式原本是$G_{\text{avg}}(\pi_{\text{avg}}^*) = \sigma^2 \text{tr}(P_{\text{Avg}})$，这里因为状态维数为1，所以直接就是$P_{\text{Avg}}$。第三式是因为$J_\gamma(\pi_{\text{avg}}^*) = \frac{1}{1-\gamma} G_{\text{avg}}(\pi_{\text{avg}}^*)$由第四式得到的。那么，经过仿真可得到四个值分别为：

Performance measure	Real	Simulation	Error
$J_\gamma(\pi_\gamma^*)$	-14.64	-14.38	1.9%
$J_\gamma(\pi_{\text{avg}}^*)$	-16.17	-16.00	1.1%
$(1-\gamma)^{-1} G_{\text{avg}}(\pi_{\text{avg}}^*)$	-16.17	-16.41	1.5%
$(1-\gamma)^{-1} G_{\text{avg}}(\pi_\gamma^*)$	-21.28	-21.22	0.2%

  与我们在第5.1.2.2节中的结论一致。

5.1.3 策略迭代vs.值迭代

就像之前第二单元博客介绍过的，策略迭代是通过PEV和PIM两个步骤交替的优化最终取得最优解的。而值迭代则是首先通过fixed-point iteration求解Bellman方程，然后再通过贪心策略来计算最优策略。两种方法的收敛性分别由以下原因保证：策略迭代通过PEV和PIM两个步骤的交替优化，每一轮经过这两步后都可以保证更新后的策略π'优于原策略π。而值迭代则是通过Bellman算子的收缩性（contractive）来保证收敛性。

5.1.3.1 策略迭代

- **Policy Evaluation（PEV）**：PEV适用于model-based和model-free的RL。尽管各种PEV表面上看起来可能有很大差别，但是其核心是一样的，就是试图为当前的策略找到一个准确的估计。
- **Policy Improvement（PIM）**：PIM需要使得更新后的策略比更新前的策略更优。这可由Policy Improvement Theorem保证。详见第二单元博客。

5.1.3.2 值迭代

值迭代把Bellman方程看作一个contractive的operator，这个operator定义了一个在Banach空间里从一个点到另一个点的非线性映射。之后值迭代采用fixed-point iteration来求最优解。

5.2 DP的策略迭代（Policy Iteration）算法

5.2.1 Policy Evaluation（PEV）

不同与MC、TD等model-free RL那里的PEV是通过采样得到的样本来更新值函数的，像DP这类model-based RL的PEV是通过迭代求解self-consistency条件来更新值函数的。先来回顾一下第一类self-consistency条件：

$$v^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left\{ \sum_{s' \in \mathcal{S}} P_{ss'}^a \left(r_{ss'}^a + \gamma v^\pi(s') \right) \right\}, \forall s \in \mathcal{S},$$

那么下面的PEV算法就是利用上述第一类self-consistency条件来更新值函数的：

Repeat j until infinity

98 5 Model-Based Indirect RL: Dynamic Programming

$$V_{j+1}^{\pi}(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a|s) \left\{ \sum_{s' \in \mathcal{S}} \mathcal{P}\left(r + \gamma V_j^{\pi}(s')\right) \right\}, \forall s \in \mathcal{S}$$

End

这里当下标j趋向于无穷的时候就可以得到在当前策略下值函数的的准确值。注意，**在上述迭代过程中策略始终不变**。其实在实际计算中，并不用到无穷，而是当更新量小于某个阈值之后即可停机。这里正好再解释一下为什么DP被称为model-based的算法。注意到上述迭代公式中有一个$\mathcal{P}$，这个$\mathcal{P}$就是环境的模型，即环境的状态转移概率，而为了知道这个概率就需要要我们对于环境模型事先有充足的了解才行。再看看之前讲过的model-free RL，它就不需要知道环境的模型，只是根据sample来估计。举我们之前讲过的Q-learning的更新公式为例：

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_a Q(s',a) - Q(s,a) \right)$$

可以发现这个更新公式中并没有出现环境的模型$\mathcal{P}$。因此，有没有在算法中出现环境的模型就是model-based和model-free的一个重要区别。下面讲到的DP的值迭代的更新公式可以按照这个角度对照参看，看看其是否符合这里讲的model-based的特点。

下面从fixed-point iteration的角度来理解一下PEV算法。注意到第一类self-consistency条件的式子并不满足fixed-point iteration的标准形式，因为它左右两侧的变量一个是值函数$v^{\pi}(s)$，一个是$v^{\pi}(s')$。因此，我们需要对上式进行一些变形。首先，设状态空间有n个可能的取值，则我们将对应的n个方程写出来：

$$\begin{aligned}
V^{\pi}(s_{(1)}) &= \sum \pi \sum \mathcal{P}^{a}_{s_{(1)}s'_{(1)}} \left(r + \gamma V^{\pi}\left(s'_{(1)}\right) \right), \\
V^{\pi}(s_{(2)}) &= \sum \pi \sum \mathcal{P}^{a}_{s_{(2)}s'_{(2)}} \left(r + \gamma V^{\pi}\left(s'_{(2)}\right) \right), \\
&\vdots \\
V^{\pi}(s_{(n)}) &= \sum \pi \sum \mathcal{P}^{a}_{s_{(n)}s'_{(n)}} \left(r + \gamma V^{\pi}\left(s'_{(n)}\right) \right),
\end{aligned}$$

然后将这n个方程构成的方程组写成矩阵形式：

$$
\begin{gathered}
X=\gamma B X+b, \\
X=\left[V^{\pi}\left(s_{(1)}\right), V^{\pi}\left(s_{(2)}\right), \cdots, V^{\pi}\left(s_{(n)}\right)\right]^{\mathrm{T}} \in \mathbb{R}^{n}, \\
B=\left\{B_{ij}\right\}_{n \times n} \in \mathbb{R}^{n \times n}, \\
b=\left\{b_{i}\right\}_{n \times 1} \in \mathbb{R}^{n}, \\
B_{ij}=\sum_{a \in \mathcal{A}} \pi(a|s_{(i)}) \mathcal{P}_{s_{(i)} s_{(j)}}^{a}, \\
b_{i}=\sum_{a \in \mathcal{A}} \pi(a|s_{(i)}) \sum_{j} \mathcal{P}_{s_{(i)} s_{(j)}}^{a} r_{s_{(i)} s_{(j)}}^{a}.
\end{gathered}
$$

这里因为π和环境的模型$\mathcal{P}$是已知的，所以B和b都是定值。下面我们来利用fixed-point iteration的相关理论来解释两点：

- **PEV算法的收敛性**：在每轮迭代是，都会扫过整个状态空间，因此本算法的收敛性可以通过Bananach定理来保证。
- **上述矩阵方程只有一个fixed-point**：定义矩阵A：

  $$
  A \stackrel{\text{def}}{=} I_{n \times n}-\gamma B.
  $$

  如果A满秩则存在唯一解。考虑到我们选取的γ在(0,1)之间，所以由如下定理可得A满秩：

  $$
  \text{定理3：} \text{If } 0<\gamma<1, \operatorname{rank}(A)=n.
  $$

  证明如下：首先，易知$0 \leq B_{ij} \leq 1$且矩阵B每行的和都为1（这可通过$\sum_{j=1}^{n} B_{ij}=\sum_{j=1}^{n} \sum_{a \in \mathcal{A}} \pi(a|s_{(i)}) \mathcal{P}_{s_{(i)} s_{(j)}}^{a}$通过双重求和遍历了状态空间里所有i、j得到）。然后，易得

  $$
  \|B\|_{\infty}=\max _{i}\left|\sum_{j=1}^{n} B_{ij}\right|=1
  $$

  这里$\|\cdot\|_{\infty}$表示矩阵的无穷范数，即矩阵元素的绝对值按行求和后取最大值$\|A\|_{\infty}=\max _{1 \leq \mathrm{i} \leq n} \sum_{j=1}^{n}\left|a_{i j}\right|$）。那么因为$\gamma$为常数，所以有：

  $$
  \|\gamma B\|_{\infty}=\gamma\|B\|_{\infty}=\gamma<1.
  $$

  接下来，需要引入矩阵的谱半径的概念：

  $$
  \rho(A) \stackrel{\text{def}}{=} \max \left\{|\lambda|: \lambda \text{ is an eigenvalue of } A\right\}.
  $$

  那么，有如下不等式：

  $$
  \rho(\gamma B) \leq\|\gamma B\|_{\infty}<1,
  $$

  因此，γB的特征值中的最大值都小于1。又因为$I_{n \times n}$的特征值都是1，因此A的特征值均大于0。所以A是满秩的。证毕。

5.2.2 Policy Improvement（PIM）

这里主要是通过greedy的搜索来从PEV得到的值函数里面获得一个更优的策略。即：

$$
a^{*}=\arg \max _{a} q^{\pi}(s, a).
$$

$$
\pi^{\prime}(a|s)=\begin{cases}1, & \text{if } a=a^{*} \\ 0, & \text{if } a \neq a^{*}\end{cases}
$$

或写成：

$$\pi'(s) = \arg\max_{\pi'} q^{\pi}\Big(s, \pi'(s)\Big), \forall s \in \mathcal{S}.$$

Greedy搜索得到的策略π'是比原策略π更优的，这里证明已经在第二单元的博客里证明过了。详情可以参考那里的博客。

但是在DP这里并不能直接使用上面的做法。因为我们在PEV阶段获得的是状态值函数而不是动作值函数。为此，利用动作值函数和状态值函数之间的关系$q^{\pi}(s,a) = \sum_{s' \in \mathcal{S}} \mathcal{P}^{a}_{ss'}(r^{a}_{ss'} + \gamma v^{\pi}(s'))$可以对上述式子进行变形得到：

$$\pi'(s) \leftarrow \arg\max_{\pi'} \left\{ \sum_{s' \in \mathcal{S}} \mathcal{P}^{a}_{ss'} \left(r + \gamma V^{\pi}_{\infty}(s') \right) \right\}.$$

5.2.3 收敛性证明

策略迭代的收敛性取决于以下两个部分：

- **每次PEV利用的self-consistency条件都是可解的（solvable）**：换句话说，即每个策略对应的状态值函数的值都是有限的。或者说，每个中间策略都是可行的（feasible），能够有效的和环境交互。而且因为策略迭代的算法使然，每次策略都是建立在之前策略的基础上。因此，就要求初始的策略是可行的。
- **PIM每次获得的新策略是更优的**：这个在上面已经证明过了。

下面分两部分给出上述两个部分收敛性的详细证明。

5.2.3.1 PEV的收敛性

回顾DP的PEV更新公式的矩阵形式：

$$X = \gamma B X + b.$$

可以看出，这个算子每次相当于对X做了一个变换后把变换后的值赋值给X。我们定义这个变换：

$$\mathcal{L}(X) = \gamma B X + b,$$

我们下面要证明这个矩阵的收敛速度为γ-contractive。证明如下：

$$\begin{aligned}
\|\mathcal{L}(X_{j+1}) - \mathcal{L}(X_j)\|_{\infty} &= \gamma \|B(X_{j+1} - X_j)\|_{\infty} \\
&\leq \gamma \left\| B \max_{i \in \{1,2,\cdots,n\}} (|X^{i}_{j+1} - X^{i}_{j}|) \right\|_{\infty} \\
&= \gamma \left\| B \|X_{j+1} - X_j\|_{\infty} \right\|_{\infty} \\
&= \gamma \|B\|_{\infty} \|X_{j+1} - X_j\|_{\infty} \\
&= \gamma \|X_{j+1} - X_j\|_{\infty}.
\end{aligned}$$

从左边到右一式把$\mathcal{L}(X)$的定义式代入即可得。右一式到右二式根据矩阵乘法的意义结合了矩阵的无穷范数的定义可得，注意X是一个列向量。右二式到右三式根据矩阵的无穷范数的定义即可。右三式到右四式是矩阵的无穷范数的性质。右四式到右五式是因为上面已经证明过$\|B\|_{\infty} = 1$。所以，我们得到最终的不等式：

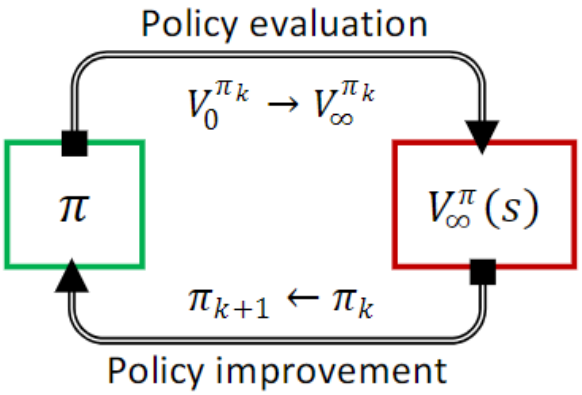
$$\|\mathcal{L}(X_{j+1}) - \mathcal{L}(X_j)\|_{\infty} \leq \gamma \|X_{j+1} - X_j\|_{\infty}.$$

接下来把$\mathcal{L}(X_{j+1}) = X_{j+2}$以及$\mathcal{L}(X_j) = X_{j+1}$代入上式，就可以得到：

$$\|X_{j+2} - X_{j+1}\|_{\infty} \leq \gamma \|X_{j+1} - X_j\|_{\infty}.$$

这就证明了每次相邻两次迭代的值函数的更新量ΔX是以指数速度收敛的。因此，$\mathcal{L}(X)$是一个γ-contractive的operator。由上述证明以及PEV那里关于只有一个fixed-point的证明，我们可以很轻松得到每次PEV在迭代次数趋于无穷时，值函数的估计值$V^{\pi}_{j}(s)$收敛于真实值函数$v^{\pi}(s)$（即那个唯一的fixed-point）。

5.2.3.2 DP的策略迭代的收敛性



欲证明DP的策略迭代的收敛性，同样需要证明以下两个部分：

- **每次PEV得到的值函数序列{$V_\infty^{\pi_0}, V_\infty^{\pi_1}, \cdots$}是单调递增的**
- **每次PIM得到的策略序列{$\pi_0, \pi_1, \cdots$}最终收敛到最优策略π^***

首先来证明第一点。即证明：

$$V_\infty^{\pi_0}(s) \leq V_\infty^{\pi_1}(s) \leq V_\infty^{\pi_2}(s) \leq \cdots \leq V_\infty^{\pi_k}(s) \leq V_\infty^{\pi_{k+1}}(s) \leq \cdots \leq v^*.$$

首先，根据我们在PEV收敛性那里的证明，我们知道每次PEV都会收敛到$v^{\pi_k}(s)$，即：

$$V_\infty^{\pi_k}(s) = v^{\pi_k}(s).$$

因为每次PIM更新后的策略都比原来更好，即：

$$v^{\pi_k}(s) \leq v^{\pi_{k+1}}(s) \quad \text{for } \forall s \in S.$$

那么我们用大V（估计值）来替换小v（真实值）就有：

$$V_\infty^{\pi_k}(s) \leq V_\infty^{\pi_{k+1}}(s).$$

证毕。

接下来证明第二点。即证明：

$$\text{When PIM stops improvement, i.e., } \pi_\infty = \pi^* \text{ if } \pi^* \text{ is unique}$$

首先，需要澄清两个2概念。可能一开始看到这个式子的时候会感到奇怪，π^*和π_∞不就是同一个东西吗？为啥还要证明啊？这样想就大错特错了。这里π^*指的是满足Bellman最优性方程的最优策略，而π_∞指的是当PIM停止迭代的时候得到的策略。这两个策略是不一样的。那么，我们接下来就证明这两个策略是一样的。针对相邻的两个策略π_k和π_{k+1}对应的状态值函数，我们有：

$$v^{\pi_{k+1}}(s) = \max_a \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \left(r_{ss'}^a + \gamma v^{\pi_k}(s') \right), \forall s \in \mathcal{S}$$

这里的max下面加了一个a表示这里a是可以自由变动的变量。同时，当PIM终止时，有

$$\pi_{\infty+1} = \pi_\infty$$

综合上面两式就有：

$$v^{\pi_\infty}(s) = v^{\pi_{\infty+1}}(s) = \max_a \sum_{s' \in \mathcal{S}} \mathcal{P}\left(r + \gamma v^{\pi_\infty}(s') \right), \forall s \in \mathcal{S},$$

即：

$$v^{\pi_\infty}(s) = \max_a \sum_{s' \in \mathcal{S}} \mathcal{P}\left(r + \gamma v^{\pi_\infty}(s')\right), \forall s \in \mathcal{S},$$

而我们知道根据第一类Bellman方程解出来的最优策略π^*满足：

$$v^{\pi^*}(s) = \max_a \sum_{s' \in \mathcal{S}} \mathcal{P}\left(r + \gamma v^{\pi^*}(s')\right), \forall s \in \mathcal{S}.$$

那么，根据上面的推导，我们就可以得到$\pi_\infty = \pi^*$。证毕。

5.2.4 使用Newton-Raphson法来解释DP的策略迭代

可能大家会对于DP的策略迭代背后的原理感到好奇，例如为什么策略迭代为什么往往比值迭代更快的收敛，为什么通常更具有鲁棒性？这背后有没有什么数学原理呢？这里就提供一种使用最优化理论里的Newton-Raphson法来解释DP的策略迭代的原理

首先介绍一下Newton-Raphson法。它的迭代公式为：

$$X_{k+1} = X_k - [\nabla g(X_k)]^{-1} g(X_k),$$

$g(X)$是一个关于向量X的向量函数，$\nabla g(X)$就是$g(X)$的Jacobian矩阵，即：

$$\nabla g(X) = \begin{bmatrix} \frac{\partial g_1(X)}{\partial X_1} & \frac{\partial g_1(X)}{\partial X_2} & \cdots & \frac{\partial g_1(X)}{\partial X_n} \\ \frac{\partial g_2(X)}{\partial X_1} & \frac{\partial g_2(X)}{\partial X_2} & \cdots & \frac{\partial g_2(X)}{\partial X_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g_m(X)}{\partial X_1} & \frac{\partial g_m(X)}{\partial X_2} & \cdots & \frac{\partial g_m(X)}{\partial X_n} \end{bmatrix}.$$

这里的$\nabla g(X)$是一个m行n列的矩阵。

现在我们尝试使用Newton-Raphson法来解释DP的策略迭代。考虑一个简化的情况，其中状态空间$\mathcal{S}$和动作空间$\mathcal{A}$都各自只有两个元素。首先回顾一下第一类Bellman方程：

$$v^*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a) \left(r^a_{ss'} + \gamma v^*(s')\right), \forall s \in \mathcal{S}.$$

那么我们首先为两个可能的状态取值$s_{(1)}$和$s_{(2)}$写出它们的第一类Bellman方程：

$$V^\pi(s_{(1)}) = \max_a \sum_{s'} \mathcal{P}^a_{s_{(1)}s'}\left(r + \gamma V^\pi(s')\right)$$
$$V^\pi\left(s_{(2)}\right) = \max_a \sum_{s'} \mathcal{P}^a_{s_{(2)}s'}\left(r + \gamma V^\pi(s')\right),$$
$$a \in \{a_{(1)}, a_{(2)}\}.$$

这里我们引入Bellman operator来做一个形式上的化简：

$$X = \mathcal{B}(X)$$

这里$X = \begin{bmatrix} V^\pi(s_{(1)}) \\ V^\pi(s_{(2)}) \end{bmatrix}$，这里注意$\mathcal{B}$是一个非线性的operator，因为在Bellman方程中包含max操作。为了进一步化简，可以引入函数$g(X)$:$\mathbb{R}^2 \to \mathbb{R}^2$：

$$g(X) = \mathcal{B}(X) - X,$$

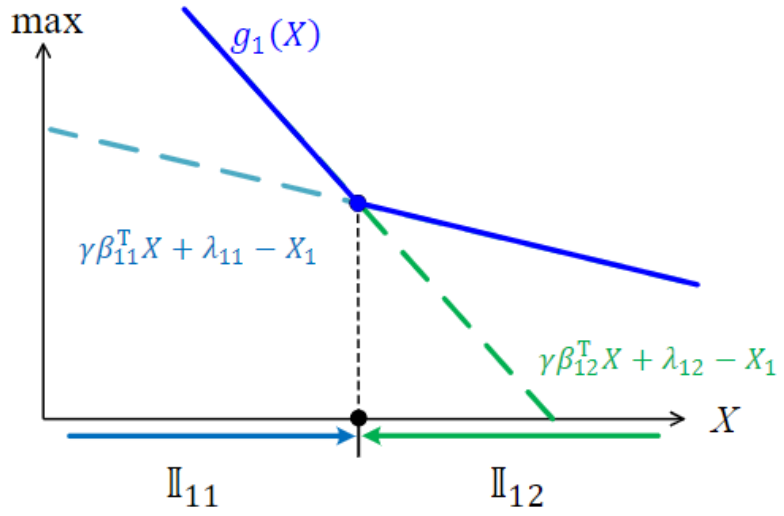
$g(X)$可以写成分量的形式：

$$g(X)=\begin{bmatrix}g_1(X)\\g_2(X)\end{bmatrix}$$

其中每个分量$g_i(X)$的具体表达式如下：

$$g_i(X)=\max_j\left(\gamma\beta_{ij}^{\mathrm{T}}X+\lambda_{ij}\right)-X_i,i,j\in\{1,2\},$$
$$\beta_{ij}=\left[\mathcal{P}_{s_{(i)}s_{(1)}}^{a_{(j)}},\mathcal{P}_{s_{(i)}s_{(2)}}^{a_{(j)}}\right]^{\mathrm{T}},\lambda_{ij}=\sum_{s'}\mathcal{P}_{s_{(i)}s'}^{a_{(j)}}r_{s_{(i)}s'}^{a_{(j)}}.$$

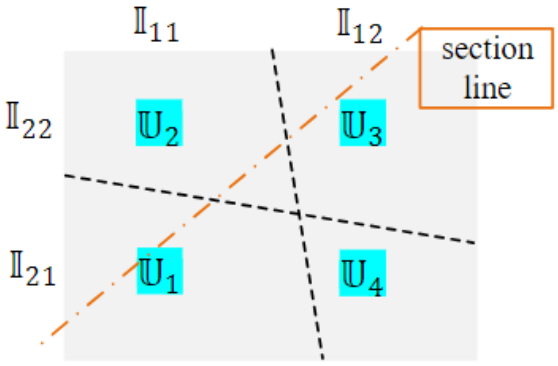
这里的$g_i(X)$仍然包含max操作，但是经过我们观察可以知道，$g_i(X)$是一个分成两段的各段均为线性函数的分段函数。



为此，可以把$g_i(X)$写成：

$$g_i(X)=\begin{cases}\gamma\beta_{i1}^{\mathrm{T}}X+\lambda_{i1}-X_i, & \text{for}X\in\mathbb{I}_{i1}\\ \gamma\beta_{i2}^{\mathrm{T}}X+\lambda_{i2}-X_i, & \text{for}X\in\mathbb{I}_{i2}\end{cases},$$

这里$\mathbb{I}_{i1}$和$\mathbb{I}_{i2}$分别表示使得$g_i(X)$在分别取动作$a_{(1)}$和$a_{(2)}$达到max的X的子区间。那么因为$g(X)$有两个分量$g_1(X)$和$g_2(X)$，而每个分量都包含两个子区间，因此关于$g(X)$的两个分量分别取什么动作达到最值就有四种情况。将平面划分为四个区域：



但是，在每个子区域内部，$g(X)$的表达式都是唯一的。因此，四个子区域内部的表达式可以统一写为：

$$g(X)=\gamma\beta_\sigma X+\lambda_\sigma-X,\text{for }X\in\mathbb{U}_\sigma,\sigma\in\{1,2,3,4\}$$
$$\beta_\sigma=\begin{bmatrix}\mathcal{P}_{s_{(1)}s_{(1)}}^{a^*} & \mathcal{P}_{s_{(1)}s_{(2)}}^{a^*}\\ \mathcal{P}_{s_{(2)}s_{(1)}}^{a^*} & \mathcal{P}_{s_{(2)}s_{(2)}}^{a^*}\end{bmatrix},\lambda_\sigma=\begin{bmatrix}\sum_{s'}\mathcal{P}_{s_{(1)}s'}^{a^*}r_{s_{(1)}s'}^{a^*}\\ \sum_{s'}\mathcal{P}_{s_{(2)}s'}^{a^*}r_{s_{(2)}s'}^{a^*}\end{bmatrix},$$

其中，a^*表示在此时的区域内应该采取的最优动作（注意，第一行的a^*和第二行的a^*不一定一样）。这样写成标准的形式后，就方便后续的求导操作了。不过，还要补充说明的一点是单看上面的分段函数图像，在拐点处并不连续，不过这里随便取左边或右边段

的导数值即可。有论文证明过这并不影响Newton-Raphson法的收敛性。那么，我们可以写出这四个区域内Jacobian矩阵的统一表达式：

$$\nabla g(X)=\gamma\beta_\sigma-I, \text{for } X\in\mathbb{U}_\sigma, \sigma=1,2,3,4$$

那么Newton-Raphson法的迭代公式就可以写成：

$$\begin{aligned}X_{k+1}&=X_k-[\nabla g(X)]^{-1}g(X_k)\\&=X_k-(\gamma\beta_\sigma-I)^{-1}(\gamma\beta_\sigma X_k+\lambda_\sigma-X_k)\\&=-(\gamma\beta_\sigma-I)^{-1}\lambda_\sigma.\end{aligned}$$

至此，我们就获得了从Newton-Raphson法的角度来得到的每轮迭代更新公式：

$$X_{k+1}=-(\gamma\beta_\sigma-I)^{-1}\lambda_\sigma.$$

下面我们**从DP的策略迭代的角度**来看看获得的迭代公式能不能于与上面刚得到的更新公式等价。在第k轮迭代时，经历过PEV后，可以得到X_{k+1}的表达式：

$$X_{k+1}=\gamma BX_k+b.$$

这里的B和b的具体表达式如下：

$$\begin{aligned}B_{\pi_{(i,j)}}&=\sum_{a\in\mathcal{A}}\pi\big(a\big|s_{(i)}\big)\mathcal{P}\big(s_{(j)}|s_{(i)},a\big),\\b_{\pi_{(i)}}&=\sum_{a\in\mathcal{A}}\pi\big(a\big|s_{(i)}\big)\sum_j\mathcal{P}\big(s_{(j)}|s_{(i)},a\big)r^a_{s(i)s(j)}.\end{aligned}$$

那么在第k轮的PIM中，更新策略其实就是寻找一个策略π_{k+1}使得：

$$\pi_{k+1}=\arg\max_\pi\{\gamma B_\pi X_k+b_\pi\}.$$

在进行第k+1轮的PEV时，X_{k+1}应该还满足在策略π_{k+1}下的self-consistency条件：

$$X_{k+1}=\gamma B_{\pi_{k+1}}X_{k+1}+b_{\pi_{k+1}},$$

这个方程的解为（**从策略迭代角度得到的更新公式**）：

$$X_{k+1}=\left(I-\gamma B_{\pi_{k+1}}\right)^{-1}b_{\pi_{k+1}}.$$

如果策略采取确定性的greedy策略，那么B和b可以简化为：

$$\begin{aligned}B^*_{ij}&=\sum_{a\in\mathcal{A}}\pi^*\big(a\big|s_{(i)}\big)\mathcal{P}\big(s_{(j)}|s_{(i)},a\big)=\mathcal{P}^{a^*}_{s_{(i)}s_{(j)}},\\b^*_i&=\sum_{a\in\mathcal{A}}\pi^*\big(a\big|s_{(i)}\big)\sum_j\mathcal{P}\big(s_{(j)}\big|s_{(i)},a\big)r=\sum_{s'\in\mathcal{S}}\mathcal{P}^{a^*}_{s_{(i)}s'}r^{a^*}_{s_{(i)}s'}.\end{aligned}$$

其中，a^*是最优策略（值函数最大）。比较$B_{\pi_{k+1}}$和β_σ，$b_{\pi_{k+1}}$和λ_σ，我们可以发现这两个矩阵和向量是一样的。因此，从策略迭代角度得到的更新公式也可以写成：

$$X_{k+1}=\left(I-\gamma B_{\pi_{k+1}}\right)^{-1}b_{\pi_{k+1}}=-(\gamma\beta_\sigma-I)^{-1}\lambda_\sigma.$$

这恰与Newton-Raphson法的更新公式一致！因此，我们就证明了策略迭代的两步cycle与Newton-Raphson法是等价的。尽管上面的分析是在一个简化的情况下进行的，但是这个结论是可以推广到一般的情况的。同时，由于Newton-Raphson法具有二阶收敛性，因此策略迭代也具有二阶收敛性。这也解释了为什么策略迭代往往比值迭代更快的收敛，更具有鲁棒性。

5.3 DP的值迭代（Value Iteration）算法

5.3.1 DP的值迭代（for Discounted Cost）

5.3.1.1 算法描述

DP的值迭代算法并不需要中间的策略，也不需要初始时指定一个可行的策略。值迭代算法首先计算出最优的值函数，然后直接从这个最优的值函数中获得最优策略。另外，值迭代算法的收敛性由Bellman operator的contraction性质保证。回顾第一类Bellman方程：

$$v^*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a) \left(r^a_{ss'} + \gamma v^*(s') \right), \forall s \in \mathcal{S}.$$

那么，一种最简单的值迭代算法就是直接利用上述方程进行迭代（其实是从Picard Fixed-Point Iteration的角度得来的）：

Repeat k until infinity

$$V_{k+1}(s) \leftarrow \max_a \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left(r^a_{ss'} + \gamma V_k(s') \right), \forall s \in \mathcal{S}.$$

End

注意，与PEV那里迭代时V的右上角有一个上标π不同，这里的V是不带上标的，因为值迭代不涉及任何的中间策略。当迭代次数达到无穷的时候，值函数的估计值$V_k(s)$就收敛于真实值函数$v^*(s)$。

5.3.1.2 DP的值迭代的收敛性证明

首先，定义Bellman operator：

$$\mathcal{B}(V) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a) \left(r^a_{ss'} + \gamma V(s') \right),$$

那么值迭代的更新公式可以写成：

$$V_{k+1} = \mathcal{B}(V_k).$$

我们下面证明这个operator是一个contraction operator：

$$\begin{aligned}
\left| B\left(V_{k+1}(s_{(i)})\right) - B\left(V_k(s_{(i)})\right) \right| &= \left| \max_a \sum_{s' \in \mathcal{S}} \mathcal{P}\left(r + \gamma V_{k+1}(s')\right) - \max_a \sum_{s' \in \mathcal{S}} \mathcal{P}\left(r + \gamma V_k(s')\right) \right| \\
&\leq \max_a \left| \sum_{s' \in \mathcal{S}} \mathcal{P}\left(r + \gamma V_{k+1}(s')\right) - \sum_{s' \in \mathcal{S}} \mathcal{P}\left(r + \gamma V_k(s')\right) \right| \\
&= \gamma \max_a \left| \sum_{s' \in \mathcal{S}} P V_{k+1}(s') - \sum_{s' \in \mathcal{S}} P V_k(s') \right| \\
&\leq \gamma \max_a \sum_{s' \in \mathcal{S}} \mathcal{P} \left| V_{k+1}(s') - V_k(s') \right| \\
&\leq \gamma \max_a \sum_{s' \in \mathcal{S}} \mathcal{P} \max_{s \in \mathcal{S}} \left| V_{k+1}(s) - V_k(s) \right| \\
&= \gamma \max_a \left\{ 1 \times \max_{s \in \mathcal{S}} \left| V_{k+1}(s) - V_k(s) \right| \right\} \\
&= \gamma \max_{s \in \mathcal{S}} \left| V_{k+1}(s) - V_k(s) \right|
\end{aligned}$$

右边第一式到第二式和第三式到第四式比较显然，举个小例子即可。第五式到第六式是因为$\sum_{s'\in\mathcal{S}}\mathcal{P}$后面乘的是一个公因式，直接提取出来。而$\sum_{s'\in\mathcal{S}}\mathcal{P}$求和由概率的性质可知等于1。注意，从第五步开始的s表示的是状态空间里的一个状态，该状态可以使得$|V_{k+1}(s)-V_k(s)|$取到最大值。而对最开始等号左边的式子取无穷范数可得：

$$\begin{aligned}\left\|\mathcal{B}\big(V_{k+1}(s)\big)-\mathcal{B}\big(V_k(s)\big)\right\|_\infty &= \max_s\left|\mathcal{B}\big(V_{k+1}(s)\big)-\mathcal{B}\big(V_k(s)\big)\right| \\ &\le \gamma\max_s|V_{k+1}(s)-V_k(s)| \\ &= \gamma\|V_{k+1}(s)-V_k(s)\|_\infty.\end{aligned}$$

注意，这里的s则表示的是状态空间里的所有状态，对应的$V_{k+1}(s)$、$V_k(s)$均表示的是一个向量。这样就证明了Bellman operator在无穷范数下的γ-contractive性质。因此，值迭代算法是收敛的，会收敛到唯一的一个fixed-point，即最优值函数$v^*(s)$。

5.3.2 DP的值迭代（for Average Cost）

这里首先需要引入一种新的return（或说cost）（不同于discounted return、average return）：**Differential Return**：

$$G_\Delta(s_t)=\sum_{i=0}^{\infty}\Big(r_{t+i}-G_{\text{avg}}(\pi)\Big),$$

其中，$G_{\text{avg}}(\pi)$是在策略π下的average return。那么根据第二单元博客里讲过的从return到值函数的的逻辑，我们可以定义一种新的状态值函数；

$$v_\Delta^\pi(s)=\mathbb{E}_\pi\{G_\Delta(s_t)|s_t=s\}=\mathbb{E}_\pi\left\{\sum_{i=0}^{\infty}\left(r_{t+i}-G_{\text{avg}}(\pi)\right)\right\},$$

这种值函数被称为**Differential State Value Function**。那么，也就顺理成章的引出了在这种值函数下的Bellman方程：

$$\begin{aligned}v_\Delta^*(s)&=\max_{a\in\mathcal{A}}\sum_{s'\in\mathcal{S}}\mathcal{P}_{ss'}^a\Big(r_{ss'}^a-G_{\text{avg}}^*(\pi)+v_\Delta^*(s')\Big)\\ v_\Delta^*(s)+G_{\text{avg}}^*(\pi)&=\max_{a\in\mathcal{A}}\sum_{s'\in\mathcal{S}}\mathcal{P}_{\text{ss}'}^a\left(r_{\text{ss}'}^a+v_\Delta^*(s')\right).\end{aligned}$$

这个式子被称为**Average Cost Optimality Equation（ACOE）**，即Bellman方程的differential return版本，也被称为**Differential Bellman Equation**。求解这个方程主要使用三种方法，分别是**Finite-Horizon Value Iteration**、**Relative Value Iteration**和**Vanishing Discount Factor Approach**。将在下面分别介绍。

对于Average Cost问题，还需要进行一些额外的操作来保证最优解存在：如采用stationary policy（只与当前状态有关，与之前地1历史状态无关）；引入一些regularity条件。

另外注意，关于DP Value Iteration for Average Cost这部分内容，仅作为介绍，并没有面面俱到。有一些内容并没有很详细，因此看起来有些地方前后可能并不是很连贯（比如differential value function在后面叙述中并没有出现很多次），这里只是作为一个引子，希望大家能够有所启发。

5.3.2.1 Finite-Horizon Value Iteration

这种方法的思路是，将原问题转化为一个有限步长的问题，而当步长很大时，就可取得较好的近似效果，不过由于计算量较大，在实际中只有迭代次数较少的时候才会使用此种方法。这种方法首先定义了一个有限步长的值函数：

$$V(s,N)=\sum_{t=0}^{N-1}r_t$$

之后可以通过求解Bellman方程得到$V^*(s,N)$，那么average cost就可以通过下式来近似得到：

$$G(\pi^*) \approx \frac{V^*(s,N)}{N}.$$

不过这个方法面临着一个困境，即当我们选取的horizon N很大时，计算量会变得很大，而当N很小时，近似效果又不是很好。因此，这种方法在实际中的应用较少。

5.3.2.2 Relative Value Iteration

这里的思路是使用引入一个额外的变量，最后求出这个变量的最优值后可以发现其与ACOE的解是一样的。

这个方法的核心思路是从所有的值函数中减去一个相同的常量。方法的具体步骤如下。首先，定义一个新的值函数：

$$h(s) = V(s,N) - V(\zeta,N), N \to \infty,$$

其中ζ是一个固定的状态。$h(s)$称为**Difference Value Function**。使用fixed-point iteration的方法，可以得到$h(s)$的更新公式：

$$\begin{aligned}
&h_{k+1}(s) \\
&= \max_a \sum \mathcal{P}_{ss'}^a \left(r_{ss'}^a + V_k(s',N)\right) - \max_a \sum \mathcal{P}_{\zeta\zeta'}^a \left(r_{\zeta\zeta'}^a + V_k(\zeta',N)\right) \\
&= \max_a \sum \mathcal{P}_{ss'}^a \left(r_{ss'}^a + h_k(s') + V_k(\zeta,N)\right) - \max_a \sum \mathcal{P}_{\zeta\zeta'}^a \left(r_{\zeta\zeta'}^a + h_k(\zeta') + V_k(\zeta,N)\right) \\
&= \max_a \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a \left(r_{ss'}^a + h_k(s')\right) - \max_a \sum \mathcal{P}_{\zeta\zeta'}^a \left(r_{\zeta\zeta'}^a + h_k(\zeta')\right).
\end{aligned}$$

当迭代达到终止（$h^*(s) = \lim_{k\to\infty} h_k(s)$）时，可以得到如下关系：

$$\begin{aligned}
h^*(s) &= -\rho^* + \max_a \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a \left(r_{ss'}^a + h^*(s')\right), \\
\rho^* &\overset{\text{def}}{=} \max_a \sum \mathcal{P}_{\zeta\zeta'}^a \left(r_{\zeta\zeta'}^a + h^*(\zeta')\right),
\end{aligned}$$

注意上式左右两侧的变量一样，都是$h^*(s)$，因此可以使用fixed-point iteration的方法来求解$h^*(s)$。最后，可以得到最优对$[\rho^*, h^*(s)]$。之后，可证明$[\rho^*, h^*(s)]$与ACOE中的量有如下关系：

$$h^*(s) = v_\Delta^*(s) \quad \rho^* = G_{\text{avg}}^*(\pi)$$

5.3.2.3 Vanishing Discount Factor Approach

这里的核心思路是把average cost问题使用discounted cost问题来近似。核心思想是使得$\gamma \to 1$，核心是下述的式子：

$$\begin{aligned}
G_{\text{avg}}(\pi) &= \lim_{N\to\infty} \lim_{\gamma\to 1} \frac{\mathbb{E}\{\sum_{i=0}^{N-1} \gamma^i r_{t+i}\}}{\sum_{i=0}^{N-1}\gamma^i} \\
&= \lim_{\gamma\to 1} \frac{\lim_{N\to\infty} \mathbb{E}\{\sum_{i=0}^{N-1} \gamma^i r_{t+i}\}}{\lim_{N\to\infty} \sum_{i=0}^{N-1}\gamma^i} \\
&= \lim_{\gamma\to 1} (1-\gamma)\mathbb{E}\left\{\sum_{i=0}^{\infty} \gamma^i r_{t+i}\right\} \\
&= \lim_{\gamma\to 1} (1-\gamma) v_\gamma(s).
\end{aligned}$$

右边第二式到第三式使用了等比数列求和公式（注意γ是小于1的）。这样，average cost问题就可以转化为discounted cost问题。这种方法的优点如下：

- 可以处理ill-conditioned问题；
- 解决Discounted Cost问题的策略迭代和值迭代算法较为成熟，稳定性和收敛性较好。

5.4 关于DP问题的几点讨论

5.4.1 对于DP的策略迭代的扩展

在DP的策略迭代中，每一轮迭代中的PEV都要进行无穷轮迭代才能得到当前策略π下的值函数估计值。但是在实际编写算法的时候，不可能进行无穷次迭代的。因此，实际中只要两次迭代之间的值函数估计值的差值小于一个阈值时，就可以停止迭代：

$$\max_{s\in\mathcal{S}}|V_{j+1}^{\pi}(s)-V_{j}^{\pi}(s)|<\Delta.$$

或者，也可以强行规定迭代次数，当迭代次数达到一定值时，就停止迭代，这也被称为Truncated DP。这里的DP在每轮大的迭代中，进行n轮PEV和1轮PIM，这样的迭代方法被称为**n-step DP**。当$n\to\infty$时，就是标准的DP的策略迭代算法。这种方法的通用框架如下：

Hyperparameters: Discount factor γ, PEV iteration number n

Initialization: $V(s)\leftarrow 0, \pi(s)\leftarrow$ arbitrary action

Repeat (indexed with k)

(1) Evaluate policy

Repeat n times

Sweep $s\in\mathcal{S}$

$$V^{\pi_k}(s)\leftarrow\sum_{s'}p\big(s'\big|s,\pi_k(s)\big)\big(r+\gamma V^{\pi_k}(s')\big)$$

End

End

(2) Greedy search

Sweep $s\in\mathcal{S}$

$$\pi_{k+1}(s)\leftarrow\arg\max_{a}\sum_{s'}p(s'|s,a)\big(r+\gamma V^{\pi_k}(s')\big)$$

End

End

注意，这里的每次PEV的更新公式：

$$V^{\pi_k}(s)\leftarrow\sum_{s'}p(s'|s,\pi_k(s))\big(r+\gamma V^{\pi_k}(s')\big)$$

与我们在策略迭代哪里的公式稍有不同：

$$V_{j+1}^{\pi}(s)\leftarrow\sum_{a\in\mathcal{A}}\pi(a|s)\left\{\sum_{s'\in\mathcal{S}}\mathcal{P}\left(r+\gamma V_{j}^{\pi}(s')\right)\right\},\forall s\in\mathcal{S}$$

这是因为我们这里采用的策略是确定性策略（greedy），因此自动就化简为这样了。

需要注意的是，上述DP被称为同步的（syncronous）的。大的迭代中每轮小的PEV迭代在进行之前都要先将此时的值函数取值储存在一个copy中，然后在该轮PEV扫过每个状态s的时候，右边的值函数的值用的都是copy中的。这样可以防止“串联更新”（学过数据结构的话会发现这里的设置copy的思路与目的与Bellman-Ford求最短路那里完全一样）。假设有m个动作和n个状态，该算法的时间复杂度为$O(mn^2)$（采用状态值函数）或$O(m^2n^2)$（采用动作值函数）。

5.4.2 用统一的视角来看Model-Based和Model-Free的RL方法

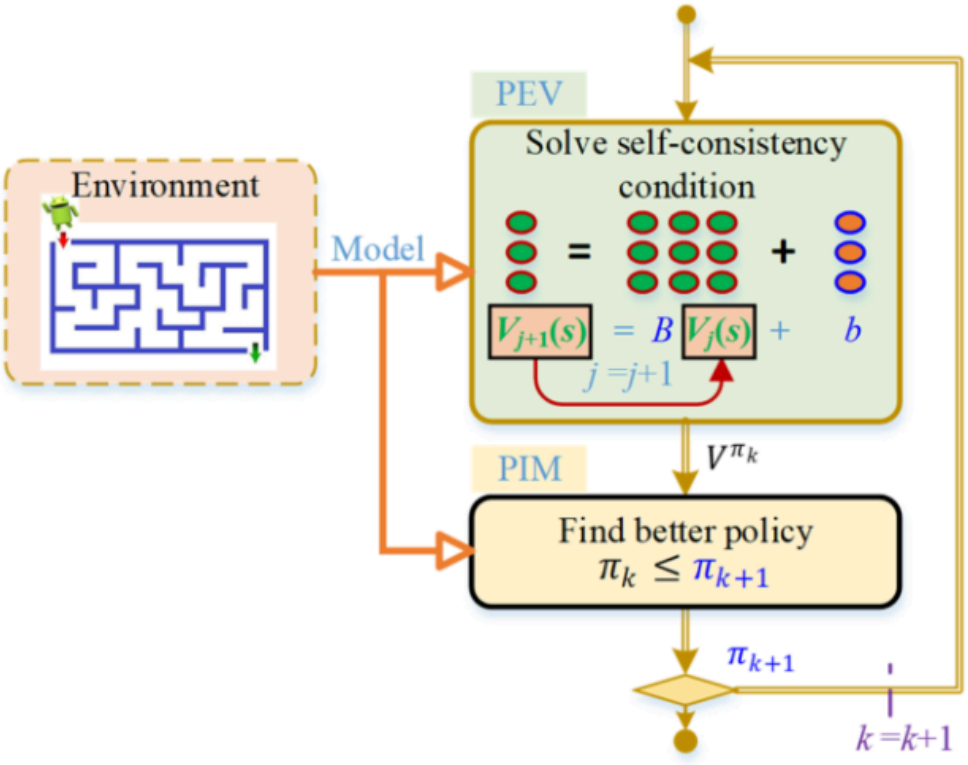


Figure 5.13 Unified framework for model-based RL

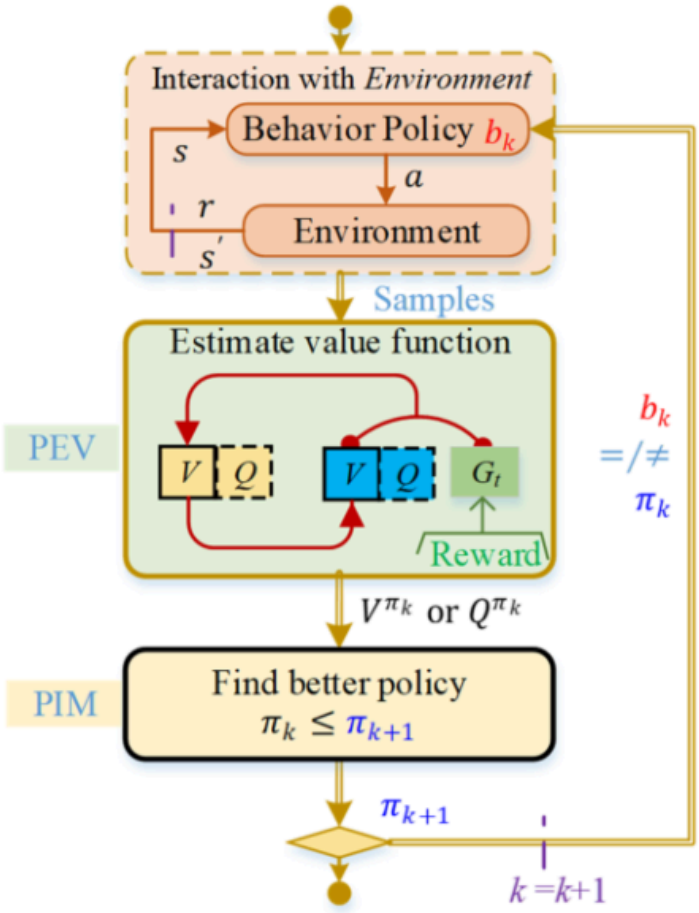


Figure 5.14 Unified framework for model-free RL

对于策略迭代的RL算法，我们可以把其中Model-Based和Model-Free的方法统一起来，放在Generalized Policy Iteration（GPI）的框架下。GPI的框架每轮更新策略的大迭代由PEV和PIM两部分构成。每次PEV是为了计算得到一个对于值函数更好的估计，而PIM则是为了在新的值函数的基础上得到一个更好的策略。

Model-Based和Model-Free的方法的区别在于PEV中如何获取环境的动力学（Environment Dynamics）的。Model-Based方法根据先验知识，***直接使用环境模型来获得环境信息***；*而Model-Free方法则是通过与环境的交互（采样）来获得环境信息**。

两种方法的优缺点如下表所示：

优缺点	Model-Based RL	Model-Free RL
优点	1. 因为有了环境模型往往比Model-Free的算法更高效， 因为环境模型是对环境的完整描述， 而sample只是对于环境的局部的描述。	1. 不需要环境模型，因此更加通用； 2. 从采样器获得的数据往往比从环境模型得到的更准确。
缺点	1. 对环境建模很难， 且获得的环境模型与实际的环境动力学之间难免有误差。	1. 效率较低， 需要与环境进行很大量的交互才能达到比较满意的效果； 2. 样本通常只能获得局部的环境信息， 很难通过采样覆盖到整个状态空间。

由上表我们也可以看出，想说出Model-Based和Model-Free哪个更好是不现实的，因为这两种方法各有优缺点，适用于不同的场景。如果一定要比较，应该指定需要处理的问题和超参数之后才能进行公平的比较。

5.4.3 使用Fixed-Point Iteration的角度来看策略迭代和值迭代

5.4.3.1 策略迭代和Fixed-Point Iteration技术

在策略迭代里面，使用fixed-point iteration的技术的地方是在PEV的过程中。总共有五种常见的PEV方法，分别是：Picard iteration，Krasnoselskij iteration，Mann iteration，Ishikawa iteration，Kirk iteration。下面汇总一下之前讲过的策略迭代算法使用的迭代分属于哪种iteration：

	Model-based	Model-free
Picard	PEV in DP policy iteration	--
Krasnoselskij	--	SARSA, Expected SARSA (with constant learning rate)
Mann	--	SARSA, Expected SARSA (with varying learning rate)
Ishikawa	--	--
Kirk	--	TD(n), TD-lambda

下面以两个例子来说明如何使用fixed-point iteration技术来解释策略迭代算法。首先，来看DP中的策略迭代算法。根据我们在使用Newton-Raphson法来解释DP的策略迭代那里的定义，这里用到的算子可以写成如下形式：

$$f(X) \overset{\text{def}}{=} \gamma BX + b.$$

我们之前已经证明了这个算子是收缩的（contractive），因此迭代最后必然会收敛到一个fixed-point。又因为之前已经证明了DP的策略迭代这里有且只有一个fixed-point，因此策略迭代算法一定会收敛到最优值函数$v^*(s)$。

下面我们来看看Model-Free算法如何使用fixed-point iteration技术进行分析。直接使用fixed-point iteration技术来分析Model-Free算法是比较困难的，因为采样具有随机性，很难在两个相邻的sample之间定义出一个contractive的operator。为了解决这个问题，需要引入期望形式的算子。下面我们就以SARSA为例来分析一下。SARSA的PEV更新公式为：

$$Q_{j+1}(s,a) = Q_j(s,a) + \alpha_j\Big(r + \gamma Q_j(s',a') - Q_j(s,a)\Big),$$

其中，α_j是第j轮迭代的学习率。将上式的同类项合并，得到：

$$Q_{j+1}(s,a) = (1-\alpha_j)Q_j(s,a) + \alpha_j\Big(r + \gamma Q_j(s',a')\Big).$$

两边取期望，得到：

$$\mathbb{E}_\pi\{Q_{j+1}(s,a)\} = (1-\alpha_j)\mathbb{E}_\pi\{Q_j(s,a)\} + \alpha_j\mathbb{E}_\pi\{r + \gamma Q_j(s',a')\}.$$

注意这里的取期望操作是针对于环境模型的，即$\mathbb{E}_\pi\{\#\}$等价于$\sum_{s'\in\mathcal{S}}\{\mathcal{P}^a_{ss'}\cdot\#\}$。那么，接下来就可以得到一个基于期望的算子：

$$f\big(Q(s,a)\big) \overset{\text{def}}{=} \mathbb{E}_\pi\{r + \gamma Q(s',a')\} = \sum_{s'\in\mathcal{S}}\mathcal{P}^a_{ss'}\big(r + \gamma Q(s',a')\big).$$

可以证明这个算子是contractive的，因此就可以得到一个类似于Mann iteration的fixed-point iteration算法：

$$Q_{j+1}(s,a) = (1-\alpha_j)Q_j(s,a) + \alpha_j f\Big(Q_j(s,a)\Big).$$

这就是SARSA的PEV迭代公式。

其它的Model-Free算法也可以使用类似的方法来进行分析，找到对应于fixed-point iteration的解释。

5.4.3.2 值迭代和Fixed-Point Iteration技术

同样的，先放上一张表，汇总一下值迭代算法使用的迭代方法：

	Model-based	Model-free
Picard	DP value iteration	--
Krasnoselskij	--	Q-learning with a constant learning rate
Mann	--	Q-learning with a varying learning rate
Ishikawa	--	--
Kirk	--	--

先来看看Model-Based的DP，正如我们之前讲过的，其值迭代算法天然就是一个fixed-point iteration算法。算子为：

$$f(X) \stackrel{\text{def}}{=} \mathcal{B}(X).$$

其中，$\mathcal{B}$是Bellman operator。其收敛性也已经证过了。还有一个问题，就是既然已经知道了值迭代可以用fixed-point iteration来解释，那么反过来能不能使用不同的fixed-point iteration技术设计新的值迭代算法呢？答案是肯定的。不过，一般对于DP的值迭代，我们使用上述基于Picard iteration的最简单的算法就可以了。

再来看看Model-Free的RL算法。这里以Q-Learning为例（为什么以Q-Learning为例参见第四单元博客。这是因为Q-Learning不显含PEV与PIM过程，而是合在一个公式中。而这个公式也可看成一种值迭代的算法）。因为Model-Free的RL算法主要通过采样来与环境进行交互，而在采样的时候自然会引入随机性。不能直接使用fixed-point iteration技术来分析。不过，我们可以通过引入期望的方式来进行分析。首先来看看学习率随迭代变化的Q-Learning算法：

$$Q_{k+1}(s,a) = Q_k(s,a) + \alpha_k \left(r + \gamma \max_{a'} Q_k(s',a') - Q_k(s,a)\right),$$

整理得：

$$Q_{k+1}(s,a) = (1-\alpha_k)Q_k(s,a) + \alpha_k \mathbb{E}_\pi\Big\{r + \gamma \max_{a'} Q_k(s',a')\Big\}.$$

那么，就可以得到一个算子：

$$f\big(Q(s,a)\big) = \mathbb{E}_\pi\big\{r + \gamma \max_{a'} Q(s',a')\big\}.$$

接下来，如果令

$$X_k = Q_k(s,a)$$

那么，上式就可以写成：

$$X_{k+1} = (1-\alpha_k)X_k + \alpha_k f(X_k).$$

这个式子恰好符合Mann iteration的形式。因此，Q-Learning的值迭代算法可以看成是一个Mann iteration的fixed-point iteration算法。那么，显然根据之前所述，我们也可以根据不同的fixed-point iteration技术来为Q-Learning设计新的值迭代算法。比如，可以使用Picard iteration来设计一个新的Q-Learning算法:

$$Q_{k+1}(s,a) = r + \gamma \max_{a'} Q_k(s',a').$$

也可以根据Ishikawa iteration来设计一个新的Q-Learning算法：

$$Q^{\text{temp}}(s,a) = (1-\alpha_k)Q_k(s,a) + \alpha_k\Big\{r + \gamma \max_{a'} Q_k(s',a')\Big\},$$
$$Q_{k+1}(s,a) = (1-\beta_k)Q_k(s,a) + \beta_k\Big\{r + \gamma \max_{a'} Q^{\text{temp}}(s',a')\Big\}.$$

这里是一个2-step的Q-Learning迭代公式，这里引入了中间变量Q^{temp}。这样的迭代公式可以加速收敛，因为相当于在一次迭代中进行了两次值迭代。但是，为了保证收敛，对于学习率α_k和β_k要求更严格：

$$0 \leq \alpha_k, \beta_k < 1,$$
$$\lim_{k\to\infty} \beta_k = 0, \lim_{k\to\infty} \alpha_k = 0,$$
$$\sum_{k=1}^{\infty} \beta_k = \infty, \sum_{k=1}^{\infty} \alpha_k = \infty.$$

另外，Ishikawa iteration对于较弱的contractive operator可能具有更好的收敛性。因此，在遇到一些ill-conditioned的问题时，可以尝试使用Ishikawa iteration来加速收敛。

5.4.4 Exact DP with Backward Recursion

让我们来考虑一个离散时间的，具有固定终止时间的T的问题。那么显然，此时的值函数不仅与状态s有关，还与时间t有关。可以这样理解，因为值函数表示的是从当前状态开始到终止获得的回报的期望。那么，所有过程到达T时刻均会终止，因此自然从相同状态但是不同时刻开始获得的回报不会一样。因此，我们可以定义一个新的值函数：

$$V^*(s,t) = \max_{\pi} \Big\{ \sum_{i=0}^{T-t} r_{t+i} | s_t = s \Big\}.$$

计算这种值函数的值通常有两种顺序，一种是从t=T开始，逆向计算；另一种是从t=0开始，正向计算。逆向计算的方法显然更符合逻辑，因此我们下面主要介绍逆向计算的方法。那么，可获得如下与实践有关的Bellman方程：

$$V^*(s,t) = \max_{\pi} \Big\{ \sum_{i=0}^{T-t} r_{t+i} | s_t = s \Big\}.$$

为了获得时刻t的值函数，我们需要递推的先获得时刻t+1的值函数。因此，这种方法也被称为backward recursion。因为终止时刻的值函数可以直接得到，因此我们就可以迭代的从后向前计算。可以发现，这与之前DP的迭代是很像的，都是从一个初始值开始不断迭代。也即是说，当终止时刻T取为∞时，这种方法就是DP的值迭代算法。

5.5 对什么是一个“更好的”策略？以及从更好的策略的角度看PIM过程

在第三单元博客的策略改进定理那里，我们定义过什么是一个更好的策略，这是使用逐个元素的方式来定义的：

$$\pi \leq \bar{\pi} \iff v^{\pi}(s) \leq v^{\bar{\pi}}(s), \forall s \in \mathcal{S}$$

该式子还有一种等价形式，即：

$$\pi \leq \bar{\pi} \iff v^{\pi}(s) \leq \sum_{a \in \mathcal{A}} \bar{\pi}(a|s) q^{\pi}(s,a), \forall s \in \mathcal{S}.$$

但是，这种定义方式只能针对元素离散的情况，无法扩展到连续的情况。

5.5.1 什么是一个“更好的”策略？用期望的形式来定义的第一种方式

为了解决上述问题，我们可以使用期望的形式来定义什么是一个更好的策略。比如，上述第一式可以写为：

$$\mathbb{E}_{s \sim d(s)}\{v^{\pi}(s)\} \leq \mathbb{E}_{s \sim d(s)}\{v^{\overline{\pi}}(s)\},$$

其中，$d(s)$是状态分布。这种定义方式可以扩展到连续的情况（求期望的时候离散情况就是按概率加权求和，连续情况就是积分）。这种期望形式相当于从原来逐个元素比较转化为了对于加权求和之后的整体比较。

引入了这种定义方式之后，就可以根据其设计PIM的策略更新方式。根据上述期望形式的定义，PIM可以被表述为一个优化问题：

$$\begin{gathered}\pi_{k+1} = \arg\max_{\pi}\{\delta - \rho(\pi, \pi_k)\}, \\ \text{s.t.} \\ \mathbb{E}_{s \sim d(s)}\{v^{\pi}(s)\} = \delta + \mathbb{E}_{s \sim d(s)}\{v^{\pi_k}(s)\}, \\ \delta \geq 0,\end{gathered}$$

这里的$\rho(\pi, \pi_k)$是一个度量两个策略之间的差异的函数。而δ是一个常数，被称为value margin。有了下面的s.t.的限制条件，就可以保证每次的新策略一定比上一次的策略更好，因为：

$$\mathbb{E}_{s \sim d(s)}\{v^{\pi}(s)\} = \delta + \mathbb{E}_{s \sim d(s)}\{v^{\pi_k}(s)\} \geq \mathbb{E}_{s \sim d(s)}\{v^{\pi_k}(s)\}.$$

而优化时为什么要使得$\delta - \rho(\pi, \pi_k)$最大呢？这是因为，$\rho(\pi, \pi_k) \geq 0$，而两个策略越接近该函数的值就越接近于0又因为$\delta$是一个常数，所以最大化$\delta - \rho(\pi, \pi_k)$就相当于最小化$\rho(\pi, \pi_k)$，也就是说，我们优化的目标是使得这相邻两轮迭代的分布尽可能的接近。当$\rho(\pi, \pi_k) = 0$时，说明两个策略完全一样，此时就可以停止迭代了。根据这个规则进行PIM可以保证至少收敛到一个局部最优解。证明的要点有两点。首先是保证$\mathbb{E}_{s \sim d(s)}\{v^{\pi_k}(s)\}$单调递增；其次是证明迭代到无穷轮时，此时的值函数$v^{*}\pi^{\infty}(s)$等于最优值函数$v^{*}(s)$。

首先，我们注意到$\mathbb{E}_{s \sim d(s)}\{v^{\pi_k}(s)\}$其实就是之前在第5.1.2.2节定义过的折扣代价函数$J_{\gamma}(\pi)$。那么，为了回答上述证明药店的第一点，先看下面的定理：

$$\text{定理4：}\ J(\pi_0) \leq J(\pi_1) \leq \cdots \leq J(\pi_k) \leq J(\pi_{k+1}) \leq \cdots \leq J(\pi^{*}),$$

其中，π^{*}是最优策略。这个定理的证明十分简单，从每次迭代时求解的优化问题的的限制条件即可看出。即：

$$J(\pi_{k+1}) = \mathbb{E}_{s \sim d(s)}\{v^{\pi_{k+1}}(s)\} = \delta + \mathbb{E}_{s \sim d(s)}\{v^{\pi_k}(s)\} \geq \mathbb{E}_{s \sim d(s)}\{v^{\pi_k}(s)\} = J(\pi_k).$$

下面我们来证明第二点，即迭代到无穷轮时，此时的值函数$v^{*}\pi^{\infty}(s)$等于最优值函数$v^{*}(s)$。我们有下述定理：

$$\begin{gathered}\text{定理5：假设}J(\cdot)\text{是严格凸的。那么当PIM在第}k \to \infty\text{轮停止时, 我们有} \\ \pi_{\infty}(s) = \pi^{*}(s), \forall s \in \mathcal{S}.\end{gathered}$$

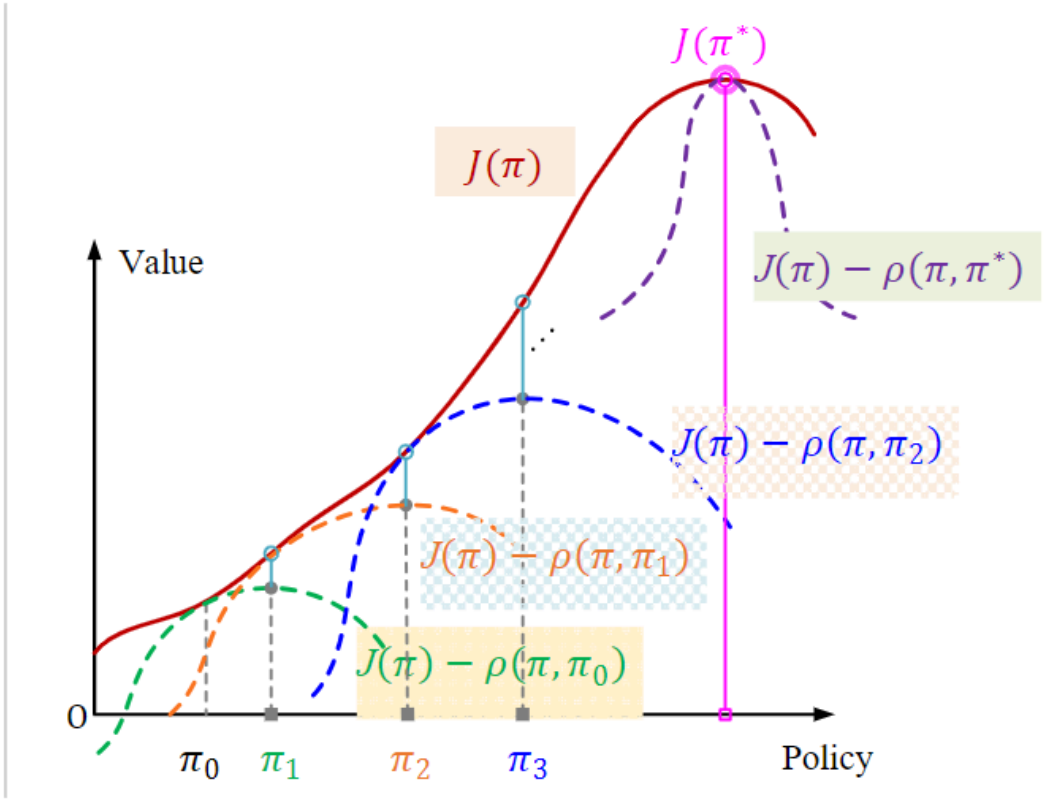
因为第k+1轮时，$J(\pi_k)$可以看成是一个常数，因此我们可以把我们刚才提出的由PIM所转化为的优化问题的δ使用$J(\pi_k)$来替代。那么，我们的优化问题就变成了：

$$\begin{gathered}\pi_{k+1} = \arg\max_{\pi}\{J(\pi) - \rho(\pi, \pi_k)\}, \\ \text{s.t.} \\ J(\pi) - J(\pi_k) \geq 0.\end{gathered}$$

下面来定义一个辅助的函数：

$$g(\pi,\pi_k)\overset{\text{def}}{=}J(\pi)-\rho(\pi,\pi_k).$$

因为$\rho(\pi,\pi_k)\geq 0$，所以$g(\pi,\pi_k)\leq J(\pi)$。引入的这个函数可以作为J的一个下界。具体证明过程省略。



上图展示了迭代过程中J函数和g函数的变化。可以看出，在迭代过程中g函数是逐渐逼近J函数的。当迭代到无穷轮时，g函数就会收敛到J函数。

另外，使用这种期望形式的定义方式，即可保证收敛。RL不需要每次去改进状态空间里的每个元素，但是在最后的策略中，每个元素仍然均达到了其最优值。

5.5.2 什么是一个“更好的”策略？用期望的形式来定义的第二种方式

仿照逐元素定义的第二种方式，可得其对应的期望形式：

$$\mathbb{E}_{s\sim d(s)}\{v^{\pi}(s)\}\leq\mathbb{E}_{s\sim d(s)}\left\{\sum_{a\in\mathcal{A}}\bar{\pi}(a|s)q^{\pi}(s,a)\right\}.$$

我们首先需要证明，对上式右端的最大化与greedy search得到的策略是一样的。

$$\text{定理6：对上式的最大化相当于greedy search，即}\\ \bar{\pi}^*\Leftrightarrow\tilde{\pi}^*,$$

其中，$\bar{\pi}^*$是上式右端的最大化策略，$\tilde{\pi}^*$是greedy search得到的策略。即：

$$\bar{\pi}^*=\arg\max_{\bar{\pi}}\mathbb{E}_{s\sim d(s)}\Big\{\sum_{a\in\mathcal{A}}\bar{\pi}(a|s)q^{\pi}(s,a)\Big\},$$
$$\tilde{\pi}^*=\arg\max_{\tilde{\pi}}\Big\{\sum_{a\in\mathcal{A}}\tilde{\pi}(a|s)q^{\pi}(s,a)\Big\},\forall s\in\mathcal{S}.$$

注意，虽然两个策略在上面均显示为一个式子，但是对于$\bar{\pi}^*$来说，确实就只有一个式子（所有的可能值根据分布加权平均了）；而对于$\tilde{\pi}^*$来说，对于每个s都有一个这样的式子，所以总共有$|\mathcal{S}|$个式子。因为$\bar{\pi}^*$是通过最大化$\mathbb{E}_{s\sim d(s)}\Big\{\sum_{a\in\mathcal{A}}\bar{\pi}(a|s)q^{\pi}(s,a)\Big\}$得到的，那么自然有：

$$\mathbb{E}_{s\sim d(s)}\Big\{\sum_{a\in\mathcal{A}}\bar{\pi}^*(a|s)q^{\pi}(s,a)\Big\}\geq\mathbb{E}_{s\sim d(s)}\Big\{\sum_{a\in\mathcal{A}}\tilde{\pi}^*(a|s)q^{\pi}(s,a)\Big\}.$$

这是因为上述不等式左右两边形式相同，而$\bar{\pi}^*$是通过最大化该形式的式子所得到最优策略，所以其自然优于包含$\tilde{\pi}^*$在内的所有其他策略。现在假设$\bar{\pi}^*$最大化了上述期望形式的目标式子，但是对于某个具体的$\sum_{a\in\mathcal{A}}\tilde{\pi}(a|s)q^{\pi}(s,a)$并没有最大化。与此同时，$\tilde{\pi}^*$是通过greedy search得到的，因为对于每个状态s，都最大化了$\sum_{a\in\mathcal{A}}\tilde{\pi}(a|s)q^{\pi}(s,a)$，所以套上对于状态s分布的期望之后，下式仍成立：

$$\mathbb{E}_{s\sim d(s)}\left\{\sum\tilde{\pi}^*(a|s)q^{\pi}(s,a)\right\}\geq\mathbb{E}_{s\sim d(s)}\left\{\sum\bar{\pi}^*(a|s)q^{\pi}(s,a)\right\},$$

有以上两式得到，只能有：

$$\mathbb{E}_{s\sim d(s)}\left\{\sum\tilde{\pi}^*(a|s)q^{\pi}(s,a)\right\}=\mathbb{E}_{s\sim d(s)}\left\{\sum\bar{\pi}^*(a|s)q^{\pi}(s,a)\right\}.$$

也即是说，$\bar{\pi}^*$和$\tilde{\pi}^*$是一样的。证毕。

在上述证明的基础上，我们可以得到下面的PIM更新方式：

$$\begin{aligned}\pi_{k+1}&=\arg\max_{\pi}\{\delta-\rho(\pi,\pi_k)\},\\&\text{s.t.}\\\mathbb{E}_{s\sim d(s)}\left\{\sum\pi(a|s)q^{\pi_k}(s,a)\right\}&=\delta+\mathbb{E}_{s\sim d(s)}\{\sum\pi_k(a|s)q^{\pi_k}(s,a)\},\\&=\delta+\mathbb{E}_{s\sim d(s)}\{v^{\pi_k}(s)\}\\&\delta\geq0.\end{aligned}$$

下面给出是否按照上述方式得到的策略π_{k+1}按照逐元素的“更好的策略”的定义也是更好的。假设存在一个状态$s_{(i)}$，使得单个元素的第二类判断方式不成立，即：

$$v^{\pi_k}(s_{(i)})>\sum\pi_{k+1}(a|s_{(i)})q^{\pi_k}(s_{(i)},a).$$

那么，构造一个替代的策略：

$$\tilde{\pi}(a|s)=\begin{cases}\pi_{k+1}(a|s),s\in S\setminus\{s_{(i)}\}\\\pi_k(a|s),s\in\{s_{(i)}\}\end{cases},$$

那么，显然对于这个新构建的策略来说，逐元素的第二类判断方式是成立的：

$$v^{\pi_k}(s)\leq\sum\tilde{\pi}(a|s)q^{\pi_k}(s,a),\forall s\in S$$

对于$\tilde{\pi}$和π_{k+1}，它们的δ分别为：

$$\begin{aligned}\tilde{\delta}&=\mathbb{E}_{s\sim d(s)}\left\{\sum\tilde{\pi}(a|s)q^{\pi_k}(s,a)\right\}-\mathbb{E}_{s\sim d(s)}\{v^{\pi_k}(s)\}.\\\delta_{k+1}&=\mathbb{E}_{s\sim d(s)}\left\{\sum\pi_{k+1}(a|s)q^{\pi_k}(s,a)\right\}-\mathbb{E}_{s\sim d(s)}\{v^{\pi_k}(s)\}.\end{aligned}$$

接下来，定义$\Delta=\tilde{\delta}-\delta_{k+1}$，那么有：

$$
\begin{aligned}
&= \mathbb{E}_{s\sim d(s)}\left\{\sum \tilde{\pi}(a|s) q^{\pi_k}(s,a)\right\} - \mathbb{E}_{s\sim d(s)}\left\{\sum \pi_{k+1}(a|s) q^{\pi_k}(s,a)\right\} \\
&= \mathbb{E}_{s\sim d(s)}\left\{v^{\pi_k}\left(s_{(i)}\right) - \sum \pi_{k+1}\left(a|s_{(i)}\right) q^{\pi_k}\left(s_{(i)},a\right)\right\} \\
&> 0.
\end{aligned}
$$

其中，第二式到第三式是根据我们之前的假设。又因为$\tilde{\pi}$与π_{k+1}除了在$s_{(i)}$处不同之外，其它地方都是一样的，而$\tilde{\pi}$在$s_{(i)}$处等于上一个策略π_k的取值。因此，$\tilde{\pi}$更接近于π_k，所以相应的策略之间的距离度量越小：

$$
\rho(\tilde{\pi}, \pi_k) \leq \rho(\pi_{k+1}, \pi_k).
$$

结合上述两式，有：

$$
\left(\tilde{\delta} - \rho(\tilde{\pi}, \pi_k)\right) > \left(\delta_{k+1} - \rho(\pi_{k+1}, \pi_k)\right),
$$

这与π_{k+1}是通过最大化$\delta - \rho(\pi, \pi_k)$得到的策略矛盾。因此，假设不成立，即对于所有的状态$s_{(i)}$，逐元素的第二类判断方式都是成立的。证毕。

5.6 Policy Entropy？以及从Policy Entropy的角度看PIM过程

5.6.1 什么是Policy Entropy

Policy Entropy定义如下：

$$
\mathcal{H}(\pi) = \sum_{a\in\mathcal{A}} -\pi(a|s)\log\pi(a|s),
$$

或写成：

$$
\mathcal{H}(\pi) = \sum_{a\in\mathcal{A}} -p(a)\log p(a),
$$

，其中$p(a) = \pi(a|s)$。显然，一个策略的随机性越大，其Policy Entropy越大。因此，可以通过最大化一个基于Policy Entropy的目标函数来获得一个随机策略。使用这种最大化基于Policy Entropy的目标函数的方法的一个好处是，获得的相应的动作分布只取决于优化时所加的限制。一些常见的case如下：

Type	Probability density	Constraints	Acton space
Uniform (discretete)	$p(a) = 1/\|cA\|$	–	$a \in \{a_1, a_2, ..., a_{\|cA\|}\}$
Uniform (cont)	$p(a) = 1/(a_{\text{high}} - a_{\text{low}})$	–	$a \in [a_{\text{low}}, a_{\text{high}}]$
Bernoulli	$p(a) = \mu^a(1-\mu)^{1-a}$	$\mathbb{E}(a) = \mu$	$a \in \{0, 1\}$
Normal	$p(a) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{(a-\mu)^2}{2\sigma^2}\right)$	$\mathbb{E}(a) = \mu$	$a \in (-\infty, +\infty)$

下面我们来看看如何通过最大化基于Policy Entropy的目标函数来获得ϵ-greedy策略。简而言之，可以通过如下的优化问题：

$$
\begin{aligned}
\pi_{k+1} &= \arg\max_{\pi}\{\pi(a^*|s) + \kappa\mathcal{H}(\pi)\}, \\
&\text{s.t.} \\
a^* &= \max_a q^{\pi_k}(s,a), \forall s \in \mathcal{S}, \\
\pi(a^*|s) &+ \sum_{a\in\mathcal{A}\setminus a^*} \pi(a|s) = 1,
\end{aligned}
$$

注意，这里的s是固定的某个状态（也就是说，对于每个状态空间中的s，都需要做一遍上述优化）。如果我们把系数κ取为：

$$\kappa = \frac{1}{\ln(1-|\mathcal{A}|(\epsilon-1)/\epsilon)}.$$

此时得到的策略就是ϵ-greedy策略。可以使用拉格朗日乘子的方法来求解上述优化问题，最后可解得：

$$\pi(a^*|s) = 1-\epsilon+\frac{\epsilon}{|\mathcal{A}|}, \quad \pi(a|s)|_{a\neq a^*} = \frac{\epsilon}{|\mathcal{A}|}.$$

这与我们在第三单元博客里讲过的ϵ-greedy策略的公式是一样的。

由此可见，通过引入Policy Entropy，可以获得更多不同的策略。

附录：Fixed-Point Iteration技术

附录1：Fixed-Point Iteration技术

先来看看固定点的定义：

> **Fixed-Point**: 当一个迭代函数的输入和输出一样时，该输入（或输出）值被称为固定点。设$\mathbb{X}$为Banach空间，$f:\mathbb{X}\to\mathbb{X}$为Banach空间上的一个映射，如果存在$X*\in\mathbb{X}$，使得$f(X^*)=X^*$，那么$X^*$就是$f$的一个固定点。

> **Contraction Mapping**: 如果一个映射f满足：
>
> $$\|f(X)-f(Y)\| \leq \gamma\cdot\|X-Y\|, \forall X,Y\in\mathbb{X}$$
>
> 这里的$\gamma\in(0,1)$是一个李普希茨系数。

Fixed-Point理论关注的两个重点是存在性和收敛性。该理论主要的作用在于求解方程，可以用来求解复杂的非线性方程。

$$\begin{gathered}\text{定理7：设}\mathbb{X}\text{是一个完备的Banach空间，}f:\mathbb{X}\to\mathbb{X}\text{是一个收缩映射，}\\ \text{那么f 有一个唯一的固定点}X^*\text{，且其柯西序列收敛到该固定点：}\\ X_n\to X^*,\text{as } n\to\infty,\end{gathered}$$

这里的$\{X_n\}$来自于Picard iteration，其初始值任选。该定理就是著名的Banach Fixed-Point定理，也被称为Contraction Mapping定理。

上面定理中的迭代点序列来自于Picard iteration。除了Picard iteration，还有其他四种常见的Fixed-Point迭代方法：

- **Picard iteration**:

  $$X_{n+1}=f(X_n)$$

  如果f是一个收缩映射，那么该迭代方法收敛到f的固定点。
- **Krasnoselskii iteration**:

  $$X_{n+1}=\lambda f(X_n)+(1-\lambda)X_n$$

  这里的$\lambda\in(0,1)$。如果引入一个算子：$f_\lambda=(1-\lambda)I+\lambda f$，那么这种迭代方式就是Picard iteration。这种迭代也于bootstrap方法类似，通过加权已知的过去值和新的信息来获得。
- **Mann iteration**:

  $$X_{n+1}=(1-\alpha_n)X_n+\alpha_n f(X_n)$$

这里的$\alpha_n \in (0,1)$。这种迭代方式是$Krasnoselskiiiteration$的一个特例，当\alpha_n=\lambda$时，就是Krasnoselskii iteration。

- **Ishikawa iteration**:

$$X_{n+1} = (1-\alpha_n)X_n + \alpha_n F(X_n)$$

这里的$\alpha_n \in (0,1)$，$\beta_n \in (0,1)$。这种迭代方式的收敛要求：

$$0 \leq \alpha_n, \beta_n < 1, \lim_{n\to\infty} \beta_n = 0, \lim_{n\to\infty} \alpha_n = 0$$
$$\sum_{n=1}^{\infty} \alpha_n = \infty, \sum_{n=1}^{\infty} \beta_n = \infty.$$

当$\alpha_n = 0$时，Ishikawa iteration就是Mann iteration。

- **Kirk iteration**:

$$X_{n+1} = c_0 X_n + c_1 f(X_n) + c_2 f\left(f(X_n)\right) + \cdots + c_k f^k(X_n),$$

这里，k是一个常数，$k \geq 1, c_i \geq 0$，$\sum_{i=0}^{k} c_i = 1$。当k=1时，Kirk iteration就是Krasnoselskii iteration。

上述四种迭代方式都有其对应的柯西序列。其收敛速度由每次迭代得到的值于最优值（固定点）的距离决定。在Picard iteration中，收敛速度至少是由李普希茨系数γ和第一次的误差$||X_1 - X_0||$决定的:

$$\|X_n - X^*\| \leq \frac{\gamma^n}{1-\gamma}\|X_1 - X_0\|,$$

有趣的是，对于well-defined的任务，其它四种迭代方式与Picard iteration的收敛速度是同阶的。

附录2：使用Fixed-Point Iteration技术解线性方程组

Fixed-Point Iteration技术在解大规模线性方程组时也有广泛的应用。对于有m个方程的方程组，使用Fixed-Point Iteration技术求解每轮迭代的操作数为m^2量级，而使用直接求解的方法的操作数为m^3量级。而且，达到收敛所需的迭代轮数通常独立于方程组的规模。

假设现在我们有一个线性方程组$Ax = b$，其中A是一个满秩的方阵。那么其迭代序列（柯西序列）为：

$$X_{n+1} = PX_n + (I-P)A^{-1}b,$$
$$P = I - A,$$

P被称为迭代矩阵。这个算法只有当$\rho(P) < 1$时才能收敛。这里的$\rho(P)$是P的谱半径。因为$\rho(P)$是P的特征值的绝对值里面的最大值，所以显然$\rho(P)$越小，迭代收敛的速度越快。为了获得更好的收敛性，可以使用一些技巧，比如把A分成两部分，即A=M-N。这样处理之后就可以使用一系列的迭代方法，比如Jacobi iteration、Gauss-Seidel iteration、relaxation method等。

@[toc]

书籍链接：Reinforcement Learning for Sequential Decision and Optimal Control

本篇博客主要介绍使用函数近似的（Function Approximation）Indirect RL算法。读者可能会注意到我们之前讲过的方法的状态空间和动作空间都是有限的，也就是说，它们均是使用表格的方式来存储的（Tabular Representation）。那么为什么我们这里要使用函数来近似呢？首先，求解具有很大的状态空间的序列决策问题很困难。比如，双陆棋的状态空间是10^{20}，而围棋的状态空间是10^{170}。其次，很多问题的状态空间本身就是连续的，其状态个数有无穷多个，自然不能使用表格来存储。

根据对哪个部分使用函数近似，这类方法可进一步细分为以下三种：

- **Value Approximation Only**：这类方法使用固定点迭代技术来更新参数化的值函数，并在得到最优的值函数（收敛）之后，每次通过最优的值函数来选择动作，而并不显式地给出策略。这类算法的代表有DQN。
- **Policy Approximation Only**：这类方法通常指一些naive的策略梯度方法，比如REINFORCE以及一些finite-horizon的neuro-DP方法。
- **Actor-Critic Approximation**：这类架构是目前RL里面最常用的架构，它同时使用了值函数近似和策略近似。在该架构里，critic通过不断更新对于值函数的估计来为当前策略的好坏提供一个更准确的评价；而actor则指导策略参数向更好表现的防线更新。这类方法因为其建立在基于梯度的优化上，因此通常具有很好的收敛性，而一些降低方差的技术，如baseline，有助于达到更好的收敛。Actor-Critic方法既可以从Indirect RL的角度来看，也可以从Direct RL的角度来看。从Indirect RL的角度来看，PEV和PIM都被基于梯度的优化方法所替代，分别对应了critic update和actor update。从Direct RL的角度来看，actor-critic方法是一种对于Policy Gradient方法的自然扩展。它的值函数被递归地计算以降低方差。虽然有种种优点，这类方法也还是有不少缺点的，比如缺少通用的收敛性保证、不可预测的deadly triad以及容易困在局部最优等等。Actor-Critic方法的代表有A3C（Asynchronous Advantage Actor-Critic）、TD3（Twin Delayed Deep Deterministic Policy Gradient）、SAC（Soft Actor-Critic）、DSAC（Distributional Soft Actor-Critic）等。

6.1 线性函数近似以及常见基函数

函数近似的目的就是使用一个已知形式的函数来近似目标函数。通常我们知道的条件仅仅包括一些等式条件以及一些data samples，而它们往往还是不准确和有噪声的。因此，选择一种合适的参数化近似方式就显得尤为重要。近似方式分为两种：线性近似和非线性近似。线性近似主要是利用一些基础函数的线性组合来近似目标函数，而非线性近似则是利用诸如神经网络等来近似目标函数。这里主要介绍线性近似。

一个线性近似可以写成如下形式：

$$
\begin{aligned}
g(\cdot;w) &= w^{\mathrm{T}} \cdot F(s) \\
w &= [w_1, \cdots, w_l]^{\mathrm{T}} \in \mathbb{R}^l \\
F(s) &= [f_1(s), f_2(s), \cdots, f_l(s)]^{\mathrm{T}} \in \mathbb{R}^l,
\end{aligned}
$$

这里$g(\cdot;w)$是最终的线性函数，写成权重向量w和特征向量$F(s)$的内积形式。$F(s)$是**特征向量**，w是权重向量，l是特征向量的维度。特征向量$F(s)$的每一个分量都是状态s的一个函数，这些函数一般取为相同形式的基函数，**因此，$g(\cdot;w)$最终可以写成一系列同类函数的线性加权组合**。基函数的具体选择多种多样，这里重点介绍几种常见的基础函数：二元基函数、多项式基函数、傅里叶基函数、径向基函数。

6.1.1 二元基函数

这里介绍两种常见的方法：state aggregation和tile coding。state aggregation是将状态空间划分为若干个不相交的子区域，接下来使用独热编码来构建二元特征（根据当前状态在哪个子区域里）。tile coding是将状态空间划分为若干个tiling，每个tiling又被划分为若干个tile。那么每个特征根据当前状态s是否落在某个tile上来取值。这里，根据我的理解，向量函数$F(s)$的维数l等于stage aggregation的子区域个数或者tile coding的tile的总个数。

这两种方法的优点是比较简单直观，缺点是随着状态空间的增大，特征维度也会增大，这会导致计算量指数级增长，很快会变得不可行。

6.1.2 多项式基函数

最常见的多项式基函数是monomial basis。其每个分量形式如下：

$$
\begin{aligned}
f_i(s) &= \prod_{j=1}^{n} s_j^{c_{i,j}}, \\
\sum_j c_{i,j} \leq d, c_{i,j} &\in \{0, 1, ..., n\},
\end{aligned}
$$

这里，假设状态向量s是一个n维向量，即$s = [s_1, s_2, ..., s_n]^{\mathrm{T}}$，$d$是多项式的最高次数。因为$f_i(s)$是通过连乘得到的，所以次数应该等于各个分量的次数相加。从$F(s)$的每个分量$f_i(s)$的形式也可以看出为什么这种基叫做monomial basis：因为每个分量均是一个单项式。最终的用于拟合的函数形式如下：

$$
g(s;w) = w_1 \prod_{j=1}^{n} s_j^{c_{1,j}} + w_2 \prod_{j=1}^{n} s_j^{c_{2,j}} + \cdots + w_l \prod_{j=1}^{n} s_j^{c_{l,j}}
$$

应指出，提高次数可以增强拟合能力，但是也会增加对于噪声的敏感度。

其它常见的多项式基函数还包括：

- **Bernstein Basis**
- **Orthogonal Polynomials**：这里的Orthogonal（正交）指的是基中任意两个基函数的在某种内积的定义下内积为0。代表有Legendre、Chebyshev、Hermite、Laguerre Polynomials等。

6.1.3 傅里叶基函数

傅里叶基函数是一种基于正弦和余弦函数的基函数。尤其适合处理周期数据和具有已知边界的函数。假设我们现在的状态s是一个n维向量，即$s = [s_1, s_2, ..., s_n]^{\mathrm{T}}$，且每个分量$s_i$均在区间$[0, 1]$上。那么d阶余弦形式傅里叶基函数的每个分量的形式如下：

$$
\begin{gathered}
f_i(s) = \cos(\pi c_i^{\mathrm{T}} s), \\
c_i = [c_{i,1}, c_{i,2}, \cdots, c_{i,n}]^{\mathrm{T}}, c_{i,j} \in \{0, 1, \cdots, d\},
\end{gathered}
$$

最后的线性函数形式如下：

$$
g(s; w) = w_1 \cos(\pi c_1^{\mathrm{T}} s) + w_2 \cos(\pi c_2^{\mathrm{T}} s) + \cdots + w_l \cos(\pi c_l^{\mathrm{T}} s)
$$

采用傅里叶基函数的优点是可以把周期函数使用一系列的正弦和余弦函数来近似（因为$F(s)$是对于不同阶数的三角函数进行加权，这与傅里叶级数的形式极为相似！），而且可以通过调整阶数d来调整拟合的精度。在某些特殊情况下，甚至可以得出闭式解从而得到理论解。

6.1.4 径向基函数

径向基函数（Radial Basis Function, RBF）是一种基于距离的基函数。这里以最常用的高斯径向基函数为例。高斯径向基函数的每个分量形式如下：

$$
f_i(s) = \exp\left(-\frac{\|s - \mu_i\|^2}{2\sigma_i^2}\right),
$$

这里μ_i是径向基函数的中心，σ_i是径向基函数的宽度。最终的线性函数形式如下：

$$
g(s; w) = w_1 \exp\left(-\frac{\|s - \mu_1\|^2}{2\sigma_1^2}\right) + w_2 \exp\left(-\frac{\|s - \mu_2\|^2}{2\sigma_2^2}\right) + \cdots + w_l \exp\left(-\frac{\|s - \mu_l\|^2}{2\sigma_l^2}\right)
$$

RBF的另一个重要的应用是RBF网络。这是一种与MLP同一级别的底层网络结构。RBF网络通常由三层组成：输入层、隐含层和输出层。输入层接收输入信号，隐含层是一组径向基函数，激活函数为高斯激活函数，输出层是一个线性输出层。RBF相对于MLP的优点是收敛快，但是缺点是在提取埋藏在高维数据中的特征方面不如MLP。其常用于插值、分类、时间序列预测。

6.2 值函数和策略的参数化

在之前的章节中，值函数和策略都是基于表格的形式来存储的。但是，这种方法在处理高维或者连续时间问题时就束手无策，陷入intractable的境地。因此，需要使用函数来对值函数和策略进行参数化。

6.2.1 值函数的参数化

值函数的参数化可以看成使用观察得到的数据来近似估计一个未知的值函数的过程。总共有三种近似值函数的方法：

- **状态值函数**

$$V(s;w) \approx v^{\pi}(s), \forall s \in S.$$

- **动作值函数**
  - **Type I**（适合连续或离散的动作空间）

$$Q(s,a;w) \approx q^{\pi}(s,a), \forall s \in \mathcal{S}, \forall a \in \mathcal{A}.$$

  - **Type II**（适合离散的动作空间）

$$Q(s,a_i;w) \approx q^{\pi}(s,a_i), \forall s \in S, i = 1, \cdots, m.$$

    这里，m是可能的离散动作数目。

上面的w是值函数的参数。

当获得了参数化的值函数后，就可以通过最优值函数来贪婪地选择动作。下面给出使用上述三种值函数来获得最优动作的方法：

- $V(s;w)$

$$a^* = \arg\max_a \left\{ \sum_{s'} P^a_{ss'} \left(r^a_{ss'} + \gamma V(s';w) \right) \right\},$$

  这里，$P^a_{ss'}$是状态转移概率，$r^a_{ss'}$是reward。这里其实使用了我们在第二单元博客里面讲到过的两种值函数的关系：

$$q^{\pi}(s,a) = \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a)(r^a_{ss'} + \gamma v^{\pi}(s'))$$

  因此这种方法实际上与下面求根据$Q(s,a;w)$来选择动作的方法是一样的。
- $Q(s,a;w)$

$$a^* = \arg\max_a \{Q(s,a;w)\},$$

- $Q(s,a_i;w)$

$$a^* = \arg \max_{\{a_1,a_2,\cdots,a_m\}} \{Q(s,a_1;w),\cdots,Q(s,a_m;w)\}.$$

这里，m是可能的离散动作数目。该式实际上是上一个式子的离散化版本。

因为状态值函数的输入空间（$\mathcal{S}$）比动作值函数的输入空间（$\mathcal{S} \times \mathcal{A}$）要小，所以估计状态值函数所需的数据量要比估计动作值函数所需的数据量要小。然而，因为动作值函数包含了环境的信息。因此当环境模型未知时，我们就只能使用动作值函数来得到最优动作。

6.2.2 策略的参数化

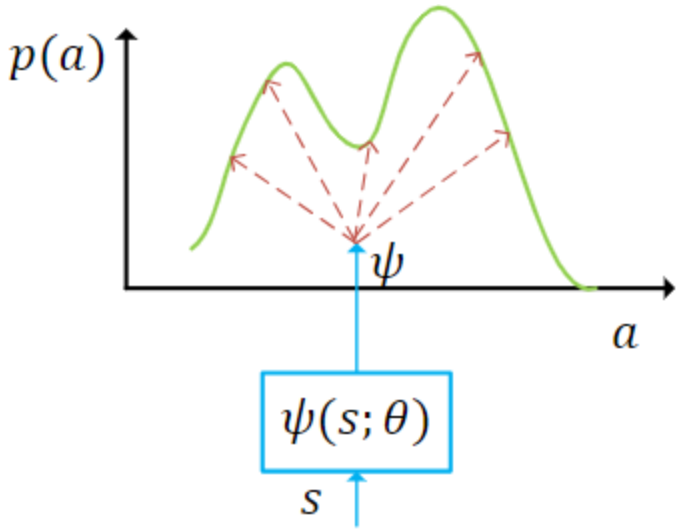
根据我们之前章节的讨论，策略可以分为确定性策略和随机策略。那么，我们对于策略的参数化也可以分为下面三类：

- **确定性策略**

$$a = \pi(s;\theta) \approx \pi(s), \forall s \in S$$

这里与之前使用表格表示的时候的情况类似，每次输入相同的状态s，会得到相同的动作a。

- **随机策略**
  在随机策略这里，通常通过选择一个策略分布来表示策略。其实在训练过程中要得到的其实是我们选定的策略的参数。
  - **Type I**（连续动作空间）



$$p(a;\psi(s,\theta)) \approx \pi(a|s), \forall s \in S, \forall a \in \mathcal{A}$$

对于此种情况，我们通常使用高斯分布（正态分布）来表示策略分布。那么，需要训练的参数就是高斯分布的均值μ和方差σ。

- **Type II**（离散动作空间）

  $$p(a_i|s;\theta) \approx \pi(a_i|s), \forall s \in \mathcal{S}, i = 1, \cdots, m,$$

  注意，这里与Type I不同，这里我们对于每个可能的动作取值a_i都要独自的使用一个函数来拟合（即每个动作individually parameterized）。这样，在当前状态s时，对于每个动作a_i，我们都可以得到一个“概率值”。之所以这里给概率打上引号，是因为这里得到的值并不符合概率每个取值在[0, 1]之间且所有可能取值的和为1的约束。为了得到一个真正的概率值，需要使用深度学习中很常见的归一化操作softmax：

  $$\pi(a_i|s) = \frac{e^{p(a_i|s;\theta)}}{\sum_{j=1}^{m} e^{p(a_j|s;\theta)}}$$

具有更好的探索能力是使用随机策略的一大原因。那么在整个流程中，应该何时使用确定性策略，何时使用随机策略呢？在训练阶段，应该使用随机策略，因为这样可以更好地探索环境；而在评估阶段，应该使用确定性策略，因为最优策略总是一个确定的策略。简略的证明如下。首先，对于任意一个策略π，状态s，都应该有以下不等式成立：

$$\int_a \pi(a|s) q^*(s,a) \mathrm{d}a \leq q^*(s,a^*) = \int_a \delta(a-a^*) q^*(s,a) \mathrm{d}a,$$

其中，$\delta(a-a^*)$是Dirac分布（该分布在$a=a^*$时取值为1，其余地方的概率密度均为0）。上面的式子为什么成立呢？上面的积分形式可能不太直观，可以写成离散的形式帮助理解：

$$\sum_{i=1}^{m} \pi(a_i|s) q^*(s,a_i) \leq q^*(s,a^*),$$

这里，a^*是最优动作（即$a^* = \arg\max_a q^*(s,a)$）。写成离散的形式就可以发现上式是显然的，因为右式（$q^*(s,a^*)$）是所有状态为s的值函数中最大的；而左氏是对于所有状态为是，值小于等于$q^*(s,a^*)$的动作值函数的加权平均。那么，显然对于一堆比$q^*(s,a^*)$小的值，其加权平均是不可能大于$q^*(s,a^*)$的。

有了上式，就可以知道，对于最优的随机策略应该是一个Dirac分布的形式，而这实际上就是一个确定性策略。注意，上述积分不等式的成立还有一个前提，就是reward不受动作分布的影响。因此，比如当我们使用Entropy Regularization时，就会使得reward受到动作分布的影响，它的最优策略也不再是一个确定性策略了。

6.2.3 该选参数化表示还是表格表示?

针对这个问题，我们可以从以下几个方面来考虑：状态空间$\mathcal{S}$和动作空间$\mathcal{A}$连续还是离散？若离散，其元素个数多少；若连续，其维度大小。根据以上要点，我们可以将RL问题分为下面几类。注意，下面的讨论中，我们将离散简记为$\mathcal{D}$，连续简记为$\mathcal{C}$。

- **离散状态空间，离散动作空间**

$\mathcal{S}$ size	$\mathcal{A}$ size	V/Q	Policy
Small	Small	Tab.	Tab.
Large	Large	Appr.	Appr.

  这种情况的一个例子是Grid World问题。比如在一个Grid World里的cleaning robot问题。

- **连续状态空间，离散动作空间**

$\mathcal{S}$ Dim.	$\mathcal{A}$ size	V/Q	Policy
Low	Small	Appr.	Appr.
High	Large	Appr.	Appr.

  在现实世界中，这种情况的一个典型任务是具有机械传动的乘用车的变速控制。其状态，如发动机转速和车速，是连续的，但其动作是离散的，因为机械传动具有有限的挡位。

- **离散状态空间，连续动作空间**

$\mathcal{S}$ size	$\mathcal{A}$ Dim.	V/Q	Policy
Small	Low	V-Tab. Q-Appr.	Tab.
Large	High	Appr.	Appr.

  一个例子是掷硬币时，动作（这里以掷硬币的力度为例）是连续的，而状态（硬币的朝向）是离散的。

- **连续状态空间，连续动作空间**

$\mathcal{S}$ Dim.	$\mathcal{A}$ Dim.	V/Q	Policy
Low	Low	Appr.	Appr.
High	High	Appr.	Appr.

此时必须使用函数近似来表示值函数和策略。

6.3 值函数的近似化（Value Function Approximation）

本节我们主要讨论如何使用带参数的函数近似来近似值函数（即Parameterization）。这个过程实际上可以看成是一个优化问题，即找到一个参数w，使得近似值函数$V(s;w)$与真实值函数$v^{\pi}(s)$的差距最小。让我们以状态值函数的近似为例来讨论这个问题。根据上面的描述，优化目标可以写成如下形式：

$$\min_{w} J(w) = \mathbb{E}_{s\sim d(s)}\{\phi(v^{\pi}(s), V(s;w))\},$$

这里，$d(s)$是加权求和时的权重，也可以看成是状态分布，但是这里需要说明的是$d(s)$不一定是一个状态分布，也可以是人为设置的权重，但是后面会说明是不是必须取为对应的状态分布。显然，这个用于加权的权重$d(s)$是一个很重要的参数，因为它决定了我们如何从目标函数中采样。ϕ是一个距离度量，用于衡量两个值函数之间的差距。

下面我们来讨论model-free的RL问题的值函近似。在model-free的RL问题中，我们不知道环境的动态特性，因此对于值函数的估计主要是通过采样得到的数据来进行的。因此，采样得到的样本的分布就十分重要，因为它决定了对于状态空间哪一部分做更多的强调（即有更多样本从这里获得）。对于**on-policy**的方法，在当前策略π下的的分布就是采样得到的样本的分布，我们也**必须使用这个分布作为$d(s)$**，这样做的好处是可以使得样本的分布与RL Agent在环境中实际的行为相符。而对于**off-policy**的方法，在理论上我们也应该使用target policy的分布作为$d(s)$，但是在实际中，我们通常使用behavior policy的分布作为$d(s)$**。这是因为在计算IS Ratio时的不确定性会带来高方差。

再来聊聊求解这个优化问题的方法。一个常见的方法是使用梯度下降法（Gradient Descent）。更新公式为：

$$w \leftarrow w - \alpha\nabla_{w}J(w),$$

这里，α是学习率，$\nabla_{w}J(w)$是$J(w)$关于w的梯度，被称为Value Gradient。上述方法对于求解小规模问题很有效。但是对于大规模的问题，data-efficiency很低，尤其是我们还要设法平衡exploration和exploitation。

从更广阔的角度来看，Value Function Approximation的作用相当于策略迭代里面的PEV，每次迭代都可以获得对于值函数的更好的估计，从而可以更好地指导策略的更新。

6.3.1 On-policy Value Approximation

根据之前的讨论，On-policy的方法需要使用当前策略的分布作为$d(s)$。那么，我们可以将$J(w)$写成如下形式：

$$\min_w J(w) = \mathbb{E}_{s\sim d_\pi}\left\{\left(v^\pi(s) - V(s;w)\right)^2\right\},$$

这里d_π是在当前策略π下的样本分布，然后我们令距离度量ϕ采用均方误差（Mean Squared Error, MSE）。那么Value Gradient就可以写成如下形式：

$$\nabla_w J(w) \propto -\mathbb{E}_{s\sim d_\pi}\left\{\left(v^\pi(s) - V(s;w)\right)\frac{\partial V(s;w)}{\partial w}\right\}.$$

这种基于MSE的$J(w)$的形式是最常见的，但是也有一些其他的形式，比如采用TD Error作为距离度量的形式。回顾一下n-step TD Error：

$$\delta_t^n(s) = r + \gamma r' + \cdots + \gamma^{n-1}r^{(n-1)} + \gamma^n V\left(s^{(n)};w\right) - V(s;w).$$

那么，我们可以将$J(w)$写成如下形式：

$$J(w) = \mathbb{E}_{s\sim d_\pi(s)}\left\{\left(\delta_t^n(s)\right)^2\right\}.$$

那么，相应的Value Gradient就可以写成如下形式：

$$\nabla_w J(w) \propto \mathbb{E}_\pi\left\{\delta_t^n(s)\left(\gamma^n\nabla V\left(s^{(n)};w\right) - \nabla V(s;w)\right)\right\}.$$

这个形式的one-step版本被称为naive residual-gradient algorithm。理论上，采用这种n-step TD Error的形式的$J(w)$具有很好的收敛性，但是实际上其收敛性不如那些semi-gradient方法（什么是semi-gradient方法会在下面讲述）。这是因为在最小化n-step TD Error的过程中，它的target（指n-step TD Error的前半部分）随着参数的变化会发生变化（因为target部分也要一起计算微分$\gamma^n\nabla V\left(s^{(n)};w\right)$），从而导致可能有多个局部最优解。因此，这种方法的收敛性是不稳定的。

对于Value Gradient的估计的质量极大的影响了训练的效率和稳定性。通常，更长更多的samples会带来更好的估计。但是，相应的计算负担和时长也会增加。因此，需要在计算效率和估计质量之间做一个权衡。

下面将以MC和TD的值函数估计为例来说明如何使用构建Value Gradient。并给出两种Value Gradient的估计方法。

6.3.1.1 True-gradient in MC and Semi-gradient in TD

使用$\nabla_w J(w) \propto -\mathbb{E}_{s\sim d_\pi}\left\{\left(v^\pi(s)-V(s;w)\right)\frac{\partial V(s;w)}{\partial w}\right\}$.来进行优化的一个难点在于，值函数的真实值（target）$v^\pi(s)$是未知的，因此我们需要人为的构建一个target：

- **MC**
  我们令其target如下：

  $$v^\pi(s) \cong \mathbb{E}_\pi\{G_{t:T}|s\},$$

  这里，$G_{t:T}$是从时刻t开始的回报。那么，我们可以得到Value Gradient的估计：

  $$\nabla J_{\mathrm{MC}}(w) = -\mathbb{E}_{s\sim d_\pi}\left\{\left(\mathbb{E}_\pi\{G_{t:T}\}-V(s;w)\right)\frac{\partial V(s;w)}{\partial w}\right\}.$$

  这里得到的Value Gradient的估计是一个True Gradient，与后文的Semi-gradient相对应。在理论上，这种基于MC的估计是一种无偏估计（对于episodic任务）。
- **TD(0)**
  我们令其target如下：

  $$v^\pi(s) \cong \mathbb{E}_\pi\{r+\gamma V(s';w)|s\},$$

  这里，s'是从状态s采取动作a后得到的下一个状态。那么，我们可以得到Value Gradient的估计：

  $$\nabla J_{\mathrm{TD}}(w) = -\mathbb{E}_{s\sim d_\pi}\left\{\left(\mathbb{E}_{\alpha\sim\pi,s'\sim\mathcal{P}}\{r+\gamma V(s';w)\}-V(s;w)\right)\frac{\partial V(s;w)}{\partial w}\right\}.$$

  这种方法为什么被称为Semi-gradient呢？这是因为$V(s')$也是关于w的函数，然而在计算$J_{\mathrm{TD}}(w)$对于w的梯度时，我们并没有考虑$V(s')$对于w的梯度。在实际中，采取这种做法却往往能够带来更好的收敛性和稳定性。还要指出，这样的估计方法在理论上是一个有偏估计，因为其估计target的时候仅仅使用了整个样本序列的一小段（一个），但是其在使用时却仍能稳定的收敛到最小值的一个有界的邻域内。

6.3.1.2 Value Gradient的估计方法

上面只是以MC和TD(0)为例，给出了Value Gradient的形式，下面将给出具体的更新计算方法。主要有两种方法：

- **Stochastic Gradient Descent**
  这是优化问题中最常见的方法。其原始的版本每次只取一个样本来更新，然而为了获得更好的随机性与稳定性，我们通常会使用mini-batch的方法。其更新公式如下：

$$\nabla_w J(w) \propto -\frac{1}{N}\sum_{\mathcal{D}_{\mathrm{SGD}}}\big(R_i - V(s_i;w)\big)\frac{\partial V(s_i;w)}{\partial w}$$
$$\mathcal{D}_{\mathrm{SGD}} = \{(s_i, R_i)\}, i \in \{1{:}N\},$$

这里，$\mathcal{D}_{\mathrm{SGD}}$是一个mini-batch，$N$是mini-batch的大小，$R_i$是我们用来近似$v^{\pi}(s_i)$的target。SGD方法的好处有以下几点：

- “Stochastic”：每次更新只使用一个样本，这样可以使得更新的方向更加随机，从而有助于跳出局部最优解。Samples是打乱的，随机的从数据集中取出的，而不是按照顺序取出的。这样有助于提高估计的质量。
- 不必等到所有数据都收集完毕再更新。可以以一种sample-by-sample的方式来更新，这样可以使得更新的速度更快。

- **Incremental least squares (ILS)**
  这是一种基于Least Squares的方法。其主要适用于我们在使用linear function来进行近似的时候。这种方法与Value Gradient就没有关系了。先来看看这种方法的原始形式，它改写了优化目标$J(w)$为如下形式：

$$\min_{w} J(w) = \sum_{\mathcal{D}}\big(R_t - w^{\mathrm{T}}\cdot F(s_t)\big)^2,$$
$$\text{s.t.}$$
$$D = \{(s_t, R_t)\}, t \in \{1{:}\,T\}$$
$$R_t = \begin{cases} G_{t:T} & \text{for} \quad \mathrm{MC} \\ r + \gamma w^{\mathrm{T}} F(s_{t+1}) & \text{for} \quad \mathrm{TD}(0). \end{cases}$$

  这里$F(s_t)$是特征向量，w是权重向量（忘记的话可以参考6.1节）。这种方法可以通过解析的方式求出最优解而不必借助于梯度下降法。这种方法也具有高度的稳定性、准确性、和可解释性。但是其可扩展性差，难以处理复杂形式的值函数近似。为了解决计算复杂度高的问题，结合Sherman-Morrison formula公式，可以得到Incremental LS 算法，其每次更新的计算复杂度是参数空间维度的平方。

6.3.2 Off-policy Value Approximation

Off-policy的方法可以实现更好的exploitation和exploration的平衡。但是，因为同时使用了两个策略，因此其估计的Value Gradient的形式会有所不同。在理论上，我们应该使用target policy的分布作为$d(s)$，沿用之前的目标函数：

$$\min_{w} J(w) = \mathbb{E}_{s\sim d_{\pi}}\left\{\big(v^{\pi}(s) - V(s;w)\big)^2\right\},$$

但是，因为实际的样本是通过behavior policy采样得到的，因此直接使用上式会导致严重的policy mismatch问题。联系之前博客里讲过的IS Ratio，我们可以使用这个Ratio在两个分布之间做一个转换：

$$\nabla_w J(w) \propto -\mathbb{E}_{s\sim d_b}\left\{\frac{d_\pi(s)}{d_b(s)}\big(v^\pi(s)-V(s;w)\big)\nabla V(s;w)\right\}.$$

这里的比值$\frac{d_\pi(s)}{d_b(s)}$是一种特殊的IS Ratio，被称为stationary distribution ratio。那么我们能否使用上面的式子来直接进行优化呢？答案是不行的。因为，这里的IS Ratio值难以计算。我们在之前博客讲过的那些off-policy的IS Ratio的计算方法都是针对离散型的问题而言的，对于连续型问题，因为两个策略的概率密度函数是不知道的，所以无法直接计算。仿照离散型那里我们进行的处理，一个简单地想法是能否通过足够长的target和behavior的samples来估计这个Ratio。但是，这种做法所需的sample数量非常大，如果所用sample不够的话就会导致低精度问题。

一种解决办法是直接使用behavior policy的分布作为$d(s)$，这样就不需要计算IS Ratio了。这样优化目标改写为：

$$\min_w J(w)=\mathbb{E}_{s\sim d_b}\left\{\big(v^\pi(s)-V(s;w)\big)^2\right\}.$$

那么，相应的Value Gradient就可以写成如下形式：

$$\nabla_w J(w) \propto -\mathbb{E}_{s\sim d_b}\left\{\big(v^\pi(s)-V(s;w)\big)\frac{\partial V(s;w)}{\partial w}\right\}.$$

尽管这样处理在理论上并不完美，但是在实际使用时效果却足够好。因此，在**之后的讨论中，如无特别说明，我们都将使用这种方法来进行Off-policy的Value Function Approximation**。

6.3.2.1 Expected Gradient in MC and TD

本小节我们主要关注如何写出off-policy的setting下的Value Gradient的形式。先来看MC的情形。与on-policy那里的情况类似，我们也需要对于实际的$v^\pi(s)$进行估计。那么，就仍然存在一个问题，就是用于估计的samples是通过behavior policy采样得到的，但是要估计的$v^\pi(s)$是在target policy下的，这就是**policy mismatch**问题。那么，我们可以使用IS Ratio来解决这个问题吗？刚才在构建目标函数时，我们不是刚被一种特殊的IS Ratio，stationary distribution ratio给难住吗？不是刚说过不能用IS Ratio来进行IS Transformation吗？为什么这里又行了？？？这是因为，我们在构建目标函数时，需要估计的那个stationary distribution ratio是在两种策略下的稳态时的状态s分布的比值，为了估计这两个完全不知道的比值，需要使用大量的数据。但是，我们这里需要估计的MC设定下的该比值仅仅是关于一段有着有限长度的trajectory的，因此其计算是可行的（tractable）。那么，我们就可以将target的$v^\pi(s)$写成如下形式：

$$\nu^\pi(s)\cong\mathbb{E}_\pi\{G_{t:T}\}=\mathbb{E}_b\{\rho_{t:T-1}G_{t:T}\}$$
$$\rho_{t:T-1}=\prod_{k=t}^{T-1}\frac{\pi(a_k|s_k)}{b(a_k|s_k)}.$$

这里的$\rho_{t:T-1}$是IS Ratio（详情参考第三单元博客的第3.3.2.2节）。紧接着，就可以得到Value Gradient的估计：

$$\nabla_w J(w) \propto -\mathbb{E}_{s\sim d_b}\left\{\left(\rho_{t:T-1}G_{t:T}-V(s;w)\right)\frac{\partial V(s;w)}{\partial w}\right\}.$$

与之类似的，我们可以得到TD(0)的情况：

$$\nu^{\pi}(s) \cong \mathbb{E}_{a\sim\pi,s'\sim\mathcal{P}}\{r+\gamma V(s';w)\} = \mathbb{E}_{a\sim b,s'\sim\mathcal{P}}\{\rho_{t:t}\big(r+\gamma V(s';w)\big)\},$$
$$\rho_{t:t} = \frac{\pi(a_t|s_t)}{b(a_t|s_t)}.$$

这里的IS Ratio为单步的IS Ratio（参考第三单元博客的第3.3.2.2节）。那么，相应的Value Gradient的估计为：

$$\begin{aligned}\nabla_w J(w)) &\propto -\mathbb{E}_{s\sim d_b}\left\{\left(\mathbb{E}_{a\sim b,s'\sim\mathcal{P}}\{\rho_{t:t}\big(r+\gamma V(s';w)\big)\}-V(s;w)\right)\frac{\partial V(s;w)}{\partial w}\right\}\\ &= -\mathbb{E}_{s\sim d_b,a\sim b,s'\sim\mathcal{P}}\left\{\left(\rho_{t:t}(r+\gamma V(s';w))-V(s;w)\right)\frac{\partial V(s;w)}{\partial w}\right.\\ &= -\mathbb{E}_b\left\{\left(\rho_{t:t}\big(r+\gamma V(s';w)\big)-V(s;w)\right)\frac{\partial V(s;w)}{\partial w}\right\}.\end{aligned}$$

Off-policy下的值函数近似的好处在于我们可以使用历史数据（从不同的behavior policy下采样得到的样本）。但是，有时我们会遇到一种棘手的情况，即behavior policy的分布未知，这种情况下对于IS Ratio的估计会很困难，且会出现很大的梯度的方差。而这种高方差又会导致训练不稳定，引起所谓的deadly triad issue。

6.3.2.2 Off-policy TD中降低方差的方法

因为我们采样所用的策略和实际的目标策略之间并不相同，而两个策略pi和b之间的差距越大，在计算IS Ratio时的不确定性就越大，因此会导致对于value gradient的估计的质量急剧降低。为了解决这个问题，需要先引入下面的等式。

我们首先要证明一个等式：

$$\mathbb{E}_b\left\{(1-\rho_{t:t})V(s;w)\frac{\partial V(s;w)}{\partial w}\right\}=0$$

证明如下：

$$
\begin{aligned}
&\mathbb{E}_b\left\{(1-\rho_{t:t})V(s;w)\frac{\partial V(s;w)}{\partial w}\right\}\\
&=\mathbb{E}_{s\sim d_b}\left\{\mathbb{E}_{a\sim b}\left\{(1-\rho_{t:t})V(s;w)\frac{\partial V(s;w)}{\partial w}\right\}\right\}\\
&=\mathbb{E}_{s\sim d_b}\left\{V(s;w)\frac{\partial V(s;w)}{\partial w}\mathbb{E}_{a\sim b}\{(1-\rho_{t:t})\}\right\}\\
&=\mathbb{E}_{s\sim d_b}\left\{V(s;w)\frac{\partial V(s;w)}{\partial w}\sum_a\left[b(a|s)\left(1-\frac{\pi(a|s)}{b(a|s)}\right)\right]\right\}\\
&=\mathbb{E}_{s\sim d_b}\left\{V(s;w)\frac{\partial V(s;w)}{\partial w}\left(\sum_a b(a|s)-\sum_a\pi(a|s)\right)\right\}\\
&=\mathbb{E}_{s\sim d_b}\left\{V(s;w)\frac{\partial V(s;w)}{\partial w}(1-1)\right\}\\
&=\mathbb{E}_{s\sim d_b}\left\{V(s;w)\frac{\partial V(s;w)}{\partial w}\cdot 0\right\}
\end{aligned}
$$

首先，第一式说明了这里求的期望是在behavior policy下的期望。然后，从第一式到第二式，是把对于策略b求期望进一步展开，把其进一步写成了在$s\sim d_b$和$a\sim b$下的期望。第二式到第三式是因为$V(s;w)\frac{\partial V(s;w)}{\partial w}$这部分与动作a没有关系。那么第三式到第四式就是使用期望和IS Ratio的定义做一个展开。之后几步都容易理解。证毕。

因此，我们就知道了在策略b期望意义下，$V(s;w)\frac{\partial V(s;w)}{\partial w}=\rho_{t:t}V(s;w)\frac{\partial V(s;w)}{\partial w}$。把该式代入到$\nabla_w J(w)=-\mathbb{E}_b\left\{\left(\rho_{t:t}\big(r+\gamma V(s';w)\big)-V(s;w)\right)\frac{\partial V(s;w)}{\partial w}\right\}$.中去，我们得到：

$$
\nabla_w J(w)\propto-\mathbb{E}_b\left\{\rho_{t:t}\big(r+\gamma V(s';w)-V(s;w)\big)\frac{\partial V(s;w)}{\partial w}\right\}.
$$

这个式子与原始的式子相比，在于把$\rho_{t:t}$后面相乘的对象从TD Target（$r+\gamma V(s';w)$）变成了TD Error（$r+\gamma V(s';w)-V(s;w)$）。为什么这样改能提升值函数近似的质量呢？这是因为TD Error要远比TD Target小，这样就算IS Ratio的估计有一定的误差，对其的放大也不会太大。

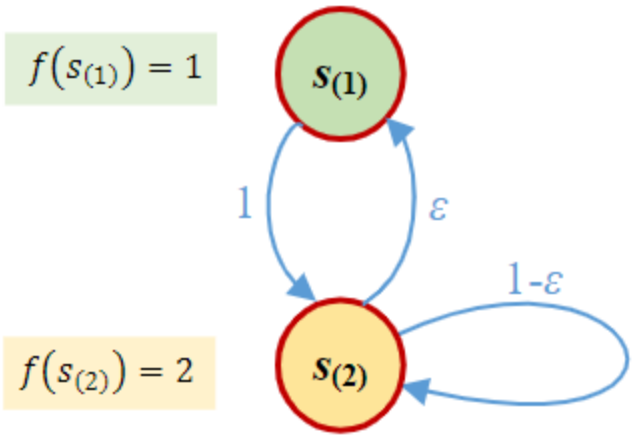
6.3.3 Deadly Triad Issue

RL中的Deadly Triad Issue是指如果一个RL算法同时集齐了下面的三个要素，则会出现训练的不稳定：

- **Function Approximation**：使用函数近似而不是精确的表格表示（Tabular Representation）来表示状态和动作空间的相关信息。
- **Bootstrapping**：在RL中，bootstrapping是指使用历史的估计值来作为target值来估计本轮值，而不依赖于完整的回报。比如TD(0)就是一种bootstrapping方法。

- **Off-policy Learning**：在RL中，off-policy learning是指在训练时使用的策略与在评估时使用的策略不同。这样可以使用历史数据来训练。

当这三者结合在一起的时候，收敛性不能得到保证，训练稳定性被大大地降低。来考虑Bertsekas and Tsitsiklis书中的一个简单的例子。



首先，环境是一个两个状态地马尔可夫链。两个状态分别为$s_{(1)}$和$s_{(2)}$，用来表征状态的特征f定义如下：

$$f(s_{(1)}) = 1, f(s_{(2)}) = 2.$$

这里的f实际上就是6.1节开头讲过的F。只不过这里不管是F还是状态s都只有1维，因此这里的f就退化一个标量了。

状态转移概率使用如下的状态转移矩阵表示：

$$H = \begin{bmatrix} 0 & \varepsilon \\ 1 & 1-\varepsilon \end{bmatrix},$$

这里的ε是一个很小的数。根据状态转移概率，我们可以知道稳定的时候$s_{(1)}$和$s_{(2)}$出现的概率。设$s_{(1)}$稳定时的出现概率为p，$s_{(2)}$稳定时的出现概率为1-p。那么结合状态转移矩阵，可得：

$$\begin{cases} 1 \to 1 : 0 \\ 1 \to 2 : p \times 1 = p \\ 2 \to 1 : (1-p) \times \varepsilon = \varepsilon(1-p) \\ 2 \to 2 : (1-p) \times (1-\varepsilon) = (1-\varepsilon)(1-p) \end{cases}$$

那么。根据稳定时的状态值可以列出下面的等式：

$$\begin{cases} p = 0 + \varepsilon(1-p) & \text{（根据状态1的稳定概率不变列恒等式）} \\ 1-p = p + (1-\varepsilon)(1-p) & \text{（根据状态2的稳定概率不变列恒等式）} \end{cases}$$

解的结果是：

$$p = \frac{\varepsilon}{1+\varepsilon}.$$

那么，我们可以得到稳态的状态分布（Stationary State Distribution，SSD）：

$$d(s) = \left[\frac{\varepsilon}{1+\varepsilon}, \frac{1}{1+\varepsilon}\right]^{\mathrm{T}}.$$

进一步，我们假设**不同的状态转移之间的reward的值都是0**。折扣因子γ是一个(0,1)之间的数。那么根据值函数的定义易知，**真实的值函数总是0**。

现在，我们规定采用一个线性的函数来近似值函数：

$$V(s;w) = w \cdot f(s), w \in \mathbb{R}$$

这里需要拟合的参数为w。那么，可以推导出下面的更新公式：

$$\begin{aligned}
w_{t+1} &= w_t + \alpha \mathbb{E}\{f(s_t)\Big(r + \gamma V(s_{t+1};w_t) - V(s_t;w_t)\Big)\} \\
&= w_t + \alpha \mathbb{E}\{f(s_t)\Big(r + \gamma f(s_{t+1})w_t - f(s_t)w_t\Big)\} \\
&= w_t + \alpha \mathbb{E}\{f(s_t)\Big(0 + \gamma f(s_{t+1})w_t - f(s_t)w_t\Big)\} \\
&= w_t + \alpha \mathbb{E}\{f(s_t)w_t\Big(\gamma f(s_{t+1}) - f(s_t)\Big)\}
\end{aligned}$$

那么，接下来怎么化简这个式子呢？我们需要讨论当前状态s_t和转移状态s_{t+1}分别是什么状态。假设根据behavior policy的采样（这里有behavior policy的概念是因为之前在讨论deadly triad issue时提到过其形成的条件之一是off-policy learning），我们有当前状态为$s_{(1)}$、$s_{(2)}$的概率分别为$Pr\big(s_t = s_{(1)}\big)$和$Pr\big(s_t = s_{(2)}\big)$（注意，这里的两个概率与上面我们给出的稳态分布不同，这里是behavior policy下采样得到状态$s_{(1)}$和$s_{(2)}$的概率）。并进一步，根据之前的讨论，当前状态为$s_{(1)}$时，下一个状态为$s_{(1)}$和$s_{(2)}$的概率分别为0和1；当前状态为$s_{(2)}$时，下一个状态为$s_{(1)}$和$s_{(2)}$的概率分别为ε和$1-\varepsilon$。那么，我们可以得到对于上式的如下化简：

$$\begin{aligned}
w_{t+1} &= w_t + \alpha \mathbb{E}\{f(s_t)w_t\Big(\gamma f(s_{t+1}) - f(s_t)\Big)\} \\
&= w_t + \alpha \mathrm{Pr}\big(s_t = s_{(1)}\big)(2\gamma - 1)w_t \\
&+ \alpha \mathrm{Pr}\big(s_t = s_{(2)}\big) f(s_t = s_{(2)}) w_t \big(\underbrace{\varepsilon(2\gamma - 4)}_{s_{t+1}=s_{(1)}} + \underbrace{(1-\varepsilon)(4\gamma - 4)}_{s_{t+1}=s_{(2)}}\big),
\end{aligned}$$

这里，我们令$\alpha = 0.1$，$Pr\big(s_t = s_{(1)}\big) = 0.5$，$Pr\big(s_t = s_{(2)}\big) = 0.5$。带入化简可得：

$$\frac{w_{t+1}}{w_t} = 1 + \frac{1}{20}\big(6\gamma - 5 + O(\varepsilon)\big),$$

这里，当$\gamma > \frac{5}{6}$时，比值$\frac{w_{t+1}}{w_t} > 1$，即权重w会发散。这就是Deadly Triad Issue的一个例子。

下面我们来说明，当deadly triad issue中的三者中的任意一项被去掉之后，无论折扣因子γ在(0, 1)之间怎么选取，权重w最终都会收敛。

- **去掉Off-policy Learning**：则经推导可得下述更新公式：

$$\frac{w_{t+1}}{w_t} = 1 + \frac{2}{5}(\gamma - 1) + O(\varepsilon) < 1$$

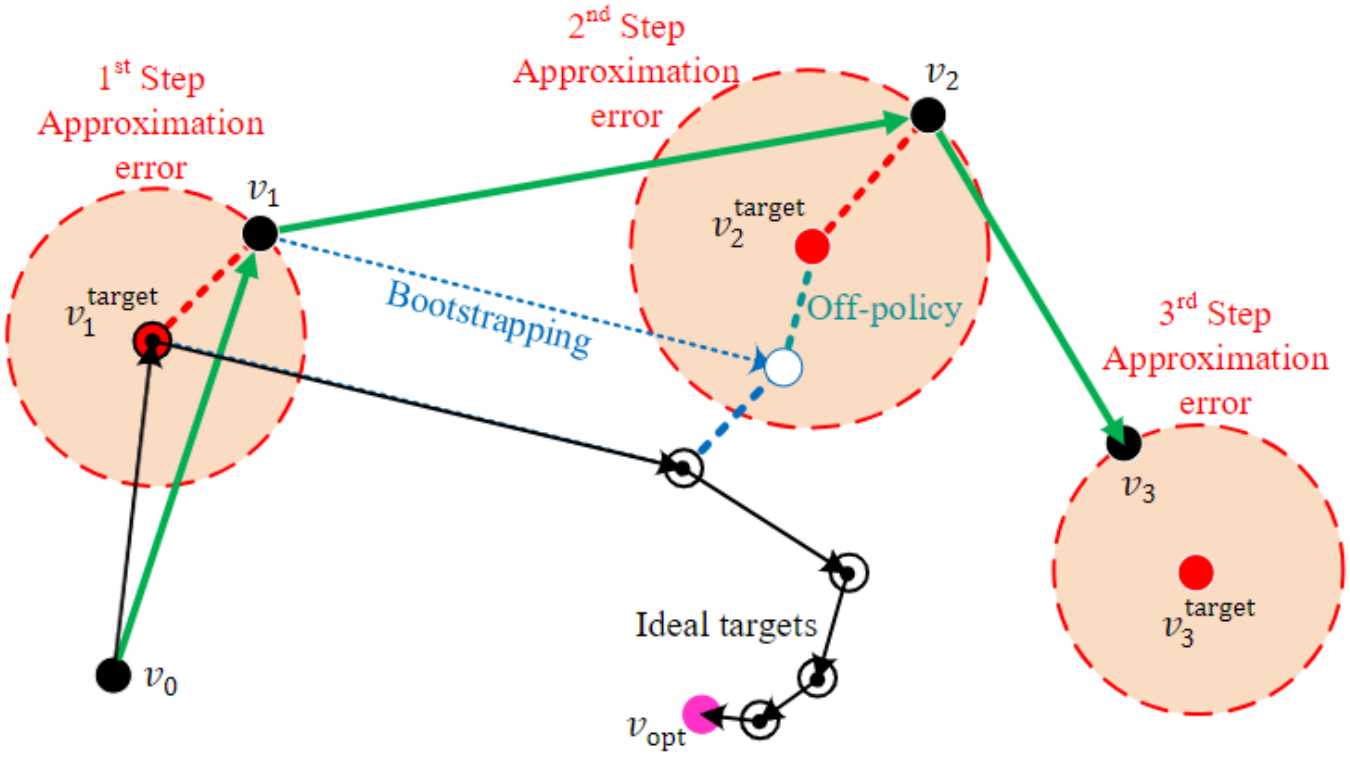
- **去掉bootstrapping**：则经推导可得下述更新公式：

$$\frac{w_{t+1}}{w_t} = 1 - \frac{1}{20}\big(5 + O(\varepsilon)\big) < 1$$

- **去掉Function Approximation**：则经推导可得下述更新公式：

$$\begin{bmatrix} V_{t+1}\big(s_{(1)}\big) \\ V_{t+1}\big(s_{(2)}\big) \end{bmatrix} = B \cdot \begin{bmatrix} V_t\big(s_{(1)}\big) \\ V_t\big(s_{(2)}\big) \end{bmatrix},$$
$$B = \begin{bmatrix} 1 - \frac{1}{20} & \frac{\gamma}{20} \\ \frac{\gamma}{20}\varepsilon & 1 + \frac{\gamma(1-\varepsilon)-1}{20} \end{bmatrix}, \rho(B) < 1.$$

可见对于这个简单问题，此时三种情况均是收敛的。那么，为什么当这三者结合在一起时就会导致算法发散呢？抛开之前简单的例子，让我们从一个更大的视角来看这个问题。下图最能说明问题。



从上图中，我们可以总结得出deadly triad issue中的三者中的每一个因素都会导致算法的累计误差增大，从而最终发散。首先，使用函数近似来表示值函数本身就会存在一定的拟合/建模误差，不可能是与实际的函数百分百一致的。因此，近似的值函数很难达到它的实际的目标值，而是会发生一定量的偏差，尤其是当处理分布外的数据时（out of Distribution，OOD）；其次，bootstrapping使用了参数化的近似函数来近似值函数，作为target，那么上一步的误差又会传给下一步；最后，对于off-policy learning，由于要使用IS Ratio来进行转换，这个Ratio的估计又是很不准确的，而且当前的target policy和behavior policy之间的差距越大，这个Ratio的估计就会越不准确。Policy mismatch问题会引起对于误差的急剧放大。因此，这三者结合在一起，会导致一个是算法的误差越来越大的循环，最终发散。

在先进的RL算法中，为了解决这个问题，通常会采用一些技巧来减小这个误差的传播。常用的方法有：

- **平衡用来近似值函数的函数的准确度与泛化能力**：研究表明，当采取泛化能力更强的函数来近似值函数时，无界的发散会消失。比如使用人工神经网络来近似值函数。
- **在bootstrapping中引入更多的真实reward值**：比如使用multi-step TD就比使用单步TD更加稳定。
- **限制IS Ratio的取值范围到一个合理的（reasonable）区间内**：这种方法就是裁剪技术（clipping technique）。这样可以限制目标估计中的方差增长。

6.4 策略的近似化（Policy Approximation）

在之前的学习中，我们已经知道了从是否使用了Bellman最优性条件可以将强化学习算法分为两类：Indirect RL和Direct RL。其中，Indirect RL是基于值函数的，而Direct RL是基于策略的。Indirect RL分为两步：PEV（求得对于值函数更好的近似）和PIM（根据值函数求得更好的策略），注意这里说的两步走的方法既包含了policy iteration也包含了value iteration，因为value iteration可以看成进行了一轮PEV和PIM。对于Indirect RL来说，在寻找更好的策略时，**值函数是固定的，不随着策略参数的变化而变化**。而Direct RL则是把值函数当成当前策略参数的一个函数，也就是说值函数在更新策略时会跟着改变。**我们在这章仅仅讨论Indirect RL中的Policy Approximation**。

回顾我们在使用表格表示（Tabular Representation）时，在PIM步是怎么得到一个更好的策略的：

$$\pi^g(s) = \arg\max_a \{Q(s,a)\}, \forall s \in \mathcal{S}.$$

也就是说，在获得了对于值函数的最新估计后，我们会采用一种贪心搜索的策略来获得一个更好的策略。但是，这种方法在我们使用函数来近似策略之后就会不可行的。因为上述搜索策略本质上是一种枚举的方法，但是在函数近似的情况下，我们无法枚举所有的动作。因此，我们需要将此处的PIM过程建模为一种随机优化过程：

$$\theta = \arg\max_\theta \{J(\theta)\}$$
$$\text{s.t.}$$

$$J(\theta) = \mathbb{E}_{s\sim d(s)}\{Q(s,a)\},$$

此处，$J(\theta)$是我们的目标函数，可以堪称是一种衡量策略好坏的指标。为了简化起见，我们先假设此处的策略是一种确定性策略，即$a = \pi(s;\theta)$（其实这里仅仅是一种为了简化说明的权宜之计，后面我们在实际推导时会使用非确定性策略）。那么，我们的目标函数可以写成如下形式：

$$\begin{gathered}\theta = \arg\max_{\theta}\{J(\theta)\} \\ \text{s.t.} \\ J(\theta) = \mathbb{E}_{s\sim d(s)}\{Q(s,\pi_\theta(s))\},\end{gathered}$$

这里的$\pi_\theta(s)$是我们的参数化的策略函数，而$d(s)$是某种用于加权的分布，显然这个分布的最终得到的策略的好坏是至关重要的，因此不能随便选取，通常取为收集到的数据的分布。那么我们就可以使用若干数值优化方法来求解这个问题。一种常见的做法是使用SGD：

$$\theta \leftarrow \theta + \beta\nabla_\theta J(\theta),$$

这里的$\nabla_\theta J(\theta) = \partial J(\theta)/\partial\theta$被称为**indirect policy gradient**，这是为了与Direct RL中的policy gradient做区分。Indirect policy gradient的好处如下：

- 结构上的简洁性
- 值函数的更新和策略的更新是分开的，值函数Q(s, a)不随着策略参数θ的变化而变化。换句话说，这里的值函数是policy-independent的，这样可以使得算法更加稳定。与之形成对比的是，Direct RL中的值函数是policy-dependent的，需要使用一系列级联的递归以及对于初始分布的假设才能进行。

6.4.1 Indirect On-Policy Gradient

这里，我们使用随机策略来改写上述有关PIM的优化问题：

$$\begin{gathered}\theta = \arg\max_{\theta}\{J(\theta)\} \\ \text{s.t.} \\ J(\theta) = \mathbb{E}_{s\sim d(s)}\left\{\sum_a \pi_\theta(a|s)Q(s,a)\right\}.\end{gathered}$$

这里，如果使用目标策略下的分布$d_\pi(s)$来代替之前的分布$d(s)$。但是，与之前的值函数近似不同，这里对于策略的近似需要处理一个顺序的问题：**先求导还是先加权**？因为这里的目标分布$d_\pi(s)$也是策略参数的函数，因此先加权再求导的话会使得求梯度变得极为复杂。因此，我们先求导再加权，即我们认为求导的时候**目标分布是固定的一组参数，与策略参数无关**。这样做的好处是可以得到更为简便的梯度形式。

6.4.1.1 使用动作值函数的Indirect Policy Gradient

注意到，根据我们之前的讨论，**在进行PIM步骤时，$Q(s,a)$和分布$d(s)$是固定的，与策略参数θ无关**。因此，$J(\theta)$关于θ的梯度可以写成如下形式：

$$\nabla_\theta J(\theta) = \sum_s d(s) \sum_a \nabla_\theta \pi_\theta(a|s) Q(s,a).$$

这里我们先求了导。接着我们再使用当前的策略下的分布来进行加权：

$$\nabla_\theta J(\theta) \quad = \sum_s d_\pi(s) \sum_a \nabla_\theta \pi_\theta(a|s) Q(s,a)$$

紧接着，我们要使用一种被称为**log-derivative trick**的技巧来进一步简化这个梯度的形式。这个技巧如下：

$$\nabla_\theta \log p(x;\theta) = \frac{\nabla_\theta p(x;\theta)}{p(x;\theta)}$$

那么，具体代入到$\nabla_\theta \pi_\theta(a|s)$可得：

$$\begin{aligned}\nabla_\theta \pi_\theta(a|s) &= \pi_\theta(a|s) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \\ &= \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s).\end{aligned}$$

这样，我们就可以把梯度写成如下形式：

$$\begin{aligned}\nabla_\theta J(\theta) &= \sum_s d_\pi(s) \sum_a \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s) Q(s,a) \\ &= \mathbb{E}_{s\sim d_\pi, a\sim\pi}\{\nabla_\theta \log \pi_\theta(a|s) Q(s,a)\},\end{aligned}$$

注意到，最后的indirect policy gradient的形式就是一个期望的形式了。写成这种形式有助于采用sample-based的方法来进行估计。

6.4.1.2 使用状态值函数的Indirect Policy Gradient

我们能不能采用状态值函数来进行策略的近似呢？当然可以！回顾我们在第二单元博客里讲过的两种值函数之间的关系：

$$Q(s,a) \cong \mathbb{E}_{s'\sim\mathcal{P}}\{r + \gamma V(s')\}.$$

把该式代入上面的indirect policy gradient中，我们可以得到如下形式：

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi\{\nabla_\theta \log \pi_\theta(a|s)\big(r + \gamma V(s')\big)\}.$$

采用这个式子的好处是它比采用动作值函数时的计算复杂度要低，因为式子中只有一个随机变量。随机变量个数的减少还有助于减小对于policy gradient的估计的方差，因为可以再使用同样规模的样本数时取得更高的估计精度。

6.4.2 Indirect Off-Policy Gradient

6.4.2.1 使用动作值函数的Indirect Off-Policy Gradient

推导的起点还是下面这个式子：

$$\nabla_\theta J(\theta) = \sum_s d(s) \sum_a \nabla_\theta \pi_\theta(a|s) Q(s,a).$$

在off-policy的情况下，我们需要选取behavior策略下的分布作为加权的分布，即$d(s) = d_b(s)$。那么，我们可以得到如下形式的梯度：

$$\nabla_\theta J(\theta) = \sum_s d_b(s) \sum_a \nabla_\theta \pi_\theta(a|s) Q(s,a)$$

那么，我们按照之前学过的知识回想，这里需不需要引入IS Ratio呢？答案是不需要。为什么呢？先来回顾一下之前引入IS Ratio的理由。之前引入IS Ratio是因为需要使用从一个分布下得到的样本来估计另一个分布下的期望。那么，两个分布之间最大的不同是什么？就是他们的状态分布，即求期望时加的权不一样。但是回顾我们上面在推导indirect off-policy gradient时的式子，可以发现，第一个加权因子是$d(s)$本来就是可以在一定情况下随便选取的，而第二个加权因子是一个条件概率$\pi_\theta(a|s)$，这个概率是可以通过当前概率计算得到的。因此，对于从behavior policy下采样得到的样本(s_b, a_b, s_b')，我们只使用s_b，而a_b可以使用当前策略产生的a_π来代替，同时s_b'因为我们使用了动作值函数而显得没有必要了。所以，对于off-policy的情况，我们不需要引入IS Ratio。因此，我们可以得到如下形式的梯度：

$$\begin{aligned}\nabla_\theta J(\theta) &= \sum_s d_b(s) \sum_a \nabla_\theta \pi_\theta(a|s) Q(s,a) \\ &= \mathbb{E}_{s\sim d_b, a\sim\pi}\{\nabla_\theta \log \pi_\theta(a|s) Q(s,a)\}.\end{aligned}$$

6.4.2.2 使用状态值函数的Indirect Off-Policy Gradient

与动作值函数的情况不同，当我们想要使用状态值函数来进行策略的近似时，我们需要引入IS Ratio。这是因为在使用状态值函数计算indirect off-policy gradient时，还需要根据当前状态s和动作a来得到下一个状态s'，但是，因为当前状态s是从behavior policy下采样得到的（即s_b），而下一个状态s'是从目标策略下采样得到的（即s_π），因此存在一个policy mismatch问题。为了解决这个问题，我们需要引入IS Ratio。那么，我们可以得到如下形式的梯度：

$$\begin{aligned}\nabla_\theta J(\theta) &= \sum_s d_b(s) \sum_a b(a|s) \frac{\pi_\theta(a|s)}{b(a|s)} \nabla_\theta \log \pi_\theta(a|s) \sum_{s'} p(s'|s,a)\big(r+\gamma V(s')\big) \\ &= \mathbb{E}_{s\sim d_b, a\sim b, s'\sim p}\left\{ \frac{\pi_\theta(a|s)}{b(a|s)} \nabla_\theta \log \pi_\theta(a|s)\big(r+\gamma V(s')\big) \right\}.\end{aligned}$$

与上面动作值函数的indirect off-policy gradient相比，这里的版本容易在计算时引入高方差，因为近似误差、IS Ratio的不连续性、bootstrapping、很少被behavior policy采样到的却可能对于target policy很重要的状态等因素。因此，在off-policy的情况下，**我们通常会使用动作值函数来进行策略的近似**。

6.4.3 使用策略近似前提下的PIM优化过程的其它构建方式

在前文中我们仿照表格表示时的贪婪搜索，构建了一个使用策略近似时的PIM随机优化问题：

$$\begin{gathered}\theta = \arg\max_\theta \{J(\theta)\} \\ \text{s.t.} \\ J(\theta) = \mathbb{E}_{s\sim d(s)}\{Q(s,a)\},\end{gathered}$$

那么，这种构建方式是唯一的吗？回想我们在第五单元第5.5节讲过的“什么是一个更好的策略”，这里的答案当然是不唯一的。我们可以使用其它方式来构建PIM阶段的优化问题。

第一种方法是在哟i话目标中加上一个policy entropy项，即使用Entropy Regularization。这样可以更好地平衡exploration和exploitation，让模型拥有更多的探索能力。第二种方法是使用基于期望的“更好的策略”的定义。这样可以保证PIM过程的单调性以及防止快速的策略改变。

6.4.3.1 使用Entropy Regularization的PIM

与第五单元第5.5节讲的一样，这里我们引入Entropy Regularization项后可以构建出一个等价于ϵ-greedy的策略。将第五单元那里的讨论扩展到连续状态空间（注意，这里并没有说动作空间也是连续的。实际上动作空间可以是连续的或离散的，下面会说明）中可得下面的式子：

$$\begin{gathered}\theta = \arg\max_\theta \{J(\theta)\} \\ \text{s.t.} \\ J(\theta) = \mathbb{E}_{s\sim d_\pi}\{\pi_\theta(a^*|s) + \lambda \cdot \mathcal{H}\big(\pi_\theta(\cdot|s)\big)\} \\ a^* = \arg\max_a Q(s,a), \\ \mathcal{H}\big(\pi_\theta(\cdot|s)\big) = -\int_a \pi_\theta(a|s)\log\pi_\theta(a|s)\mathrm{d}a,\end{gathered}$$

这里的λ是一个超参数，用于平衡解的最优性和探索能力。这里我们在每次求解关于θ的最优化问题时，都要重新计算a^*。**注意，这里如果动作空间是离散的，那么可以使用枚举的方法来获得a^*；如果动作空间是连续的，则要在PEV步更新获得最新的动作值函数估计值之后，使用数值优化方法来求解a^***。可以

看出，构建的$J(\theta)$包含两项，第一项用于最大化选择greedy action的概率，第二项用于保证学得的策略有一定程度的随机性。

为了得到indirect policy gradient的解析形式，我们需要实例化一种参数化的策略形式。不失普遍性地，我们可以选择一个高斯分布来表示策略：

$$\pi_\theta \sim \mathcal{N}(\mu(\theta), \mathcal{K}(\theta))$$

这样，我们可以得到如下形式的indirect policy gradient：

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim d_\pi}\left\{\nabla_\theta \pi_\theta(a^*|s) + \lambda \cdot \nabla_\theta \log\det\left(\mathcal{K}(\theta)\right)\right\}.$$

使用Entropy Regularization的好处是可以鼓励RL Agent去探索更多的状态空间，从而可以更好地平衡exploration和exploitation，实现更好的sample efficiency。但是这样也会牺牲一部分的策略的最优性。一种可能的解决办法是使用可变的λ，即在训练的前期使用较大的λ，在训练的后期逐渐减小λ，这样可以在训练的前期更多地探索状态空间，而在训练的后期更多地利用已有的经验。

6.4.3.2 使用基于期望的“更好的策略”的定义的PIM

这里我们要用到第五单元博客第5.5.2节讲过的“更好的策略”的第二种定义。先来回顾一下那里是怎样定义PIM优化问题的：

$$\begin{aligned}
&\pi_{k+1} = \arg\max_\pi \{\delta - \rho(\pi, \pi_k)\},\\
&\text{s.t.}\\
&\mathbb{E}_{s\sim d(s)}\left\{\sum \pi(a|s) q^{\pi_k}(s,a)\right\} = \delta + \mathbb{E}_{s\sim d(s)}\left\{v^{\pi_k}(s)\right\}\\
&\delta \geq 0.
\end{aligned}$$

注意到，这里的$\rho(\pi, \pi_k)$是用来衡量两个策略之间的相似性的。针对我们这里的情况，我们将$\rho(\pi, \pi_k)$选取为KL散度：

$$D_{\mathrm{KL}}(\pi_\theta, \pi_{\mathrm{old}}) \overset{\mathrm{def}}{=} \int \pi_\theta(a|s) \log \frac{\pi_\theta(a|s)}{\pi_{\mathrm{old}}(a|s)} \mathrm{d}a,$$

那么，我们的优化问题就可以构造如下：

$$\begin{aligned}
&\theta = \arg\max_\theta \{\delta - \rho \cdot \mathbb{E}_{s\sim d(s)} D_{\mathrm{KL}}(\pi_\theta, \pi_{\mathrm{old}})\}\\
&\text{s.t.}\\
&\mathbb{E}_{s\sim d(s), a\sim\pi_\theta}\{Q^{\pi_{\mathrm{old}}}(s,a)\} = \delta + \mathbb{E}_{s\sim d(s), a\sim\pi_{\mathrm{old}}}\{Q^{\pi_{\mathrm{old}}}(s,a)\},
\end{aligned}$$

注意，上式没有写错，左右两边的动作值函数的上标都是π_{old}，原因参见第五单元5.5节以及之前的博客。接下来，我们将δ改写为两个期望相减：

$$J(\theta) = \mathbb{E}_{s\sim d(s),a\sim\pi_\theta}\{Q^{\pi_{\text{old}}}(s,a)\} - \mathbb{E}_{s\sim d(s),a\sim\pi_{\text{old}}}\{Q^{\pi_{\text{old}}}(s,a)\} \\ - \rho\cdot\mathbb{E}_{s\sim d(s)}D_{\text{KL}}(\pi_\theta,\pi_{\text{old}})$$

接下来，我们对于这个式子求导。注意到第二项与策略参数θ无关，因此求完导之后的形式如下：

$$\nabla_\theta J = \mathbb{E}_{s\sim d(s)}\left\{\sum_a \nabla_\theta\pi_\theta(\alpha|s)Q^{\pi_{\text{old}}}(s,\alpha)\right\} - \rho\mathbb{E}_{s\sim d(s)}\{\nabla_\theta D_{\text{KL}}(\pi_\theta,\pi_{\text{old}})\}.$$

但是，注意到我们此时手中的数据都是来源于上一轮的老策略π_{old}，直接使用上面的梯度来更新存在一个分布mismatch的问题，因此需要做一步转换：

$$\begin{aligned}\nabla_\theta J &\approx \mathbb{E}_{s\sim d_{\pi_{\text{old}}}}\left\{\sum_a \pi_{\text{old}}(a|s)\frac{\nabla_\theta\pi_\theta(a|s)}{\pi_{\text{old}}(a|s)}Q^{\pi_{\text{old}}}(s,a)\right\} - \rho\mathbb{E}_{s\sim d_{\pi_{\text{old}}}}\{\nabla_\theta D_{\text{KL}}(\pi_\theta,\pi_{\text{old}})\} \\ &= \mathbb{E}_{s\sim d_{\pi_{\text{old}}},a\sim\pi_{\text{old}}}\left\{\frac{\nabla_\theta\pi_\theta(a|s)}{\pi_{\text{old}}(a|s)}Q^{\pi_{\text{old}}}(s,a)\right\} - \rho\mathbb{E}_{s\sim d_{\pi_{\text{old}}}}\{\nabla_\theta D_{\text{KL}}(\pi_\theta,\pi_{\text{old}})\}\end{aligned}$$

为了化简上述式子，我们将策略实例化为一个高斯分布。那么，上面梯度的解析解形式可以写成：

$$\nabla_\theta J = \mathbb{E}_{\pi_{\text{old}}}\left\{\frac{\nabla_\theta\pi_\theta(a|s)}{\pi_{\text{old}}(a|s)}Q^{\pi_{\text{old}}}(s,a) + \rho\nabla_\theta\left[\log\det(\mathcal{K}(\theta)) - \text{tr}\left(\mathcal{K}_{\text{old}}^{-1}\mathcal{K}(\theta)\right)\right.\right. \\ \left.\left. - \left(\mu(\theta)-\mu_{\text{old}}\right)^T\mathcal{K}_{\text{old}}^{-1}(\mu(\theta)-\mu_{\text{old}})\right]\right\},$$

这里的μ_{old}和$\mathcal{K}_{\text{old}}$是上一轮策略的均值和协方差矩阵。

使用KL散度的好处在于可以避免相邻两次更新时的参数剧烈改变，从而有助于稳定训练过程。

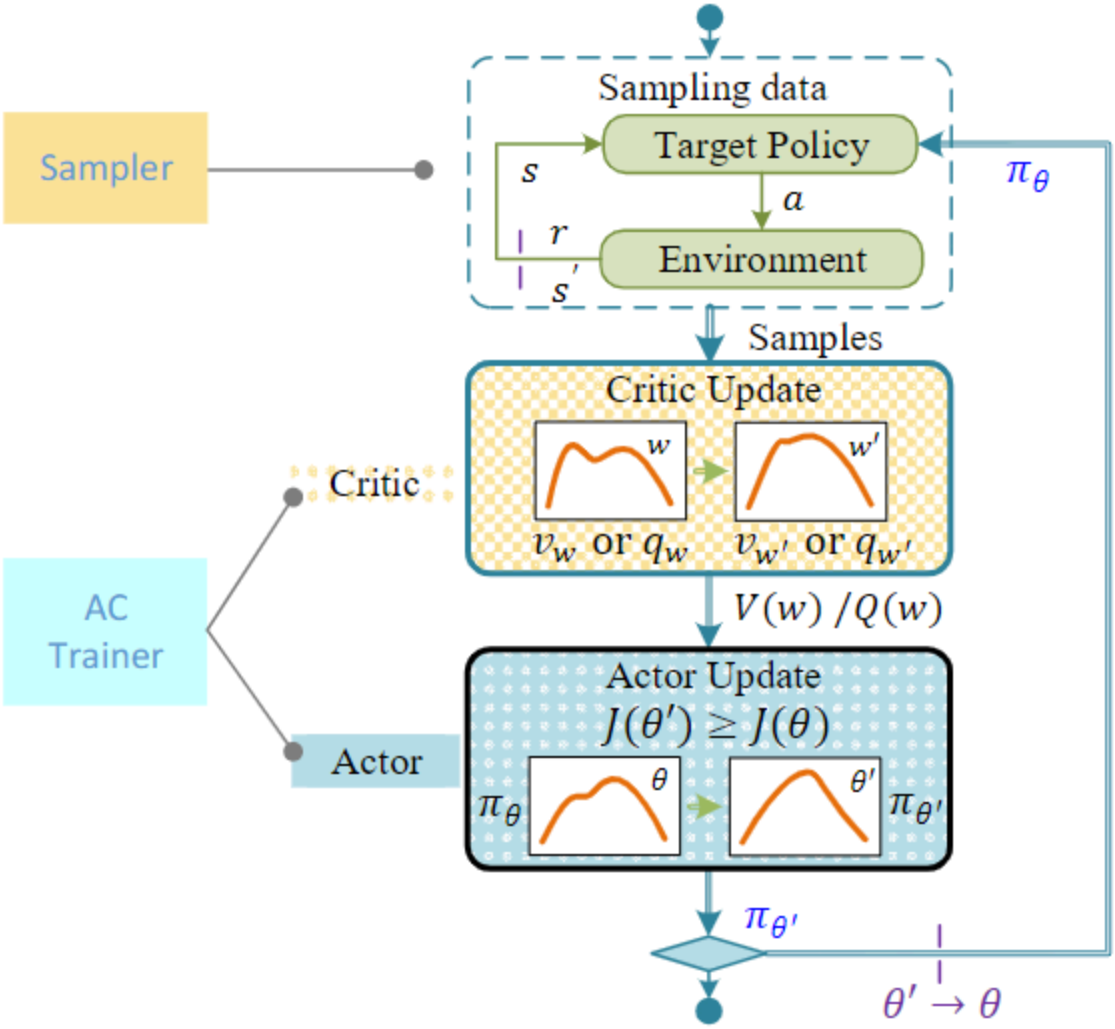
注意到，最终我们得出的梯度式子的形式与TRPO算法的梯度形式十分相似，甚至与TRPO的扩展算法PPO的一种变体的梯度式子一模一样。但是二者的出发点是大相径庭的。此处的式子的推导在于定义了什么是一个更好的策略之后，使用KL散度作为惩罚项来求解一个无约束优化问题；而TRPO的梯度公式则来自minorize-
maximization optimization问题。

6.5 Actor-Critic算法

标准的Indirect RL包含以下两步（不管是策略迭代还是值迭代）：Policy Evaluation（PEV）和Policy Improvement（PIM）。在标准的Indirect RL中，这两步都是“完整的”，即PEV会经过无穷步的迭代以及PIM会采用greedy来获取最优策略。当我们把表格表示的Indirect RL使用函数近似之后，就得到了著名的Actor-Critic算法。Actor-Critic算法通常包含了函数近似以及基于梯度的更新算法。

Actor-Critic算法包含两个部分，一个是Actor，它负责控制agent怎么和环境交互，相当于之前Indirect RL中的PIM；另一个是Critic，它负责评估Actor的策略，相当于之前Indirect RL中的PEV。类似于之前使用表格表示时的情况，用上函数近似后，actor和critic两部分也需要进行无穷此梯度更新才能被称之为“完整的”。但是，这样做会带来很大的计算开销，因此实际的Actor-Critic算法通常只会进行有限步骤的更新。但是，这个有限的迭代次数一定要仔细选取，因为如果迭代次数太少，由于Policy Evaluation的不准确性以及Policy Improvement实际并没有达到最优而致使算法出现不稳定以至于发散。

Actor-Critic算法的框架如下所示：



从上图也可看出在本单元中我们讨论的Actor-Critic算法都是off-policy的。上图共有三个重要组件：Sampler负责根据target策略与环境交互得到sample；Critic负责更新用于近似值函数的函数的参数来对于值函数进行更好的估计；Actor负责更新用于近似策略的函数的参数来对于策略进行更好的近似。

6.5.1 Actor-Critic算法的分类与概述

根据我们为Critc选择的值函数是动作值函数还是状态值函数以及我们为Actor选择的策略是确定性策略还是随机策略，我们可以将Actor-Critic算法分为四类：

	Action-value	State-value
Deterministic policy	$J_{\text{Critic}} = \mathbb{E}_{s,a}\left\{\left(q - Q(w)\right)^2\right\}$	$J_{\text{Critic}} = \mathbb{E}_s\left\{\left(v - V(w)\right)^2\right\}$
	$J_{\text{Actor}} = \mathbb{E}_s\{Q(s, \pi_\theta(s); w)\}$	$J_{\text{Actor}} = \mathbb{E}_{s,s'}\{r + \gamma V(s'; w)\}$
Stochastic policy	$J_{\text{Critic}} = \mathbb{E}_{s,a}\left\{\left(q - Q(w)\right)^2\right\}$	$J_{\text{Critic}} = \mathbb{E}_s\left\{\left(v - V(w)\right)^2\right\}$
	$J_{\text{Actor}} = \mathbb{E}_{s,a}\{Q(s, a; w)\}$	$J_{\text{Actor}} = \mathbb{E}_{s,a,s'}\{r + \gamma V(s'; w)\}$

观看上表我们会发现其中采用确定性策略且采用状态值函数的那一类被用斜线划掉了。这是因为这类方法在model-free（即不知道环境模型或者说状态转移概率$\mathcal{P}(s'|s,a)$）的情况下是不存在的。这是因为当使用状态值函数的时候，需要用到状态转移概率（环境模型），但此时的环境模型是未知的。

6.5.2 On-Policy Actor-Critic with Action-Value Function

6.5.2.1 Deterministic Policy版本

回顾我们在第6.4.1.1节讲过的indirect on-policy gradient的形式：

$$\nabla_\theta J(\theta) = \sum_s d(s) \sum_a \nabla_\theta \pi_\theta(a|s) Q(s,a).$$

因为这里的策略是确定性的，因此第二重对于动作的求和就直接去掉了：

$$\nabla_\theta J(\theta) = \sum_s d(s) \nabla_\theta \pi_\theta(a|s) Q(s,a).$$

因此，可进一步的写成下面的期望形式：

$$\nabla_\theta J_{\text{Actor}} = \mathbb{E}_{s\sim d_\pi}\{\nabla_\theta \pi_\theta(s) \nabla_a Q(s,a;w)\},$$

这个公式也被称为**deterministic policy gradient**（DPG）。简要介绍一下里面各个量的维度。这里，动作s是一个m维的列向量，策略参数θ是一个n维的列向量，动作值函数Q是一个标量，$\nabla_\theta \pi_\theta(s) \overset{\text{def}}{=} \partial \pi^{\mathrm{T}} / \partial \theta \in \mathbb{R}^{l\theta \times m'}$是一个$l_\theta \times m$的矩阵，$\nabla_a Q(s,a;w) \overset{\text{def}}{=} \partial Q / \partial a \in \mathbb{R}^{m'}$是一个$m'$维的列向量。那么最后的梯度就是一个$l_\theta \times 1$的列向量。

算法的整体框架如下：

Hyperparameters: critic learning rate α, actor learning rate β, discount factor γ, critic update frequency n_c, actor update frequency n_a, number of environment resets M, length of each episode B, variance of exploration noise σ_ϵ^2

Initialization: action-value function $Q(s,a;w)$, policy function $\pi(s;\theta)$

Repeat (indexed with k)

(1) Data collection

Initialize memory buffer $\mathcal{D} \leftarrow \emptyset$

Repeat M environment resets

$s_0 \sim d_{\text{init}}(s)$

For i in $0,1,2,\dots,B-1$ or until episode termination

$a_i \leftarrow \pi(s_i;\theta) + \epsilon\,,\epsilon \sim \mathcal{N}(0,\sigma_\epsilon^2)$

Apply a_i in environment, observe s_{i+1} and r_i

$\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_i,a_i,r_i,s_{i+1})\}$

End

End

(2) Critic update

Repeat n_c times

$$\nabla_w J_{\text{Critic}} \leftarrow \frac{1}{|\mathcal{D}|}\sum_{\mathcal{D}}\left(r+\gamma Q(s',a';w)-Q(s,a;w)\right)\frac{\partial Q(s,a;w)}{\partial w}$$

$$w \leftarrow w - \alpha \cdot \nabla_w J_{\text{Critic}}$$

End

(3) Actor update

Repeat n_a times

$$\nabla_\theta J_{\text{Actor}} \leftarrow \frac{1}{|\mathcal{D}|}\sum_{\mathcal{D}}\nabla_\theta \pi(s;\theta)\nabla_a Q(s,a;w)$$

$$\theta \leftarrow \theta + \beta \cdot \nabla_\theta J_{\text{Actor}}$$

End

End

下面来解释一下算法的关键部分：

- **参数**
  - α：critic的学习率
  - β：actor的学习率

- γ：discount factor
- n_c：critic每轮大循环里面的迭代次数。前面说过，这个值按理论上来说应该达到无穷次，但是实际上我们只会进行有限次的迭代。
- n_a：actor每轮大循环里面的迭代次数。
- M：在每轮大循环里面采样时将环境重置回初始状态的次数，与下面的episode长度共同决定了每轮数据集大小（Batch Size）。
- B：每轮大循环里面的episode长度。在M轮中的每轮探索环境收集数据时达到该长度或者达到终止条件后就会结束。

这里超参数的选取是AC算法里面十分重要的一点，需要根据实际情况和经验进行调整。比如$n_c=5\sim20, n_a=1$或者$n_c=n_a=1$是两种常见的选择critic和actor的迭代次数的方法。

- **Data Collection**：这里之所以在探索环境时引入了一个随机噪声ϵ是因为我们这里本来就是一个确定性的策略，如果在探索环境时还不加上噪声的话，那么对于训练出来的模型就缺乏对于环境的全局认识，自然远远达不到最优的。这里的ϵ是一个服从高斯分布$\mathcal{N}(0,\sigma_\epsilon^2)$的随机噪声。
- **Critic Update**：这里用的就是前面值函数近似那里（第5.3.1节）的公式，只不过把状态值函数换成了动作值函数且把取期望换成了使用大小为$\mathcal{D}$的数据集来估计。
- **Actor Update**：这里的更新公式就是DPG的梯度公式。也是吧取期望换成了使用大小为$\mathcal{D}$的数据集来估计。

6.5.2.2 Stochastic Policy版本

回顾我们在第6.4.1.1节讲过的采用动作值函数的indirect on-policy gradient的形式：

$$\begin{aligned}\nabla_\theta J(\theta) &= \sum_s d_\pi(s)\sum_a \pi_\theta(a|s)\nabla_\theta \log \pi_\theta(a|s) Q(s,a)\\ &= \mathbb{E}_{s\sim d_\pi, a\sim\pi}\{\nabla_\theta \log \pi_\theta(a|s) Q(s,a)\},\end{aligned}$$

这里面使用了log-derivative trick。但是，做了这种变换后仍然不能直接使用，因为这样的做法会使得在sample-based的估计中带来高方差。为了获得一个低方差的stochastic gradient版本，我们需要使用重参数技巧（Reparameterization Trick）。为此，我们不使用上述引入log-derivative trick的$J(\theta)$版本，而是先回到原始版本：

$$\nabla_\theta J_{\text{Actor}} = \nabla_\theta \mathbb{E}_{s\sim d_\pi, a\sim\pi_\theta}\{Q(s,a;w)\}.$$

紧接着，我们对动作a进行重参数化：

$$a = g_\theta(s,\epsilon)$$

这里的$g_\theta(s,\epsilon)$是一个确定性的函数，而ϵ是一个服从某种分布$p(\epsilon)$的随机变量。那么，上式就可以进一步化简：

$$\begin{aligned}\nabla_\theta \mathbb{E}_{s,a\sim\pi_\theta}\{Q(s,a;w)\} &= \nabla_\theta \mathbb{E}_{s,\epsilon\sim p(\epsilon)}\{Q(s,g_\theta(s,\epsilon);w)\} \\ &= \mathbb{E}_{s,\epsilon\sim p(\epsilon)}\{\nabla_\theta Q(s,g_\theta(s,\epsilon);w)\} \\ &= \mathbb{E}_{s,\epsilon\sim p(\epsilon)}\{\nabla_\theta g_\theta(s,\epsilon)\nabla_a Q(s,a;w)\},\end{aligned}$$

注意这里之所以右边第一式到第二式可以把梯度符号移到期望里面是因为这里期望的两个下标s和ϵ均与策略参数θ无关。再来说说右边第三式的含义，化简到这里就可以看出，这里求梯度可以看成是对于一个有关策略参数的确定性函数和动作值函数分别对于策略参数θ和动作a求梯度。

接下来的问题就变成了如何构建这个重参数化函数$g_\theta(s,\epsilon)$。这里举两个例子。假如我们的原始策略是一个高斯分布，即$\pi_\theta(a|s)\sim\mathcal{N}(\mu_\theta(s),\sigma_\theta^2(s))$ 那么，我们可以使用下面的重参数化函数：

$$g_\theta(s,\epsilon)=\mu_\theta(s)+\epsilon\cdot\sigma_\theta(s),$$

这里的ϵ是一个服从$\mathcal{N}(0,1)$的随机变量。另一个例子是，如果我们的原始策略服从均匀分布，即$\pi_\theta(a|s)\sim\mathcal{U}(a_{\min},a_{\max})$，那么我们可以使用下面的重参数化函数：

$$g_\theta(s,\epsilon)=\frac{\theta_{\max}+\theta_{\min}}{2}+\epsilon\cdot\frac{\theta_{\max}-\theta_{\min}}{2},$$

这里的ϵ是一个服从$\mathcal{U}(-1,1)$的随机变量。

使用了重参数技巧的stochastic policy gradient的好处在于求期望的时候期望下标里面不包含动作分布，这样可以减小估计的方差。这种好处的取得是因为我们在构建重参数化函数的时候引入了一个分布已知的随机变量。这样就提供了一定量的确定性信息。但是，这种方法因为其认为假定了一个分布，自然也不是没有代价的。只有当我们假设的分布与真实分布贴近时，才可能降低方差。

算法的整体框架如下：

Hyperparameters: critic learning rate α, actor learning rate β, discount factor γ, critic update frequency n_c, actor update frequency n_a, number of environment resets M, length of each episode B

Initialization: action-value function $Q(s,a;w)$, policy function $\pi(a|s;\theta)$

Repeat (indexed with k)

(1) Data collection

Initialize memory buffer $\mathcal{D} \leftarrow \emptyset$

Repeat M environment resets

$s_0 \sim d_{\text{init}}(s)$

For i in 0,1,2, … , $B-1$ or until episode termination

$\epsilon \sim \mathcal{N}(0,1)$

$a_i = g_\theta(s_i,\epsilon) = \mu_\theta(s_i) + \epsilon \cdot \sigma_\theta(s_i)$

Apply a_i in environment, observe s_{i+1} and r_i

$\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_i, a_i, r_i, s_{i+1})\}$

End

End

(2) Critic update

Repeat n_c times

$$\nabla_w J_{\text{Critic}} \leftarrow \frac{1}{|\mathcal{D}|}\sum_{\mathcal{D}}\left(r+\gamma Q(s',a';w)-Q(s,a;w)\right)\frac{\partial Q(s,a;w)}{\partial w}$$

$$w \leftarrow w - \alpha \cdot \nabla_w J_{\text{Critic}}$$

End

(3) Actor update

Repeat n_a times

$$\nabla_\theta J_{\text{Actor}} \leftarrow \frac{1}{|\mathcal{D}|}\sum_{\mathcal{D}} \nabla_\theta g_\theta(s,\epsilon)\nabla_a Q(s,a;w)$$

$$\theta \leftarrow \theta + \beta \cdot \nabla_\theta J_{\text{Actor}}$$

End

End

这里要说明的事项基本上在上面的Deterministic Policy版本中都有提到，这里就不再赘述。唯一需要注意的是Actor Update部分里的梯度公式是使用了重参数技巧的stochastic policy gradient的梯度公式。

6.5.3 On-policy AC with State-Value Function

很遗憾，我们在使用状态值函数时灵活性较小。首先，使用状态值函数的时候策略只能选择随机策略，这在第6.5.1节中提到过。在使用了随机策略的前提下，也不能像动作值函数那里使用reparameterization trick来减小方差，只能使用log-derivative trick。不过，这里也有好处，就是使用随机策略和状态值函数的AC算法需要更少的样本来达到精确的估计。

首先，再次回顾一下6.4.1.2节里那里得出的indirect policy gradient的形式：

$$\nabla_\theta J_{\text{Actor}}(\theta) = \mathbb{E}_{\pi_\theta}\{\nabla_\theta \log \pi_\theta(a|s)\,(r + \gamma V(s';w))\}.$$

算法的整体框架如下：

Hyperparameters: critic learning rate α, actor learning rate β, discount factor γ, critic update frequency n_c, actor update frequency n_a, number of environment resets M, length of each episode B

Initialization: state-value function $V(s;w)$, policy function $\pi(a|s;\theta)$

Repeat (indexed with k)

(1) Data collection

Initialize memory buffer $\mathcal{D} \leftarrow \emptyset$

Repeat M environment resets

$s_0 \sim d_{\text{init}}(s)$

For i in 0,1,2, … , $B-1$ or until episode termination

$a_i \sim \pi(a|s_i;\theta)$

Apply a_i in environment, observe s_{i+1} and r_i

$\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_i, a_i, r_i, s_{i+1})\}$

End

End

(2) Critic update

Repeat n_c times

$$\nabla_w J_{\text{Critic}} \leftarrow \frac{1}{|\mathcal{D}|}\sum_{\mathcal{D}}\big(r + \gamma V(s';w) - V(s;w)\big)\frac{\partial V(s;w)}{\partial w}$$

$$w \leftarrow w - \alpha \cdot \nabla_w J_{\text{Critic}}$$

End

(3) Actor update

Repeat n_a times

$$\nabla_\theta J_{\text{Actor}} \leftarrow \frac{1}{|\mathcal{D}|}\sum_{\mathcal{D}} \nabla_\theta \log \pi(a|s;\theta)\,(r + \gamma V(s';w))$$

$$\theta \leftarrow \theta + \beta \cdot \nabla_\theta J_{\text{Actor}}$$

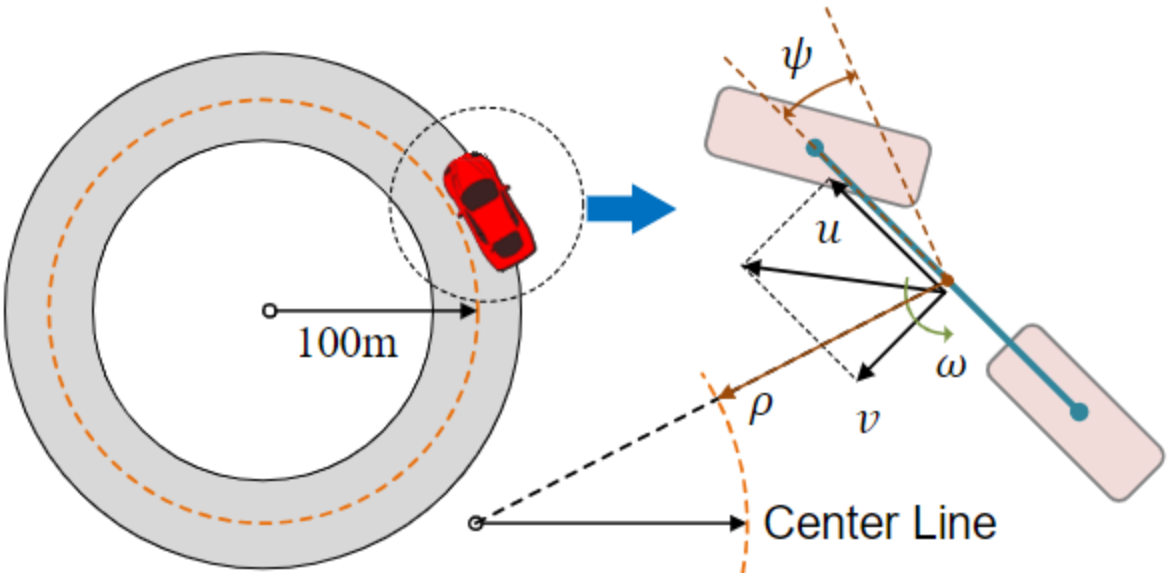
End

End

与刚才讲的两个算法基本一致，这里就不再赘述。

6.6 一个例子：环形路上的自动驾驶

这里以一个实例来结束本单元。我们考虑一个环形路上的自动驾驶问题。控制的目标是使得横向位置、Yaw角和纵向速度的误差以及车辆控制所需的转向以及加速所付出的代价最小。注意，这里我们重点关注是怎么把这个问题建模成一个RL问题的，至于具体的车辆动力学建模以及超参数选取等请参考原书第6.6节。



首先，定义状态空间和动作空间如下：

$$s = [\rho, \psi, u, v, \omega]^{\mathrm{T}} \in \mathbb{R}^5$$
$$a = [\delta, a_x]^{\mathrm{T}} \in \mathbb{R}^2.$$

含义分别如下：

- **状态空间**
  - ρ：横向位置误差
  - ψ：车辆车头与此时道路切线位置夹角
  - u：纵向速度
  - v：横向速度
  - ω：Yaw角速度
- **动作空间**
  - δ：前轮转向角
  - a_x：纵向加速度

关于车辆动力学以及环境的随机扰动的建模、重要的车辆以及环境参数参考原书178到179页。接下来，我们要定义奖励函数。这里我们定义的奖励函数如下：

$$
\begin{gathered}
r(s)=c_0-c_\rho|\rho|-c_\psi\psi^2-c_u|u-u_{\exp}|-c_\delta\delta^2-c_a a_x^2-I_{\text{fail}} \\
\text{where } I_{\text{fail}}=\begin{cases}100, & \text{if out of lane} \\ 0, & \text{otherwise}\end{cases}.
\end{gathered}
$$

这个奖励函数对上面提到优化目标里面各个环节都进行了考虑。并且注意到，最前面添加了一个常数项c_0，这是因为在添加这个常数项之前所有的奖励项都是负的，这样直接训练会导致车辆倾向于采取直接撞击来停止车辆，这样早点结束反而能取得更高的return。因此，我们添加了这个常数项来保证车辆不会在开始的时候就直接撞击。

接下来看看如何对于策略进行建模。对于随机策略，我们可以使用高斯分布来建模：

$$
\begin{gathered}
a\sim\pi(a|s)=\mathcal{N}(\mu,\sigma^2), \\
\mu=\mathrm{NN}(s;\theta_{\mathrm{NN}})\in\mathbb{R}^2, \\
\sigma=\exp(\theta_{\exp})\in\mathbb{R}^2,
\end{gathered}
$$

这里的NN表示一个神经网络。我们的策略是在最外层的高斯分布里面套上了可训练的参数（NN和指数函数）。因此，最终的策略参数为$\theta=[\theta_{\mathrm{NN}},\theta_{\exp}]$。对于确定性的动作，直接使用神经网络来建模：

$$
a=\pi(s)=\mathrm{NN}(s;\theta)\in\mathbb{R}^2.
$$

接着，我们使用NN来拟合值函数：

$$
\begin{gathered}
V(s)=\mathrm{NN}(s;\theta) \\
or \\
Q(s,a)=\mathrm{NN}(s,a;\theta).
\end{gathered}
$$

下面使用了三种不同的AC算法来解决这个问题：stochastic policy with action-value function（Sto-Q），deterministic policy with action-value function（Det-Q）和stochastic policy with state-value function（Sto-V）。

有关具体的实验结果请参考原书第6.6.3节。

@[toc]

书籍链接：Reinforcement Learning for Sequential Decision and Optimal Control

本篇博客主要介绍了使用Policy Gradient的直接强化学习（Direct RL with Policy Gradient）。在之前的博客中，我们介绍的MC、TD等方法都是基于贝尔曼方程的间接RL方法。而本单元讲的Direct RL则不利用贝尔曼方程，直接通过优化一个预先定义的目标函数来学习策略。这样处理RL问题有很多好处，最大的好处我认为是形式的简洁性以及能够与现有的很多数值优化方法进行结合。采用这样直接优化策略的做法，可以保证至少收敛到一个局部最优解，以及处理众多不是完全马尔科夫的问题。

这类方法最早的时候是随机的（Stochastic）的，输出为一个针对动作的概率分布。经典的工作包括REINFORCE算法、Vanilla Policy Gradient算法等。但是，动作的随机性也导致了stochastic policy gradient方法的高方差。在此基础上，Silver等人在2014年提出了Deterministic Policy Gradient（DPG）方法，该算法因其避免了对于随机的动作的积分而避免了高方差。之后，基于深度学习的改进版本DDPG、TD3等也相继提出。

Direct RL，尤其是那些off-policy gradient的版本，很容易导致训练时的不稳定。解决办法是避免训练时过快的调整策略。著名的改进做法比如Trust Region Policy Optimization（TRPO），该方法通过限制每次更新的策略的KL散度来保证策略的稳定性。但是其每次要计算Hessian矩阵，计算量较大。之后，PPO算法提出，有效降低了计算量。这些算法的核心都是通过限制策略更新的幅度来保证策略的稳定性。

7.1 Direct RL的总目标函数（Overall Objective Function）

为了说明如何构建Direct RL算法，我们首先规定在一个统一的问题下讨论后续的操作：具有随机策略的连续任务（Continuing Tasks with Stochastic Policy）。因为episodic任务可以看作是连续任务的特例，只需要认为在episode终止后的奖励全为0即可。同时，确定性策略的任务可以看作是随机策略的特例。因此，我们只讨论连续任务和随机策略的情况。

首先，我们先来看看我们的目标函数：长时间的累积奖励（Long-Term Accumulative Reward）：

$$\begin{aligned}\max_\theta J(\theta) &= \mathbb{E}_{s_t\sim d(s_t)}\{v^{\pi_\theta}(s_t)\}\\ &= \int d(s_t)v^{\pi_\theta}(s_t)\mathrm{d}s_t\\ &= \mathbb{E}_{s_t,a_t,s_{t+1},\ldots\sim\rho_{\pi_\theta}}\left\{\sum_{\tau=t}^{\infty}\gamma^{\tau-t}r_\tau\right\},\end{aligned}$$

从右边第一式到第二式使用的是期望的定义，从第二式到第三式是将状态值函数$v^{\pi_\theta}(s_t)$按照定义展开，即：

$$v^{\pi_\theta}(s) = \mathbb{E}_{a_t,s_{t+1},a_{t+1},s_{t+2},a_{t+2},\ldots\sim\rho\pi_\theta}\left\{\sum_{\tau=t}^{\infty}\gamma^{\tau-t}r_\tau|s_t=s\right\},$$

其中$\mathbb{E}$的下标$a_t, s_{t+1}, a_{t+1}, s_{t+2}, a_{t+2}, \ldots \sim \rho\pi_\theta$表示的是对于轨迹上的状态和动作的联合概率分布。

这里的$J(\theta)$被称为“Overall” Objective Function，是为了和AC算法中的Critic loss function和Actor loss function区分开来。还需要提请注意的一点是$s_t \sim d(s_t)$中的初始状态分布$d(s_t)$。首先，这个分布**必须与策略参数θ无关**，否则我们无法通过优化θ来优化$J(\theta)$。其次，引入这个分布其实就是相当于一种加权操作，在出现更多的状态上放置更多的注意。那么，我们能不能选取其他的分布作为加权的因子呢？在前面的博客中，我们提到过只要待优化的策略没有结构上的限制，那么无论初始的状态分布怎么选择都不会影响最终学得的策略的最优性。因此，理论上，这里用于加权的分布可有选择诸如平均的分布或者就优化从某个特定状态开始的$J(\theta)$（相当于一个只在某一个特定点有非零概率，在其他点概率均为0的分布）。但是，在实际中，在实际中，随便选取用于加权的分布会导致训练的不稳定以及策略的不准确。这是因为在用于加权的分布和真实的分布之间状态出现的频率会出现巨大的不匹配。因此，我们通常就选择**Stationary State Distribution（SSD）**作为加权的分布。

7.1.1 MDP过程的Stationary State Distribution（SSD）

SSD是马尔可夫决策过程的一个重要的性质。我们以马尔可夫决策过程的一个重要的实例化：马尔可夫链来讨论SSD。在第二单元的博客中我们介绍过，马尔可夫链具有有限的和可数的状态集合。我们可以使用有向图来描述马尔可夫链，其中每个节点代表一个状态，每条边代表一个状态转移概率。下面我们来演示怎么引出SSD的概念。

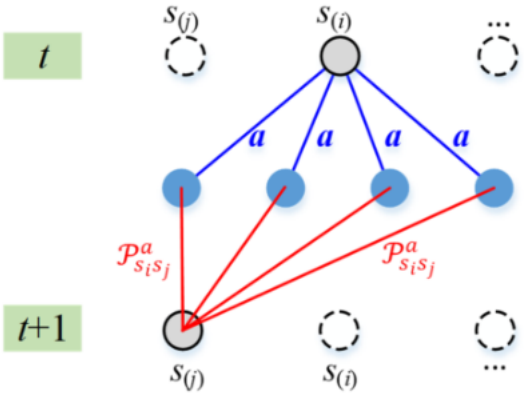
首先，我们定义状态空间$\mathcal{S}$，其中包含n个状态：

$$\mathcal{S} = \{s_(1), s_(2), ..., s_(n)\}.$$

然后，我们定义状态转移概率:

$$\zeta_{i,j} = \sum_{a\in\mathcal{A}}\pi(a|s=s_{(i)})p\big(s'=s_{(j)}|s=s_{(i)},a\big),$$

其中，$\zeta_{i,j}$表示从状态$s_{(i)}$转移到状态$s_{(j)}$的概率。上面关于$\zeta_{i,j}$的定义通过枚举从状态$s_{(i)}$开始采用动作集合$\mathcal{A}$中的每一个动作，然后根据环境模型进行转移的各种情况之和得到的。该过程可以使用下面的图来形象的表示：



接下来，为了分析和定义方便，我们需要使用状态空间模型来描述马尔可夫链。但是，马尔可夫过程是一个随机过程，每个时刻的状态是一个随机变量，无法直接使用状态空间模型来描述。因此我们需要将状态表示为一个向量，向量的每一维都是状态空间中的一种可能的状态的边缘分布：

$$d_t(s) = \begin{bmatrix} d_t(s_{(1)}) & d_t(s_{(2)}) & \cdots & d_t(s_{(n)}) \end{bmatrix}^{\mathrm{T}} \in \mathbb{R}^{n\times 1}.$$

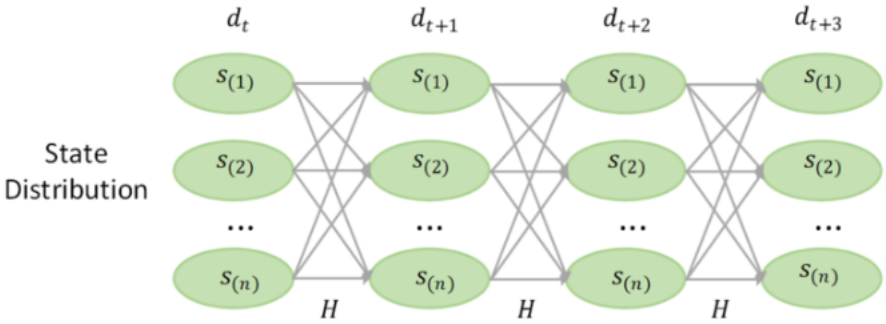
这样我们就把马尔可夫链的状态这个随机变量描述为了一个n维的列向量，该向量在每个固定的时刻都是一个确定值。这样，就可以写出马尔可夫链的状态转移方程：

$$d_{t+1} = H_{n\times n} d_t,$$

其中

$$H = \begin{bmatrix} \zeta_{1,1} & \zeta_{1,2} & \cdots & \zeta_{1,n} \\ \zeta_{2,1} & \zeta_{2,2} & \cdots & \zeta_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \zeta_{n,1} & \zeta_{n,2} & \cdots & \zeta_{n,n} \end{bmatrix}.$$

是单步的状态转移概率构成的矩阵。每个时刻之间的状态转移如下图所示：



d_t代表了在时刻t当前状态为状态集合里面各个状态的概率，d_t的所有元素的和为1。

很多随机的RL算法都假设算法是独立于时间t的。具体到马尔可夫链来说，当$t \to \infty$时，马尔可夫链会收敛到一个平衡态，也就是我们之前一直说的Stationary State Distribution（SSD）。这里的stationary意思就是与时间t无关。下面，我们给出SSD的定义：

> **定义1：Stationary State Distribution（SSD）**：
>
> $$d_\pi(s) = \begin{bmatrix} d_\pi(s_{(1)}) & d_\pi(s_{(2)}) & \cdots & d_\pi(s_{(n)}) \end{bmatrix}^{\mathrm{T}}$$
>
> 这里$\sum_{i=1}^{n} d_\pi(s_{(i)}) = 1$并且$d_\pi(s_{(i)}) \geq 0$，满足$d_\pi(s) = H d_\pi(s)$（换句话说，即$d_\pi(s_{(j)}) = \sum_{s_{(i)}\in\mathcal{S}} d_\pi(s_{(i)})\zeta_{i,j}$）。

注意，这里的SSD针对的是与时间无关的有限MDP而言的。可以证明，对于任意有限的、不可约的、具有遍历性的MDP，具有以下性质：

> - 该MDP的SSD是唯一的；
> - 对任意$s_(i), s_(j) \in \mathcal{S}$，当$t \to \infty$时，初始状态$s_0$为$s_(i)$且时刻t的时候的状态为$s_(j)$的概率值存在且与初始状态$s_0$无关。且这个概率值等于SSD中的$d_\pi(s_(j))$。

这个性质说明了两点，第一，说明了SSD的存在性；第二，说明了SSD的含义，即不管初始状态是什么，最后收敛到某个具体状态的的概率都是一定的。

SSD被认为是instantaneous state distribution的一种特殊形式。并且，上述说明MDP具有唯一的SSD是在很多的强假设下做出的，实际上一个一般的MDP的平衡态可能有多个。对于具有唯一性的SSD，在策略π下满足下面的式子：

$$d_\pi(s_{(j)}) = \sum_{s\in\mathcal{S}} d_\pi(s) \sum_{a\in\mathcal{H}} \pi(a|s) p\big(s' = s_{(j)}|s,a\big),$$
$$j = 1, 2, \cdots, n.$$

7.1.2 折扣目标函数（Discounted Objective Function）

我们刚才讨论的目标函数的形式为：

$$J(\theta) = \mathbb{E}_{s_t\sim d(s_t)}\{v^{\pi_\theta}(s_t)\}$$

为了在实际中使用这个目标函数，我们需要将其实例化。本小节我们讨论将目标函数实例化的一种形式：折扣目标函数（Discounted Objective Function）。

使用折扣目标函数的原因很简单，因为我们讨论的continuous task的时间是无限的，但是目标函数的值却必须是有界的。实现这个要求的一种简单的办法就是引入在[0,1]之间的折扣因子γ。这样，我们就可以将目标函数改写为：

$$\begin{aligned}
J(\theta) &= \mathbb{E}_{s_t\sim d(s_t)}\{v^{\pi_\theta}(s_t)\} \\
&= \mathbb{E}_{s_t,a_t,\ldots\sim\rho_{\pi_\theta}}\left\{\sum_{\tau=t}^{\infty}\gamma^{\tau-t}r_\tau\right\} \\
&= \sum_{s_t} d(s_t)\sum_{a_t}\pi_\theta(a_t|s_t)\sum_{s_{t+1}}p(s_{t+1}|s_t,a_t)\sum_{a_{t+1}}\pi_\theta(a_{t+1}|s_{t+1})\sum_{s_{t+2}}p(s_{t+2}|s_{t+1},a_{t+1})\cdots\sum_{\tau=t}^{\infty}\gamma^{\tau-t}r_\tau \\
&= \sum_{s} p(s_t=s|\pi_\theta)\sum_{a_t}\pi_\theta(a_t|s)\sum_{s_{t+1}}p(s_{t+1}|s,a_t)r_t + \sum_{s} p(s_{t+1}=s|\pi_\theta)\sum_{a_{t+1}}\pi_\theta(a_{t+1}|s)\sum_{s_{t+2}}p(s_{t+2}|s,a_{t+1})\cdots\sum_{\tau=t+1}^{\infty}\gamma^{\tau-t}r_\tau \\
&= \sum_{s} p(s_t=s|\pi_\theta)\sum_{a_t}\pi_\theta(a_t|s)\sum_{s_{t+1}}p(s_{t+1}|s,a_t)r_t + \gamma\sum_{s} p(s_{t+1}=s|\pi_\theta)\sum_{a_{t+1}}\pi_\theta(a_{t+1}|s)\sum_{s_{t+2}}p(s_{t+2}|s,a_{t+1})r_{t+1} + \sum_{s} p(s_{t+2}=s|\pi_\theta)\sum_{a_t} \\
&\qquad\qquad\qquad\qquad \cdots\ \cdots \\
&= \sum_{\tau=t}^{\infty}\gamma^{\tau-t}\sum_{s} p(s_\tau=s|\pi_\theta)\sum_{a}\pi_\theta(a|s)\sum_{s'}p(s'|s,a)r(s,a,s') \\
&= \sum_{s} p(s_\tau=s|\pi_\theta)\sum_{\tau=t}^{\infty}\gamma^{\tau-t}\sum_{a}\pi_\theta(a|s)\sum_{s'}p(s'|s,a)r(s,a,s') \\
&= \sum_{s}\sum_{\tau=t}^{\infty}\gamma^{\tau-t} p(s_\tau=s|\pi_\theta)\sum_{a}\pi_\theta(a|s)\sum_{s'}p(s'|s,a)r(s,a,s') \\
&= \frac{1}{1-\gamma}\sum_{s} d^\gamma_{\pi_\theta}(s)\sum_{a}\pi_\theta(a|s)\sum_{s'}p(s'|s,a)r(s,a,s')
\end{aligned}$$

上面列了一大串公式，现在就逐一讲讲怎么来的。首先从右边第一行到第二行，就是通过引入了折扣因子γ实现了构建折扣目标函数。第三式省略号前面那一长串就是把$\mathbb{E}$的那一长串下标展开写。从第三式到第四式是因为r_t只与初始状态s_t和动作a_t有关，与后面的动作和状态无关，所以可以提到前面来。然后对于后面的那一堆连乘的式子，可以采用相同的方法进行展开，最终可以得到第五式以及下面省略号后面的式子。那么，就可以得到一种类似于这样的式子：$1\times()+\gamma\times()+\gamma^2\times()+\cdots$，那么我们交换求和顺序，第一重求和是对于时间$\tau$的求和。最后几式的推导已经写得很清楚了，这里就不再赘述。注意，我们把$d^\gamma_{\pi_\theta}(s)$定义为：

$$d^\gamma_{\pi_\theta}(s) = (1-\gamma)\sum_{\tau=t}^{\infty}\gamma^{\tau-t}p(s_\tau=s|\pi_\theta).$$

这里，我们称$d^\gamma_{\pi_\theta}(s)$为discounted state distribution。它卡面乘了一个因子$(1-\gamma)$，这个因子是为了保证$\sum_s d^\gamma_{\pi_\theta}(s)$的和为1。

最终，我们把目标函数写成了三重求和（或者在连续状态空间中三重积分）的形式。写成这种形式有助于后续推导likelihood ratio gradients。

7.1.3 平均目标函数（Average Objective Function）

另一种使得目标函数有界的方法是引入平均目标函数（Average Objective Function），也就是把总的reward之和除以总步数，然后令总步数趋于无穷。平均目标函数和折扣目标函数的最大区别在于前者对于每个时间步的reward都赋予相同的重视程度，而后者则是对于越靠后的reward赋予越小的重视程度。因此。平均目标函数适合处理那些每个时间步都有相同重要性的任务，一个典型的例子就是交通运输问题中的燃油节省问题。

平均目标函数的定义如下：

$$J(\theta) = \lim_{N\to\infty}\frac{1}{N}\mathbb{E}_{\pi_\theta}\left\{\sum_{\tau=t}^{t+N-1} r_\tau\right\}$$
$$= \sum_s d_{\pi_\theta}(s)\sum_a \pi_\theta(a|s)\sum_{s'} p(s'|s,a)r(s,a,s').$$

这个式子是怎么来的呢？可以从上面我们推导折扣目标函数的过程中看出端倪。回到上面推导折扣目标函数的第五式：

$$= \sum_s p(s_t=s|\pi_\theta)\sum_{a_t}\pi_\theta(a_t|s)\sum_{s_{t+1}} p(s_{t+1}|s,a_t)r_t + \gamma\sum_s p(s_{t+1}=s|\pi_\theta)\sum_{a_{t+1}}\pi_\theta(a_{t+1}|s)\sum_{s_{t+2}} p(s_{t+2}|s,a_{t+1})r_{t+1} + \gamma^2\sum_s p(s_{t+2}=s|\pi_\theta)\sum_{a_{t+2}}$$

我们可以将此时的γ设置为1，即每步同等重要。将共N步这样的式子加在一起，得到了一个和。将这个和除以N，再令N趋于无穷，就得到了平均目标函数。因为上述求和的每项都是形如$\sum_s p(s|\pi_\theta)\sum_a \pi_\theta(a|s)\sum_{s'} p(s'|s,a)r$的形式，所以最终的目标函数就是这个形式。这里的$\sum_s p(s|\pi_\theta)$在$N\to\infty$时就是$d_{\pi_\theta}(s)$，即SSD。也就是说：

$$d_{\pi_\theta}(s) = \lim_{\tau\to\infty} p(s_\tau = s|\pi_\theta).$$

7.2 Likelihood Ratio Gradient

首先，先来回顾一下梯度下降法。根据高数的知识我们可以知道，对于多元函数来说，梯度是其上升最快的方向，那么如果我们要求最小值，那么每次沿着梯度反方向更新即可。而在RL中，我们更常要处理的是最大化目标函数，那么只需要沿着梯度上升的方向更新即可：

$$\theta \leftarrow \theta + \beta\cdot\nabla_\theta J(\theta),$$

可以证明，只要对于梯度的更新是无偏的，再加上一些关于学习率β较弱的条件，那么至少可以收敛到一个局部最优解。从上个世纪九十年代早期，人们就开始使用一些梯度下降的方法来求解RL问题。但是，早期的探索主要是一些actor-only的算法，这些算法在数值上不稳定，并且容易产生高方差。一个重要的改进是vanilla policy gradient算法，该算法是actor-critic算法的一个重要的里程碑。Vanilla policy gradient可以看成是一种likelihood ratio gradient的特例。有两种方法可以推导出policy gradient，一种是基于轨迹（trajectory）的观念的，discounted return的期望是直接优化的；另一种使用了Cascading（级联）的思想，优化的是状态值函数的期望。下面将分别介绍这两种方法。

7.2.1 基于轨迹概念的Gradient推导

7.2.1.1 从轨迹角度来推导Likelihood Ratio Gradient

在本处推导时，我们不失一般性的以离散的状态和动作空间来推导，这样可以简化记号的使用和概念的说明，比如我们就可以使用求和符号来代替积分符号。

我们首先将轨迹$\mathcal{T}$定义为一个状态-动作序列的集合：

$$T \overset{\text{def}}{=} \{s_t, a_t, ..., s_{t+n}, a_{t+n}, ...\}$$

那么，接下来就可以将discounted return写成轨迹$\mathcal{T}$的函数：

$$G(T) = \sum_{\tau=t}^{\infty}\gamma^{\tau-t}r(s_\tau, a_\tau)$$

那么，我们在上一节讲过的折扣目标函数也可以写成$\mathcal{T}$的函数：

$$J(\theta) = \mathbb{E}_{T\sim\rho\pi_\theta}\{G(T)\} = \sum_T \rho_{\pi_\theta}(T)G(T),$$

这里的$\rho_{\pi_\theta}(T)$是轨迹$\mathcal{T}$的概率分布（或者说是轨迹上众多状态和动作的联合概率分布）。即：

$$\rho_{\pi_\theta}(T) = d(s_t)\prod_{\tau=t}^{\infty}\pi_\theta(a_\tau|s_\tau)p(s_{\tau+1}|s_\tau,a_\tau).$$

那么，我们接下来就可以来推导目标函数梯度的形式。首先，我们可以写出目标函数的梯度：

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \sum_T \nabla_\theta \rho_{\pi_\theta}(T) G(T) \\
&= \sum_T \rho_{\pi_\theta}(T) \frac{\nabla_\theta \rho_{\pi_\theta}(T)}{\rho_{\pi_\theta}(T)} G(T) \\
&= \mathbb{E}_{T\sim\rho_{\pi_\theta}} \left\{ \nabla_\theta \log \rho_{\pi_\theta}(\mathcal{T}) \cdot G(\mathcal{T}) \right\},
\end{aligned}
$$

从右边第一式到右边第二式就是做了一个恒等变形，但是为什么要做一个这样的变形呢？主要是为了凑出score function的形式，其定义如下：

> **定义2：Score Function**：对于一个概率分布$p(x)$，其score function定义为：
>
> $$\nabla_\theta \log p(x) = \frac{\nabla_\theta p(x)}{p(x)}.$$

其实这里的score function就是我们在第六单元博客讲过的log-derivative trick。score function的一个重要的性质是它的期望总是0，即：

> **性质1：Score Function的期望为0**：
>
> $$\mathbb{E}_{x\sim p(x)} \left\{ \nabla_\theta \log p(x) \right\} = 0.$$

对应到我们这里的情况就是$\sum_{\mathcal{T}} \rho(\mathcal{T}) \nabla_\theta \log \rho(\mathcal{T}) = 0$。我们先来看看本处的score function怎么化简：

$$
\begin{aligned}
\nabla_\theta \log \rho_{\pi_\theta}(\mathcal{T}) &= \nabla_\theta \log d(s_t) + \nabla_\theta \sum_{\tau=t}^{\infty} (\log \pi_\theta(a_\tau|s_\tau) + \log p(s_{\tau+1}|s_\tau, a_\tau)) \\
&= \sum_{\tau=t}^{\infty} \nabla_\theta \log \pi_\theta(a_\tau|s_\tau).
\end{aligned}
$$

这里其实就是把$\rho_{\pi_\theta}(\mathcal{T})$展开，然后因为我们这里是对于策略参数$\theta$求导，所以一些与$\theta$无关的项（比如初始的状态分布以及环境动力学）求导自然为0。那么最后剩下的项实际上就是若干策略项取log后求和（这里我们也可以看出为什么我们要引入log的原因，因为这样可以把乘法转化为加法，方便求导）。那么，我们就可以得到最终的梯度形式：

$$
\nabla_\theta J(\theta) = \mathbb{E}_{T\sim\rho_{\pi_\theta}} \left\{ \sum_{\tau=t}^{\infty} \nabla_\theta \log \pi_\theta(a_\tau|s_\tau) \cdot \sum_{\tau=t}^{\infty} \gamma^{\tau-t} r_\tau \right\}.
$$

这个式子就是我们的likelihood ratio gradient的形式。再观察这里score function的形式，我们发现其分母为$\rho_{\pi_\theta}(\mathcal{T})$，为了使得分式有意义，分母不能为0。这也从侧面解释了为什么likelihood ratio gradient方法只适用于stochastic policy gradient方法，因为对于deterministic policy gradient方法，大部分轨迹的概率为0（只要里面有一步$\pi_\theta(a|s) = 0$，那么整个轨迹的概率就为0），这就导致了分母为0，无法计算梯度。

7.2.1.2 使用Temporal Causality（时间因果性）来化简Likelihood Ratio Gradient

上面的likelihood ratio gradient的形式虽然已经进行了化简，但是其计算仍不易，因其包含了一系列相乘和相加的操作。为了解决这个问题，我们可以利用Temporal Causality（时间因果性）来化简这个梯度。其实这里说起来很高级，其实利用的思想特别简单，就是**未来的动作/决策不会影响之前已经获取到的奖励**。也就是说，这两部分之间没有因果联系。为此，我们可以如下化简：

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{T\sim\rho_{\pi_\theta}} \left\{ \sum_{\tau=t}^{\infty} \nabla_\theta \log \pi_\theta(a_\tau|s_\tau) \cdot \sum_{\tau=t}^{\infty} \gamma^{\tau-t} r_\tau \right\} \\
&= \mathbb{E}_{T\sim\rho_{\pi_\theta}} \left\{ \sum_{\tau=t}^{\infty} \nabla_\theta \log \pi_\theta(a_\tau|s_\tau) \cdot \left(\sum_{i=t}^{\tau-1} \gamma^{i-t} r_i + \sum_{i=\tau}^{\infty} \gamma^{i-t} r_i \right) \right\} \\
&= \mathbb{E}_{T\sim\rho_{\pi_\theta}} \left\{ \sum_{\tau=t}^{\infty} \nabla_\theta \log \pi_\theta(a_\tau|s_\tau) \gamma^{\tau-t} \sum_{i=\tau}^{\infty} \gamma^{i-\tau} r_i \right\}.
\end{aligned}
$$

从右边第一式到右边第二式是这样来的：原式是两个同一级的求和式相乘（两个计数下标均为τ），我们化简的目的是使得两个同级并列相乘的求和式化成内外层的求和式。外层的下标是τ，内层的下标是i。这样，对于外层每个确定的τ，内层的式子就是一个与τ有关的式子。从第二式到第三式是将第二式的小括号里的第一项省去，这就是里哟个了我们刚才提到的Temporal Causality，因为前面的$\nabla_\theta \log \pi_\theta(a_\tau|s_\tau)$是从时刻$\tau$开始的，而小括号里的第一项是$\tau$之前的reward，$\nabla_\theta \log \pi_\theta(a_\tau|s_\tau)$与其无关。这样我们就得到了一个更加简化的梯度形式。化成这个形式有效的降低了计算量并提高了在基于样本进行估计使的准确性。

7.2.1.3 Vanilla Policy Gradient算法

在使用了Temporal Causality进行化简之后，还可以进一步的化简，这就是Vanilla Policy Gradient算法。我们可以将上述的梯度从关于轨迹$\mathcal{T}$的函数转化为关于初始的状态-动作对(s_t, a_t)的函数。这样，我们就可以得到Vanilla Policy Gradient算法的形式：

$$\begin{aligned}\nabla_\theta J(\theta) &= \sum_s \sum_{\tau=t}^{\infty} \gamma^{\tau-t} p(s_\tau = s|\pi_\theta) \sum_a \nabla_\theta \pi_\theta(a|s) q^{\pi_\theta}(s,a) \\ &= \frac{1}{1-\gamma} \sum_s d^\gamma_{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(a|s) q^{\pi_\theta}(s,a).\end{aligned}$$

注意，这里的$d^\gamma_{\pi_\theta}(s)$就是之前提到过的discounted state distribution。从右一式化简到右二式也是使用了discounted state distribution的定义。

7.2.2 基于Cascading的Gradient推导

下面我们使用另一种方法——Cascading来推导policy gradient。不同于基于轨迹的方法，Cascading方法是一种递归的方式，不断利用状态值函数和动作值函数之间的关系来推导梯度。首先，我们知道Overall Objective Function可以写成：

$$J(\theta) = \mathbb{E}_{s\sim d_{\pi_\theta}}\{v^{\pi_\theta}(s)\} = \sum_s d_{\pi_\theta}(s) v^{\pi_\theta}(s).$$

那么，我们可以对其求导：

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta \sum d(s_t) v^{\pi_\theta}(s_t) \\ &= \nabla_\theta \sum_{s_t} d(s_t) \sum_{a_t} \pi_\theta(a_t|s_t) q^{\pi_\theta}(s_t,a_t) \\ &= \sum_{s_t} d(s_t) \sum_{a_t} \Big[\nabla_\theta \pi_\theta(a_t|s_t) q^{\pi_\theta}(s_t,a_t) + \pi_\theta(a_t|s_t) \nabla_\theta q^{\pi_\theta}(s_t,a_t)\Big] \\ &= \sum_{s_t} d(s_t) \sum_{a_t} \Big[\nabla_\theta \pi_\theta(a_t|s_t) q^{\pi_\theta}(s_t,a_t) + \pi_\theta(a_t|s_t) \gamma \nabla_\theta \sum_{s_{t+1}} p(s_{t+1}|s_t,a_t) v^{\pi_\theta}(s_{t+1})\Big] \\ &= \sum_{s_t} d(s_t) \sum_{a_t} \nabla_\theta \pi_\theta(a_t|s_t) q^{\pi_\theta}(s_t,a_t) + \gamma \sum_{s_t} d(s_t) \sum_{a_t} \pi_\theta(a_t|s_t) \sum_{s_{t+1}} p(s_{t+1}|s_t,a_t) \nabla_\theta v^{\pi_\theta}(s_{t+1}) \\ &= \sum_{s_t} d(s_t) \sum_{a_t} \nabla_\theta \pi_\theta(a_t|s_t) q^{\pi_\theta}(s_t,a_t) + \gamma \sum_{s_{t+1}} d(s_{t+1}|\pi_\theta) \nabla_\theta v^{\pi_\theta}(s_{t+1}) \\ &= \sum_{s_t} d(s_t) \sum_{a_t} \nabla_\theta \pi_\theta(a_t|s_t) q^{\pi_\theta}(s_t,a_t) + \gamma \sum_{s_{t+1}} d(s_{t+1}|\pi_\theta) \sum_{a_{t+1}} \nabla_\theta \pi_\theta(a_{t+1}|s_{t+1}) q^{\pi_\theta}(s_{t+1},a_{t+1}) + \gamma^2 \sum_{s_{t+2}} d(s_{t+2}|\pi_\theta) \nabla_\theta v^{\pi_\theta}(s_{t+2}) \\ &\qquad\qquad\qquad\qquad \cdots\cdots \\ &= \sum_{\tau=t} \sum_{s_\tau}^{\infty} \gamma^{\tau-t} p(s_\tau = s|\pi_\theta) \sum_a \nabla_\theta \pi_\theta(a|s) q^{\pi_\theta}(s,a) \\ &= \sum_{\tau=t} \sum_{s_\tau}^{s} d(s_\tau) \sum_{a_\tau} \pi_\theta(a_\tau|s_\tau) \nabla_\theta \log \pi_\theta(a_\tau|s_\tau) \gamma^{\tau-t} q^{\pi_\theta}(s_\tau,a_\tau) \\ &= \frac{1}{1-\gamma} \sum_s d^\gamma_{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(a|s) q^{\pi_\theta}(s,a).\end{aligned}$$

右边第一式到右边第二式是利用了：

$$v^{\pi_\theta}(s_t) = \sum_{a_t} \pi_\theta(a_t|s_t) q^{\pi_\theta}(s_t,a_t).$$

第二式到第三式是利用了基本的求导法则。然后第三式到第四式是利用了

$$q^{\pi_\theta}(s_t,a_t) = \sum_{s_{t+1}\in\mathcal{S}} \mathcal{P}(s_{t+1}|s_t,a_t)(r^{a_t}_{s_t s_{t+1}} + \gamma v^{\pi_\theta}(s_{t+1}))$$

然后因为reward $r^{a_t}_{s_t s_{t+1}}$与θ无关，所以求导为0，因此只剩下了$v^{\pi_\theta}(s_{t+1})$那一项。第五式到第六式是利用了：

$$\sum_{s_t} d(s_t) \sum_{a_t} \pi_\theta(a_t|s_t) \sum_{s_{t+1}} p(s_{t+1}|s_t,a_t) = \sum_{s_{t+1}} d(s_{t+1}|\pi_\theta)$$

可能大家还会疑问，为什么这里的$\sum_{s_{t+1}} d(s_{t+1}|\pi_\theta)$不用对它求导，答案是第六式是从第五式里面来的。而第五式里面就没有对这项求导。之后，反复利用状态值函数和动作值函数之间的关系，我们就可以得到最终的梯度形式。这个梯度公式是Sutton等人率先提出的，这也是对于discounted objective function的第一个可计算的policy gradient。这也为之后的其他policy gradient方法提供了一个很好的基础，比如TRPO、PPO、SAC等。

7.3 On-Policy Gradient

尽管我们刚才已经给出了Vanilla Policy Gradient的形式，但是其仍然不能用来进行计算。这是因为我们对于计算公式中的discounted state distribution仍然束手无策。而且计算整个动作和状态空间上的积分/求和通常来说计算量也太大了。为了解决以上的问题，我们可以使用sample-based的方法来计算Vanilla Policy Gradient。

7.3.1什么时候可以使用SSD来代替Discounted State Distribution?

如果我们能利用SSD来代替discounted state distribution，那么我们就可以极大地化简Vanilla Policy Gradient的计算。那么，什么时候我们可以使用SSD来代替discounted state distribution呢？严格来说，可以使用SSD来代替discounted state distribution的情况只有两种。下面将分别阐述。

第一种情况是**初始分布恰等于SSD**，即$d(s_t)=d_{\pi_\theta}(s)$，这里的$d(s_t)$是初始状态分布，$d_{\pi_\theta}(s)$是策略π下的SSD。在这种情况下，我们可以进行如下推导：

$$\begin{aligned}
d_{\pi_\theta}^{\gamma}(s) &= (1-\gamma)\sum_{\tau=t}^{\infty}\gamma^{\tau-t}p(s_\tau=s|\pi_\theta) \\
&= (1-\gamma)\sum_{\tau=t}^{\infty}\gamma^{\tau-t}d_{\pi_\theta}(s) \\
&= d_{\pi_\theta}(s),
\end{aligned}$$

右边第一式就是discounted state distribution的定义。从第一式到第二式利用的是SSD的特点，即到达SSD之后，后续的状态分布就不再变化了，一直处于SSD。所以时间步τ之后的任意一个$p(s_\tau=s|\pi_\theta)$都等于$d_{\pi_\theta}(s)$。又因为$\sum_{\tau=t}^{\infty}\gamma^{\tau-t}=1/(1-\gamma)$，所以我们就得到了$d_{\pi_\theta}^{\gamma}(s)=d_{\pi_\theta}(s)$。这样，我们就可以使用SSD来代替discounted state distribution。当我们使用SSD来代替discounted state distribution时，就可以极大程度的降低计算量。

第二种情况是**折扣因子**$\gamma\to 1$。其证明如下：

$$\begin{aligned}
\lim_{\gamma\to 1} d_{\pi_\theta}^{\gamma}(s) &= \lim_{\gamma\to 1}\frac{d_{\pi_\theta}^{\gamma}(s)}{1} \\
&= \lim_{\gamma\to 1}\frac{d_{\pi_\theta}^{\gamma}(s)}{\sum_s d_{\pi_\theta}^{\gamma}(s)} \\
&= \lim_{\gamma\to 1}\frac{\sum_{\tau=t}^{\infty}\gamma^{\tau-t}p(s_\tau=s|\pi_\theta)}{\sum_s\sum_{\tau=t}^{\infty}\gamma^{\tau-t}p(s_\tau=s|\pi_\theta)} \\
&= \lim_{\gamma\to 1}\lim_{N\to\infty}\frac{\sum_{\tau=t}^{t+N}\gamma^{\tau-t}p(s_\tau=s|\pi_\theta)}{\sum_s\sum_{\tau=t}^{t+N}\gamma^{\tau-t}p(s_\tau=s|\pi_\theta)} \\
&= \lim_{\gamma\to 1}\lim_{N\to\infty}\frac{\sum_{\tau=t}^{t+N}1\cdot p(s_\tau=s|\pi_\theta)}{\sum_{\tau=t}^{t+N}\sum_s 1\cdot(s_\tau=s|\pi_\theta)} \\
&= \lim_{N\to\infty}\frac{(N+1)\cdot(s_\tau=s|\pi_\theta)}{N+1} \\
&= d_{\pi_\theta}(s).
\end{aligned}$$

右一式到右二式利用了任意一种状态分布的和为1的特点（那么当然对于$d_{\pi_\theta}^{\gamma}(s)$也是成立的）。其余式子的推导我已经展开写的很清楚了。注意，在上面的推导中我们假设了$\gamma\to 1$，那么实际中究竟γ取多少才算是趋近于1呢？在实践中，可以得出当$0.95<\gamma<1.0$时，我们就可以认为γ趋近于1了。要是再加上我们搜集了足够多的样本来做估计，那么discounted state distribution就和SSD非常接近了。

在得出上述两种情况下我们可以使用SSD来替换discounted state distribution的结论后，我们就可以进一步改写Vanilla Policy Gradient的计算公式：

$$\begin{aligned}
\nabla_\theta J(\theta) &= \frac{1}{1-\gamma}\sum_s d_{\pi_\theta}^{\gamma}(s)\sum_a \nabla_\theta\pi_\theta(a|s)q^{\pi_\theta}(s,a). \\
&= \frac{1}{1-\gamma}\sum_s d_{\pi_\theta}^{\gamma}(s)\sum_a \pi_\theta(a|s)\nabla_\theta\log\pi_\theta(a|s)q^{\pi_\theta}(s,a) \\
&\approx \frac{1}{1-\gamma}\sum_s d_{\pi_\theta}(s)\sum_a \pi_\theta(a|s)\nabla_\theta\log\pi_\theta(a|s)q^{\pi_\theta}(s,a) \\
&\propto \mathbb{E}_{s\sim d_{\pi_\theta}}\{\mathbb{E}_{a\sim\pi_\theta}\{q^{\pi_\theta}(s,a)\nabla_\theta\log\pi_\theta(a|s)\}\} \\
&= \mathbb{E}_{\pi_\theta}\{q^{\pi_\theta}(s,a)\nabla_\theta\log\pi_\theta(a|s)\}.
\end{aligned}$$

右一式到右二式使用了log-derivative trick，右二式到右三式使用了SSD可以代替discounted state distribution的结论。右四式到右五式为什么两层期望变成一层了呢？其实右五式中的期望下标π_θ是一个笼统的说法，展开写就是右四式的样子。但是其实具体的期望下标不重要，**最重要的是我们要将梯度写成期望的形式**，这样就可以通过sample-based的方法来做估计。这里得到的Vanilla Policy Gradient的期望形式十分有用，因为它不包含状态分布或者环境动力学信

息，只通过收集sample即可对其进行估计。而且，该式子还是一个无偏估计。但是，此方法仍然存在一些缺点，比如如果手机的sample不够多的话就会产生高方差。下面我们将介绍几种针对性的改进。

7.3.2 On-Policy Gradient的改进

这里介绍三种常见的On-Policy Gradient的改进方法：

- **Monte Carlo policy gradient**：使用Monte Carlo方法来估计动作值函数
- **Baseline**：使用一个baseline来减小方差
- **使用状态值函数来代替动作值函数**

7.3.2.1 Monte Carlo policy gradient

我们注意到想利用$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\{q^{\pi_\theta}(s,a)\nabla_\theta \log \pi_\theta(a|s)\}$.进行计算我们必须知道动作值函数的真实值。但是，在实际的问题中，这个值往往是我们无法获取的。那么怎么办呢？这就得请出老办法——Monte Carlo估计。也就是说，我们可以利用从当前(s_t, a_t)开始的多个样本的Return的平均值来作为$q^{\pi_\theta}(s_t, a_t)$的估计。即：

$$\theta \leftarrow \theta + \beta \cdot \mathrm{Avg}\{G_t|s_t, a_t\}\nabla \log \pi_\theta(a_t|s_t).$$

这种算法也被称为REINFORCE算法。注意到这里并没有使用任何函数来对于动作值函数进行参数化，因此REINFORCE算法是一个actor-only的算法。REINFORCE算法的缺点是其方差较大，且在处理大规模问题时效率很低。其伪代码如下：

Hyperparameters: learning rate β, discount factor γ, number of environment resets M, number of episodes B

Initialization: policy function $\pi(a|s;\theta)$

Repeat (indexed with k)

(4) Data collection

Initialize memory buffer $\mathcal{D} \leftarrow \emptyset$

Repeat M environment resets

$s_0 \sim d_{\mathrm{init}}(s)$

$a_0 \sim \pi(a|s_0;\theta)$

For i in 1,2, … , B

Sample an episode τ_i until termination

$\tau_i \leftarrow \{s_0, a_0, r_0, \cdots, s_{T-1}, a_{T-1}, r_{T-1}, s_T\}$

$G_i(s_0, a_0) \leftarrow \sum_{t=0}^{T-1} \gamma^t r_t$

End

$\bar{G}(s_0, a_0) \leftarrow \mathrm{Avg}\{G_i(s_0, a_0), i = 1,2, \cdots, B\}$

$\mathcal{D} \leftarrow \mathcal{D} \cup \{(\bar{G}, s_0, a_0)\}$

End

(5) Update with MC policy gradient

$$\nabla_\theta J \leftarrow \frac{1}{|\mathcal{D}|}\sum_{\mathcal{D}} \bar{G}(s,a)\nabla \log \pi(a|s;\theta)$$

$\theta \leftarrow \theta + \beta \cdot \nabla_\theta J$

End

现对于关键部分解释如下：

- **Data Collection**：这里最外层大循环总共进行k次。每次大循环中，先清空buffer $\mathcal{D}$，然后往其中添加M个sample，每个sample又是由B个episodes平均得到的。
- **Update with MC Policy Gradient**：这里直接根据公式即可。

7.3.2.2 Baseline技术

Baseline技术是一种减小方差的技术。这个Baseline可以是任何一个和状态s有关的函数，我们把值高于这个Baseline的部分的Q函数对应的动作认为是一个“好”的动作，反之则是一个“坏”的动作。引入了Baseline之后，就可以通过更多的选择好的动作而去掉坏的动作来实现更准确的梯度估计和更快的收敛。引入了Baseline之后，我们的梯度公式变成了：

$$\nabla_\theta J(\theta) \propto \mathbb{E}_{\pi_\theta}\{(q^{\pi_\theta}(s,a)-\zeta(s))\,\nabla \log \pi_\theta(a|s)\},$$

这里，$\zeta(s)$就是Baseline。引入baseline并不会改变我们的结果，因为上述梯度公式中含有baseline的部分的期望为0：

$$\begin{aligned}\mathbb{E}_{\pi_\theta}\{\zeta(s)\nabla \log \pi_\theta(a|s)\} &= \sum_s d(s)\sum_a \pi_\theta(a|s)\cdot\zeta(s)\frac{\nabla\pi_\theta(a|s)}{\pi_\theta(a|s)}\\ &= \sum_s d(s)\zeta(s)\nabla\sum_a \pi_\theta(a|s)\\ &= \sum_s d(s)\zeta(s)\nabla 1\\ &= 0.\end{aligned}$$

从上述推导中，也可看出$\zeta(s)$可以是任意一个和状态s有关的函数，**但是必须是一个与动作无关的函数**，否则在右一式到右二式的时候梯度符号就不会只加在$\sum \pi_\theta(a|s)$上了。那样最后的期望就不会为0了。

上面我们只是说明了Baseline选择必须满足的条件，那么满足上述条件的Baseline中有没有个好坏之分呢？当然有！我们引入Baseline的目的是什么？是为了减小方差，那么我们的Baseline应该尽可能的使得引入Baseline后的方差的减小量最大。方差减小量的公式如下：

$$\begin{aligned}\Delta D &= \mathbb{D}\{\nabla J_{\mathrm{BL}}\} - \mathbb{D}\{\nabla J\}\\ &= \mathbb{D}_{\pi_\theta}\{\big(q^{\pi_\theta}(s,a)-\zeta(s)\big)\nabla\log\pi_\theta\} - \mathbb{D}_{\pi_\theta}\{q^{\pi_\theta}(s,a)\nabla\log\pi_\theta\}\\ &= -\mathbb{E}_{\pi_\theta}\{(\nabla\log\pi_\theta)^2\}\mathbb{E}_{\pi_\theta}\{\big(2v^{\pi_\theta}(s)-\zeta(s)\big)\zeta(s)\}.\end{aligned}$$

从最后一式可以看出，$DeltaD$是$\zeta(s)$的二次函数，当$\zeta(s)=v^{\pi_\theta}(s)$时，$\Delta D$取得最值。此时，方差减小量的绝对值最大（因为$\Delta D$是一个负数，即$\Delta D$取最小值）：

$$\Delta\mathbb{D}_{\min} = -\mathbb{E}_{\pi_\theta}\{(\nabla\log\pi_\theta)^2\}\mathbb{E}_{\pi_\theta}\left\{\left(v^{\pi_\theta}(s)\right)^2\right\}.$$

因此，baseline函数的最佳选择就是状态值函数$v^{\pi_\theta}(s)$。使用状态值函数作为baseline的方法就是著名的Advantage Learning Architecture。在该框架下，我们令$A(s,a)=q(s,a)-v(s)$。这也可以解释为什么引入baseline之后的效果更好，因为之前我们$\nabla\log\pi_\theta$相乘的是$q(s,a)$，而现在是$q(s,a)-v(s)$，这样对于对数形式的policy gradient的不确定性的放大更小。但是，这里的设计还是比较naive的，因为我们需要同时关照状态值函数和动作值函数，这样就会导致更多的计算量。能不能只用其中一种呢？当然可以。详见下面的方法。

7.3.2.3 使用状态值函数来代替动作值函数

首先回顾以下动作值函数和状态值函数之间self-consistency的关系：

$$q^{\pi_\theta}(s,a) = \mathbb{E}_{s'\sim P}\{r+\gamma v^{\pi_\theta}(s')\}.$$

那么我么就可以得出使用$v^{\pi_\theta}(s)$来作为baseline的公式：

$$\begin{aligned}\nabla_\theta J(\theta) &\propto \mathbb{E}_{\pi_\theta}\left\{\left(\mathbb{E}_{s'\sim P}\{r+\gamma\nu^{\pi_\theta}(s')\}-\nu^{\pi_\theta}(s)\right)\nabla\log\pi_\theta(a|s)\right\}\\ &\propto \mathbb{E}_{\pi_\theta}\{(r+\gamma\nu^{\pi_\theta}(s')-\nu^{\pi_\theta}(s))\nabla\log\pi_\theta(a|s)\}.\end{aligned}$$

为什么从右一式可以直接省去$\mathbb{E}_{s'\sim P}$化到右二式呢？因为我们下面反正都是要利用sample进行估计，因此期望的下标无所谓。

在本质上来看，一个状态值函数反映了在当前状态下所有动作值函数的平均值。因此使用advantage function之后，我们就相当于可以提高高于平均值的动作的概率，降低低于平均值的动作的概率。有趣的是，这里的$A(s,a)=r+\gamma v(s')-v(s)$.形式与one-step TD error的形式是一样的。因此，我们就可以从两个角度来理解Advantage Learning Architecture：一方面，从baseline的角度，advantage function来自于对于方差的最大降低量的考虑；另一方面，从TD error的角度，advantage function来自于bootstrapping技术。那么一种自然而然的延申就在于可以使用多步的TD error或者TD(λ)来代替advantage function。这样能够增加估计的准确性、训练的稳定性和收敛的速度。

那么，加上baseline之后的究竟对于算法有什么好处呢？原书7.3.3节给出了一个例子来解释。这里使用的环境还是第六章最后的那个环形路上的自动驾驶车。观察图7.7可以印证我们上面说过的使用baseline的好处（训练稳定性、收敛速度等）。图7.8和7.9给出了达到稳态后的各个状态量的值。从图中可以看出，使用/不使用baseline的情况下最终值基本一样但有些微差别。

7.4 Off-Policy Gradient

只使用target policy收集数据对于策略的更新来说不是很高效。更严重的是，如果没有足够的数据来产生一个更好的策略，训练过程甚至会发散。平行的探索可以部分缓解这种现象，但是又会极大的增加计算负担。因此，我们需要利用历史收集的数据。这也就要求使用off-policy的方法来提升data efficiency。这方面比较有影响力的工作来自于Sutton等人在2012年提出的方法。这个工作之后又启发了一些off-policy的方法，比如off-policy DPG、ACER、IPG等。下面就介绍以下off-policy gradient的推导。

7.4.1 Off-Policy True Gradient

我们推导的起点来自于之前讲过的两个公式：

$$
\begin{aligned}
J(\theta) &= \mathbb{E}_{s\sim d_{\pi_\theta}}\{v^{\pi_\theta}(s)\} \\
\nabla_\theta J(\theta) &= \frac{1}{1-\gamma}\sum_s d^\gamma_{\pi_\theta}(s)\sum_a \nabla_\theta \pi_\theta(a|s) q^{\pi_\theta}(s,a)
\end{aligned}
$$

但是注意到上面的第二式是使用target policy进行sample的，因此我们需要使用behavior policy进行改写。因此就需要引入IS Ratio进行修正。修正后的公式如下（推导过程就先省略了）：

$$
\nabla_\theta J(\theta) = \mathbb{E}_{\mathcal{T}\sim b}\left\{\sum_{\tau=t}^{\infty}\nabla_\theta \log \pi_\theta(a_\tau|s_\tau)\left(\prod_{k=t}^{\tau}\frac{\pi_\theta(a_k|s_k)}{b(a_k|s_k)}\right)\gamma^{\tau-t}q^{\pi_\theta}(s_\tau,a_\tau)\right\}.
$$

这个梯度公式被称为Off-Policy True Gradient。但显然在实际中我们无法使用上述公式来构建算法，因为这个公式的计算量太大，而且由于IS Ratio的连乘导致了对于不确定性的放大而引起了高方差。换句话说，直接使用这个公式不仅计算十分费力而且估计的误差很大。下面就来介绍一种改进的方法。

7.4.2 Off-Policy Quasi-Gradient

为了解决上述问题，一个解决的办法是从源头入手，修改目标函数。我们可以将目标函数改写为：

$$
J(\theta) = \mathbb{E}_{s\sim d_b}\{v^{\pi_\theta}(s)\}.
$$

那么我们对于心得目标函数进行求梯度，得到下面的式子：

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{s\sim d_b}\left\{\sum_{a_t}[\nabla_\theta \pi_\theta(a_t|s_t)q^{\pi_\theta}(s_t,a_t) + \pi_\theta(a_t|s_t)\nabla_\theta q^{\pi_\theta}(s_t,a_t)]\right\} \\
&\propto \mathbb{E}_{s\sim d^\gamma_{\pi_\theta}(s|d_b),a\sim\pi_\theta}\{q^{\pi_\theta}(s,a)\nabla_\theta \log \pi_\theta(a|s)\},
\end{aligned}
$$

注意，从右一式到右二式与on-policy那里的推导类似，这里就不再赘述了。现在重点是$d^\gamma_{\pi_\theta}(s|d_b)$是个什么东西？我们之前讲过的discunted state distribution长这样：$d^\gamma_{\pi_\theta}(s) = \sum_{\tau=t}^{\infty}\gamma^{\tau-t}p(s_\tau = s|\pi_\theta)$。或者可以写成$d^\gamma_{\pi_\theta}(s|s_t)$。那么我们下面来推导一下这两个相似的东西之间的关系：

$$
\begin{aligned}
d^\gamma_{\pi_\theta}(s|d_b) &= (1-\gamma)\sum_{\tau=t}^{\infty}\gamma^{\tau-t}p(s_\tau = s|\pi_\theta, d_b) \\
&= (1-\gamma)\sum_{s_t} d_b(s_t)\sum_{\tau=t}^{\infty}\gamma^{\tau-t}p(s_\tau = s|\pi_\theta, s_t) \\
&= \sum_{s_t} d_b(s_t)\Big((1-\gamma)\sum_{\tau=t}^{\infty}\gamma^{\tau-t}p(s_\tau = s|\pi_\theta, s_t)\Big) \\
&= \sum_{s_t} d_b(s_t) d^\gamma_{\pi_\theta}(s|s_t).
\end{aligned}
$$

但是，上述化简仍然存在问题，就是$d^\gamma_{\pi_\theta}(s|s_t)$的计算包含大量$p(s_\tau = s|\pi_\theta, s_t)$项，这些项在我们没有足够多的sample时仍然是无法计算的。为了解决上述问题，一种简单而有效的方法如下。先回到我们在Off-Policy Quasi-Gradient中得到的式子：

$$
\nabla_\theta J(\theta) \propto \mathbb{E}_{s\sim d^\gamma_{\pi_\theta}(s|d_b),a\sim\pi_\theta}\{q^{\pi_\theta}(s,a)\nabla_\theta \log \pi_\theta(a|s)\},
$$

我们不妨把上式中的$d^\gamma_{\pi_\theta}(s|d_b)$直接替换为behavior policy下的SSD，那么就可以得到：

$$
\begin{aligned}
\nabla J(\theta) &\approx \sum_s d_b(s) \sum_a \nabla_\theta \pi_\theta(a|s) q^{\pi_\theta}(s,a) \\
&= \mathbb{E}_{s\sim d_b}\left\{ \sum_a b(a|s) \frac{\pi_\theta(a|s)}{b(a|s)} q^{\pi_\theta}(s,a) \frac{\nabla \pi_\theta(a|s)}{\pi_\theta(a|s)} \right\} \\
&= \mathbb{E}_{s\sim d_b}\left\{ \mathbb{E}_{a\sim b}\left\{ \frac{\pi_\theta(a|s)}{b(a|s)} q^{\pi_\theta}(s,a) \nabla \log \pi_\theta(a|s) \right\} \right\} \\
&= \mathbb{E}_{b}\left\{ \frac{\pi_\theta(a|s)}{b(a|s)} q^{\pi_\theta}(s,a) \nabla \log \pi_\theta(a|s) \right\}.
\end{aligned}
$$

最终得到的这个梯度式子就是被称为off-policy quasi-gradient的著名式子。注意到，其实上式中的$\frac{\pi_\theta(a|s)}{b(a|s)}$就是IS Ratio，只不过我们这里推导的时候不是从Importance Sampling的角度来看的。这样使用单步的IS Ratio来代替连乘的IS Ratio，就大大减小了计算量并降低了不确定性。

那么我们还可以继续给出上式的状态值函数版本：

$$
\begin{aligned}
\nabla J(\theta) &= \mathbb{E}_{s\sim d_b, a\sim b}\left\{ \frac{\pi_\theta(a|s)}{b(a|s)} \mathbb{E}_{s'\sim\mathcal{P}}\{r + \gamma v^{\pi_\theta}(s')\} \nabla \log \pi_\theta(a|s) \right\} \\
&= \mathbb{E}_{s\sim d_b, a\sim b, s'\sim\mathcal{P}}\left\{ \frac{\pi_\theta(a|s) p(s'|s,a)}{b(a|s) p(s'|s,a)} \big(r + \gamma v^{\pi_\theta}(s')\big) \nabla \log \pi_\theta(a|s) \right\} \\
&= \mathbb{E}_{b}\left\{ \frac{\pi_\theta(a|s)}{b(a|s)} \big(r + \gamma v^{\pi_\theta}(s')\big) \nabla \log \pi_\theta(a|s) \right\}
\end{aligned}
$$

注意，最后把期望的下标简写为b仅仅是一种省略的表示。在实际的算法中，我们使用下述公式来更新：

$$
\nabla J \propto \frac{1}{|\mathcal{D}_b|} \sum_{\mathcal{D}_b} \frac{\pi_\theta(a|s)}{b(a|s)} (r + \gamma v^{\pi_\theta}(s^{'})) \nabla \log \pi_\theta(a|s)
$$

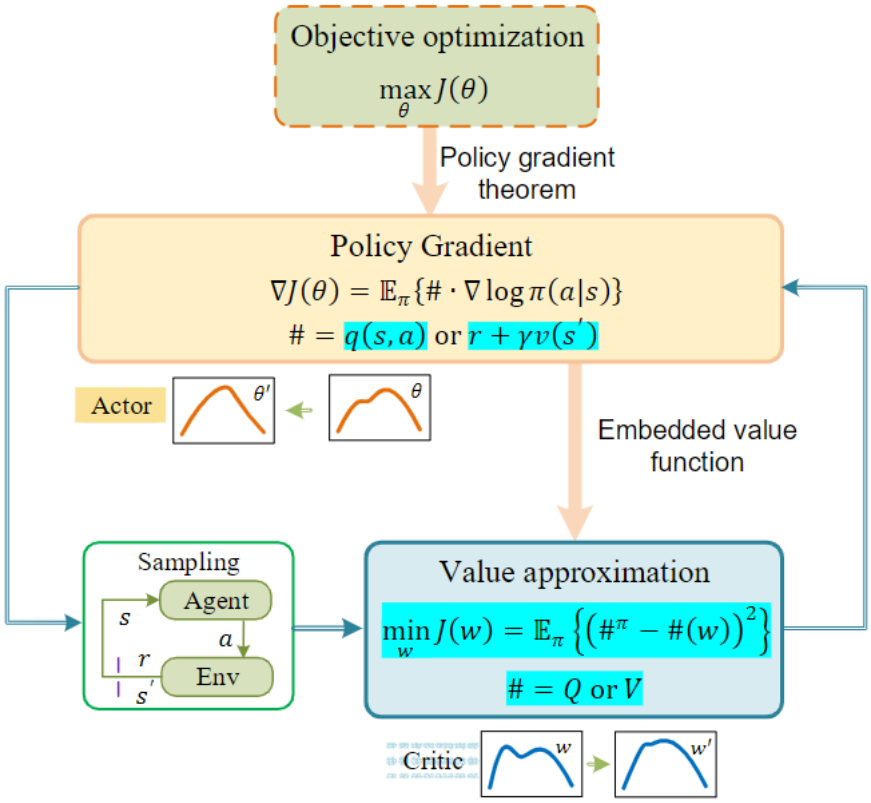
这里的$\mathcal{D}_b$是从behavior policy中采样得到的数据集。

另外，我们还可以仿照on-policy的方法，使用baseline来减小方差。

Off-Policy Quasi-Gradient的优点是很明显的，最大的好处就在于可以使用历史数据来进行训练，这尤其适用于我们的sample数目不多的时候。在现代的DRL中，利用历史数据的方法被称为Experience Replay，可以用来稳定训练过程以及利用多种来源的数据来进行训练，只要我们知道数据来源的分布。

7.5 从Direct RL角度看Actor-Critic架构

之前的博客我们已经介绍了从indirect RL角度如何看Actor-Critic架构。当时，critic被解释为PEV过程，通过critic的更新来更好的拟合值函数；actor被解释为PIM过程，通过actor的更新来寻找更好的策略。那么从Direct RL的角度来看，Actor-Critic架构又是什么呢？在Direct RL的视角下，actor可以被解释为基于policy gradient的更新过程，而critic则被解释为嵌入policy gradient估计中的值函数拟合。一个典型的Direct RL的Actor-Critic架构如下：



另外需要说明的是，对于Direct RL的Actor-Critic架构，收敛的保证不依赖于每次策略更新找到一个“更好”的策略，而依赖于所使用的数值优化方法的收敛性。下面来介绍几种常见的Direct RL的Actor-Critic算法。

7.5.1 off-policy A2C

A2C，即Advantage Actor-Critic，是一种利用了Advantage Function的Actor-Critic算法。Advantage Function在7.3.2.3节中已经介绍过了，也就是说我们把动作值函数使用状态值函数来表示并且将baseline设置为状态值函数。只不过那里的是on-policy的版本，这里我们将要介绍的是off-policy的版本。off-policy A2C的伪代码如下：

Hyperparameters: critic learning rate α, actor learning rate β, discount factor γ, number of environment resets M, episode length N, mini-batch size $|\mathcal{B}|$

Initialization: state-value function $V(s;w)$, policy function $\pi(a|s;\theta)$, memory buffer $\mathcal{D} \leftarrow \emptyset$

Repeat (indexed with k)

(1) Data collection

Repeat M environment resets

Generate an initial state $s_0 \sim d_{\text{init}}(s)$

For i in 0,1,2, … , N or until episode termination

$a_i \sim b(a|s_i;\theta)$

$p_i \leftarrow b(a_i|s_i;\theta)$

Apply a_i in environment and observe s_{i+1} and r_i

$\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_i, a_i, r_i, s_{i+1}, p_i)\}$

End

End

(2) Re-sampling from memory buffer

Randomly select a mini-batch $\mathcal{B} \subset \mathcal{D}$

Sweep mini-batch $\mathcal{B}$

$\rho_i \leftarrow \pi(a_i|s_i;\theta)/p_i, i \in \mathcal{B}$

End

(3) Critic update

$$\nabla_w J_{\text{Critic}} \leftarrow \frac{1}{|\mathcal{B}|}\sum_{\mathcal{B}} \rho \cdot \big(r + \gamma V(s';w) - V(s;w)\big)\frac{\partial V(s;w)}{\partial w}$$

$$w \leftarrow w - \alpha \cdot \nabla_w J_{\text{Critic}}$$

(4) Actor update

$$\nabla_\theta J_{\text{Actor}} \leftarrow \frac{1}{|\mathcal{B}|}\sum_{\mathcal{B}} \rho \cdot \nabla_\theta \log \pi(a|s;\theta)\big(r + \gamma V(s';w) - \zeta(s)\big)$$

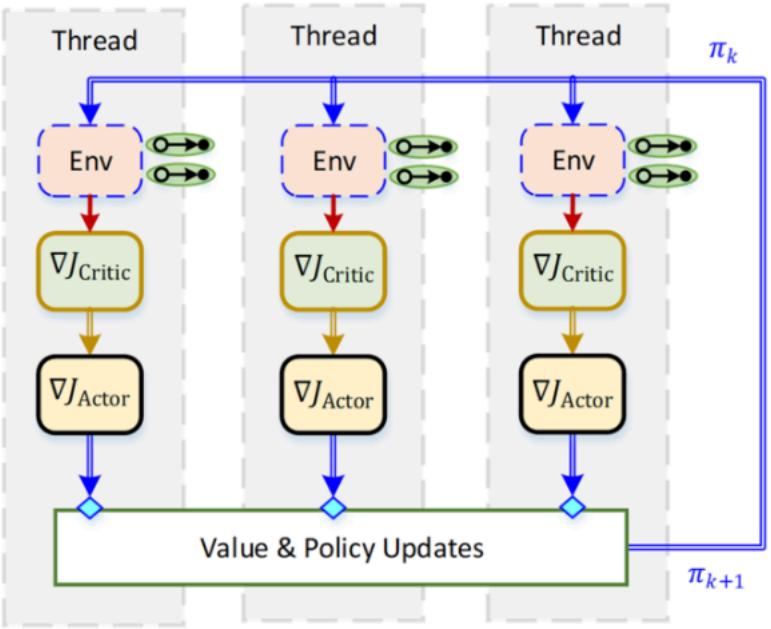
$$\theta \leftarrow \theta + \beta \cdot \nabla_\theta J_{\text{Actor}}$$

End

这里需要对于上述伪代码中的几个关键部分进行解释：

- **Data Collection**
  - a_i**和**p_i**的含义**：注意到这里的a_i与右边的连接符号是$\sim$，这表示a_i是从$b(a|s_i;\theta_b)$这个策略中采样得到的。而p_i与右边的连接符号为$\leftarrow$，这表示只是把右边那个概率值赋值给p_i，此时右边并不表示一个分布，只表示在策略b下在状态s_i下采取动作a_i的概率。
  - **为什么要计算**p_i**并将其一并存储进buffer中**：这是因为我们后面要使用p_i来计算IS Ratio。还有一个点不知道大家发现没有，就是这里的buffer $\mathcal{D}$只初始化（清空）一次。这也是因为只要我们为一个数据绑定了其p_i，之后就可以很方便的计算出IS Ratio，也就实现了利用历史数据的目的。
- **Re-sampling from memory buffer**：这里实际上就是从总的数据集中抽取一个mini-batch，并计算其中每个sample的IS Ratio。
- **Critics Update**：这里的critic更新使用的就是第六单元讲过的更新值函数的方法。
- **Actor Update**：利用的就是第7.3.2.3节讲过的公式，只不过这里还要乘上IS Ratio。

尽管使用了Experience Replay技术，但是上述算法仍然效率不高。一种解决办法是并行计算。其中一个著名的例子就是A3C算法——Asynchronous Advantage Actor-Critic。该算法由众多的thread组成，每个thread都有一个独立的sampler和learner。然后众多thread为训练出一个共同的值函数和策略而共同努力，但是对于这两个共同的参数的更新是异步的，即看哪个thread到了需要更新的时候就更新，不用等其他的thread。A3C的算法框架如下：



一开始的时候，A3C都是on-policy的，为了进一步提高样本效率，Google DeepMind提出了IMPALA算法，一个可以大规模并行并利用Importance Sampling技术的off-policy算法。该算法最主要的特点就是将sampler与learner解耦，这样sampler不必等learner更新完了再继续采样，而是可以一直采样，然后将采样的数据放入buffer中，等到learner需要更新的时候再从buffer中取出数据进行更新。这样就大大提高了样本效率。

7.5.2 off-policy DPG

使用Stochastic Policy Gradient的方法有很多好处，比如可以较好地探索环境等。但是，使用随机策略也带来了高方差等困境。为此，我们可以使用Deterministic Policy Gradient的方法。

首先，我们先定义目标函数：

$$J(\theta) \overset{\text{def}}{=} \mathbb{E}_{s\sim d(s)}\{q^{\pi_\theta}(s,\pi_\theta(s))\}.$$

注意，这里是通过动作值函数来推导DPG的。实际上，正如我们在第六章博客的第6.5.2.1节讲的那样，也可以先以状态值函数为目标函数，再通过两种值函数之间的self-consistency关系来推导DPG。但是此处我们就以动作值函数为例来推导吧。紧接着我们要来对目标函数求梯度了，注意这里求梯度需要满足几个约束（求梯度时遇到的所有函数都是连续且有界的）。那么我们可以得到下面的式子：

$$\nabla J(\theta) \approx \mathbb{E}_{s\sim d_b/s\sim d_\pi}\{\nabla_\theta \pi_\theta(s) \nabla_a q^{\pi_\theta}(s,a)|_{a=\pi_\theta(s)}\}.$$

注意，这里期望的下标$s\sim d_b/s\sim d_\pi$用于区分on/off-policy。如果是on-policy的话，那下标就是$s\sim d_\pi$；如果是off-policy的话，那下标就是$s\sim d_b$。注意，这里的梯度公式里面最重要的一点就是**不显含IS Ratio，对于on-policy和off-policy的梯度公式都是一样的**。这点可以这样简单理解，即IS Ratio是在两个分布之间进行转换的，然而，我们此处的情况是即使你用了off-policy，即使你的behavior policy是一个分布，但是你的target policy是一个确定性的策略而不是一个状态分布，因此不存在在两个分布之间进行转换的问题。最后，注意到上述公式是以≈进行连接的，那么什么时候可以用等号呢？有两种情况：

- $d(s)$**严格等于**$d_b(s)$或$d_\pi(s)$
- $\gamma \to 1$

从计算量的角度来看，DPG只在状态空间进行积分/求和。而在随机版本中，需要同时在状态空间和动作空间进行积分/求和。因此，DPG的计算量要小很多。另外注意的是，尽管DPG具有种种好处，然而其“确定性”也带来了对于环境的探索不充足的问题。因此，我们一般不使用样本效率低且探索不充足的on-policy版本，而是使用off-policy版本。因为off-policy版本可以利用历史数据且能利用一个随机的behavior policy来进行探索。下面给出off-policy DPG的伪代码：

Hyperparameters: critic learning rate α, actor learning rate β, discount factor γ, number of environment resets M, episode length N, mini-batch size $|\mathcal{B}|$

Initialization: action-value function $Q(s,a;w)$, policy function $\pi(s;\theta)$, and memory buffer $\mathcal{D} \leftarrow \emptyset$

Repeat (indexed with k)

(1) Data collection

Repeat M environment resets

Generate an initial state $s_0 \sim d_{\text{init}}(s)$

For i in 0,1,2, … , N or until episode termination

$a_i \sim b(a|s_i;\theta)$

Apply a_i in environment and observe next state s_{i+1} and reward r_i

$\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_i, a_i, r_i, s_{i+1})\}$

End

End

(2) Re-sampling from memory buffer

Randomly select a mini-batch $\mathcal{B} \subset \mathcal{D}$

(3) Critic update

$a' = \pi(s';\theta)$

$$\nabla_w J_{\text{Critic}} \leftarrow \frac{1}{|\mathcal{B}|}\sum_{\mathcal{B}}\left(r + \gamma Q(s', a'; w) - Q(s, a; w)\right)\frac{\partial Q(s,a;w)}{\partial w}$$

$w \leftarrow w - \alpha \cdot \nabla_w J_{\text{Critic}}$

(4) Actor update

$$\nabla_\theta J_{\text{Actor}} \leftarrow \frac{1}{|\mathcal{B}|}\sum_{\mathcal{B}} \nabla_\theta \pi(s;\theta) \nabla_a Q(s,a;w)$$

$\theta \leftarrow \theta + \beta \cdot \nabla_\theta J_{\text{Actor}}$

End

这里需要注意几点：

-**不需要显式的计算和存储IS Ratio**：原因在上面已经讲过了。

-**Critics Update**：使用的还是第六章讲过的更新参数化的值函数的方法。注意这里计算梯度公式的上一行那个$a' = \pi(s';\theta)$我的理解是对于mini-batch $\mathcal{B}$中的每一个sample，我们都要根据当前策略以及sample中储存的状态来计算下一步的动作a'，因为在下面的梯度公式中会用到。

7.5.3 AC算法汇总

下面用一点篇幅简要总结一下AC算法的发展历程。

早在1983年，Barto、Sutton和Anderson等人提出的算法可能是AC算法最早的版本。在这个工作中，actor和critic分别被称为associative search elements和“adaptive critic elements。这两个部分的功能与现代的AC算法中的actor和critic是几乎一模一样的。之后，AC算法逐步发展。普通的AC算法都有两个核心组件：actor和critic。但是，在众多的AC算法中，也出现了一些极端的版本：actor-only架构和critic-only架构。前者的代表是我们在本篇博客前面部分说过的REINFORCE算法，后者的代表是Q-learning算法。

时至今日，AC架构已经成为强化学习领域最著名的架构之一了，也发展出了许多变种。典型的算法包括DDPG（Deep Deterministic Policy Gradient）、A2C（Advantage Actor-Critic）、A3C（Asynchronous Advantage Actor-Critic）、SACh（Soft Actor-Critic）DSAC（Distributional Soft Actor-Critic）、TD3（Twin Delayed Deep Deterministic Policy Gradient）TRPO（Trust Region Policy Optimization）PPO（Proximal Policy Optimization）等等。下面的表格从五个维度对这些算法进行了对比，分别是policy type、value function type、critic update mechanism、policy update mechanism和on/off-policy。

Algorithm	Policy	Value	Critic Update	Actor Update	On/Off

DDPG	D	Q	TD-based	Vanilla PG	Off
TRPO	S	V	TD-based	Natural PG	On
PPO	S	V	TD-based	Clipped PG	On
TD3	D	Q	Clipped Double Q	Vanilla PG	Off
D4PG	D	Q	Discrete Distributional Q-TD	Vanilla PG	Off
ACKTR	S	V	TD-based	Natural PG	On
A2C/A3C	S	V	TD-based	Vanilla PG	On
Off-PAC	S	V	TD-based	Vanilla PG	Off
ACER	S	Q	TD-based	Vanilla PG	Off
IMPALA	S	V	TD based	Vanilla PG	Off
Soft Q-learning	S	Q	Soft Q-iteration	Soft PG	Off
SAC	S	Q	Clipped Double-Q	Soft PG	Off
DSAC	S	Q	Continuous Distributional Q-TD	Soft PG	Off

在不少的AC算法的更新公式中，会使用baseline，以至于advantage function。使用这项技术可以有效地降低方差。正如我们在本篇博客靠前的部分提到的，advantage function的形式实际上与one-step TD error的形式是一样的。那么我们自然可以想到，能不能使用多步的TD error或者TD(λ)来代替advantage function呢？答案是肯定的。先来回顾一下我们在第四单元博客里讲过的n-step TD error的形式：

$$\delta_V^{\mathrm{TD}(n)}(s) \stackrel{\mathrm{def}}{=} \underbrace{G_{t:t+n-1} + \gamma^n V^\pi(s_{t+n})}_{n-\text{step TD target}} - V^\pi(s_t).$$

下面的表格就汇总了一些**更新actor的方法（即不同的policy gradient的形式）**，里面那些基于状态值函数的方法的推导思路就是本来是状态值函数，然后引入self-consistency关系以及advantage function的概念，最后再由one-step TD error的角度拓展到n-step TD error：

	Stochastic	Deterministic
V	$\nabla J_{\mathrm{Actor}} = \mathbb{E}_\pi\left\{\delta_V^{\mathrm{TD}(n)}(s)\nabla_\theta \log \pi_\theta(a\|s)\right\}$	
Q	$\nabla J_{\mathrm{Actor}} = \mathbb{E}_{s\sim d_\pi, a\sim\pi}\{Q(s,a)\nabla_\theta \log \pi_\theta(a\|s)\}$	$\nabla J_{\mathrm{Actor}} = \mathbb{E}_{s\sim d_\pi}\{\nabla_\theta \pi_\theta(s)\nabla_a Q(s,a)\}$

	Stochastic	Deterministic
V	$\nabla J_{\mathrm{Actor}} = \mathbb{E}_b\left\{\rho_{t:t+n-1}\delta_V^{\mathrm{TD}(n)}\nabla_\theta \log \pi_\theta(a\|s)\right\}$	
Q	$\nabla J_{\mathrm{Actor}} = \mathbb{E}_{s\sim d_b, a\sim b}\left\{\frac{\pi_\theta(a\|s)}{b(a\|s)}Q(s,a)\nabla_\theta \log \pi_\theta(a\|s)\right\}$	$\nabla J_{\mathrm{Actor}} = \mathbb{E}_{s\sim d_b}\{\nabla_\theta \pi_\theta(s)\nabla_a Q(s,a)\}$

注意里面off-policy的方法除了off-policy加上deterministic之外的情况都要使用IS Ratio做转换。还要注意到，在deterministic的情况下，梯度中是对于$\pi_\theta(s)$而不是对于$\log \pi_\theta(a|s)$求梯度。

讨论完actor的更新之后就要来说说critic的更新了。下表就给出了利用n-step TD error来进行**critic的更新**的公式：

	On-policy	Off-policy
V	$J_{\text{Critic}} = \mathbb{E}_s\left\{\left(R^{(n)} - V(s_t;w)\right)^2\right\}$ $R^{(n)} = G_{t:t+n-1} + \gamma^n V(s_{t+n};w)$	$J_{\text{Critic}} = \mathbb{E}_s\left\{\left(\rho_{t:t+n-1}R^{(n)} - V(s_t;w)\right)^2\right\}$ $R^{(n)} = G_{t:t+n-1} + \gamma^n V(s_{t+n};w)$
Q	$J_{\text{Critic}} = \mathbb{E}_{s,a}\left\{\left(R^{(n)} - Q(s_t,a_t;w)\right)^2\right\}$ $R^{(n)} = G_{t:t+n-1} + \gamma^n Q(s_{t+n},a_{t+n};w)$	$J_{\text{Critic}} = \mathbb{E}_{s,a}\left\{\left(\rho_{t+1:t+n-1}R^{(n)} - Q(s_t,a_t;w)\right)^2\right\}$ $R^{(n)} = G_{t:t+n-1} + \gamma^n Q(s_{t+n},a_{t+n};w)$

还有一点值得注意的是，在使用这种基于TD的梯度更新时，通常更常使用semi-gradient而不是true gradient，即我们在求梯度时虽然TD error也含有要更新的参数但是我们却并不对其求梯度而是把它当成一个常数。这样做的理由很简单，如果求true gradient的话，因为函数形式更加复杂，也就意味着可能出现更多的局部极小，而且由于随机优化的性质，当达到最优值之后停在那里也并不容易。

除了上述的TD-based的critic和actor的更新方法，还有一些其它的方法。比如说，clipped double-Q critic、discrete distributional Q-TD、continuous distributional Q-TD等。

7.6 Direct RL中的一些问题

众所周知，Direct RL问题可以被视为一种随机优化问题。本节就将从优化问题的视角，来讨论其中遇到的一些问题。本节将从以下三个方面展开：优化技巧、优化方法、目标函数。
下面先来放几张表格来作为本节内容的总览：

Table 7.5 Direct RL with a stochastic policy

		Derivative-free optimization	First-order optimization	Second-order optimization
Objective function	Primal	--	Vanilla PG#	--
	MM optimization	--	Natural PG# (TRPO)	--
	Entropy regularization	--	Soft PG&	--

#- Log-derivative trick; &- reparameterization trick

Table 7.6 Direct RL with deterministic policy

		Derivative-free optimization	First-order optimization	Second-order optimization
Objective function	Primal	FD; EA	Deterministic PG	--
	MM optimization	--	--	--
	Entropy regularization	--	--	--

* FD-Finite difference; EA-evolutionary algorithm

7.6.1 随机优化里的优化技巧

在RL中，一个常见的问题是计算对于一个损失函数的梯度。让我们来考虑哟个随机变量z，其服从分布$p_\theta(z)$。给定一个z的函数$h(z)$，我们希望计算$h(z)$的expected gradient：

$$\nabla_\theta \mathbb{E}_{z\sim p_\theta}\{h(z)\} = \sum_z \nabla_\theta p_\theta(z) h(z).$$

显然，右式（离散的情况为求和，连续的情况为积分）不是一个期望的形式（求导之前$\mathcal{E}_{z\sim p_\theta}\{h(z)\}$确实是一个期望的形式，但是求导之后就不是了，因为对于原本用于加权的概率/概率密度求导了，因此不能保证新的系数$\nabla_\theta p_\theta(z)$仍然是一个分布）。可能大家会有疑问，为什么一定要化成梯度的形似才能计算呢？因为对于model-free的RL来说，我们手头有的只是与环境交互得到的samples。这其实就是一个使用样本来估计统计量的统计问题。而只有将统计量化为期望的形式才比较好估计。因此，我们下面就来介绍一些常见的技巧来将上述的式子化为期望的形式。

7.6.1.1 Log-derivative trick

这个其实我们早就说过了。这里在理解了为什么期望形式对于基于sample的估计这么重要之后，我们再来重新审视一下这个技巧的本质。还是针对我们上面使用z的函数$h(z)$举的例子，在应用了后，有：

$$\nabla_\theta \mathbb{E}_{z\sim p_\theta}\{h(z)\} = \sum_z p_\theta(z) \frac{\nabla_\theta p_\theta(z)}{p_\theta(z)} h(z) = \mathbb{E}_{z\sim p_\theta}\{\nabla_\theta \log p_\theta(z) h(z)\}.$$

这个技巧其实就是利用了复合函数求导的规则：

$$\nabla_\theta \log p_\theta(z) = \frac{\nabla_\theta p_\theta(z)}{p_\theta(z)}.$$

将原本前面已经有梯度符号的$p_\theta(z)$（此时可能已经不是一个分布了）换出来换到前面，那么就可以重新得到一个期望的形式。这个技巧在RL中的应用非常广泛，其优缺点列举如下：

- **优点**
  - 将非期望形式化为期望形式，方便使用sample进行估计。
  - 对于$h(z)$没有要求，也不要求其可导。
- **缺点**
  - 因为吧$p_\theta(z)$换到了分母上，可能会引起高方差和数值不稳定（对于那些概率很小的z来说）。

7.6.1.2 Reparameterization trick

Reparameterization trick是另一种将非期望形式化为期望形式的技巧。这个技巧的核心思想在于，**将一个复杂的随机变量使用一个确定性的变换（结构已知）和一个简单的随机变量的结合来表示**。比如，我们可以对于z进行重参数化：

$$z\sim p_\theta \rightarrow z = g_\theta(\epsilon), \epsilon\sim p(\epsilon),$$

这里，$g_\theta(\cdot)$是一个确定性的函数，其表达式已知；ϵ是一个简单的随机变量，其分布$p(\epsilon)$也是已知的。那么，我们就可以将原本的期望形式化为：

$$\nabla_\theta \mathbb{E}_{z\sim p_\theta}\{h(z)\} = \nabla_\theta \mathbb{E}_{\epsilon\sim p(\epsilon)}\{h(g_\theta(\epsilon))\} = \mathbb{E}_{\epsilon\sim p(\epsilon)}\{\nabla_\theta h(g_\theta(\epsilon))\}.$$

这里，从上面的一式到中间那个式子，就是对于期望的下标进行了简单的替换。但是这个替换却因为我们是对于θ而不是ϵ求导而至关重要！因为我们是对于θ求导，所以期望的形式始终不会改变（求偏导也求不到ϵ头上去）。这个技巧的优缺点如下：

- **优点**
  - 方差比log-derivative trick要小。这可能是因为我们上面通过reparameterization trick将一个复杂的随机变量转化为了一个简单的随机变量和一个确定性的函数的组合，这样就将原来的stochastic policy gradient中的随机和确定部分分开。也可以说是因为我们选取的结构已知的函数$g_\theta(\cdot)$将一定的确定性引入到了原本的随机过程中。
- **缺点**
  - 有时候可能会比较难找到合适的$g_\theta(\cdot)$，它的选取是一个大问题。如果不能很好的贴近真实的分布效果可能就不太好。

7.6.2 随机优化里的优化方法

这里主要介绍三种优化方法：Derivative-free方法、First-order方法和Second-order方法。

7.6.2.1 Derivative-free方法（DFO）

Derivative-free方法是一种不需要计算梯度的优化方法，也被称为零阶优化或者黑盒优化。这种方法的优点在于不需要计算梯度，计算量小，且因为不需要计算梯度而可以处理一些非凸、不可微、ill-conditioned的问题。

如何把这类方法应用于RL中呢？一般是忽略问题的MDP特性，而直接在参数空间中进行优化。首先，我们需要定义一个目标函数；然后，每次更新的时候需要随机在参数空间中进行某种微小的扰动；然后在此基础上进行更新。下面介绍两种常见的DFO方法：Finite Difference（FD）Evolution Strategies（ES）。

先来看看FD。这里只简介FD最简单的版本。该版本的扰动方向是这样选择的：假设我们的参数为$\theta \in \mathbb{R}^n$，那么我们此处就有n个扰动方向（对应于n个维度），每个扰动方向都是$\mathbb{R}^n$中的一个单位向量，且这n个方向两两之间正交（即这些方向沿着n维空间的n个坐标轴）。那么，最后的更新公式就是：

$$J(\theta) \leftarrow J(\theta) + \widehat{\nabla J}(\theta)$$

其中，$\widehat{\nabla J}(\theta)$的计算公式为：

$$\widehat{\nabla J}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \frac{J(\theta + \sigma \epsilon_i) - J(\theta)}{\sigma} \epsilon_i,$$

这里，σ是一个很小的数（标量），$\epsilon_i \in \mathbb{R}^n$是一个单位向量。但是，显而易见的，使用这个方法要求我们参数空间有多少个维度，就要引入多少个扰动方向，这样该方法的scalability就很差，在参数空间维度很大的时候，计算量会很大。

接下来是ES。ES引入扰动的方式与FD不同，它的扰动是从一个分布中采样得到的。这个分布一般是一个高斯分布。其具体推导这里就不放了，只给出最后的公式：

$$\widehat{\nabla J}(\theta) \approx \frac{1}{m\sigma} \sum_{i=1}^{m} \big(J(\theta + \sigma \epsilon_i) - J(\theta)\big) \epsilon_i,$$

这里，$\epsilon_i \sim \mathcal{N}(0, I_{n \times n})$，$m$是采样的个数。而$\sigma$是一个超参数，是一个定值。

DFO方法的一些更高级的改进方法能够实现参数的自适应，可以自动地调整高斯分布的均值和方差。这些方法包括CEM（Cross-Entropy Method）、CMA-ES（Covariance Matrix Adaptation Evolution Strategies）等。

7.6.2.2 First-order方法（一阶优化）

一阶优化是大规模的RL问题中最常见的优化方法。这其中SGD（Stochastic Gradient Descent）是最常见的一种。最基本的SGD一次只使用一个样本来计算梯度，但是这样会有很大的随机性。为了在计算效率和稳定性之间取得平衡，我们可以使用mini-batch SGD。mini-batch SGD为了取得良好的效果，需要对几个参数进行调参，比如batch size、learning rate等。但是，SGD容易陷入局部最优或鞍点。因此，后续学术界提出了很多的改进方法，比如Momentum-SGD、RMSprop、Adam等。相比于只使用当前步的梯度，Momentum-SGD法都引入了之前的梯度信息来决定更新的方向和步长。RMSprop可以自动的调节学习率并且为每个不同的参数单独选取一个学习率。Adam结合了Momentum-SGD和RMSprop的优点，是目前深度学习中最流行的方法。

7.6.2.3 Second-order方法（二阶优化）

一个比较著名的二阶优化方法是Newton-Raphson法。该方法具有二阶的收敛速度。我们可以这样理解这个方法：如何使用一个截断的泰勒展开式来逼近原始的目标函数并连续的求解QP（Quadratic Programming）问题。每次迭代可以看成求解一个这样的QP问题：

$$\max_{\Delta\theta} g^{\mathrm{T}} \Delta\theta + \frac{1}{2} \Delta\theta^{\mathrm{T}} F \Delta\theta,$$

这里，$\Delta\theta$是参数的更新量，$g = \nabla_\theta J(\theta)$是梯度，$F = \nabla_\theta^2 J(\theta)$是Hessian矩阵。这里的QP问题实际上具有一个解析解：

$$\Delta\theta^* = F^{-1} g = [\nabla_\theta^2 J(\theta)]^{-1} \nabla_\theta J(\theta).$$

求出上述更新量的解析解之后，我们就可以更新参数了：

$$\theta \leftarrow \theta + \Delta\theta^*.$$

这里主要的困难在于如何准确且高效地计算Hessian矩阵。这里结合问题的特点，使用类似于我们在7.2节Likelihood Ratio Gradient里面讲过的推导一阶导数（梯度）的方法，可以得出Hessian矩阵的一种计算方法：

$$\nabla_\theta^2 J(\theta) \approx \sum_s d_\pi(s) \sum_a \nabla_\theta^2 \pi_\theta(a|s) q^{\pi_\theta}(s,a).$$

尽管如此，此时的计算量对于高维问题来说仍然不小。在一些工作中使用了BFGS（Broyden-Fletcher-Goldfarb-Shanno）方法来近似Hessian矩阵。但是，经过实验验证这种改进对于深度学习等大规模问题并没有太大的改进，因此没有被广泛接受。

7.6.3 目标函数的变体

在RL中，除了我们在本篇博客最开始提到过的那种原始的目标函数之外，还可以在目标函数的构造上下功夫，以提高训练的稳定性和更好的探索能力。下面介绍几种常见的目标函数的变体。

7.6.3.1 Surrogate function optimization

大规模的RL问题中，训练的稳定性是一个很大的问题。Minorize-Maximization（MM）算法可以用来处理这个问题，它能保证策略的改进是单调的。MM算法的核心思想在于不断优化一个surrogate function，而不是原函数。Surrogate function是原函数的一个**下界**。可以证明，如果我们不断优化surrogate function，那么对于原函数来说，会不断改进，至少不会变得更差。因此关键问题在于如何合理的构建一个surrogate function。下面我们来简单介绍一下这种算法的一个典型代表：TRPO（Trust Region Policy Optimization）。

TRPO（Trust Region Policy Optimization）算法是由Schulman等人在2016年提出的。从其名字中的trust region可以看出，这个算法在更新时限制了更新的幅度。首先，我们先来构造一个surrogate function：

$$
\begin{aligned}
S(\pi) &= L_{\pi_{\text{old}}}(\pi) - C \cdot D_{\text{KL}}^{\max}(\pi_{\text{old}}, \pi) \\
C &= 2\epsilon\gamma/(1-\gamma)^2 \\
L_{\pi_{\text{old}}}(\pi) &= J(\pi_{\text{old}}) + \sum_s d_{\pi_{\text{old}}}^{\gamma}(s) \sum_a \pi(a|s) A^{\pi_{\text{old}}}(s,a).
\end{aligned}
$$

这里的$S(\pi)$就是我们的surrogate function，它由两个部分组成。第一部分（$L_{\pi_{\text{old}}}(\pi)$）是对于$J(\pi)$的一个local approximate function，第二部分是一个KL散度的惩罚项。这个惩罚项的作用在于保证更新的幅度不会太大。$L_{\pi_{\text{old}}}(\pi)$的计算公式中的$A^{\pi_{\text{old}}}(s,a)$是advantage function，$d_{\pi_{\text{old}}}^{\gamma}(s)$是discounted state distribution。那么我们就将原始的优化问题转化为了下述关于surrogate function的优化问题：

$$
\max_{\pi} \{ L_{\pi_{\text{old}}}(\pi) - C \cdot D_{\text{KL}}^{\max}(\pi_{\text{old}}, \pi) \}.
$$

上式关于策略参数的一阶导数就是著名的natural policy gradient。与naive policy gradient相比，natural policy gradient的优点在于其可以保证策略的更新是单调的且更新步长被限制在一个合理的范围内。

7.6.3.2 Entropy Regularization

按照我们之前的目标函数训练出来的RL算法倾向于采取保守的动作而不是更多的探索环境。这些算法倾向于采用之前的经验，而不是去探索新的动作。因此，这种RL算法容易陷入局部最优而不是全局最优。为了解决这个问题，我们可以在目标函数中加入一个熵正则项。策略熵（policy entropy）是一个衡量策略的随机性的指标。一个策略越随机，其熵就越大。

加入了熵正则项的目标函数主要有以下两种形式：

- **每步都加入熵正则项**：这种形式的目标函数是在每一步都加入熵正则项，即：

$$
J_{\mathcal{H}}(\pi) = \mathbb{E}_{s_t \sim d(s_t)} \left\{ \sum_{i=t}^{\infty} \gamma^{i-t} \left(r_i + \alpha \mathcal{H}\left(\pi(\cdot|s_i)\right) \right) \right\},
$$

  这个里面对于每一步都加入了一个熵正则项，策略熵里的状态时每步的状态s_i。
- **只包含初始状态的熵正则项**：这种形式的目标函数是只在初始状态加入熵正则项，即：

$$
J_{\mathcal{H}}(\pi) = \mathbb{E}_{s_t \sim d(s_t)} \left\{ \mathbb{E}_{\pi} \left\{ \sum_{i=t}^{\infty} \gamma^{i-t} r_i \right\} + \alpha \mathcal{H}\left(\pi(\cdot|s_t)\right) \right\},
$$

  这个里面的正则项只对于初始状态s_t进行惩罚。

这里来解释一下为什么加入熵正则项可以提高探索性。可以看出，这里熵正则项前面的符号是加号，因此该正则项越大，目标函数就会越大。而我们的优化目标恰恰就是最大化。因此在优化的过程中，必然会鼓励采取熵更大的策略，即策略更加随机，探索性更强。

还需要指出的是，细心的读者可能已经注意到，我们之前的博客似乎也介绍过一种很相像的熵函数。但是那里与这里讲的熵正则项有所不同。那里的熵函数只是在actor的损失函数种加入，是在已经更新好了值函数的情况下对于PIM的ϵ-greedy策略进行优化，是Indirect RL的PIM的一种改进。而这里的熵正则项是直接在目标函数中加入的，是直接在参数空间种优化策略时增加随机性的一种方法，是一种Direct RL的改进。另外，从收敛性来看，在PIM中加入熵函数不一定还能保证收敛；而在Direct RL中加入熵正则项不会改变收敛性。

@[toc]

书籍链接：Reinforcement Learning for Sequential Decision and Optimal Control

本篇博客讲述ADP（Approximate Dynamic Programming）。ADP是最优控制与RL的集合。它适用于处理这样的问题：问题本身被建模为最优控制问题，**系统的环境模型已知**，但是如果直接使用最优控制的receding horizon control方法，需要在线计算，计算量大。因此，ADP尝试使用offline learning的方法来学习一个近似的最优控制器。ADP主要有两种主要类型，值迭代的ADP和策略迭代的ADP。

ADP的概念最早由Bellman等人提出，用于解决exact DP遭遇的维度灾难问题。二十世纪七十年代后期，Werbos提出将连续状态空间参数化，在参数空间中重新构建ADP问题。在二十世纪八十年代早期，Barto等人开始使用神经网络来近似ADP的actor和critic。在九十年代早期，基于模型的ADP与model-free的RL之间的联系逐渐变得明朗起来，ADP也因此取得了很大的发展，逐渐成为沟通最优控制与RL的桥梁。现在，ADP特指一种**在确定性环境下工作，基于模型的RL方法**，可以用来求解在非线性和有约束的环境中复杂的最优控制问题；可以实现较好的实时控制。

8.1 离散时间、无限视野（Discrete-Time, Infinite-Horizon）的ADP

现在的控制问题基本上都是使用数字控制器，也就说是在离散的时间域内进行的。然而，现实中的物理系统基本上都是连续时间的，因此需要对于连续时间的系统进行离散化，构建离散时间的最优控制问题。Consistent approximation理论提供了关于离散化之后的系统收敛到原连续时间系统的最优解的理论保证。

一个可解的**最优控制问题**（OCP）必须具有以下两个特点：

- **环境模型具有马尔科夫性质**
- **代价函数对于每个时间步是可分离的**

那么，对于ADP来说，其确定性的环境模型在离散化之后自然满足马尔科夫性质。而这里我们的代价函数是由每步的utility function的和构成的，因此也是可分离的。我们可以这样构建对应于离散时间、无限视野的ADP问题：

> **定义1**：离散时间、无限视野的ADP问题是这样的一个最优控制问题：
>
> $$\min_{\{u_t,u_{t+1},\cdots,u_\infty\}} V(x)=\sum_{i=0}^{\infty} l(x_{t+i},u_{t+i}),$$
> $$\text{s.t.}$$
> $$x_{t+1}=f(x_t,u_t),$$

这里的$u \in \mathcal{U} \subset \mathbb{R}^m$代表动作（即之前RL里的a，这里最优控制与RL的记号系统略有不同），$x \in \mathcal{X} \subset \mathbb{R}^n$代表状态（即之前RL里的s）。$f(x,u)$是环境模型，它**必须是确定性的**。$l(x,u) \geq 0$是utility function（或者按照R了的习惯，称为reward signal）。$V(x)$由各步的utility function的加和构成，被称为代价函数（或者按照RL的习惯，称为状态值函数）。另外，还需满足以下的假设：

> **假设1**：$f(x,u)$只在原点有一个平衡态，即$f(0,0)=0$，且$\partial f(x,u)/\partial u$必须可以获得。
> **假设2**：$l(x,u)$**正定**，即除了在原点处$l(0,0)=0$外，$l(x,u)>0$。且$\partial l(x,u)/\partial u$、$\partial l(x,u)/\partial x$必须可以获得。

注意，实际上不一定系统只能有一个平衡态，但是我们作上述假设可以简化问题。对于有多个平衡态的系统，它在各个平衡态附近也可以看成只有一个平衡态。

离散的时间步的一个重要好处是可以递归的进行计算，从而促进了self-consistency条件和Bellman方程的数值计算。

8.1.1 Bellman方程

读者可以发现，我们在本章对于问题的建模与符号体系是根据最优控制的习惯来的。因此，在本节的讨论中，我们也应该先提出对应于最优控制语境的self-consistency条件和Bellman方程。

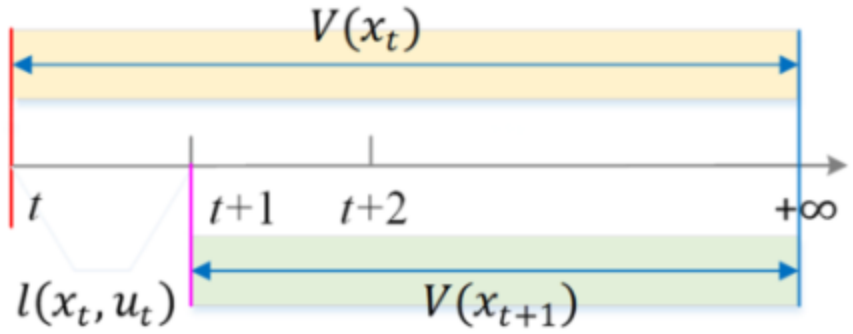
首先来看看self-consistency条件：

$$V(x) = l(x,u) + V(x'), \forall x \in X.$$

这里我们再把RL里的self-consistency条件放出来对比一下：

$$v^{\pi}(s) = \mathbb{E}_{\pi}\left\{r + \gamma v^{\pi}(s') | s\right\} = \sum_{a \in s^a} \pi(a|s) \left\{ \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a) \left(r^a_{ss'} + \gamma v^{\pi}(s')\right) \right\}$$

可以看出，两者本质是一样的。形式上的区别在于在此处我们采用的是确定策略、确定环境，因此也自然不存在$\pi(a|s)$和$\mathcal{P}(s'|s,a)$了，也自然没有求和式了。本处的self-consistency条件的图示如下：



接下来，在引入Bellman方程之前，需要现对于最优值函数进行定义：

$$V^*(x) \overset{\text{def}}{=} V^{\pi^*}(x) = \min_{\{u_t, u_{t+1}, \cdots, u_\infty\}} V(x),$$

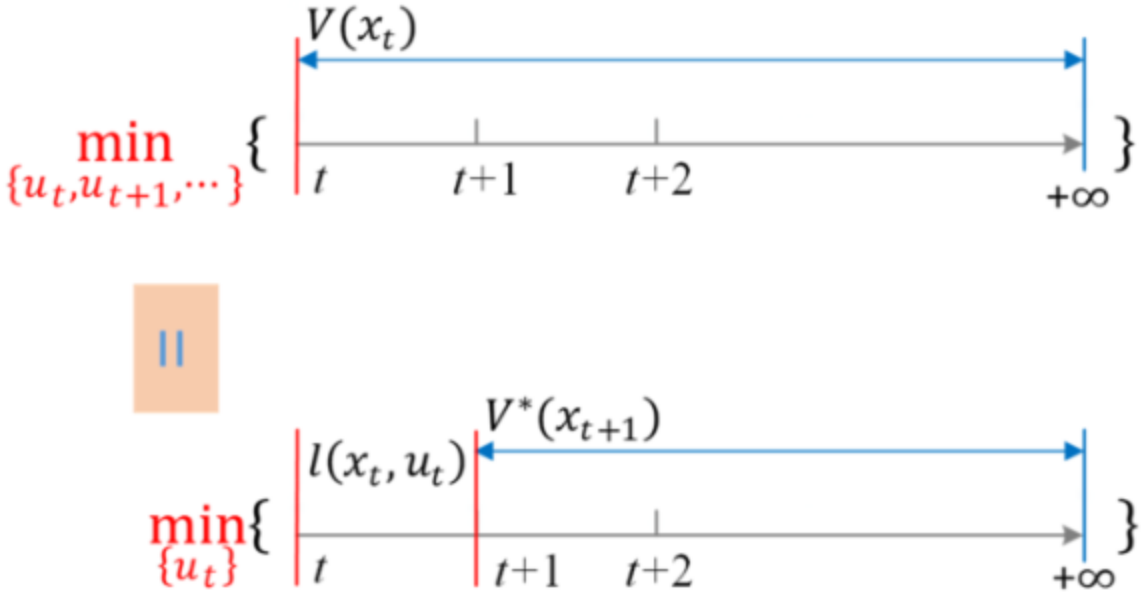
这里$\{u_t, u_{t+1}, \cdots, u_\infty\}$从时刻t开始的控制序列。Bellman方程的形式如下：

$$\begin{aligned} V^*(x) &= \min_{\{u_t, u_{t+1}, \cdots, u_\infty\}} \{l(x_t, u_t) + V(f(x_t, u_t))\} \\ &= \min_{u_t} \{l(x_t, u_t) + V^*(f(x_t, u_t))\} \\ &= \min_{u} \{l(x, u) + V^*(f(x, u))\}. \end{aligned}$$

而我们之前在RL中提到的Bellman方程是这样的：

$$v^*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s, a) \left(r^a_{ss'} + \gamma v^*(s')\right), \forall s \in \mathcal{S}.$$

两者对比，我们可以看出，差别同样在于因为确定策略、确定环境的原因，省去了枚举状态进行求和的形式。Bellman方程的图示如下：



Bellman方程是我们在本处构造的最优控制问题的最优解的充要条件。它的好处在于如果知道了下一个状态的最优值，那么就只需要搜索当前状态下单个最优策略，这样就可以通过从后往前的方式递归的计算出最优值函数，而不用考虑很多步之后。

现实问题中对于马尔可夫性质的破坏是很常见的，比如移动平均滤波器、时延控制等。这种情况下的解决办法通常是对于状态空间进行重构（扩大状态空间），然后使用新的状态来选择最优动作。不过这样也肯定会带来计算量的增加。

最后我们还需要解释两个问题：

- **为什么Bellman方程的解就是原来最优控制问题的最优解？**
  以上两个解相等的原因在于我们之前做过的两个假设：环境的马尔科夫性质和代价函数的可分离性。这两个假设保证了我们把原始的问题分解成一系列multistage的子问题提供了方便。更具体地来说，我们考虑以下两个优化问题：

$$u^* = \arg\min_u \{\cdot\},$$
$$\pi^* = \arg\min_\pi \{\cdot\}.$$

  以上两式看似除了下标之外完全一样，但是实际上它们来自于两个开环和闭环最优控制问题。开环OCP直接优化出一系列的控制序列，而不考虑与当前及之前状态的关联；而在闭环OCP中，我们要优化得到的闭环策略的结构上是有限制的。一旦找到了这样一个最优策略，我们就得到了一个在线的反馈控制器。我们刚才讲的ADP版的Bellman方程实际上是一个开环的最优条件，但是当我们引入了一个结构上有限制的策略时，我们就可以得到一个闭环的最优控制问题：

$$\pi^*(x) = u^* = \arg\min_u \{l(x,u) + V^*(x')\}, \forall x \in \mathcal{X},$$

  这里的$u = \pi(x), \forall x \in \mathcal{X}$。那么下面我们只要说明开环和闭环OCP的等价性即可。开环Bellman方程的解是一个序列$\{u_t^*, u_{t+1}^*, \cdots, u_\infty^*\}$，而闭环Bellman方程的解是一个策略$\pi^*(x)$。易知对于每一个固定的状态，其最优动作要不是只有唯一一个，要不就是有多个，而这多个选哪个作为最优动作是无所谓的。因此，根据闭环策略，我们可以轻松得到一系列的最优动作序列。其实，只要我们用来拟合策略$\pi(x)$的函数具有足够的表示能力（比如神经网络），那么我们就可以认为开环和闭环Bellman方程的解之间没有差别。
- **为什么不需要考虑初始的状态？**
  可能大家也已经注意到了，我们在构建ADP对应的OCP问题时与之前的RL问题最大的区别就是没有将初始状态分布作为加权系数考虑进去。这是因为如果$\pi^*(s)$是对于某个特定的初始状态得到的最优策略，那么对于其它初始状态它也是最优的。

8.1.2 离散时间的ADP算法框架

ADP的目的是找到闭环的Bellman方程的解。类似于之前讲过的RL算法，这里ADP的算法也可以被归类为值迭代和策略迭代两种，而且这两种算法都可以使用Actor-Critic的框架来表述，即可分为PEV（对应于Critic）和PIM（对应于Actor）两个部分。策略迭代和值迭代的ADP对比如下：

	Policy iteration ADP	Value iteration ADP
PEV iteration	Infinite steps	One step
Initial policy	Must be admissible	No admissibility requirement
Is each intermediate policy admissible?	Yes	No guarantee
Iteration efficiency	High, but every iteration is more computationally demanding	Low
Online/Offline	Online/Offline	Mostly offline

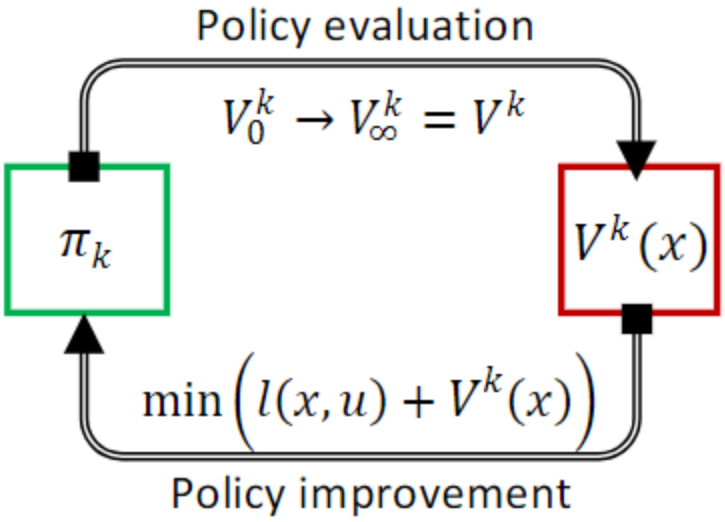
这里需要解释一下上表中什么是Iteration Efficiency。它被定义为达到某个performance level所需要的迭代次数。

在介绍算法之前，需要再次强调的是，我们在ADP中的环境模型不仅是已知的，而且是**确定性的**。这种环境莫i选哪个带来种种简化：

- reward signal（或者说utility function）不需要包含有关下一个状态的信息（回顾我们之前定义的reward都是三元组(s, a, s')的函数），因为下一个状态可通过环境模型由当前状态和动作唯一确定。
- 值函数的定义式中不包含折扣因子γ。这是因为状态和动作都能达到零平衡态，因此值函数是有界的，并不需要使用折扣因子来保证其有界性。

尽管有上述种种好处，但是也存在一些困难。比如不使用折扣因子可能会导致算法的收敛性变差、策略的准确性下降、训练稳定性下降，这是由于缺少了γ-contraction性质导致的。需要更仔细的调参和一些技巧（如zero boundary condition、high-order value approximation）来保证算法的收敛性。

下面我们主要来讲一下策略迭代的ADP算法框架。策略迭代的ADP算法框架如下：



对于具有确定环境模型的ADP问题，PEV就是要求解self-consistency条件。但是，没有折扣因子的话，self-consistency条件的解可能不唯一！怎么办呢？我们可以在该条件的左右两侧各加一个相同的常数，这并不会改变左右两侧的相等。而通过加这个常数，我们可以使得当$x = x_{equ}$时$V^k(x) = 0$，这样就可以得到一个唯一解了。这个操作就是zero boundary condition。这样，接下来，我们把self-consistency条件看成是一个代数方程，就可以通过fixed-point iteration来求解了。这个过程就是PEV的过程：

Repeat until $j \to \infty$

$$V_{j+1}^k(x) \leftarrow l(x, \pi_k) + V_j^k(x') - V_j^k\left(x_{\text{equ}}\right), \forall x \in \mathcal{X}$$

End

注意，上式最右边减去$V_j^k(x_{\text{equ}})$就是利用上面说的zero boundary condition，如果不见去掉这一项，训练可能会非常不稳定。在很多任务中，x_{equ}为0。这里的上下标需要注意，k表示的是最外层的循环的轮数，也可以理解为当前策略为π^k，而j表示的是内层循环的轮数。

在第k轮大循环的PEV按照上述的方式迭代完成后，就要进行PIM了。PIM时遵守的公式如下：

$$\pi_{k+1}(x) = \arg\min_u \{l(x,u) + V^k(x')\}, \forall x \in \mathcal{X}.$$

上式右边使用的式子也被称为弱Bellman方程。因为右边的$V^k(x')$不是最优值函数而是第k轮PEV后对于值函数的估计。

8.1.3 收敛性和稳定性

收敛性和稳定性是ADP算法的两个相互紧密联系的性质。它们既有紧密的联系，又有明显的区别。让我们先来看看它们各自的定义。

> **收敛性**：ADP算法的收敛性描述了一个算法是否能从中间策略π_k逐渐收敛到最优策略π^*。证明收敛性的方法是看随着迭代轮数k的增加，值函数的值是否变得更好（在我们这里最小化代价函数的假设下即值函数的值是否变得更小）：
>
> $$V^{k+1}(x) \leq V^k(x), \forall x \in \mathcal{X}.$$

> **稳定性**：ADP算法的稳定性描述了**最优策略**π^*是否能够使得被控对象稳定下来。证明方法是检查随着时间t的增加，**最优值函数**$V^*(x)$是否持续下降：
>
> $$V^*(x_{t+1}) \leq V^*(x_t), \forall x_t \in \mathcal{X}.$$

让我们再来辨别一下二者的区别与联系：

- **联系**：
  - **稳定性是收敛性的前提**：一个ADP算法不可能收敛到一个不稳定的策略上。而且不光是最终的策略，只要中间策略是不稳定的，就无法进行有效的环境交互。
- **区别**：
  - **与什么有关**：收敛性与主循环的迭代次数有关，而稳定性与时间t有关。前者预训练算法有关，后者与得到策略有关。
  - **一个策略可以是稳定而非最优的**：一个不好的算法可能最终会达到一个稳定的策略，但是这个策略并不是最优的。那么这样的算法因为缺少最优性，因此没有意义。

有关二者的内在联系在有噪声的观测和不确定的环境模型下就更加复杂，这里就不展开了。

8.1.3.1 闭环稳定性

对于没有限制的OCP问题，闭环稳定性等价于策略的可接受性。一个可接受的策略，以其为反馈控制器的闭环系统必须是稳定的。换句话说，不能有不稳定的状态（即值函数无穷大的状态）。

在策略迭代ADP中，为了能够与环境成功的进行交互，迭代的过程必须是recursive stable的，即如果当前策略π_k是可接受的，那么本轮迭代结束后得到的下一轮策略π_{k+1}也必须是可接受的。如果能够保证这一点，那么只要我们选取的初始策略是可接受的，那么一直到最后得到最优策略的整个过程中所有的中间策略都是可接受的。我们下面来证明这一点。

> **定理1**：在策略迭代的ADP算法中，每个策略π_k都是可接受的，只要初始的策略π_0是可接受的。

证明如下。首先我们来回顾一下迭代过程的记号变化情况。在第k轮迭代的PEV阶段，我们的策略是π_k。本轮PEV开始之前，值函数为$V^{k-1}(x)$，在本轮PEV结束后得到的值函数是$V^k(x)$。在PIM阶段，我们在最新得到值函数$V^k(x)$的基础上搜索新的最佳策略，PIM结束后得到的策略是π_{k+1}。下面我们来证明这个定理。我们在第k轮PIM结束后得到策略π_{k+1}，根据其搜索的方式，我们有：

$$l(x, \pi_{k+1}) + V^k\big(f(x, \pi_{k+1})\big) = \min_u \{l(x, u) + V^k\big(f(x, u)\big)\}.$$

即π_{k+1}对于式子$l(x,u)+V^k\big(f(x,u)\big)$是最优的（此处为最小的）。那么易知因为$\pi_k$此时对于$l(x,u)+V^k\big(f(x,u)\big)$不是最优的，因此有：

$$l(x,\pi_k)+V^k\big(f(x,\pi_k)\big)\geq\min_u\{l(x,u)+V^k\big(f(x,u)\big)\}.$$

又由self-consistency条件，我们有：

$$V^k(x)=l(x,\pi_k)+V^k\big(f(x,\pi_k)\big).$$

因此有：

$$V^k(x)\geq l(x,\pi_{k+1})+V^k\big(f(x,\pi_{k+1})\big).$$

又因为utility function是正定的，所以有：

$$l(x,\pi_{k+1})+V^k\big(f(x,\pi_{k+1})\big)\geq V^k\big(f(x,\pi_{k+1})\big)$$

最后再结合$V^k\big(f(x,\pi_{k+1})\big)=V^k(x')$，我们就得到了：

$$V^k(x)\geq V^k(x'),\quad\forall x\in\mathcal{X}.$$

很明显，只有当达到了平衡态之后才取等号。注意到，x'是由策略π_{k+1}决定的，这就说明了新策略得到值函数必然是越来越小的，自然不会发散，因此是有界的，也就是说是可接受的。这就证明了定理1。

下面我们再来证明策略迭代ADP算法最终得到的最优策略π*是可渐近稳定的。

> **定理2**：在策略迭代的ADP算法中，最终得到的最优策略π*是渐近稳定的。

证明如下。首先，最优值函数也必定满足self-consistency条件：

$$V^*(x)=l(x,\pi^*)+V^*\big(f(x,\pi^*)\big)=l(x,\pi^*)+V^*(x').$$

再由utility function的正定性，我们有：

$$V^*(x)\geq V^*(x').$$

根据稳定性的定义，即得证。

从上面的证明过程也可以看出，utility function的正定性是证明稳定性的关键。而对于值迭代的ADP算法，因为没有中间策略，随意不要求recursive stability，但是最终得到的最优策略仍然必须是可接受的。

8.1.3.2 收敛性

稳定性不一定意味着收敛性。也就是说，一个策略是可接受的，并不意味着在它之后的策略是更好的。极端的情形是。所有中间策略都是一模一样的，也就是说，策略一直没有改进。这样只要初始策略是可接受的，后续的策略也自然是可接受的，但是却一直不能收敛。证明ADP的收敛需要证明以下两点：

- **Value Decreasing Property**：

$$V^{k+1}(x) \leq V^k(x), \forall x \in \mathcal{X}$$

- **最终的策略可以达到最优**：

$$V^{\infty}(x) = V^*(x), \forall x \in \mathcal{X}.$$

  这里必须解释一下，$V^{\infty}(x)$和$V^*(x)$是两个完全不同的东西。前者是从迭代的角度来说的，说的是迭代最终停止时得到的值函数。而后者是Bellman方程的解。二者的相等不是那么显然的，需要我们去证明的。

下面试分别证明之。先来证明Value Decreasing Property。

> **定理3**：随着迭代轮数k的增长，值函数$V^k(x)$是单调递减的。即：
>
> $$V^{k+1}(x) \leq V^k(x), \forall x \in \mathcal{X}.$$

证明如下。假设我们当前的策略是π_{k+1}，那么我们根据这个策略与环境进行交互，产生一系列状态序列：

$$x, x', x'', x''', \cdots, x^{(\infty)}$$

因为π_{k+1}是可接受的（即稳定的），因此根据我们之前的稳定性定义，随着时间t的增长，值函数是单调递减的。又因为我们之前假设过平衡态只在原点取到，因此$x^{(\infty)} = 0$。由self-consistency条件，我们有：

$$V^k(x) = l(x, \pi_k) + V^k\big(f(x, \pi_k)\big).$$

又由于第k轮的PIM搜索后得到的策略π_{k+1}是最优的，因此有：

$$\pi_{k+1} = \arg\min_u \{l(x, u) + V^k\big(f(x, u)\big)\}.$$

由此我们有：

$$l(x, \pi_k) + V^k\big(f(x, \pi_k)\big) \geq l(x, \pi_{k+1}) + V^k\big(f(x, \pi_{k+1})\big).$$

再结合self-consistency条件，我们有：

$$V^k(x) \geq l(x, \pi_{k+1}) + V^k\big(f(x, \pi_{k+1})\big).$$

再根据第k+1轮的值函数的self-consistency条件，我们有：

$$l(x, \pi_{k+1}) = V^{k+1}(x) - V^{k+1}\big(f(x, \pi_{k+1})\big).$$

代入上式，我们就得到了：

$$V^k(x) \geq V^{k+1}(x) - V^{k+1}\big(f(x, \pi_{k+1})\big) + V^k\big(f(x, \pi_{k+1})\big).$$

我们将$f(x, \pi_{k+1})$记成x'，并递归的使用上面的不等式，我们就得到了：

$$V^{k+1}(x) - V^k(x) \leq V^{k+1}(x') - V^k(x') \leq \cdots \leq V^{k+1}(x^{(\infty)}) - V^k(x^{(\infty)})$$

由因为我们之前提到过的zero boundary condition，我们有$V^k(0) = 0$、$V^{k+1}(0) = 0$及$x^{(\infty)} = 0$，因此我们有：

$$V^{k+1}(x) - V^k(x) \leq V^{k+1}(x^{(\infty)}) - V^k(x^{(\infty)}) = V^{k+1}(0) - V^k(0) = 0.$$

即：

$$V^{k+1}(x) \leq V^k(x), \forall x \in \mathcal{X}.$$

证毕。

> **定理4**：最终的策略可以达到最优。即：
>
> $$V^{\infty}(x) = V^*(x), \forall x \in \mathcal{X}.$$

证明如下。当迭代停止时，相邻的两个策略是相同的，对应的值函数也是完全一样的。考虑到每轮的PIM过程和Bellman方程：

$$\pi_{k+1}(x) = \arg\min_u \{l(x, u) + V^k(x')\}, \forall x \in \mathcal{X}$$
$$V^*(x) = \min_u \big\{l(x, u) + V^*\big(f(x, u)\big)\big\}.$$

则易知，迭代停止时相邻的两个策略必定满足Bellman方程。因此，可证此时策略对应的值函数即为Bellman方程的解。证毕。

从上述证明中也可看出，ADP的收敛机制与之前讲过的具有discounted cost的DP的收敛机制是不同的。前者是靠utility function的正定性和环境的确定性来保证的。而且zero boundary condition也是一个很重要的技巧。而后者是靠折扣因子的γ-contraction性质来保证的。

8.1.4 Inexact Policy Iteration ADP的收敛性

在我们上面的策略迭代的PEV中，每次大循环我们在理论上需要进行无数次的内层迭代才能获得值函数的估计值。但这样的计算量太大了。我们能不能像之前的RL那里说过的一样，只进行有限次数的内层迭代呢？当然可以，这就是Inexact Policy Iteration（Inexact PI）算法。IPI算法的框架如下：

Repeat j until $N-1$

$$V_{j+1}^{k}(x) \leftarrow l(x,\pi_k) + V_j^k(x'), \forall x \in \mathcal{X}$$

End

在采取了上述框架后，我们每次这样更新值函数每次PEV结束后$V^k(x) = V_N^k(x)$（即迭代k轮），每次PEV开始前使用上一轮的值函数来进行初始化$V_0^{k+1}(x) = V^k(x)$。那么现在的关键问题就是这样进行迭代是否仍然收敛呢？答案是肯定的。我们下面来证明这一点。

首先，我们需要对于初始的策略做一点假设：

> **假设3**：初始策略满足：
>
> $$V^0(x) \geq 0, V^0(x) \geq \min_u\{l(x,u) + V^0(x')\}, \forall x \in X$$

如果上面的假设成立，那么我们就有如下的定理：

> **定理5**：在**假设3**成立的情况下，值函数序列$\{V^k(x)\}$是单调递减的。

证明如下。我们的核心思路是要证明以下两点

- $V^k(x) \geq V^{k+1}(x)$（核心要证明的式子）
- $V^{k+1}(x) \geq \min_u\{l(x,u) + V^1\big(f(x,u)\big)\}$ （证明中的辅助项）

那么我们首先来证明k=0的情况。根据假设3，我们有：

$$V^0(x) \geq \min_u\{l(x,u) + V^0\big(x'\big)\}.$$

又因为策略π_1是通过最小化$l(x,u) + V^0\big(x'\big)$得到的，因此有：

$$V^0(x) \geq l(x,\pi_1) + V^0\big(x'\big).$$

那么对于$V^0(x')$，仿照上述的证明过程，我们有：

$$V^0(x') \geq l(x',\pi_1) + V^0\big(x''\big).$$

那么不断这样展开，我们就有：

$$
\begin{aligned}
l(x,\pi_1)+V^0(x') &\geq l(x,\pi_1)+l\big(x',\pi_1(x')\big)+V^0(x'') \\
&= \sum_{i=0}^{1} l\left(x^{(i)},\pi_1(x^{(i)})\right)+V^0(x^{(2)}) \\
&\geq \sum_{i=0}^{1} l\left(x^{(i)},\pi_1\big(x^{(i)}\big)\right)+l\left(x^{(2)},\pi_1\big(x^{(2)}\big)\right)+V^0\big(x^{(3)}\big) \\
&\geq V^0(x^{(N)})+\sum_{i=0}^{N-1} l\left(x^{(i)},\pi_1(x^{(i)})\right) \\
&= V^1(x).
\end{aligned}
$$

为什么$V^0(x^{(N)})+\sum_{i=0}^{N-1} l\left(x^{(i)},\pi_1(x^{(i)})\right)=V^1(x)$呢？这个直接根据上面Inexact PI的迭代框架展开写即可看出。因此，我们就证明了第一点，即：

$$
V^0(x)\geq l(x,\pi_1)+V^0\big(x'\big)\geq V^1(x).
$$

下面我们来证明第二点。首先把上述框架展开写出来，我们有：

$$
\begin{aligned}
V^1(x) &= \sum_{i=0}^{N-1} l\left(x^{(i)},\pi_1(x^{(i)})\right)+V^0(x^{(N)}) \\
&\geq l(x,\pi_1)+\sum_{i=1}^{N} l\left(x^{(i)},\pi_1(x^{(i)})\right)+V^0(x^{(N+1)}) \\
&= l(x,\pi_1)+V^1(x'),
\end{aligned}
$$

这个式子实际上就是$V^1(x)\geq l(x,\pi_1)+V^1(x')$。可能大家会疑惑，这不就是self-consistency条件吗？但是为什么这里的等号变成了不等号呢？这是因为我们这里的迭代是Inexact PI，因此不能保证self-consistency条件的成立。现在我们来看看这个式子怎么证明的。上式最关键的就是右边第一式到右边第二式。去掉相同项之后，其实就是证明$V^0(x^{(N)})\geq l(x^{(N)},\pi_1(x^{(N)}))+V^0(x^{(N+1)})$。那么这里根据我们的初始的假设即可得到。再有下面这个式子：

$$
l(x,\pi_1)+V^1(x')\geq \min_u\{l(x,u)+V^1(x')\}.
$$

这个式子实际上利用了PIM的机理，右边可以看成是策略π_2，而根据其最优性可知上式成立。综合以上两式，我们有：

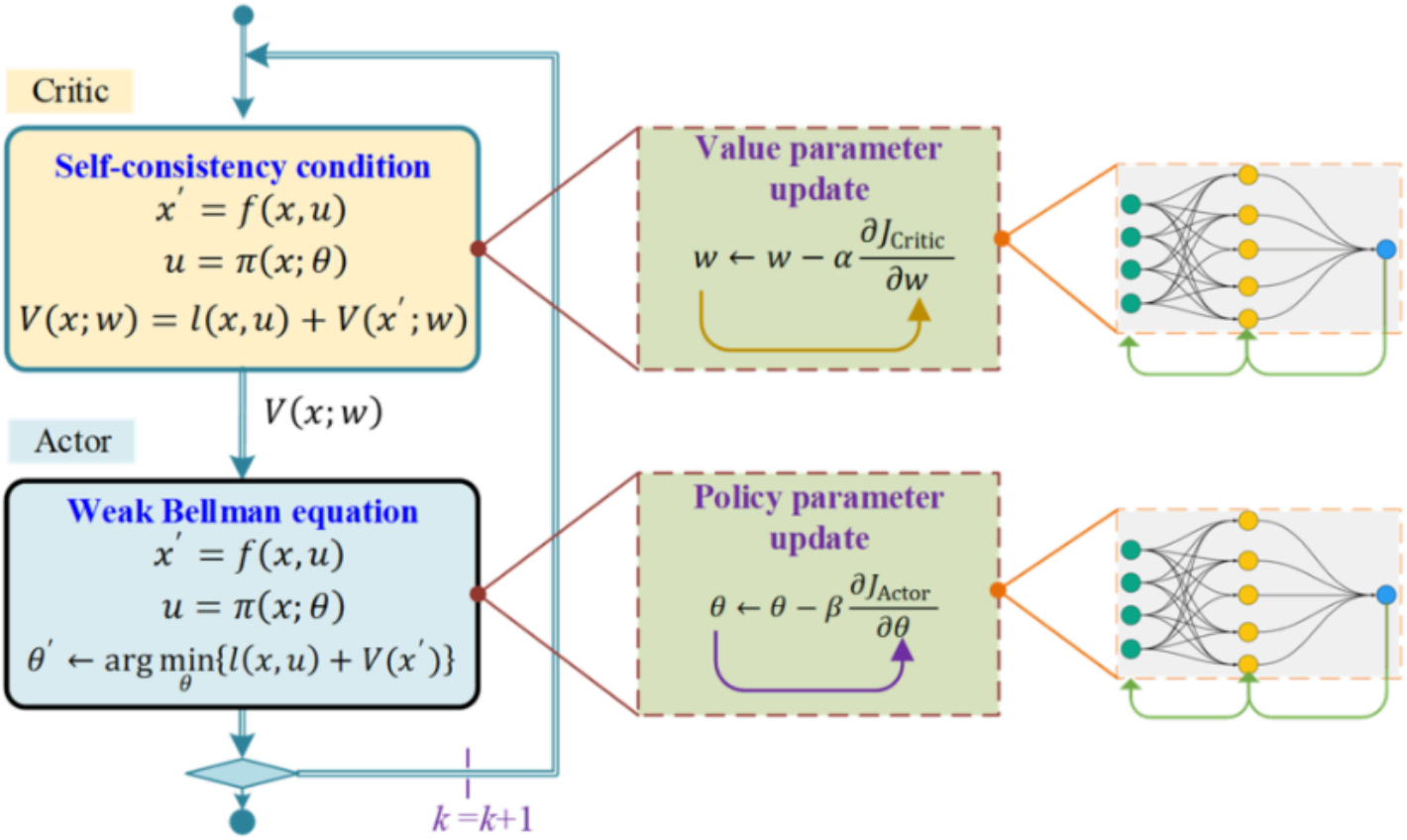
$$
V^1(x)\geq \min_u\{l(x,u)+V^1(x')\}.
$$

第二点也证明完毕。因此我们仿照上面的证明方式，递归的进行下去，就可以证明定理5，即Inexact PI的值函数是单调递减的。因此算法是收敛的。

Inexact PI的重要的超参数N的选择是一个很重要的问题。通常要结合具体问题仔细调参。常见的取值为$N = 5 \sim 10$。从Exact PI到Inexact PI的转变实际上为我们提供了一个重要的insight，即我们每轮对于值函数的估计不必做到十分精确也能保证收敛。遮掩就为我们后面使用函数近似（Function Approximation）提供了一个很好的理论保证。

8.1.5 使用函数近似的离散时间ADP

当问题规模变大时，Tabular ADP的方法就越来越不适用了。这是我们需要使用函数来近似值函数和策略，尽管这会损失一些精度。这里我们还是使用前面在讲函数近似的时候使用过的Actor-Critic框架来表述，并且我们选取的函数是神经网络。这样的近似方法也是ADP被称为Neuron Dynamic Programming的原因。



我们先来看看actor梯度的推导。这是因为在第k轮主循环时，先进行的是critic（即值函数）的更新，此时的actor（即策略）是固定的，由第k-1轮迭代所确定的。为了表述方便，我们需要把这里的critic的更新写成一个优化问题：

$$J_{\text{Critic}}(w) = \frac{1}{2}\big(l(x,u) + V(x';w) - V(x;w)\big)^2.$$

$$\text{where}$$

$$x' = f(x,u),$$

$$u = \pi(x;\theta).$$

上式实际上就利用了self-consistency条件$V(x;w)=l(x,u)+V(x';w)$，只不过我们把上述条件写成等式两边做差的形式并最小化这个目标函数。但是这个优化问题面临严重的问题，即极值不唯一。我们给值函数加上一个相同的常数（相当于平移，即对于值函数的所有取值都加上一个相同的常数），那么因为目标函数是做差的形式，因此仍然成立。这实际上就造成了有无穷多个极值，训练稳定性因此很差。下面是几种常见的解决方法：

- **Semi-gradient**：我们按照下面的semi-gradient的方式来更新值函数：

$$\nabla J_{\text{Critic}}=\frac{\partial J_{\text{Critic}}}{\partial w}=-\big(l(x,u)+V(x';w)-V(x;w)\big)\frac{\partial V(x;w)}{\partial w}.$$

  这里之所以是semi-gradient是因为我们没有考虑到$V(x';w)$的梯度，认为它是一个常数。这样处理之后可以稳定训练，因为上式与单步的bootstrapping方法是具有形式上的相似性的。
- **将zero boundary condition正则项加入到目标函数中**：

$$J_{\text{Critic}}(w)=\frac{1}{2}\left\{\Big(l(x,u)+V(x';w)-V(x;w)\Big)^2+\rho V(0;w)^2\right\}$$

  这里的ρ是一个超参数，是一个正的常数。该方法还可以与上面的semi-gradient方法结合使用。

可以看出，这里的critic gradient与之前讲过的model-free的actor-critic方法的actor gradient是很相似的，除了环境信息是怎样采样的略有不同。这里使用过一个解析模型$f(x,u)$，而model-free的方法是通过直接与环境交互来采样的。不过，因为我们采用了semi-gradient的方法，并未直接对$V(x';w)$求梯度，因此这里的critic gradient中没有包含环境模型的梯度。拥有模型的好处在此处还看得不是很明显，但是在下面的actor gradient中就会看得很明显了。

下面我们来看看actor gradient的推导。其对应的优化问题是：

$$\begin{gathered}J_{\text{Actor}}(\theta)=l(x,u)+V(x';w),\\ \text{where}\\ x'=f(x,u),\\ u=\pi(x;\theta),\end{gathered}$$

注意，这里在第k轮我们进行actor更新时，Critic（$V(\cdot;w)$）是固定的，我们需要更新的是actor（$\pi(\cdot;\theta)$）的参数θ。Actor gradient的推导如下：

$$\begin{aligned}\nabla J_{\text{Actor}}&=\frac{\partial J_{\text{Actor}}}{\partial\theta}=\frac{\partial u^{\mathrm{T}}}{\partial\theta}\frac{\partial l}{\partial u}+\frac{\partial u^{\mathrm{T}}}{\partial\theta}\frac{\partial x'}{\partial u}\frac{\partial V}{\partial x'}\\&=\frac{\partial\pi^{\mathrm{T}}(x;\theta)}{\partial\theta}\left(\frac{\partial l(x,u)}{\partial u}+\frac{\partial f^{\mathrm{T}}(x,u)}{\partial u}\frac{\partial V(x';w)}{\partial x'}\right).\end{aligned}$$

注意，这里的$u=\pi(x;\theta)$，$x'=f(x,u)$。上述推导使用高数中学过的链式法则即可轻松得到。这里我们就可以看出具有模型的好处了。这里的模型不仅包括环境模型$f(x,u)$，还包括reward signal $l(x,u)$。我们在上述梯度中反复利用这两者的梯度，从而极大的减少了与环境交互带来的巨大的计算量，并增加了精度（只要模型足够准确）。下面我们给出整个算法的伪代码：

Hyperparameters: critic learning rate α, actor learning rate β, critic update frequency n_c, actor update frequency n_a, number of environment resets M

Initialization: state-value function $V(x;w)$, policy function $\pi(x;\theta)$.

Repeat (indexed with k)

(1) Use environment model

Initialize memory buffer $\mathcal{D}\leftarrow\emptyset$

Repeat M environment resets

$x\sim d_{\text{init}}(x)$

$u\leftarrow\pi(x;\theta)$

$x'\leftarrow f(x,u)$

Calculate $l, V, \partial V/\partial w, \partial V/\partial x, \partial l/\partial u, \partial f^{\mathrm{T}}/\partial u$, and $\partial\pi^{\mathrm{T}}/\partial\theta$

$$\mathcal{D}\leftarrow\mathcal{D}\cup\left\{\left(l,V,\frac{\partial V}{\partial w},\frac{\partial V}{\partial x},\frac{\partial l}{\partial u},\frac{\partial f^{\mathrm{T}}}{\partial u},\frac{\partial\pi^{\mathrm{T}}}{\partial\theta}\right)\right\}$$

End

(2) Critic update

Repeat n_c times

$$\nabla_w J_{\text{Critic}}\leftarrow-\frac{1}{M}\sum_{\mathcal{D}}\left(l(x,u)+V(x';w)-V(x;w)\right)\frac{\partial V(x;w)}{\partial w}$$

$$w\leftarrow w-\alpha\cdot\nabla_w J_{\text{Critic}}$$

End

(3) Actor update

Repeat n_a times

$$\nabla_\theta J_{\text{Actor}}\leftarrow\frac{1}{M}\sum_{\mathcal{D}}\frac{\partial\pi^{\mathrm{T}}(x;\theta)}{\partial\theta}\left(\frac{\partial l(x,u)}{\partial u}+\frac{\partial f^{\mathrm{T}}(x,u)}{\partial u}\frac{\partial V(x';w)}{\partial x'}\right)$$

$$\theta\leftarrow\theta-\beta\cdot\nabla_\theta J_{\text{Actor}}$$

End

End

这里需要做几点说明：

- **Use environment model**：注意这里我们每轮主循环的数据集大小为M。且这里的数据集与我们之前在model-free的actor-critic中的数据集是不同的，这里因为不用于环境交互而依靠模型，因此数

据集每条数据的构成也是一些后面要用到的梯度项。

- n_a**和**n_c：与我们之前在model-free的actor-critic中的n_a和n_c的定义是一样的，即每轮主循环中的actor和critic的更新次数。更多的更新次数意味着更多的计算量，但是也意味着更高的精度。还有一种极端的情况是$n_a = n_c = 1$，即每轮主循环只进行一次actor和critic的更新。这样虽然损失了一些精度，但是大大减少了计算量。并且可以证明，这样并不会影响算法的收敛性。

8.2 连续时间、无限视野（Continuous-time, Infinite-horizon）的ADP

8.2.1 问题建模及Hamilton-Jacobi-Bellman（HJB）方程

在实际问题中，很多时候我们要处理的问题是连续时间的。连续时间的问题与之前讲的离散时间问题有很强的联系。概括点来说，就是把求和变成积分，差分变成求导，差分方程变成微分方程。下面我们来看看连续时间的问题具体是怎么建模的。

> **定义2**：连续时间、无限视野的ADP问题如下定义：
>
> $$\begin{aligned}\min_{u(\tau),\tau\in[t,\infty)} V\big(x(t)\big) &= \int_t^{\infty} l\big(x(\tau),u(\tau)\big)\mathrm{d}\tau,\\ \text{s.t.}\\ \dot{x}(t) &= f\big(x(t),u(t)\big),\end{aligned}$$

这里的$V(\cdot)$是代价函数（或按照RL的习惯称之为值函数），且在原点取到0（$V(0)=0$）。$l(x,u)$是utility函数，它是正定的，除了在原点(0, 0)处之外处处大于零。环境模型$f(x,u)$与离散的时候稍有不同，它不像其对应的离散版本给出的是下一个状态，而是状态的导数（即状态的变化率）。$f(x,u)$在原点这个平衡态也取到0。上面的定义给出的值函数满足一个被称为Hamilton-Jacobi-Bellman（HJB）方程的微分方程，该方程是离散时间Bellman方程的连续时间版本。一旦我们求出了这个方程的解，就可以通过最小化哈密顿量来得到最优的控制策略。下面我们来看看HJB方程是怎么推导的。

对定义2中的关于值函数的积分式两侧求导，并注意到右边时间t在下限上，求导之后有个负号，那么就有：

$$\frac{\partial V(x)}{\partial x^{\mathrm{T}}} f(x,u) = -l(x,u).$$

这里的$\partial V(x)/\partial x^{\mathrm{T}} \in \mathbb{R}^{1\times n}$是一个行向量。为了下面的推导方便，我们还需要作以下几点假设，比如值函数$V\big(x(t)\big)$关于状态x和时间t都是连续可微的。仿照离散时的self-consistency条件，我们有：

$$V\big(x(t)\big) = \int_t^{t+\mathrm{d}t} l\big(x(\tau),u(\tau)\big)\mathrm{d}\tau + V\big(x(t)+\mathrm{d}x(t)\big).$$

这个式子根据我们的定义2很好得到。对上式应用Bellman最优性原理，我们有：

$$V^*\big(x(t)\big) = \min_{t\le\tau\le t+\mathrm{d}t}\left\{\int_t^{t+\mathrm{d}t} l\big(x(\tau),u(\tau)\big)\mathrm{d}\tau + V^*\big(x(t)+\mathrm{d}x(t)\big)\right\}.$$

我们对于$V^*\big(x(t)+\mathrm{d}x(t)\big)$在$x(t)$处进行泰勒展开：

$$V^*\big(x(t)+\mathrm{d}x(t)\big) = V^*\big(x(t)\big) + \frac{\partial V^*\big(x(t)\big)}{\partial x^{\mathrm{T}}(t)}\mathrm{d}x(t) + \mathcal{O}\big(\mathrm{d}x(t)\big).$$

再利用$\mathrm{d}x(t)=\dot{x}(t)\mathrm{d}t, \mathrm{d}t\to 0, \tau\to t,$，我们有：

$$V^*\big(x(t)\big) = \min_{u(t)}\left\{l\big(x(t),u(t)\big)\mathrm{d}t + \frac{\partial V^*\big(x(t)\big)}{\partial x^{\mathrm{T}}}\dot{x}(t)\mathrm{d}t + V^*\big(x(t)\big)\right\}.$$

又因为$V^*\big(x(t)\big)$与动作u(t)无关，因此左右两边可以消掉$V^*\big(x(t)\big)$，我们就得到了HJB方程：

> **定义3**：HJB方程如下：
>
> $$\min_{u(t)}\left\{l\big(x(t),u(t)\big) + \frac{\partial V^*\big(x(t)\big)}{\partial x^{\mathrm{T}}} f\big(x(t),u(t)\big)\right\} = 0.$$

接下来我们根据上述HJB方程来定义一个哈密顿量：

> **定义4**：哈密顿量如下：
>
> $$H\left(x,u,\frac{\partial V^*(x)}{\partial x}\right) \overset{\text{def}}{=} l(x,u) + \frac{\partial V^*(x)}{\partial x^{\mathrm{T}}} f(x,u).$$

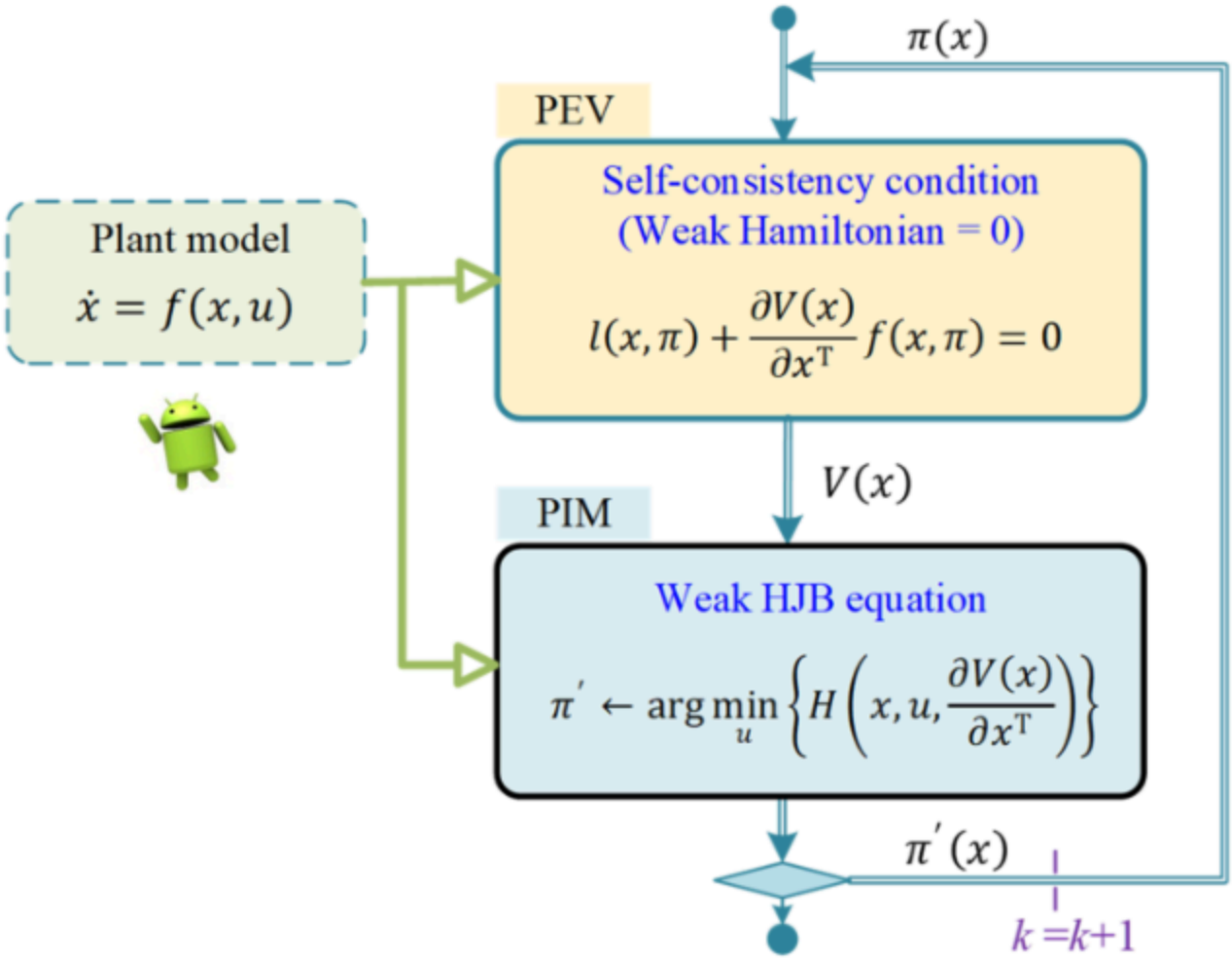
有了HJB方程和哈密顿量，我们就可以先通过求解HJB方程得到最优的值函数，再通过哈密顿量根据下式来得到最优的控制策略：

$$\pi^*(x) = \arg\min_u H\left(x,u,\frac{\partial V^*(x)}{\partial x}\right)$$

由于实际问题的复杂性，HJB方程很难直接求解。但是它却提供了一个检验策略最优性的方法。这种方法的一个缺点在于HJB方程只接受足够光滑的值函数，这在一些特殊的情况下是很难满足的。

8.2.2 连续时间ADP的框架

HJB方程是一个高度非线性的偏微分方程（PDE），而且通常用于求解PDE的三种方法都不适用于HJB方程。因此，这里我们用连续时间的ADP框架来求解HJB方程。ADP的框架如下：



这也是一个两步的迭代框架。下面我们来分别看看这两步的具体内容：

- **PEV**：这一步与我们之前在model-free的框架里讲的PEV相对应。它的本质是在策略π_k固定的情况下，通过**求解一个关于值函数$V(x)$的微分方程**。
- **PIM**：这一步与我们之前在model-free的框架里讲的PIM相对应。这步的本质是在我们本轮值函数已经更新完成的情况下找到一个能最小化哈密顿量的策略π'。

下面我们在从宏观的角度来看一下这个框架为什么能求解HJB方程。再回顾一下HJB方程：

$$\min_{u(t)}\left\{l\big(x(t),u(t)\big)+\frac{\partial V^*\big(x(t)\big)}{\partial x^{\mathrm{T}}}f\big(x(t),u(t)\big)\right\}=0.$$

直接求解HJB方程的难度是很高的，这个微分方程同时具有min符号和多个与x有关的函数（$\pi^*(x)$和$V^*(x)$）。但是，PEV过程**首先把min符号去掉，且固定了策略π，使得HJB方程变成了一个关于值函数$V(x)$的简单的微分方程**；而PIM过程则是在**固定了值函数$V(x)$的情况下，找到一个能最小化哈密顿量的策略π，这就是一个简单地优化问题**。另外注意，$V(0)=0$是连续时间ADP框架中一个很重要的条件。它的作用可以理解为给第一步PEV求解微分方程提供了一个初始条件。根据高数中求解微分方程的

知识可知，如果不给出初始条件，那么微分方程的解是不唯一的。因此，$V(0)=0$是唯一确定本轮的值函数的一个很重要的条件。

8.2.3 连续时间ADP的收敛性和稳定性

正如我们在上一节离散时间的ADP中所讲的，这里我们也需要讨论连续时间的ADP的收敛性和稳定性。这里大体的证明思路与离散那时候基本上一致，稳定性就是证明当前策略能够随着时间的推移使得被控对象趋于一个稳定的状态；而收敛性就是证明策略能够随着迭代轮数的增长而达到最优。不过因为这里我们处理的是连续的情况，因此具体的证明方法会有所不同。

8.2.3.1 稳定性

这里我们证明的方法就是现代控制理论里面的李雅普诺夫第二方法，**即找到一个Lyapunov函数，它本身是正定的，且它的导数是负定的**。这里我们选取值函数$V(x)$作为Lyapunov函数，那么根据我们之前的定义，$V(x)$本来就是正定的。那么证明稳定性，就变成了如下的问题：

> **稳定性**：选取值函数$V(x)$作为Lyapunov函数，证明其导数是负定的：
>
> $$\frac{\mathrm{d}V(x)}{\mathrm{d}t}\leq 0,\forall x\in X.$$

这里的证明思路与离散时的情况相似，核心是要证明递归稳定性。首先，我们需要找到一个可接受度厄初始策略π_0，其对应的值函数对于所有的$x\in\mathcal{X}$都是有限值。下面我们来证明这个定理：

> **定理6**：如果初始策略π_0是可接受的，那么根据连续时间的ADP框架，迭代过程中包含最终策略的任何中间策略都是可接受的。

证明如下。我们的核心是证明，对于任意两个相邻的策略π_k和π_{k+1}，如果π_k是可接受的，那么π_{k+1}也是可接受的。在迭代框架的第k次迭代中，在进行PEV之前，策略为π_k，值函数为$V^{k-1}(x)$。在PEV后，值函数变为$V^k(x)$。在进行PIM之后，策略变为π_{k+1}。那么根据迭代框架，策略π_{k+1}是通过最小化哈密顿量得到的，即：

$$\pi_{k+1}=\arg\min_u\left\{l(x,u)+\frac{\partial V^k(x)}{\partial x^{\mathrm{T}}}f(x,u)\right\}.$$

那么我们可以得到：

$$H\left(x,\pi_{k+1},\frac{\partial V^k(x)}{\partial x}\right)=\min_u\left\{l(x,u)+\frac{\partial V^k(x)}{\partial x^{\mathrm{T}}}f(x,u)\right\}.$$

那么同样易知，策略π_k肯定不是最优的，而且根据PEV过程，我们知道其满足：

$$l(x,\pi_k)+\frac{\partial V^k(x)}{\partial x^{\mathrm{T}}}f(x,\pi_k)=0$$

那么我们就有：

$$\begin{aligned} H\left(x, \pi_{k+1}, \frac{\partial V^k(x)}{\partial x}\right) &= \min_u \left\{ l(x,u) + \frac{\partial V^k(x)}{\partial x^{\mathrm{T}}} f(x,u) \right\} \\ &\leq H\left(x, \pi_k, \frac{\partial V^k(x)}{\partial x}\right) = l(x,\pi_k) + \frac{\partial V^k(x)}{\partial x^{\mathrm{T}}} f(x,\pi_k) = 0 \end{aligned}$$

即：

$$H\left(x, \pi_{k+1}, \frac{\partial V^k(x)}{\partial x}\right) \leq 0.$$

又因为：

$$\begin{aligned} \frac{\mathrm{d}V^k(x)}{\mathrm{d}t} &= \frac{\partial V^k(x)}{\partial x^{\mathrm{T}}} \dot{x} \\ &= \frac{\partial V^k(x)}{\partial x^{\mathrm{T}}} f(x,u) \\ &= H\left(x, \pi_{k+1}, \frac{\partial V^k(x)}{\partial x^{\mathrm{T}}}\right) - l(x, \pi_{k+1}) \\ &\leq 0 - l(x, \pi_{k+1}) \\ &\leq 0. \end{aligned}$$

因此，我们知道作为值函数的$V^k(x)$的导数是负定的。再结合其本身是正定的，且等号只在本轮PEV停止时成立，且在平衡态（即原点）处取到0，那么根据李雅普诺夫第二方法，我们就证明了策略π_{k+1}可以使得系统渐进稳定。证毕。

在上面的证明中，utility函数$l(x,u)$的正定性是一个很重要的条件。这个utility函数被视为一种推广的能量函数。一个闭环系统，如果它的广义能量随着时间推移单调下降。特别的，如果这个能量函数只有一个全局最小值，那么沿着任意非稳态的解，这个广义能量函数都会下降。

8.2.3.2 收敛性

收敛性的证明与之前的离散时间的ADP的证明方法是一致的。

> **收敛性**：证明随着迭代轮数的增长，对于任意的$x \in \mathcal{X}$，值函数$V(x)$的值都是单调递减的。收敛性的证明与离散的情况还是类似，我们要证明以下定理：
> **定理7**：随着迭代轮数k的增长，对于$x \in \mathcal{X}$，值函数$V^k(x)$是单调递减的。即：
>
> $$V^{k+1}(x) \leq V^k(x), \forall x \in X$$

证明如下。考虑第k+1轮的PEV过程，其就是求解一个微分方程。那么其解$V^{k+1}(x)$必然满足：

$$\frac{\partial V^{k+1}(x)}{\partial x^{\mathrm{T}}} f(x, \pi_{k+1}) + l(x, \pi_{k+1}) = 0$$

即：

$$\frac{\partial V^{k+1}(x)}{\partial x^{\mathrm{T}}} f(x, \pi_{k+1}) = -l(x, \pi_{k+1}),$$

再加上zero boundary condition，我们有：

$$V^{k+1}(0) = 0.$$

下面来考察第k轮的值函数$V^k(x)$和第k+1轮的值函数$V^{k+1}(x)$的对时间的导数。首先，由哈密顿函数的定义，我们有：

$$\frac{\mathrm{d}V^k(x)}{\mathrm{d}t} = H\left(x, \pi_{k+1}, \frac{\partial V^k(x)}{\partial x}\right) - l(x, \pi_{k+1})$$

又因为我们在第8.2.3.2节已经证明了$H\left(x, \pi_{k+1}, \frac{\partial V^k(x)}{\partial x}\right) \leq 0$，因此有：

$$\frac{\mathrm{d}V^k(x)}{\mathrm{d}t} = H\left(x, \pi_{k+1}, \frac{\partial V^k(x)}{\partial x}\right) - l(x, \pi_{k+1}) \leq -l(x, \pi_{k+1}),$$

这是$V^k(x)$的导数满足的关系。那么我们再来看看$V^{k+1}(x)$的导数。根据第k+1轮的PEV过程所满足的方程，我们有：

$$\frac{\partial V^{k+1}(x)}{\partial x^{\mathrm{T}}} f(x, \pi_{k+1}) + l(x, \pi_{k+1}) = 0$$

又因为$\frac{\mathrm{d}V^{k+1}(x)}{\mathrm{d}t} = \frac{\partial V^{k+1}(x)}{\partial x^{\mathrm{T}}} f(x, \pi_{k+1})$，因此有：

$$\frac{\mathrm{d}V^{k+1}(x)}{\mathrm{d}t} = \frac{\partial V^{k+1}(x)}{\partial x^{\mathrm{T}}} f(x, \pi_{k+1}) = -l(x, \pi_{k+1}).$$

所以：

$$\frac{\mathrm{d}V^k(x)}{\mathrm{d}t} \leq \frac{\mathrm{d}V^{k+1}(x)}{\mathrm{d}t} \leq 0.$$

根据牛顿——莱布尼茨公式并移向得到：

$$V\big(x(t)\big) = V\big(x(\infty)\big) - \int_t^\infty \frac{\mathrm{d}V\big(x(\tau)\big)}{\mathrm{d}\tau} \mathrm{d}\tau.$$

又因为根据我们刚才已经证明了稳定性，因此我们有$lim_{t\to\infty}x(t)=0$再结合zero boundary condition的$V^k\big(x(\infty)\big)=V^{k+1}\big(x(\infty)\big)=0$，我们有：

$$V^k\big(x(t)\big)=-\int_t^\infty\frac{\mathrm{d}V^k\big(x(\tau)\big)}{\mathrm{d}\tau}\mathrm{d}\tau$$
$$V^{k+1}\big(x(t)\big)=-\int_t^\infty\frac{\mathrm{d}V^{k+1}\big(x(\tau)\big)}{\mathrm{d}\tau}\mathrm{d}\tau$$

又因为$\frac{\mathrm{d}V^k(x)}{\mathrm{d}t}\leq\frac{\mathrm{d}V^{k+1}(x)}{\mathrm{d}t}\leq0$恒成立，因此积分之后该关系仍然保持，因此：

$$V^{k+1}(x)\leq V^k(x),\forall x\in X.$$

证毕。

8.2.4 连续时间ADP的函数近似

上面我们讲过的方法针对的还是tabular的情况，但是在实际问题中，我们常常需要使用函数近似。这里我们就以actor-critic框架为例，来讲解如何使用函数近似来求解连续时间的ADP问题。

首先来看看critic的梯度的推导。我们先来看看critic的优化问题：

$$J_{\mathrm{Critic}}(w)=\frac{1}{2}\left(H^2\left(x,u,\frac{\partial V(x;w)}{\partial x}\right)+\rho V^2(0;w)\right).$$

根据8.2.2节讲过的ADP的迭代框架，critic部分实际上就是解一个哈密顿量等于0的方程。那么我们就可以把对应的优化问题写成最小化这个哈密顿量的形式，并且为了保证非负性，使用了平方项的形式。然后为了保证解的唯一性，我们加入了一个正的常数$\rho V^2(0;w)$。这个项实际上就是为了保证zero boundary condition成立，即$V(0)=0$。我们对上述目标函数求梯度，并结合复合函数求导法则及$H(x,u,\partial V(x;w)/\partial x)=l(x,u)+\partial V(x;w)/\partial x^{\mathrm{T}}f(x,u)$，我们有：

$$\nabla J_{\mathrm{Critic}}=H\left(x,u,\frac{\partial V}{\partial x}\right)\frac{\partial^2V(x;w)}{\partial x^{\mathrm{T}}\partial w}f(x,u)+\rho\frac{\partial V(0;w)}{\partial w},$$
$$\frac{\partial^2V}{\partial x^{\mathrm{T}}\partial w}\overset{\mathrm{def}}{=}\frac{\partial}{\partial w}\left(\frac{\partial V}{\partial x^{\mathrm{T}}}\right)\in\mathbb{R}^{l_w\times n}.$$

这里的$\nabla J_{\mathrm{Critic}}\in\mathbb{R}^{l_w}$被称作**Critic Gradient**。但是，这里有一个问题，就是要计算二阶导数，这带来了很大的计算量。一种改进的方法是Dual Heuristic Programming（DHP）方法，它使用神经网络来近似值函数的一阶导而不是值函数本身,这可以有效的减少计算量。

再来看看actor的梯度的推导。我们先来看看actor的优化问题，这其实就不用说了吧，因为在8.2.2节的ADP框架中，我们已经讲过了，actor的优化问题就是在值函数固定的情况下最小化哈密顿量，即：

$$J_{\text{Actor}}(\theta) = l\big(x, \pi(x;\theta)\big) + \frac{\partial V(x;w)}{\partial x^{\mathrm{T}}} f\big(x, \pi(x;\theta)\big)$$

求梯度得到：

$$\nabla J_{\text{Actor}} = \frac{\partial \pi^{\mathrm{T}}(x;\theta)}{\partial \theta} \left(\frac{\partial l(x,u)}{\partial u} + \frac{\partial f^{\mathrm{T}}(x,u)}{\partial u} \frac{\partial V(x;w)}{\partial x} \right)$$

这里的$\nabla J_{\text{Actor}} \in \mathbb{R}^{l_\theta}$被称作**Actor Gradient**。

8.2.5 连续时间的Linear Quadratic控制

对于一个线性系统的最优控制对于理解ADP的理论上的完备性是很重要的。尽管很多的实际问题是非线性的，但是它们都可以在一个平衡态的邻域内进行线性化来转化为线性系统。在有关的最优控制理论中，Linear Quadratic Regulator（LQR）是其中最重要的一个方法。它具有简洁的线性形式、具有收敛性的保障以及易于根据具体的问题进行部署。下面我们先来看看LQR的具体形式，再使用上面说过的ADP的框架来分析LQR问题。

首先，LQR需要定义一个cost function：

$$\begin{gathered} V\big(x(t)\big) = \int_t^\infty \big(x^{\mathrm{T}}(\tau) Q x(\tau) + u^{\mathrm{T}}(\tau) R u(\tau)\big) \mathrm{d}\tau, \\ \text{s.t.} \\ \dot{x}(t) = A x(t) + B u(t), \end{gathered}$$

这里的$u(t)$是控制输入，$x(t)$是状态，矩阵A和B的组合可以使得系统稳定下来。$Q \geq 0$和$R > 0$是对称矩阵，分别是状态和控制输入的权重。我们的目标是找到一个最优的控制策略$\pi^*(x)$，使得$V\big(x(t)\big)$最小。经过研究发现，这个问题与求解Riccati方程有着很深的内在关系。首先我们根据Riccati方程，求解出一个最优的矩阵P^*：

$$Q + P^* A + A^{\mathrm{T}} P^* - P^* B R^{-1} B^{\mathrm{T}} P^* = 0$$

可以证明，在我们之前假设的条件下（即矩阵A和B的组合可以使得系统稳定），则上述Riccati方程的解是唯一且正定的。那么求解出P^*之后，我们就可以得到最优的控制策略：

$$u = -\frac{1}{2} R^{-1} B^{\mathrm{T}} \frac{\partial V^*(x)}{\partial x} = -R^{-1} B^{\mathrm{T}} P^* x$$

同时最优的cost function也可使用P^*来表示：

$$V^*(x) = x^{\mathrm{T}} P^* x$$

下面，我们来看看如何使用ADP的框架来求解LQR问题。我们把按照ADP框架迭代，在第k轮主循环时的cost function使用二次型来参数化：

$$V^k(x) = x^{\mathrm{T}} P^k x,$$

这里的P^k就是我们之前框架中的值函数的参数w。那么在第k轮主循环的PEV过程中，可以这样来更新：

$$\begin{aligned} P_{j+1}^k &= P_j^k - \alpha H\left(x, \pi_k, \frac{\partial V\left(x; P_j^k\right)}{\partial x}\right) x f^{\mathrm{T}} \\ &= P_j^k - \alpha\left(x^{\mathrm{T}} Q x + u^{\mathrm{T}} R u + 2 f^{\mathrm{T}} P_j^k x\right) x f^{\mathrm{T}} \end{aligned}$$

这里的更新公式实际上就是用了我们在刚才的ADP框架讲过的Critc的梯度的推导（$\nabla J_{\mathrm{Critic}} = H\left(x, u, \frac{\partial V}{\partial x}\right) \frac{\partial^2 V(x;w)}{\partial x^{\mathrm{T}} \partial w} f(x,u) + \rho \frac{\partial V(0;w)}{\partial w}$，只不过这里我们不含后面含$\rho$的那一项，并且这里涉及到一些矩阵求导的运算，这里就不展开了）。那么当PEV的迭代步数趋于无穷大时，我们就可以得到最优的P^k。之后，在PIM过程中，可以直接得到一个解析解形式的最优控制策略：

$$\pi_{k+1}(x) = -\frac{1}{2} R^{-1} B^{\mathrm{T}} \frac{\partial V^k(x)}{\partial x} = -R^{-1} B^{\mathrm{T}} P^k x.$$

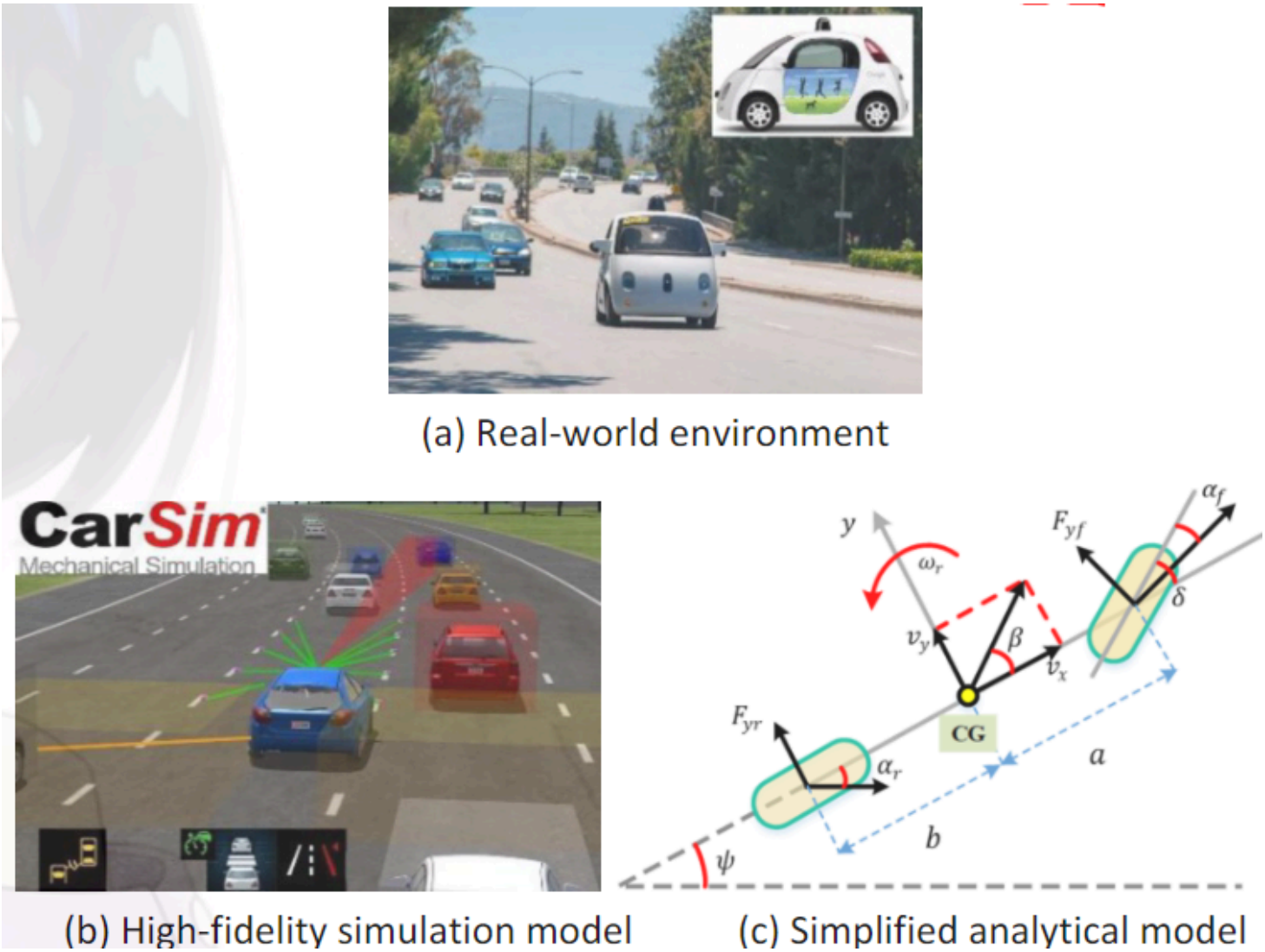
那么根据我们之前的ADP框架，当主循环的迭代次数趋于无穷大时，我们就可以得到最优的参数矩阵P^*。注意，这里的LQR问题可以看成是通用的ADP框架的一个特例，因为我们这里对于策略参数和值函数参数统一都使用了一个P矩阵来表示。到这里我们也可以总结一下了。实际上，使用ADP的框架来求解LQR问题的过程就是使用了迭代的方法来替代直接求解Riccati方程。

另外需要注意的是，LQR方法要求我们具有对于状态的完全的观测，但是在很多实际问题中，这是不现实的，会带来额外的观测误差，需要使用状态估计器来更好的估计状态。

8.3 怎么运行RL或者ADP算法

在这一节中，我们主要讨论如何运行RL或者ADP算法。注意，这里我们采用狭义的说法，RL特指model-free的方法，而ADP特指model-based的方法。ADP具有模型的好处是（解析）模型的可微性使得我们可以更有效的利用模型的信息；而RL因为没有模型，只能通过和环境不断交互来获得梯度的信息，因此效率低于ADP。

8.3.1 两种环境模型



环境模型是用来刻画环境的动力学信息的。通常来说，我们可以把环境模型分为两种：

- **高保真的仿真模型**
  - **简介**：这种模型是使用仿真软件进行模拟的，它可以提供高保真的环境信息，与真实的环境十分接近。
  - **优缺点**
    - **优点**：高保真的仿真模型可以提供准确的环境信息，这对于RL算法的训练是非常有帮助的；相比于在真实的环境中收集数据，使用仿真模型可以大大减少训练的时间和成本，并降低风险。
    - **缺点**：缺乏解析表达式，不容易获得导数信息。
- **简化的解析模型**
  - **简介**：这种模型是使用一些显式的解析表达式来描述环境的动力学信息的，比如常微分方程（ODE）。
  - **优缺点**

- **优点**：简化的解析模型可以提供解析的导数信息，这对于ADP算法的训练是非常有帮助的。在我们下面的讨论中，我们假设**model-based的方法使用的模型必须是可微的**。
- **缺点**：模型的表现很大程度上取决于建模的准确性。如果模型的准确性不够，那么对于梯度的估计的精度也会大大受到影响。

下表展示了三种模型（上述两种加上真实环境）在RL和ADP算法中的应用情况（实心圈代表适用，空心圈代表不适用）：

Environment	Model-free RL	Model-based ADP
(a) Real-world environment	●	○
(b) High-fidelity simulation model	●	○
(c) Simplified analytical model	●	●

8.3.2 Implementation of RL and ADP算法

8.3.2.1 梯度的计算

让我们来以具有随机策略的RL和具有确定性策略的ADP为例，来比较一下两种方法计算梯度的时候有什么不同。注意，我们下面的分析都是在actor-critic框架下进行的。

先来看看二者是如何计算critic的梯度的：

$$\nabla J_{\text{Critic}}^{\text{RL}} = -\mathbb{E}_s \left\{ \left(V^{\text{target}}(s) - V(s;w) \right) \frac{\partial V(s;w)}{\partial w} \right\}$$
$$\nabla J_{\text{Critic}}^{\text{ADP}} = -\mathbb{E}_x \left\{ \left(V^{\text{target}}(x) - V(x;w) \right) \frac{\partial V(x;w)}{\partial w} \right\}$$

可以发现，RL和ADP的critic的梯度计算是一样的（注，这里ADP说的是离散时间的版本，连续时间的ADP的critic公式与此形式不同）。再来看看actor的梯度的推导：

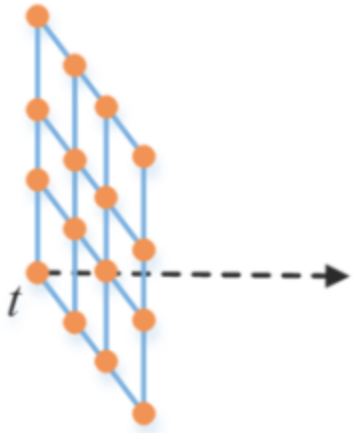
$$\nabla J_{\text{Actor}}^{\text{RL}} = \mathbb{E}_{s,a,s'} \{ \nabla_\theta \log \pi_\theta(a|s) \left(r + \gamma V(s';w) \right) \}$$
$$\nabla J_{\text{Actor}}^{\text{ADP}} = \mathbb{E}_x \left\{ \frac{\partial \pi^{\text{T}}(x;\theta)}{\partial \theta} \left(\frac{\partial l(x,u)}{\partial u} + \frac{\partial f^{\text{T}}(x,u)}{\partial u} \frac{\partial V(x';w)}{\partial x'} \right) \right\}$$

可以发现，此处关于梯度的计算就出现了很大的不同。区别就在于RL中的actor的梯度计算实际上是一种使用samples对于梯度进行的估计；而ADP则是直接利用了模型的导数信息。相比较而言，ADP算法的更新efficiency更高，但是相应的，其计算模型导数的代价也更高。

8.3.2.2 两种常见的采样方法

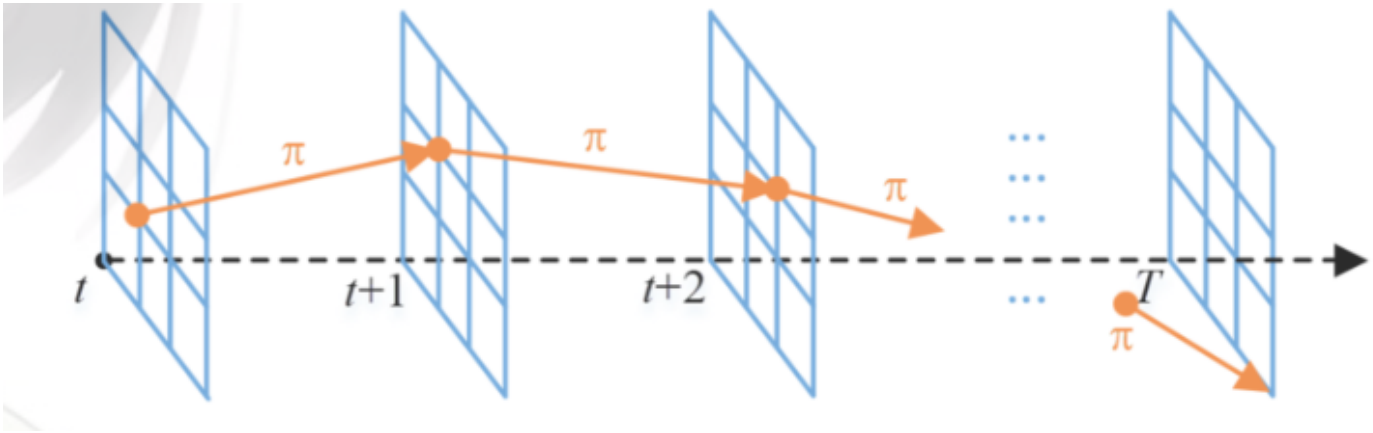
首先我们必须先明确一点，虽然ADP利用了模型的信息，但并不是说它就不需要和环境交互（即采样）。例子可以看看第8.1.5节的伪代码。在这一节中，我们主要讨论两种常见的采样方法：State Initial Sampling（SIS）State Rollout Forward（SRF）：

- **State Initial Sampling（SIS）**



  这种方法一次选择一个初始状态的分布，比如均匀分布或者高斯分布，然后根据这个初始状态的分布来采样。这个方法形象的理解就是一次选择若干个起点，然后同时出发向后至少走一步。

- **State Rollout Forward（SRF）**



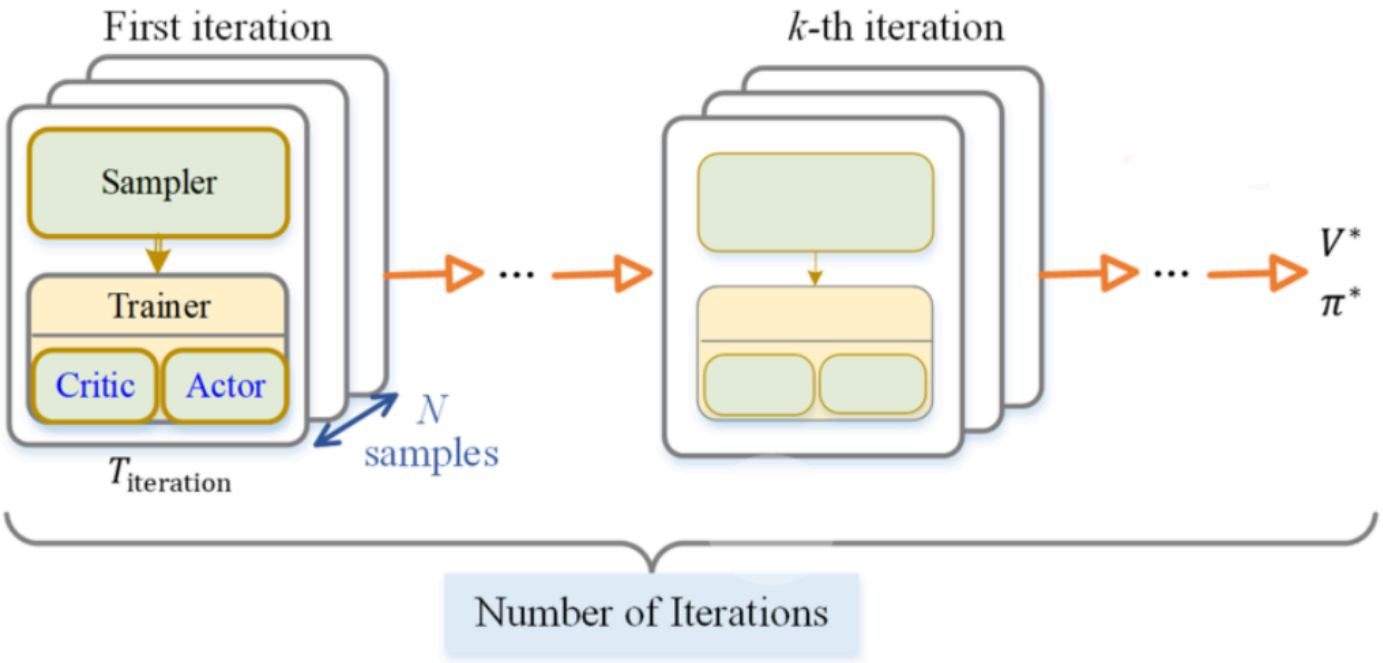
  这种方法是一次选择一个初始状态，然后从这个初始状态开始，根据当前的策略，一直向后rollout，然后得到一条包含state和action的轨迹。这个方法形象的理解就是一次选择一个起点，然后一直走到终点。对于RL来说，需要利用的是轨迹中每对(state,action)的信息；而对于ADP来说，需要利用的是轨迹中蕴含的state distribution的信息。与SRF相比，SIS的优点是其state distribution更接近于SSD。

一个更为常见的做法是结合两种方法，即同时选择若干个初始状态然后同时向后rollout。这种方法的优点有很多，比如一次可以产生一个batch而非单一的一条轨迹，这样有利于矩阵运算的加速；另外，这种方法可以更好地探索环境，尤其是当环境和策略都是确定性的时候。

8.3.3 RL与ADP算法的训练效率比较

训练效率主要由两个因素决定：

- **每轮主循环复杂度$T_{interation}$**
  这个复杂度由三部分决定：sampler、actor update和critic update。
- **达到某个表现指标所需的主循环轮数**
  这个轮数主要由对于梯度估计的质量决定。容易想到，高质量的梯度估计可以使得我们在更少的主循环轮数下达到某个表现指标。



下面让我们来讨论一下RL和ADP在每轮主循环复杂度$T_{interation}$的区别。先直接放结论再解释（我们在下面的讨论中假定使用神经网络来近似值函数和策略，这也是目前最广为使用的方法）：

	RL	ADP
Interaction	$T_{\text{env}} \cdot N$	$T_{\text{model}} \cdot N$
Critic	$(2T_{\text{FD}}^{V} + T_{\text{BP}}^{V}) \cdot n_c N$	$(2T_{\text{FD}}^{V} + T_{\text{BP}}^{V}) \cdot n_c N$
Actor	$(T_{\text{FD}}^{\pi} + T_{\text{BP}}^{\pi}) \cdot n_a N$	$(T_{\text{FD}}^{\pi} + T_{\text{BP}}^{\pi} + T^{f} + T^{l} + T_{\text{BP}}^{V}) \cdot n_a N$

- **Interaction**：这里的Interaction指的就是sampler，T_{env}和T_{model}分别代表了在真实环境和模型环境中所需的时间。N则代表样本的数量（表中下面的N意思相同）。
- **Critic**：根据我们在8.3.2.1节的讨论，RL和ADP的critic的梯度计算是一样的，因此这里的复杂度是一样的。在进行critic的更新时，主要涉及到三个部分的计算：V^{target}、$V(x;w)$和$\frac{\partial V(x;w)}{\partial w}$。前两者的计算复杂度是一样的，即一次critic网络的前向传播的时间，记作T_{FD}^{V}。而计算$\frac{\partial V(x;w)}{\partial w}$的复杂

度是一次critic网络的反向传播的时间，记作T_{BP}^V。因此，critic的单个样本复杂度是$2T_{FD}^V + T_{BP}^V$。再考虑到样本的数量N及critic的更新次数n_c，那么critic的总复杂度就是$n_c N(2T_{FD}^V + T_{BP}^V)$。

- **Actor**：actor部分是RL和ADP主循环复杂度的主区别所在。先来看RL。RL的actor的梯度计算主要有两个部分：$\nabla_\theta \log \pi_\theta(a|s)$和$V(s';w)$。前者的计算复杂度是一次actor网络的前向传播的时间加上一次actor网络的反向传播的时间，记作$T_{FD}^\pi + T_{BP}^\pi$。而后者已经在critic的更新中算过了，故不需要再计入。因此，actor的单个样本复杂度是$n_a N(T_{FD}^\pi + T_{BP}^\pi)$。再来看看ADP的复杂度，其主要与以下四个部分有关：$\frac{\partial l}{\partial u}$、$\frac{\partial f}{\partial u}$、$\frac{\partial V}{\partial x'}$和$\frac{\partial \pi}{\partial \theta}$。首先，我们把计算$\frac{\partial l}{\partial u}$和$\frac{\partial f}{\partial u}$的复杂度记作$T^l$和$T^f$，而这个两个时间因为已知环境和utility函数，因此计算开销是很小的。接下来是计算$\frac{\partial V}{\partial x'}$。这里需要经过一次critic网络的前向传播和反向传播，但是因为前向传播的那个计算已经在critic更新的时候算过，因此其复杂度是T_{BP}^V。可能有人会问，为什么这里需要的是一次正向加一次反向，而之前我们算过的$\frac{\partial V(x;w)}{\partial w}$只需要一次反向呢？答案在于$w$是网络的参数而状态$x'$是输入而不是参数，因此这里面牵扯到一个链式法则求导的问题。最后是计算$\frac{\partial \pi}{\partial \theta}$的复杂度，这个复杂度是一次actor网络的前向传播和反向传播的时间加上一次actor网络的反向传播的时间。综上，actor的单个样本复杂度是$n_a N(T^l + T^f + T_{BP}^V + T_{FD}^\pi + 2T_{BP}^\pi)$。

从上面的讨论我们可以看出，每轮主循环里ADP比RL更耗时。那么为什么我们说一般来说ADP比RL更高效呢？原因就在于在计算actor gradient的时候，我们直接利用了模型的信息而非从环境中直接采样得到的样本。这就避免了样本的随机性以及稀疏性带来的高方差，从而提高了梯度估计的质量，因此影响总的复杂度的第二个因素——达到某个表现指标所需的主循环轮数远远小于RL的，因此总的复杂度更小。

8.4 应用于追踪（Tracking）任务的ADP算法和它的策略结构

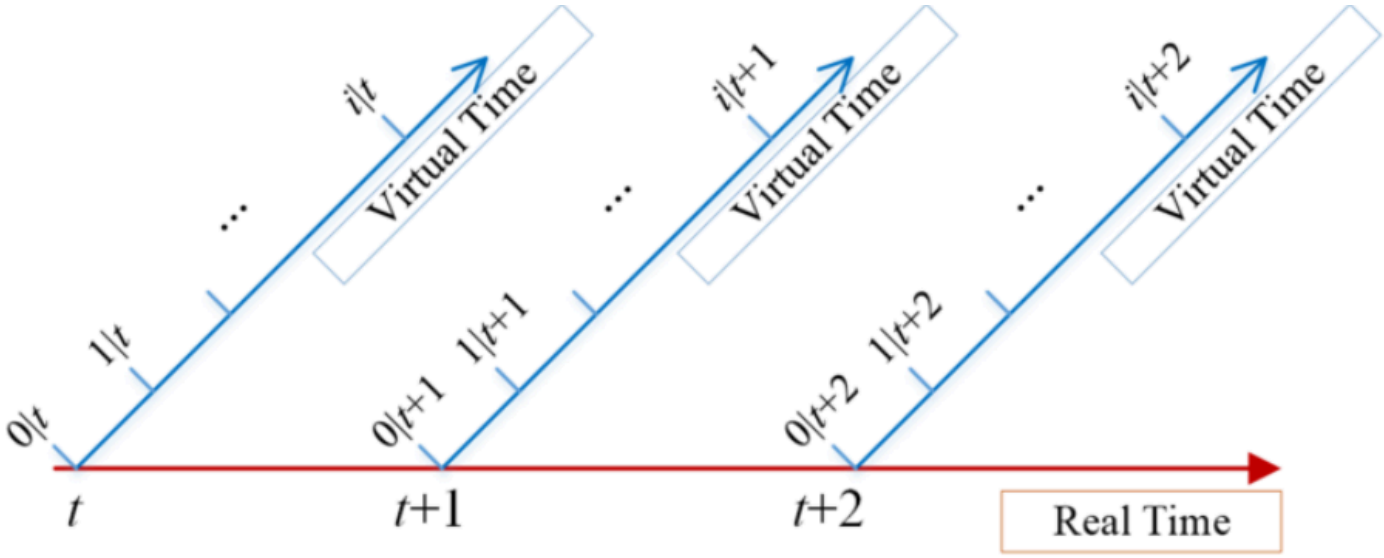
ADP和MPC都可以被视为一种求解通常的最优控制问题（OCP（的数值求解器。它们之间一个重要的区别在于怎么找到和应用最优策略。大多数的ADP算法是offline的，也就是说先在整个状态空间中进行离线的训练，之后就可以把训练好的策略视为一个最优控制器，来在online的快速给出最优动作。这样的offline的特点也使得ADP算算法具有很好的实时性。但是ADP算法常常会有一些performance loss，这是因为其策略通常是使用函数来近似的，那么如果函数本身的结构无法很好的近似我们的策略的话，就会出现上述情况。因此针对具体问题选择一个合适的策略结构是很重要的。

而MPC则是online的，它实时的计算一系列未来的最优策略，然后只执行第一个动作。这样很耗时间，但是好处是采取这样的控制策略得到的闭环策略与实际的开环的OCP的解是（几乎）一致的（除了一些可能的优化误差）。

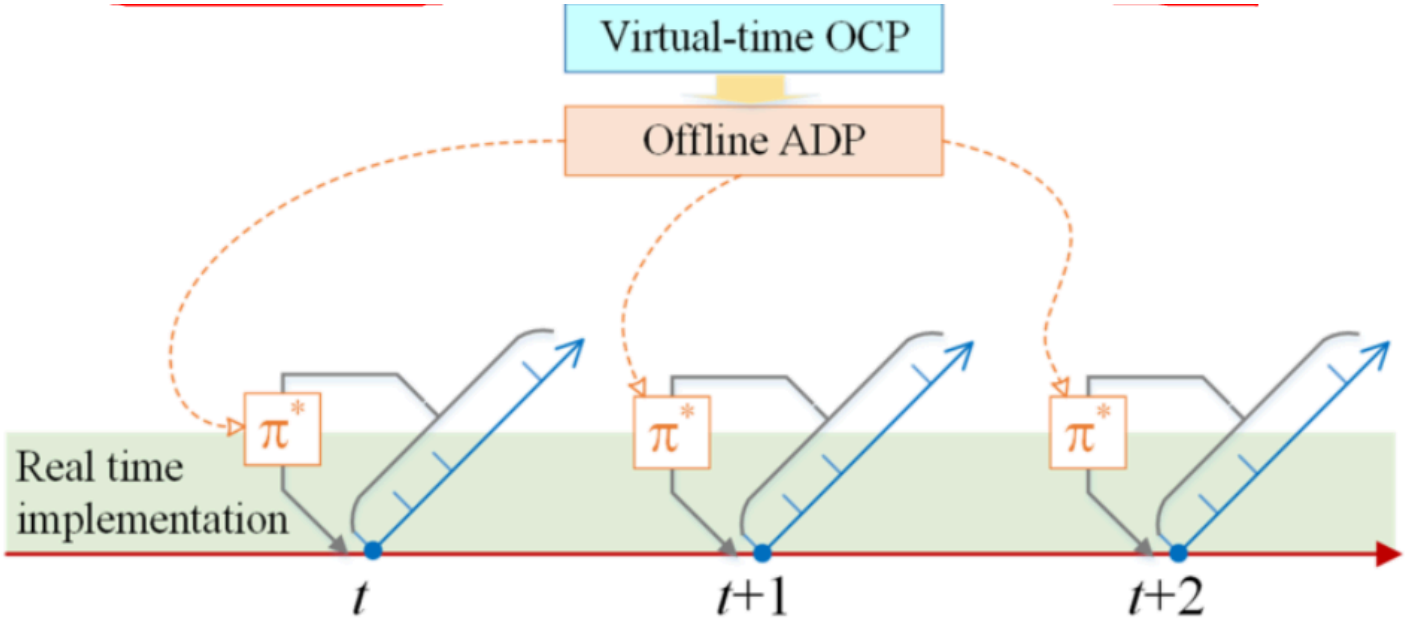
8.4.1 Receding Horizon Control（RHC）中的两种时间域

最优追踪器包含了一系列控制任务，它的设计需要仔细地匹配问题定义和策略结构。为了不失一般性，我们定义一个标准的无限视野的（infinite-horizon）的追踪问题来比较不同的策略结构。我们下面引入Reciding Horizon Control（RHC）来更好地解释怎么为最优追踪器选择一个合适的策略结构。

尽管没有明确说明，但是实际上在RHC的视角下有两种时间域：real-time domain和virtual-time domain。如下图所示：



把这两种时间域区分开也可以为我们分析ADP提供一个新的视角，而这在之前的工作中几乎从未被提及。在ADP的框架下，我们可以把最优策略的训练视为在virtual-time domain中进行，而最优策略的应用视为在real-time domain中进行。



这样的视角可以帮助我们设计一个更高效的采样方式和更好的策略结构。

8.4.2 在Real-time Domain中定义的Tracking ADP

首先，我们来定义一个infinite-horizon的，在real-time domain中定义的cost function（也就是值函数，只不过这里我们要最小化）：

$$V(x_t, X^R) = \sum_{i=0}^{\infty} l(x_{t+i}^R - x_{t+i}, u_{t+i}),$$

这里的t是real-time domain中的时间步，$X^R = \{x_t^R, x_{t+1}^R, \cdots, x_\infty^R\}$的概念。这里的轨迹不仅包含位置信息，还包含时间信息。而我们通常所说的那种只包含位置信息的其实是路径（path）。因此才会在我们的参考轨迹X^R中看到带有时间下标的状态，可以简单理解为在参考轨迹上的每个点代表在每个时间点应该到达什么状态。这里我们的值函数的输入不再只有当前状态，还包含了参考轨迹。而且这里的utility function里通常应该是状态的的位置现在被替换为了当前状态与参考轨迹上的状态的差值$x_{t+i}^R - x_{t+i}$。

我们将开环的最优动作序列记为$\{u_t^*, u_{t+1}^*, \cdots, u_\infty^*\}$，那么根据这个最优的开环最优动作序列，可以得到cost function的最优值。那么开环的Bellman方程可以写成：

$$V_{\text{Open}}^*(x_t, X^R) = \min_u \{l(x_t^R - x_t, u) + V_{\text{Open}}^*(x_{t+1}, X^R)\}.$$

下面我们来考察一下策略的结构。实际上，有下面两种常见的策略结构：

$$u_t = \pi(x_t, x_t^R),$$
$$u_t = \varphi(x_t, x_t^R, x_{t+1}^R, x_{t+2}^R, \cdots, x_\infty^R).$$

第一种策略的输入只有当前状态和参考轨迹上的当前参考点，而第二种策略的输入还包含了参考轨迹上的未来的所有参考点。那么，通过ADP的框架来最小化cost function，我们可以得到两种策略下的最优值函数$V_\pi^*(x)$ 和 $V_\varphi^*(x)$。实际上，第一种策略结构使用的最多。但是，这能否就能代表这第一种策略结构就是最优的呢？实际上并不是。因为$\pi(\cdot)$的结构十分受限，只有当前时间步的参考信息，不能很好的处理后续参考轨迹点。而$\varphi(\cdot)$才是理论上更好的选择，但是其受限于输入的高维度和训练的复杂度，因此难以广泛应用。

8.4.3 在Virtual-time Domain中重新定义的Tracking ADP

为了更好的理解之前我们说过的两种策略结构，我们需要把tracking这个在real-time中的OCP转换为一个在virtual-time中类似MPC的问题。注意，下面我们要从infinite-horizon的MPC角度来切入，虽然infinite-horizon的MPC在实际中不是很常用，但是他却与我们的ADP问题有着很大的联系。我们下面要把ADP写成一种infinite-horizon的MPC形式。根据我们下面的virtual-time的下标的不同，我们可以把重新构建的这个infinite-horizon的MPC问题分为两种情况：full-horizon追踪器和first-point追踪器。

在时间t（真实时间）时，full-horizon追踪器的目标函数可以写成：

$$J_{\mathrm{MPC}}^{\varphi}=\sum_{i=0}^{\infty}l(x_{i|t}^{R}-x_{i|t},u_{i|t}),$$

这里因为我们引入了virtual-time的概念，为了避免混淆，需要对于下面要用到的符号系统解释一下。我们用$(i|t)$中的t表示实际的时间（real-time），而i表示虚拟的时间（virtual-time）。而上述求和式中涉及到的$\chi_{0|t}^{R},\chi_{1|t}^{R},\cdots,\chi_{\infty|t}^{R}$是在当前**真实时间t固定**的条件下在virtual-time domain中的predicting horizon。这样子把真实时间和虚拟时间区分开来的做法为我们构建ADP追踪器提供了很大的灵活度。下面的问题就是$x_{i|t}^{R}$到底等于什么。其实这里的定义方法不止一种，下面就将一种最常用的定义方法：

$$x_{i|t}^{R}=x_{t+i}^{R},i\in\mathbb{N}.$$

这种定义方法实际上就是把从真实时间t开始的虚拟时间i等价为从t开始的真实时间。那么在最小化$J_{\mathrm{MPC}}^{\varphi}$之后我们就可以得到一个full-horizon追踪器的最优策略：

$$u(t)=\varphi\left(x_{0|t},x_{0|t}^{R},x_{1|t}^{R},\cdots,x_{\infty|t}^{R}\right),$$

当环境模型准确时，上述最优策略的开环控制最优性就等于原问题的闭环最优性。

下面我们来看看first-point追踪器。在时间t（真实时间）时，first-point追踪器的目标函数可以写成：

$$J_{\mathrm{MPC}}^{\pi}=\sum_{i=0}^{\infty}l(x_{0|t}^{R}-x_{i|t},u_{i|t})$$

注意这里的J_{MPC}^{π}和$J_{\mathrm{MPC}}^{\varphi}$的区别在于，这里与$x_{i|t}$做差的不再是$x_{i|t}^{R}$而是一个固定值$x_{0|t}^{R}$。在我们上面$x_{i|t}^{R}=x_{t+i}^{R},i\in\mathbb{N}$的设定下，这实际上就相当于拿t时刻的真实的轨迹上的值x_{t}^{R}与$x_{i|t}$做差。这样其实就蕴含了一个重要的假设，即：

$$x_{i|t}^{R}=x_{0|t}^{R}=x_{t}^{R},i\in\mathbb{N}.$$

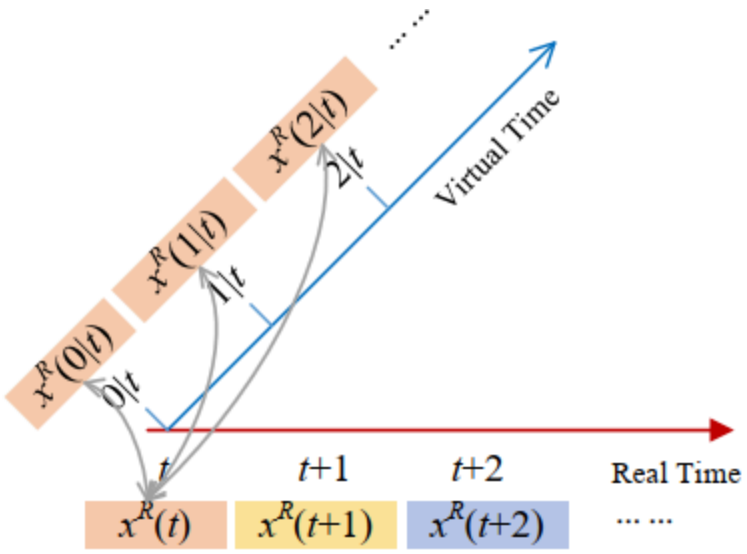
其实这里的$x_{0|t}^{R}$的选择也不是唯一的，比如还可以令其等于x_{t}^{R}的一个函数，只要令$x_{0|t}^{R}$由x_{t}^{R}唯一决定即可。通过最小化first-point追踪器的目标函数，我们可以得到一个first-point追踪器的最优策略：

$$u(t)=\pi\left(x_{0|t},x_{0|t}^{R}\right).$$

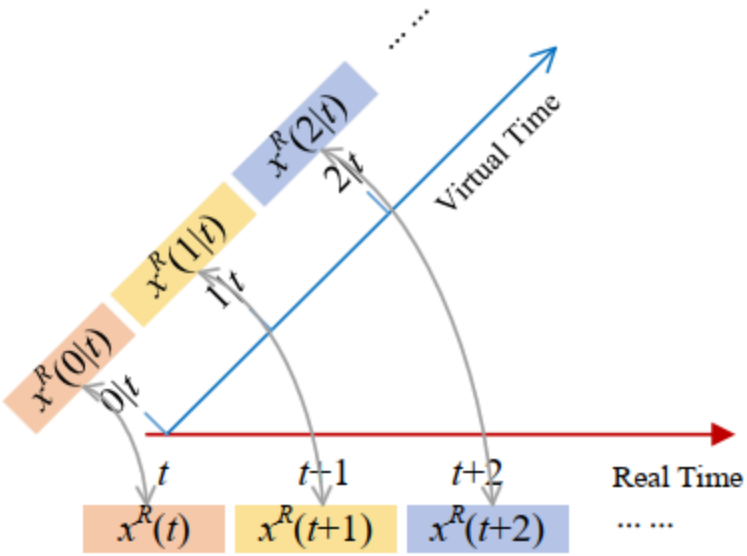
容易想到，π策略的最优性比φ策略的最优性要差，因此也达不到开环控制下的最优值，即：

$$V_{\pi}^{*}(x)\geq V_{\varphi}^{*}(x)=V_{\mathrm{Open}}^{*}(x),\forall x\in X$$

这种最优性的丧失可以被理解为π策略的结构受限（只有当前时间步的参考信息）。二者的比较可以由下图形象的表示：

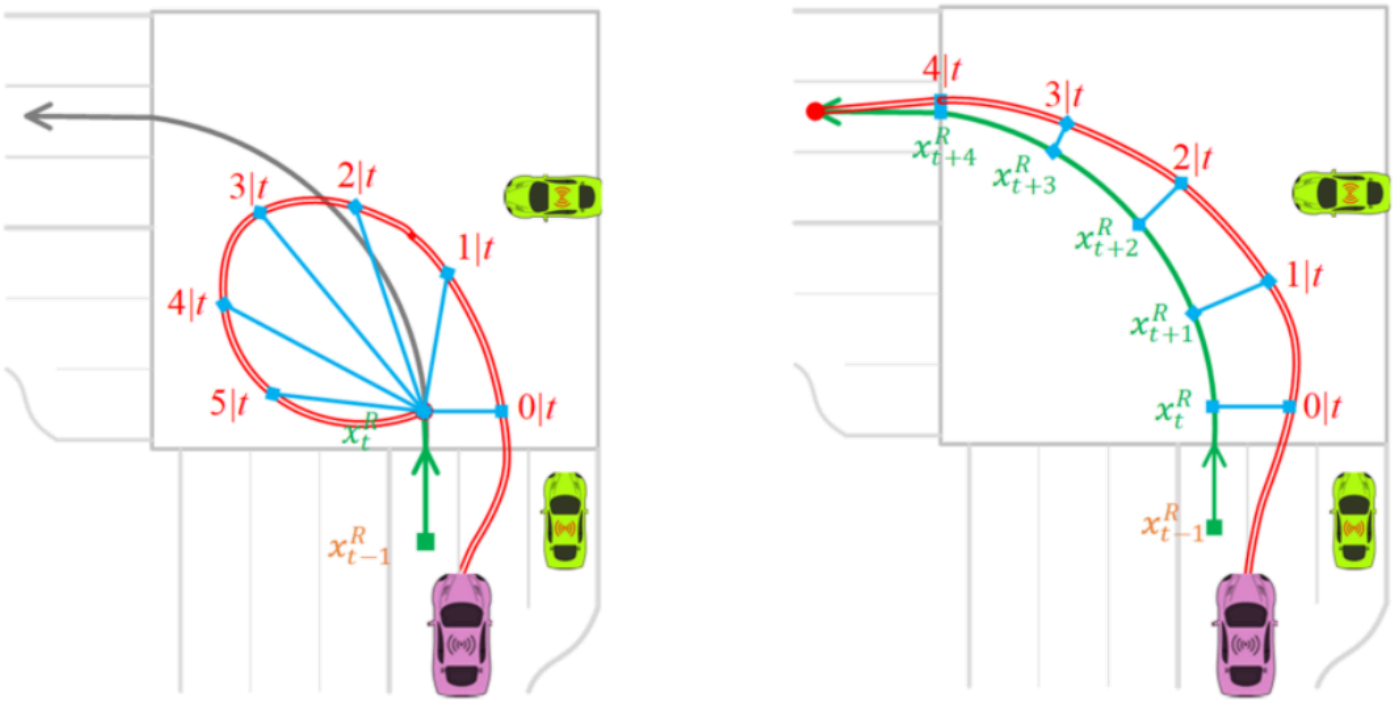


(a) Policy π (i.e., first-point tracker)



(b) Policy φ (i.e., full-horizon tracker)

或者我们再举一个更形象的例子。下面是一个自动驾驶汽车在拐弯处跟踪参考轨迹的例子。下图展示了某个真实时刻t时，汽车在virtual-time domain中的跟踪情况。其中蓝色的线代表参考轨迹，红色的线代表汽车的轨迹（在virtual-time domain中的轨迹）。



(a) Policy π (i.e., first-point tracker)　　(b) Policy φ (i.e., full-horizon tracker)

可以看出，左图（first-point追踪器）与右图（full-horizon追踪器）相比，跟踪效果有一定的损失。不过这在实际应用中是可以接受的，因为full-horizon追踪器的计算量太大。因此在实际应用中，我们通常会选择first-point追踪器，也可以使用下面的折中的改进办法。即我们不使用∞-horizon，而改成有限的horizon，并在其中令$x_{i|t}^R = x_{t+i}^R$。

8.4.4 以OCP的一个经典例子LQC为例来进行定量分析

Linear Quadratic Control（LQC）是一个经典的OCP问题。其实，LQC问题主要洋浦两类：LQ追踪器和LQ调节器（LQR）。它们的求解都是通过求解Riccati方程来得到最优的控制策略。对于LQ调节器来说，**状态总是被要求保持在平衡态附近（通常是0）**；而对于LQ追踪器来说，状态则是被要求跟踪一个参考轨迹。实际上，在应用LQC时，上述两种不同的任务经常被搞混。大部分使用者也不会去区分最优的regulator和tracker之间的区别。一个常见的错误做法是把追踪误差当作一个新的“状态”来使用LQR来解决追踪问题。令人惊奇的是，这样一个“Fake Tracker”在实际应用竟然也能取得不错的效果。但是，就它优化的目标函数来看，得到的策略显然不是最优的。这也是工程中常见的理论上错误但是实际上却能取得不错效果的例子。但是，既然我们这节就是讲这个的，我们就来好好看一下LQ追踪器和LQ调节器到底有什么区别。

先来看看LQ追踪问题的cost function：

$$V(x_t, X^R) = \sum_{i=t}^{T} \left((x_i^R - x_i)^{\mathrm{T}} Q (x_i^R - x_i) + u_i^{\mathrm{T}} R u_i \right),$$

$$\text{s.t.}$$

$$\chi_{t+1} = Ax_t + Bu_t$$

注意，这里有我们刚才在第8.4.2节中定义的目标函数的形式相比，区别在于这列我们既然已经明确了是线性结构，那么我们把$l(x_t^R - x_t, u_t)$具体化为了上述形式。并且我们把horizon从无穷改为了(T-t)。这里我们假设矩阵组合(A,B)是可控的，且系数矩阵$Q \geq 0, R > 0$。那么，我们可以得到LQ Tracking问题的的Bellman方程：

$$V^*(x_t, X^R) = \min_u \{(x_t^R - x_t)^{\mathrm{T}} Q(x_t^R - x_t) + u_t^{\mathrm{T}} R u_t + V^*(x_{t+1}, X^R)\}$$

由LQC的相关知识可知，最优值函数$V^*(x_t, X^R)$是一个二次型函数，可以写成如下形式：

$$V^*(x_t, X^R) = x_t^{\mathrm{T}} P_t x_t + 2x_t^{\mathrm{T}} \beta_t + \alpha_t$$

这里的最优值函数里面包含三个系数：P_t, β_t, α_t。这三个系数里面嵌入了包含时间信息的参考轨迹信息。最优的策略也可用上述三个系数中的两个P_t, β_t来表示：

$$u_t^* = -R^{-1}B^{\mathrm{T}}(P_{t+1}x_{t+1} + \beta_{t+1})$$

回顾之前在某篇博客的某处讲过的LQR的最优策略的形式，我们会发现区别就是多了一个前馈项β_{t+1}，它里面就包含了参考轨迹的信息。这也是LQ追踪器和LQ调节器的一个重要区别。站在更高的角度来看，LQR可以被视为LQ Tracking的一个特例，即参考轨迹是一个常数（通常为0）。下面我们需要关心的就是怎么在每一时刻求解该时刻对应的三个系数。可以通过被称为**Differential Riccati Equation**的方程来求解这三个系数：

$$\begin{aligned}
P_t &= Q + A^{\mathrm{T}} P_{t+1} A - A^{\mathrm{T}} P_{t+1} B(R + B^{\mathrm{T}} P_{t+1} B)^{-1} B^{\mathrm{T}} P_{t+1} A, \\
\beta_t &= (A^{\mathrm{T}} - A^{\mathrm{T}} P_{t+1} B(R + B^{\mathrm{T}} P_{t+1} B)^{-1} B^{\mathrm{T}})\beta_{t+1} - Q x_t^R, \\
\alpha_t &= \alpha_{t+1} + {x_t^R}^{\mathrm{T}} Q x_t^R - \beta_{t+1}^{\mathrm{T}} B(R + B^{\mathrm{T}} P_{t+1} B)^{-1} B^{\mathrm{T}} \beta_{t+1}
\end{aligned}$$

可以看出，P_{t+1}的求解根据第一式可以完全的由P_t确定，这与LQR的求解方式是一样的。但是，求解β_{t+1}时，就不仅需要β_t的值，还需要P_{t+1}和参考轨迹x_t^R的信息。而求解α_{t+1}时，需要的就更多了，同时需要α_t、x_t^R、P_{t+1}和β_{t+1}的信息才行。怎么求解上述方程对于定量分析不同的策略结构对于LQ Tracking问题的影响是很重要的。下面我们就来看看。

8.4.4.1 递归的求解Finite-horizon LQ Tracking问题

让我们考虑具有终止条件的finite-horizon LQ Tracking问题。我们把终止时间记为T，我们假设在T时刻，最优值函数$V^*(x_T, X^R) = 0$。而最优值函数根据我们上面的讨论具有二次型的形式$V^*(x_t, X^R) = x_t^{\mathrm{T}} P_t x_t + 2x_t^{\mathrm{T}} \beta_t + \alpha_t$。那么再考虑到终止状态的任意性，为了使得最优值函数恒为0，易知三个系数都应该为0，即：

$$P_T = 0, \beta_T = 0, \alpha_T = 0$$

那么，我们就可以根据diffential Riccati方程递归的从后向前求解出每一个时刻的三个系数。

另外，注意到最优策略u_t^*中包含系数β_{t+1}，这暗示了最优策略其实是受到时间t之后的参考轨迹信息的影响的。因此，我们的策略结构$u_t = \pi(x_t, x_t^R)$不是最优的，因为它只包含了当前时刻的参考轨迹信息，而忽视了之后的轨迹信息。真正的最优策略应该是

$$u_t = \varphi(x_t, x_t^R, x_{t+1}^R, \cdots, x_T^R)$$

8.4.4.2 Infinite-horizon LQ Tracking问题的稳定状态解

Infinite-horizon LQ追踪器与它对应的finite版本非常不同。对于infinte-horizon LQ tracking问题，我们可以将differential Riccati方程视为一个离散的动态系统，将他的三个系数视为三个状态(P_t, β_t, α_t)。为了保证稳定状态的解的存在，我们需要保证这个离散的动态系统是稳定的。那么既然这个系统是稳定的，自然就意味着上述三个状态变量可以收敛到它们的稳态，挡在时刻t达到稳态之后，我们有：

$$\begin{aligned} P = P_t = P_{t+1} = \cdots = P_\infty, \\ \beta = \beta_t = \beta_{t+1} = \cdots = \beta_\infty, \\ \alpha = \alpha_t = \alpha_{t+1} = \cdots = \alpha_\infty, \end{aligned}$$

这里我们使用P, β, α来表示稳态的解。那么我们把稳态的这三个值代入到differtial Riccati方程中，并记$D \overset{\text{def}}{=} A^{\mathrm{T}} - A^{\mathrm{T}}PB(R + B^{\mathrm{T}}PB)^{-1}B^{\mathrm{T}} - I$，则我们可以得到：

$$\beta^{\mathrm{T}}B(R + B^{\mathrm{T}}PB)^{-1}B^{\mathrm{T}}\beta = \beta^{\mathrm{T}}D^{\mathrm{T}}Q^{-1}D\beta,$$

那么上述等式在下面两个条件下成立：

$$\begin{cases} \beta = 0 \\ \qquad\qquad or \\ B(R + B^{\mathrm{T}}PB)^{-1}B^{\mathrm{T}} = D^{\mathrm{T}}Q^{-1}D \end{cases}$$

而后一个条件的等号成立可以进一步的等价于下面的式子：

$$Q = P$$

那么我们再把$Q = P$代入到differtial Riccati方程中，我们可以得到下述等式：

$$B(R + B^{\mathrm{T}}QB)^{-1}B^{\mathrm{T}} = Q^{-1}.$$

那么是否对于我们这里的infinite-horizon LQ Tracking问题来说，两种情况（$\beta = 0$和$B(R + B^{\mathrm{T}}QB)^{-1}B^{\mathrm{T}} = Q^{-1}$）是否都是合理的呢？其实根据秩的相关推导，可以证明后一种情况存在矛盾（详见原书第8.4.4.2节），因此只有$\beta = 0$是合理的。这个时候我们把稳态时$\beta = 0$代入到differtial Riccati方程的第二式，我们可以得到在稳态时下述很强的结论：

$$x_t^R = x_{t+1}^R = \cdots = x_\infty^R = 0.$$

那么**对于infinite-horizon LQ Tracking问题来说，最优解存在的当且仅当参考轨迹恒等于0**。这里的0-稳态也被称为self-harmonized reference。那么我们同样可以得知，如果我们的参考轨迹不是恒等于0的，那么具有infinite-horizon的LQ Tracker实际上不存在最优解。原因也很好理解，因为如果参考轨迹不是恒等于0的话，稳态的状态的误差就是一个非零值，这样得到的值函数就不是有界的。因此，**实际上我么不可能设计出一个infinite-horizon的LQ Tracker**，除非我们让它跟踪一个恒等于0的参考轨迹，但是这样的轨迹在实际中基本上是没有什么意义的，而且此时LQ Tracker就退化为了LQR了。

这样其实也解释了为什么在实际应用中，一些finite-horizon的LQ Tracker比它们对应的infinite版本更好。人们普遍认为，finite-horizon的LQ Tracker总是比infinite-horizon的LQ Tracker更差。这种想法是因为前者是一个local optimizer而后者是一个global optimizer。但是，正如我们上面所分析的，由于self-harmonized reference的限制，infinite的版本不可能跟踪任意的参考轨迹（特别是对于快速变化的参考轨迹）。因此，正因为infinite-horizon的LQ Tracker在理论上的不完备，其自然也就不具备最优性。

8.4.5 考虑到Reference Dynamics的Sampling机制

根据我们上面的讨论，Infinite-horizon Tracker的一个关键问题是**如果参考轨迹不是self-harmonized的话，那么最优策略就不存在**。尽管我们上面得出这个结论靠的是针对LQ问题的分析，但是得出的结论不变，只不过**对于更一般的问题其轨迹为self-harmonized的条件不一定是恒等于0了**。不是self-harmonized的轨迹在工程实践中很常见，但是却很少被讨论。因此本小节就来讨论一下。首先，我们先通过一个例子来直观的认识一下什么是非self-harmonized的参考轨迹（这里的轨迹只是一个形象的说法，实际上是状态随着时间变化的一个函数）。考虑到汽车的纵向控制，这里的状态向量由两个分量构成，分别是车的纵向位置和纵向速度。那么，一个非self-harmonized的参考轨迹可以这样设置：从某个时刻t开始，车的位置恒定不变，但是车的速度为一个非零值。这样的轨迹显然在物理上是不成立的，表现在utility function上就是utility function一直为非零值（即实际的轨迹始终无法跟上参考轨迹，因为参考轨迹在物理上就不可行），这样对于utility function求和的值函数就会发散到无穷大。

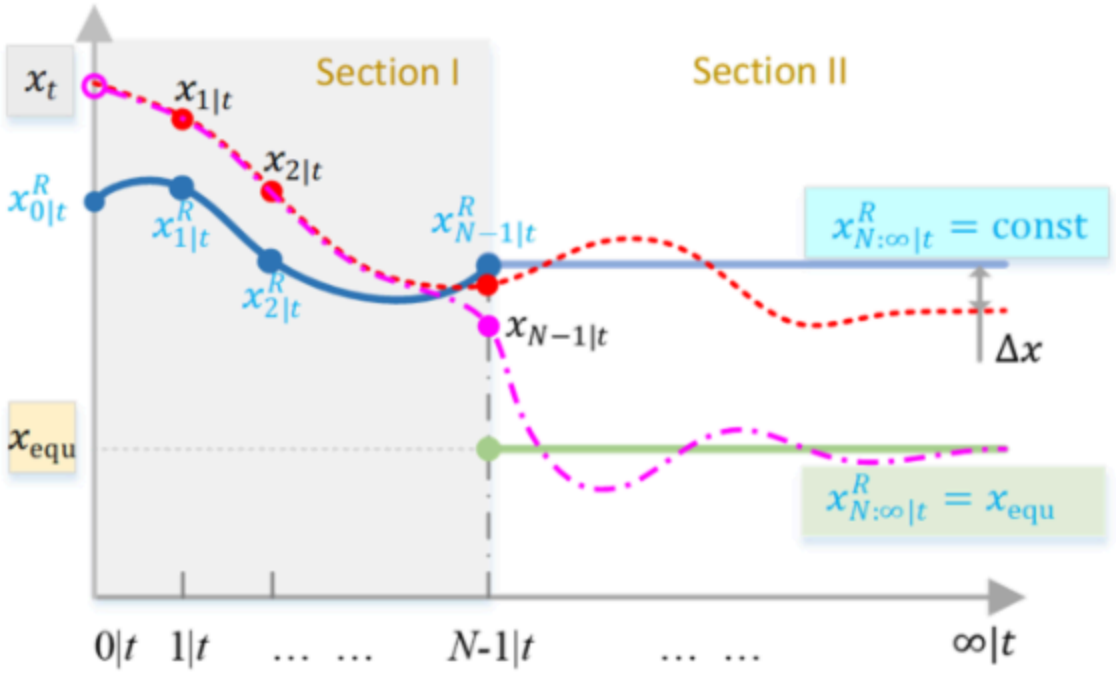
那么，如果轨迹本身就不是self-harmonized的，而我们又想跟踪这个轨迹并得到实用的结果，应该怎么做呢？答案就是去训练一个使用有限horizon的参考轨迹点作为输入的准最优策略。这里的核心思想是建立一个splitting cost function来代替原来的infinite-horizon的版本。为了理解这个splitting cost function工作的原理，我们需要在virtual-time domain中来定义和分析。在virtual-time domain中，我们把实际时间t对应的虚拟时间轴分成两段，分别是$\left[0|t,\ (N-1)|t\right]$和$\left[N|t,\ \infty|t\right]$。第一段的virtual-time reference使用实际的参考轨迹来保证期望的追踪能力，同时也作为准最优策略的输入的一部分。即：

$$x_{i|t}^{R}=x_{t+i}^{R}, i\in\mathbb{N}.$$

怎么选择**定义第二段的方式才是重新通过splitting cost function来重新得到self-harmonized的参考轨迹的关键**。这里有两种可能的选择：

- **选择第一段结尾的那个点作为第二段的值**：这样相当于任选一个常数作为第二段的参考轨迹的做法显然不合理，因为此时稳态误差仍然存在，因此刚才说的值函数无界的问题仍然存在。

- **选择最终的稳态值作为第二段的值**：这样的选择是合理的，因为这样的选择可以保证最终的稳态误差为0。



到这里为止，我们已经构建了一个splitting cost function。但是现在阻碍我们构建一个理论完备的ADP算法的鸿沟是什么？在于我们的cost function定义在virtual-time domain中，但是我们收集数据是在real-time domain中。因此我么必须先详细的分析reference dynamics和sampling机制之间的关系。

实际上，任何对于reference的修改都可以被视为项原始的环境中引入了一个特殊的reference dynamics。而新的reference dynamics可以被视为一个新的reference预测模型：

$$x^R_{i+N|t} = g\left(x^R_{i|t}, x^R_{i+1|t}, \cdots, x^R_{i+N-1|t}\right)$$

这里$g(\cdot)$可以根据过去的N步的参考信号来预测下一步的参考信号。这也解释了我们把虚拟时间分成两段的合理性，因为这个滑动窗口的大小为N，因此一开始必须有N个已知值才能向后滑动。怎么定义这个函数也是一个学问，需要根据实际需要来确定。如果按照我们之前说过的把最终的稳态值作为第二段的参考轨迹的话，那么这个函数就可以简单的定义为：

$$g_{t+N} \overset{\text{def}}{=} g\left(x^R_{i|t}, x^R_{i+1:i+N-1|t}\right) = x_{\text{equ}}$$

为了配合上述重构的追踪问题，我们需要给出追踪版本的Bellman方程：

$$
\begin{aligned}
&V^*\left(x_{0|t}, x^R_{0:N-1|t}\right) \\
&= \min_{u_{0:\infty|t}} \left\{ l\left(x^R_{0|t} - x_{0|t}, u_{0|t}\right) + \sum_{i=1}^{\infty} l\left(x^R_{i|t} - x_{i|t}, u_{i|t}\right) \right\} \\
&= \min_{u_{0|t}} \left\{ l\left(x^R_{0|t} - x_{0|t}, u_{0|t}\right) + \min_{u_{1:\infty|t}} \left\{ \sum_{i=1}^{\infty} l\left(x^R_{i|t} - x_{i|t}, u_{i|t}\right) \right\} \right\} \\
&= \min_{u_{0|t}} \{ l\left(x^R_{0|t} - x_{0|t}, u_{0|t}\right) + V^*\left(x_{1|t}, x^R_{1:N|t}\right) \}
\end{aligned}
$$

最小化这个Bellman方程的过程就能得到一个准最优的策略：

$$
u^*_t = \pi\underbrace{\left(x_{0|t}, x^R_{0:N-1|t}\right)}_{\text{virtual time}} = \pi\underbrace{(x_t, x^R_{t:t+N-1})}_{\text{real time}}.
$$

上式中从virtual time到real time之间的转换是因为我们的splitting的virtual时间轴的第一段采用的是和real-time同步的形式。可以看出，上述策略共有N+1个输入：当前的实际状态以及在virtual-time domain中的N个参考轨迹点。

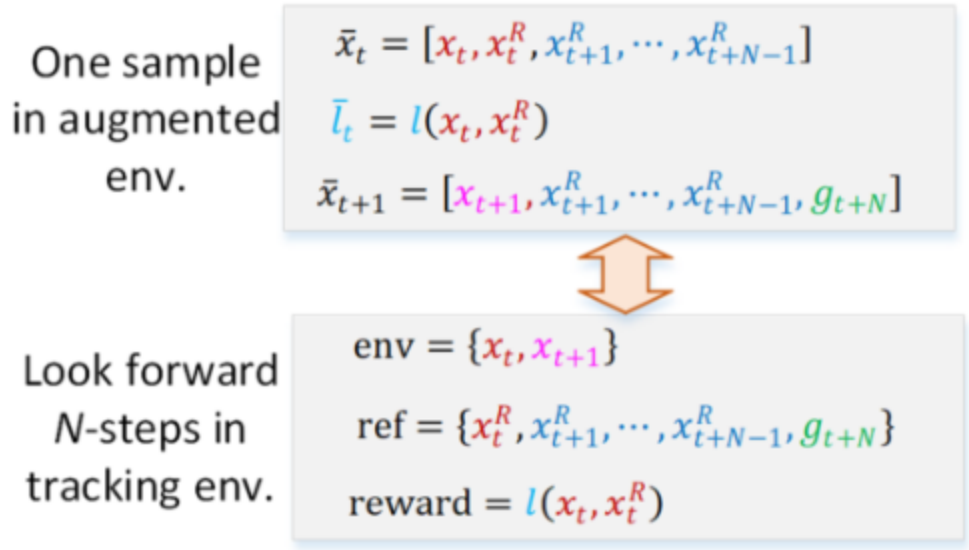
需要考虑的另一个问题是，在引入了reference dynamics之后，我们的环境不在保持Markov性质。为了保持Markov性质，我们需要构建如下的augmented state：

$$
\bar{x}_{i|t} = \left[x_{i|t}, \underbrace{x^R_{i|t}, x^R_{i+1|t}, \cdots, x^R_{i+N-1|t}}_{\text{Reference}} \right]^{\mathrm{T}}, i \in \mathbb{N}
$$

那么augmented的环境模型可以写成：

$$
\bar{x}_{i+1|t} = \begin{bmatrix} x_{i+1|t} \\ x^R_{i+1|t} \\ \cdots \\ x^R_{i+N-1|t} \\ x^R_{i+N|t} \end{bmatrix} = \begin{bmatrix} f\left(x_{i|t}, u_{i|t}\right) \\ x^R_{i+1|t} \\ \cdots \\ x^R_{i+N-1|t} \\ g\left(x^R_{i|t}, x^R_{i+1|t}, ..., x^R_{i+N-1|t}\right) \end{bmatrix} = f_{\text{aug}}\left(\bar{x}_{i|t}, u_{i|t}\right)
$$

其中，对于$\bar{x}_{i+1|t}$的最后一个元素是由我们刚才定义的reference dynamics模型$g(\cdot)$给出的，reference dynamics信息也就这样被嵌入了原始的环境模型中去。那么我们就可以这样收集samples：先收集N步的真实reference信号，再使用reference dynamics模型来预测最后一个参考信号。这样不断使用新的环境模型f_{aug}（已嵌入了reference dynamics信息）rollout就可以获得我们需要的samples，每个sample包含了两个实际的状态和N个实际的参考信号以及1个预测的参考信号，如下图所示：



那么，如果我们坚持使用real-time reference来采集samples会怎么样呢？这个时候，我们来分成有一个or多个reference轨迹来讨论（当然这时只有当这些轨迹都self-harmonized的时候才有意义）。当只有一个self-harmonized的参考轨迹时，最终的得到的是一个time-dependent的策略，其输出只由当前状态和当前实践决定。而当有多个self-harmonized的参考轨迹时，我们学得的策略里面隐含了某种在参考轨迹之间转移的分布，这样学得的策略就具有更好的探索和泛化能力。

到此为止，我们就已经介绍完了使用考虑到reference dynamics的sampling机制来解决infinite-horizon追踪器来追踪非self-harmonized参考轨迹的方法。可以看出，splitting cost function和reference dynamics的引入使得我们有能力跟踪一个非self-harmonized的参考轨迹，而不至于陷入值函数（cost function）无界的困境。

8.5 设计Finite-horizon ADP的方法

之前我们讨论的ADP都是基于infinite-horizon的，我们需要很仔细地保证值函数是有界的。而对于finite-horizon的最优控制问题，我们就可以轻松一点了。因为这里我们很容易把值函数做到有界，而它的policy gradient的推导也是一种基于cascading的方式。

但是，现在大多数实用的finite-horizon的最优控制问题都是基于MPC来做的。MPC确实具有很多优点，例如可以处理非线性系统而不用可以对于系统进行线性化、显式的考虑了系统的物理上或者操作上的限制以及对于多种控制问题（比如set-point调调节、追踪等）都适用。但是，它的online优化的特性也使得它的计算负担很重，因此只能处理那些动态很缓慢、采样周期很长、纬度低的系统。因此我们下面有必要来讨论一下如何设计一个finite-horizon的ADP算法来为finite-horizon的最优控制问题提供一个替代的方法。

8.5.1 Finite-horizon ADP的基础

我们首先要明确我们究竟要讨论的是什么问题。在infinite-horizon的情况下，我们既讨论了离散时间的版本也讨论了连续时间的版本。但是在finite-horizon的版本的讨论中，情况就完全不同了。不同于连续时间版本中的HJB方程，finite-horizon的问题中的值函数具有time-dependent的结构，因此此时的HJB方程常常在计算上是不可行的（尤其在系统具有很强的非线性和高维度的情况下）。因此，我们下面都只讨论定义在离散时间上的finite-horizon的ADP算法。这类算法也具有一些额外的特点，比如说值函数是time-dependent的，对于reward采不采取折扣因子没有那么敏感以及内在的multistage的策略结构。

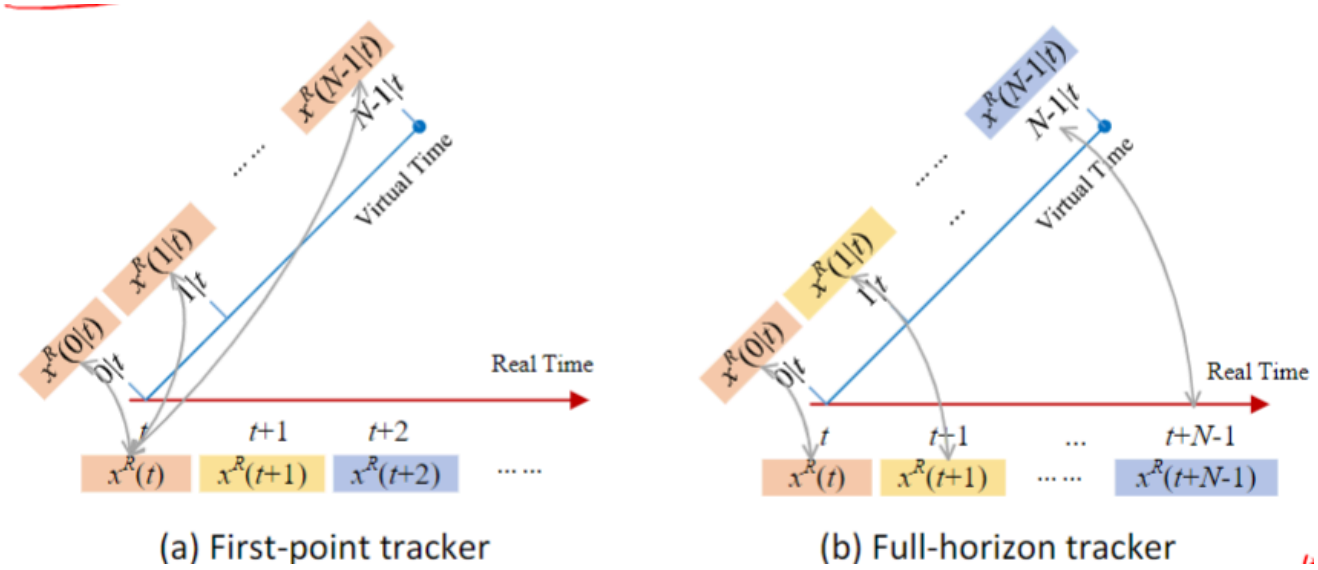
首先，为了下面构建算法的需要，我们仍然需压迫保持在infinite那里做的两个假设，即化境具有Markov性质以及cost function的可分离性。这里易知我们的finite-horizon的最优控制问题自然满足这两个假设。那么我们可以把finite-horizon的最优控制问题写成如下形式：

$$\min_{u_t,\dots,u_{t+N-1}} V(x, X^R) = \sum_{i=0}^{N-1} l(x_{t+i}, x_{t+i}^R, u_{t+i})$$
$$\text{s.t.}$$
$$x_{t+1} = f(x_t, u_t)$$

注意，这里的utility function不一定是正定的了，因为这里的值函数的有界性很好保证。这里可能有读者会疑惑，为什么这里的utility function中包含了参考信息x_{t+i}^R，这里又没有说一定是tracking问题。实际上，这只是一种统一建模的方法，在下面会讲。

实际上，我们可以将finite-horizon的OCP分为以下两类：

- **Regulator**：这里的参考轨迹恒为一个常数，即稳态值$x_e qu$。对于大多数问题来说，这个稳态值就是0。
- **Tracker**：这里的参考轨迹是一个随时间变化的函数。我们可以进一步将tracker分为下面两类：
  - **First-point tracker**：这里当每步在virtual-time域中求解时只用到第一个此时轨迹上的参考点（即位于时间t的参考点）。
  - **Full-horizon tracker**：这里在每一步求解时用到了整个horizon上的参考点。



对于finite-horizon的问题，标准的Bellman方程不在存在，我们需要引入multistage的Bellman方程。stage的数量取决于horizon的长度。在Exact DP中，这个multistage的Bellman方程是使用stage-by-stage的backward的方式计算的。而在ADP中，我们需要给出新的算法来求解这个multistage的Bellman方程。

我们再来考察一下对于一个finite-horizon的ADP算法很重要的一点：怎么使用函数对策略进行近似。因为每步的最有动作不仅取决于现在的状态，还取决于参考序列（含时间信息），因此我们的策略也应该是采取time-dependent的形式（在我们上面统一纳入了参考信息的OCP建模下）。下面来看看对于tracker和regulator的情况下，策略的形式：

- **对于最优regulator来说的策略结构**：

  $$u_{t+i} = \pi(x_t, N-i; \theta), i = 0, 1, \ldots, N-1$$

  注意，**与infinite-horizon的情况不同，这里的策略必须包含时间信息**。
- **对于最优tracker来说的策略结构**：

  $$u_{t+i} = \pi(x_t, x_{t+i:t+N-1}; \theta_i), i = 0, 1, \ldots, N-1$$

  注意，**这里对于一个horizon长度为N的问题，我们需要N套不同的参数**$\theta_0 \in \mathbb{R}^{l_0}, \theta_1 \in \mathbb{R}^{l_1}, \ldots, \theta_{N-1} \in \mathbb{R}^{l_{N-1}}$。**这是因为参考轨迹是不断变化的，没法使用一套固定的参数来跟踪（上面的regulator的情况是因为参考轨迹是恒定的，因此只需要一套参数）**。

最后，求解finite-horizon的ADP问题有两种主流的算法：value-based ADP和policy-based ADP。前者先计算出最优的值函数再通过最优值函数来计算最优策略，而后者直接通过梯度下降来求解最优策略。

8.5.2 最优Regulator的Finite-horizon ADP算法

首先，我们为了下面能够得出multi-stage的Bellman方程，需要对上面提到过的值函数形式做一点变形：

$$V^*(x,t)=\min_{u_t,u_{t+1},\ldots,u_T}\sum_{i=t}^{T}l(x_i,u_i).$$

可以看到，与之前在8.5.1节的Problem Formulation那里定义的值函数的形式略有不同。那里的值函数形式及是**固定horizon长度**（horizon长为N），而这里的值函数是**固定终止时间为T，初始时间t可变**。为什么在之前已经定义了一种值函数的形式的情况下要再顶i一种值函数形式呢？是多此一举吗？**当然不是**！这两种形式必然是在我们后面的推导中各有用处。本处引入的这种终止时间固定为T的值函数的好处在于可以推导出下面的multi-stage的Bellman方程，而之前的那种固定horizon长度的值函数因为其horizon长度固定且输入参数不显含时间t（其形式为$V(x,X^R)$），因此无法得出multi-stage的Bellman方程。那么，我们就来看看这个multi-stage的Bellman方程是怎么样的：

$$\begin{aligned}V(x,t)&=\min_u\{l(x,u)+V(x',t+1)\},\\V(x,t+1)&=\min_u\{l(x,u)+V(x',t+2)\},\\V(x,T-1)&=\min_u\{l(x,u)+V(x',T)\},\\V(x,T)&=\min_u\{l(x,u)\}.\end{aligned}$$

这里的方程个数等于horizon的长度（由初始时间t和终止时间T决定）。

那么我们应该怎样由这个Bellman方程得到最优策略呢？一种直接的想法是我们为每个时间步都使用一个函数来拟合该时间步的值函数（即$V(x,t;w_t),V(x,t+1;w_{t+1}),\ V(x,t+2;w_{t+2}),\ \ldots,\ \mathrm{and}V(x,T;w_T)$），然后使用值迭代的技术来求解这些值函数的最优值。在得到最优值之后就可以得到最优策略。但是这样进行参数化的方式会带来超高维的问题，并导致计算效率极其低下（我的理解是因为这里的初始时间t不确定，我们要想覆盖所有的t，那么我们的参数空间就会变得非常大）。因此，我们需要另辟蹊径。我们可以利用**这里值函数对于utility function的求和是有限步**来作为突破口。下面的方法中就不需要显式的对于值函数进行建模了，只需要递归的计算policy gradient就可以了。下面我们就来看看这个方法。

首先，我们可以这样更新策略参数θ：

$$\theta\leftarrow\theta-\alpha\frac{\mathrm{d}V(x)}{\mathrm{d}\theta}$$

与之前得到multi-stage的Bellman方程时必须使用固定终止时间T的值函数不同，这里我们**必须用回固定horizon长度N的值函数**。另外，考虑到这两种值函数实际上在t=0且T=N-1时是等价的，因此我们可以

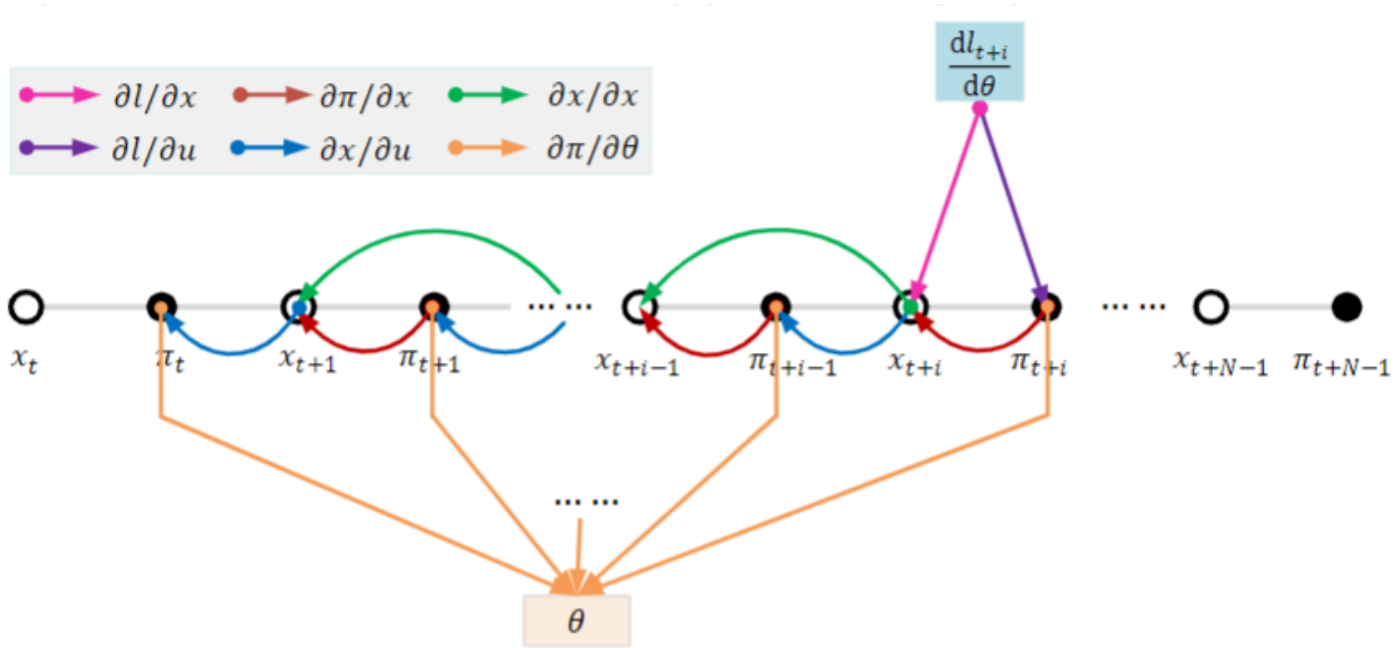
使用由固定终止时间T的值函数得到的multi-stage的Bellman方程来计算要求值函数是固定horizon长度N的值函数的梯度。另外为了递归计算的方便，我们需要改写8.5.1节中的optimal regulator的策略结构：

$$u_{t+i} = \pi(x_t, N-i; \theta), i = 0, 1, \ldots, N-1$$
$$\downarrow$$
$$u_{t+i} = \pi(x_{t+i}, N-i; \theta), i = 0, 1, ..., N-1$$

这里最大的变化就是我们上面那个策略结构是根据当前状态x_t来决定未来预测horizon（长度为N）的各个最优动作值的，而我们修改后的策略结构是根据每个时刻的状态x_{t+i}来决定对应时间步的最优动作。那么，我们的policy就可以写成如下形式：

$$\frac{\mathrm{d}V(x)}{\mathrm{d}\theta} = \sum_{i=0}^{N-1} \frac{\mathrm{d}l(x_{t+i}, u_{t+i})}{\mathrm{d}\theta}$$

现在的问题归结为怎么计算每个$\frac{\mathrm{d}l(x_{t+i}, u_{t+i})}{\mathrm{d}\theta}$。其实答案就是很简单，就是使用链式法则，只不过这里的依赖关系之间比较复杂，可以被建模为一个有向无环图，这样一张图的起点是$l(x_{t+i}, u_{t+i})$，终点是θ：



为了简单起见，我们引入下面的记号：

$$l_{t+i} \overset{\text{def}}{=} l(x_{t+i}, u_{t+i}),$$
$$f_{t+i} \overset{\text{def}}{=} f(x_{t+i}, u_{t+i}),$$
$$\pi_{t+i} \overset{\text{def}}{=} \pi(x_{t+i}, N-i; \theta).$$

引入上述记号后，我们有：

$$\frac{\mathrm{d}l_{t+i}}{\mathrm{d}\theta}=\frac{\mathrm{d}x_{t+i}^{\mathrm{T}}}{\mathrm{d}\theta}\frac{\partial l_{t+i}}{\partial x_{t+i}}+\frac{\mathrm{d}u_{t+i}^{\mathrm{T}}}{\mathrm{d}\theta}\frac{\partial l_{t+i}}{\partial u_{t+i}}$$

这个式子就是上图里面从$dl_{t+i}/d\theta$指向下面的红色和紫色的线。接下来，继续引入记号来化简：

$$\phi_{t+i}\overset{\text{def}}{=}\frac{\mathrm{d}x_{t+i}^{\mathrm{T}}}{\mathrm{d}\theta},\psi_{t+i}\overset{\text{def}}{=}\frac{\mathrm{d}u_{t+i}^{\mathrm{T}}}{\mathrm{d}\theta}=\frac{\mathrm{d}\pi_{t+i}^{\mathrm{T}}}{\mathrm{d}\theta}$$

然后通过观察上图，我们可以得到如下ϕ和ψ在相邻两个时间不得递归关系：

$$\begin{aligned}\phi_{t+i}&=\phi_{t+i-1}\frac{\partial f_{t+i-1}^{\mathrm{T}}}{\partial x_{t+i-1}}+\psi_{t+i-1}\frac{\partial f_{t+i-1}^{\mathrm{T}}}{\partial u_{t+i-1}}\ ,\\\psi_{t+i}&=\phi_{t+i}\frac{\partial \pi_{t+i}^{\mathrm{T}}}{\partial x_{t+i}}+\frac{\partial \pi_{t+i}^{\mathrm{T}}}{\partial \theta}\ ,\end{aligned}$$

注意，这里的推导中利用了两个关系式：$x_{t+i}=f_{t+i-1}$（环境模型）和$u_{t+i}=\pi_{t+i}$（因为策略是确定性的）。并且在ψ_{t+i}的递归关系中，左边有一个ϕ_{t+i}（即$d\pi_{t+i}^{\mathrm{T}}/d\theta$）而右边又有一个$\partial\pi_{t+i}^{\mathrm{T}}/\partial\theta$，可能有读者会疑惑，这两者难道不相等吗？确实不相等！结合高数知识，就可以知道二者根本不是一个东西。$d\pi_{t+i}^{\mathrm{T}}/d\theta$的含义是我这个$\pi_{t+i}$就是一个关于$\theta$的函数，我这里的对于$\theta$求导数（**这里不是偏导**！！！）就是要遍历上述DAG中从π_{t+i}开始到θ的所有路径，求导求到底；而$\partial\pi_{t+i}^{\mathrm{T}}/\partial\theta$这里我们不用求导求到底，只需要求最浅层的那个导数就可以了（及上述DAG中从π_{t+i}到θ的黄色那条边）。那么我们就可以把policy gradient改写为下述形式：

$$\frac{\mathrm{d}V(x)}{\mathrm{d}\theta}=\sum_{i=0}^{N-1}\frac{\mathrm{d}l(x_{t+i},u_{t+i})}{\mathrm{d}\theta}=\sum_{i=0}^{N-1}\left(\phi_{t+i}\frac{\partial l_{t+i}}{\partial x_{t+i}}+\psi_{t+i}\frac{\partial l_{t+i}}{\partial u_{t+i}}\right)$$

上式中的ϕ_{t+i}和ψ_{t+i}可以通过递归的方式计算得到。给定初始条件$\phi_{t+N-1}=0$（因为初始状态是已经给定的，与策略无关，自然也就与策略参数无关）和$\psi_{t+N-1}=\partial\pi^{\mathrm{T}}(x_0,N;\theta)/\partial\theta$。推到到这里，我们就可以看出我们上面这样重构问题的好处了，这里我们不再需要计算值函数相关的项了，只需要计算一些偏导数即可。这也是在离散时间域上的finite-horizon的ADP算法的一个好处，我们不需要花费计算资源去计算一个time-dependent的值函数。反观在连续时间域上的finite-horizon的ADP算法，它的HJB方程因为含有time-dependent的值函数因此是一个非线性的PDE，这样的PDE的求解是非常困难的。

最后来看看整个算法的伪代码：

Hyperparameters: learning rate α, predictive horizon N, number of environment resets M

Initialization: policy function $\pi(x, \#; \theta)$

Repeat

(1) Use environment model

Initialize memory buffer $\mathcal{D} \leftarrow \emptyset$

Repeat M environment resets

$x_0 \sim d_{\text{init}}(x)$ 这里 x_0 是从初始分布中采样的

$u_0 = \pi(x_0, N; \theta)$ 因此与策略参数 θ 无关

$\phi_0 = 0$ 所以 $\phi_0 = 0$

$\psi_0 = \partial \pi^{\mathrm{T}}(x_0, N; \theta)/\partial\theta$ $\psi_0 = 0 + \frac{\partial \pi_t}{\partial \theta} = \partial \pi_t / \partial \theta$

For i in1,2, … , $N-1$

//Rollout with model f and policy π

$x_i = f(x_{i-1}, u_{i-1})$

$u_i = \pi(x_i, N-i; \theta)$

$$\text{Calculate}\left(l, V, \frac{\partial l}{\partial u}, \frac{\partial l}{\partial x}, \frac{\partial f^{\mathrm{T}}}{\partial u}, \frac{\partial f^{\mathrm{T}}}{\partial x}, \frac{\partial \pi^{\mathrm{T}}}{\partial x}, \frac{\partial \pi^{\mathrm{T}}}{\partial \theta}\right)_i$$

$$\phi_i = \phi_{i-1}\left(\frac{\partial f^{\mathrm{T}}}{\partial x}\right)_{i-1} + \psi_{i-1}\left(\frac{\partial f^{\mathrm{T}}}{\partial u}\right)_{i-1}$$

$$\psi_i = \phi_i\left(\frac{\partial \pi^{\mathrm{T}}}{\partial x}\right)_i + \frac{\partial \pi^{\mathrm{T}}(x_i, N-i; \theta)}{\partial \theta}$$

逆向计算中间变量

共M条MPC sample，每个的 horizon 长为 N

End

$$\frac{\mathrm{d}V(x_0)}{\mathrm{d}\theta} = \sum_{i=0}^{N-1}\left(\phi_i\left(\frac{\partial l}{\partial x}\right)_i + \psi_i\left(\frac{\partial l}{\partial u}\right)_i\right)$$

$$\mathcal{D} \leftarrow \mathcal{D} \cup \left\{\frac{\mathrm{d}V(x_0)}{\mathrm{d}\theta}\right\}$$

End

(2) Actor update

$$\nabla_\theta J_{\mathrm{ADP}} \leftarrow \frac{1}{M}\sum_{\mathcal{D}}\frac{\mathrm{d}V(x_0)}{\mathrm{d}\theta}$$

$$\theta \leftarrow \theta - \alpha \cdot \nabla_\theta J_{\mathrm{ADP}}$$

End

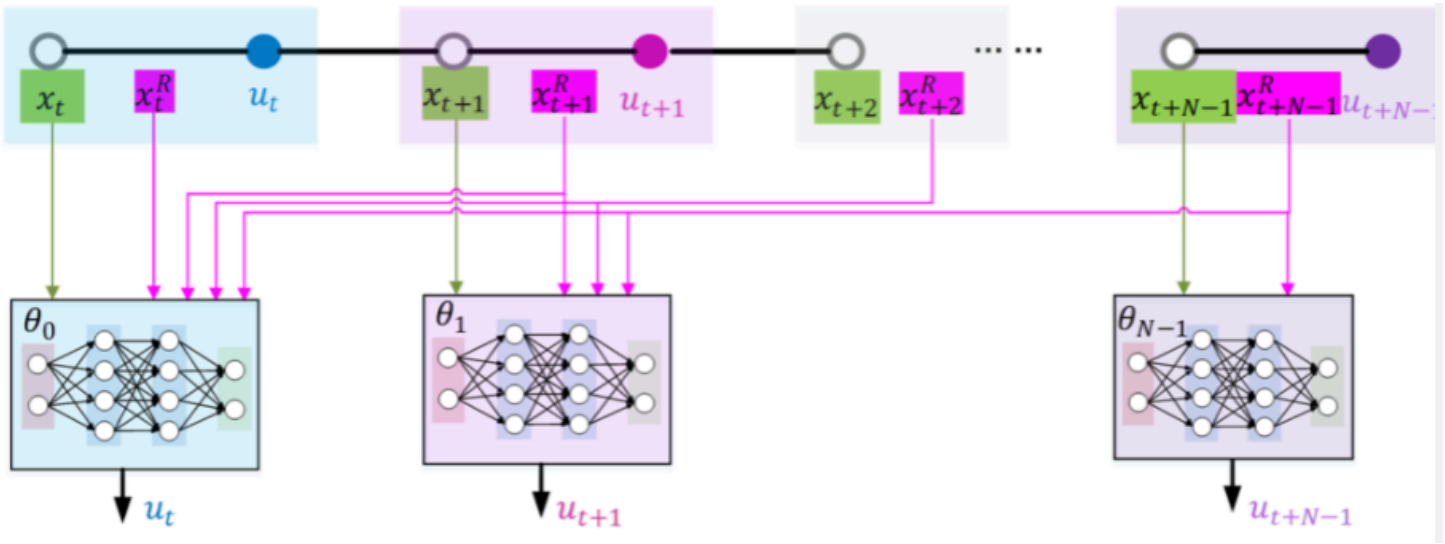
从上面的算法框架中也可以看出，因为这里我们不用显示的对于值函数进行建模，因此也不存在通常意义上的“Critic”，所以上述框架中只有对于actor的梯度更新。

8.5.3 使用Multi-stage策略的最优Tracker的Finite-horizon ADP算法

不同于刚才讲过的regulator的情况，tracker的情况下我们需要使用不同的参数来表示不同时间步的策略，即multi-stage的策略。那么到底需要几套策略参数呢？其实这个数就等于我们的horizon的长度N。我们将在8.5.1里面讲过的策略结构重新再拿出来：

$$\begin{aligned} u_t &= \pi(x_t, x^R_{t:t+N-1}; \theta_0), \\ u_{t+1} &= \pi(x_{t+1}, x^R_{t+1:t+N-1}; \theta_1), \\ u_{t+N-1} &= \pi(x_{t+T-1}, x^R_{t+N-1}; \theta_{N-1}) \end{aligned}$$

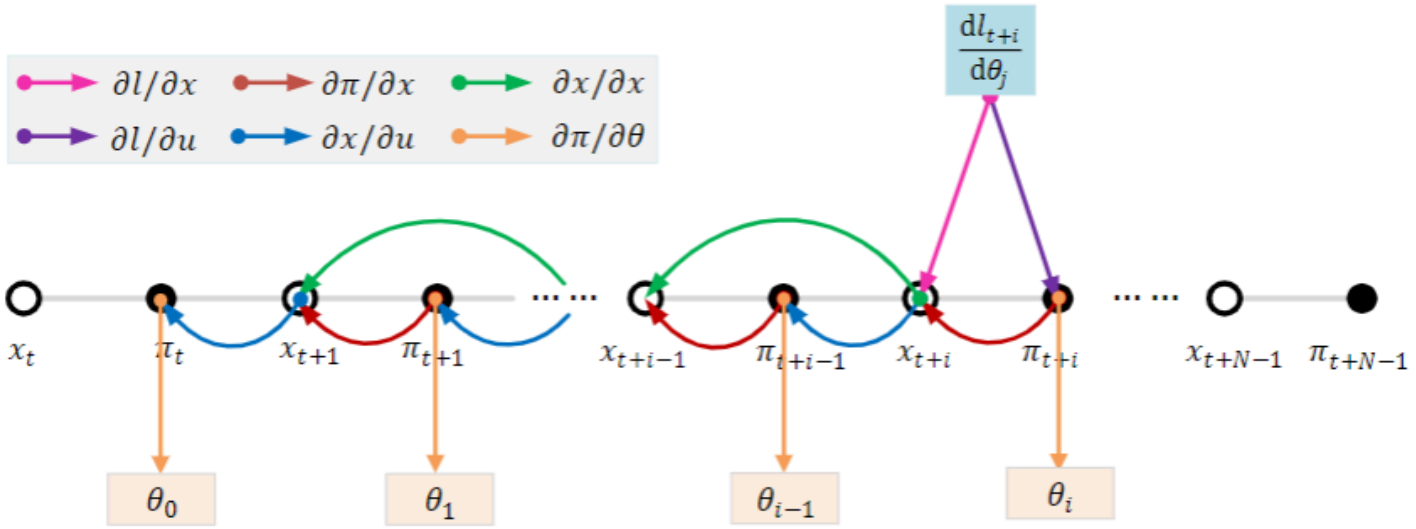
这里，$\theta_0, \theta_1, ..., \theta_{N-1}$就是我们的N套策略参数。值得注意的是，这些策略参数以及策略函数本身通常都具有不同的维度，这是因为对于预测问题来说，在时间t的时候，我们不仅需要使用当前的状态x_t作为输入，还需要使用未来的N个参考点$x^R_{t:t+N-1}$。因此，也易知，输入的维度随着时间步越向后越少（因为此时剩下的参考信号数量也越少）。这种multi-stage的策略可以使用下图形象的描述：



那么，仿造regulator的情况，我们可以得到N个policy gradient：

$$\frac{\mathrm{d}V(x,X^R)}{\mathrm{d}\theta_j}=\sum_{i=0}^{N-1}\frac{\mathrm{d}l(x_{t+i},x_{t+i}^R,u_{t+i})}{\mathrm{d}\theta_j},j\in\{0,1,\cdots,N-1\}$$

与8.5.2节一样，上述这N个梯度的计算也同样依赖于一个DAG：



于那里的DAG的区别在于，终点不再是一个单独的参数θ，而是N个不同的参数$\theta_0,\theta_1,...,\theta_{N-1}$。同样的我们引入下述记号来简化式子：

$$\begin{aligned}l_{t+i}(x_{t+i}^R)&\overset{\text{def}}{=}l(x_{t+i},x_{t+i}^R,u_{t+i}),\\f_{t+i}&\overset{\text{def}}{=}f(x_{t+i},u_{t+i}),\\\pi_{t+i}(\theta_i)&\overset{\text{def}}{=}\pi(x_{t+i},x_{t+i:t+N-1}^R;\theta_i).\end{aligned}$$

引入了上述记号后，可以把每个梯度写成如下形式：

$$\frac{\mathrm{d}l(x_{t+i}, x_{t+i}^R, u_{t+i})}{\mathrm{d}\theta_j} = \frac{\mathrm{d}x_{t+i}^{\mathrm{T}}}{\mathrm{d}\theta_j}\frac{\partial l_{t+i}(x_{t+i}^R)}{\partial x_{t+i}} + \frac{\mathrm{d}u_{t+i}^{\mathrm{T}}}{\mathrm{d}\theta_j}\frac{\partial l_{t+i}(x_{t+i}^R)}{\partial u_{t+i}}.$$

再像8.5.2节一样，引入下述两个额外的记号（只不过这里的ϕ和ψ右上角都有一个j，来表示其是对于不同的策略参数来求导的）：

$$\phi_{t+i}^j = \frac{\mathrm{d}x_{t+i}^{\mathrm{T}}}{\mathrm{d}\theta_j} = \frac{\mathrm{d}f_{t+i-1}^{\mathrm{T}}}{\mathrm{d}\theta_j}, \psi_{t+i}^j = \frac{\mathrm{d}u_{t+i}^{\mathrm{T}}}{\mathrm{d}\theta_j} = \frac{\mathrm{d}\pi_{t+i}^{\mathrm{T}}(\theta_i)}{\mathrm{d}\theta_j}.$$

然后我们可以得到如下的递归关系：

$$\begin{aligned}\phi_{t+i}^j &= \phi_{t+i-1}^j\frac{\partial f_{t+i-1}^{\mathrm{T}}}{\partial x_{t+i-1}} + \psi_{t+i-1}^j\frac{\partial f_{t+i-1}^{\mathrm{T}}}{\partial u_{t+i-1}},\\ \psi_{t+i}^j &= \phi_{t+i}^j\frac{\partial \pi_{t+i}^{\mathrm{T}}(\theta_i)}{\partial x_{t+i}} + \frac{\partial \pi_{t+i}^{\mathrm{T}}(\theta_i)}{\partial \theta_j},\end{aligned}$$

值得注意的是，这里DAG的终点不是一个而是多个带来了下面的两个性质：

$$\begin{aligned}\frac{\partial \pi^{\mathrm{T}}(x_{t+i}, x_{t+i:t+N-1}^R; \theta_i)}{\partial \theta_j} &= 0, \mathrm{if} i \neq j,\\ \phi_{t+i}^j = \frac{\mathrm{d}x_{t+i}^{\mathrm{T}}}{\mathrm{d}\theta_j} &= 0, \mathrm{if} i \leq j.\end{aligned}$$

第一个性质是因为从上述DAG可以看出每个$\pi_{t+i}(\theta_i)$只有唯一的一条黄色的边指向θ_j；而第二个性质是因为是由时间因果性决定的，先发生的状态显然与后续的策略参数无关。因此，最后的policy gradient归结为下面的N个形式相同的式子：

$$\frac{\mathrm{d}V(x, X^R)}{\mathrm{d}\theta_j} = \sum_{i=0}^{N-1}\left(\phi_{t+i}^j\frac{\partial l_{t+i}(x_{t+i}^R)}{\partial x_{t+i}} + \psi_{t+i}^j\frac{\partial l_{t+i}(x_{t+i}^R)}{\partial u_{t+i}}\right)$$

上面的式子中的基本元素都可以通过递归的方式计算得到。最后们需要指出的是，因为各个stage之间紧密相连，因此更新参数时一次就要更新所有的参数，而不能只更新其中的几个：

$$\begin{aligned}\theta_0 &\leftarrow \theta_0 - \alpha_0\frac{\mathrm{d}V(x, X^R)}{\mathrm{d}\theta_0},\\ \theta_1 &\leftarrow \theta_1 - \alpha_1\frac{\mathrm{d}V(x, X^R)}{\mathrm{d}\theta_1},\\ \theta_{N-1} &\leftarrow \theta_{N-1} - \alpha_{N-1}\frac{\mathrm{d}V(x, X^R)}{\mathrm{d}\theta_{N-1}}.\end{aligned}$$

最后，我们给出整个算法的伪代码：

Hyperparameters: learning rate α, predictive horizon N, number of environment resets M

Initialization: policy function $\pi(x, x_{1:N}^R; \theta_0)$, $\pi(x, x_{2:N}^R; \theta_1)$,$\cdots$, $\pi(x, x_N^R; \theta_{N-1})$

Repeat

(1) Use environment model

Initialize memory buffer $\mathcal{D} \leftarrow \emptyset$

Repeat M environment resets

$x_0 \sim d_{\text{init}}(x)$

$u_0 = \pi(x_0, x_{1:N}^R; \theta_0)$

$\phi_0^j = 0, j = 0,1, \dots N-1$

$\psi_0^j = \partial \pi^{\mathrm{T}}(x_0, x_{1:N}^R; \theta_0)/\partial \theta_j, j = 0,1, \dots N-1$

For i in 1,2, … , $N-1$

//Rollout model and policy

$x_i = f(x_{i-1}, u_{i-1})$

$u_i = \pi(x_i, x_{i+1:N}^R; \theta)$

Calculate $\left(l, V, \frac{\partial l}{\partial u}, \frac{\partial l}{\partial x}, \frac{\partial f^{\mathrm{T}}}{\partial u}, \frac{\partial f^{\mathrm{T}}}{\partial x}, \frac{\partial \pi^{\mathrm{T}}}{\partial x}, \frac{\partial \pi^{\mathrm{T}}}{\partial \theta}\right)_i$

For j in 0,1, … , $N-1$

$$\phi_i^j = \phi_{i-1}^j \left(\frac{\partial f^{\mathrm{T}}}{\partial x}\right)_{i-1} + \psi_{i-1}^j \left(\frac{\partial f^{\mathrm{T}}}{\partial u}\right)_{i-1}$$

$$\psi_i^j = \phi_i^j \left(\frac{\partial \pi^{\mathrm{T}}}{\partial x}\right)_i + \frac{\partial \pi^{\mathrm{T}}(x_i, x_{i+1:N}^R; \theta_i)}{\partial \theta_j}$$

End

End

$$\frac{\mathrm{d}V(x_0)}{\mathrm{d}\theta_j} = \sum_{i=0}^{N-1} \left(\phi_i^j \left(\frac{\partial l}{\partial x}\right)_i + \psi_i^j \left(\frac{\partial l}{\partial u}\right)_i\right), j = 0,1, \dots N-1$$

$$\mathcal{D} \leftarrow \mathcal{D} \cup \left\{\frac{\mathrm{d}V(x_0)}{\mathrm{d}\theta_0}, \frac{\mathrm{d}V(x_0)}{\mathrm{d}\theta_1}, \cdots, \frac{\mathrm{d}V(x_0)}{\mathrm{d}\theta_{N-1}}\right\}$$

End

(2) Actor update

For j in 0,1, … , $N-1$

$$\nabla_{\theta_j} J_{\text{ADP}} \leftarrow \frac{1}{M} \sum_{\mathcal{D}} \frac{\mathrm{d}V(x_0)}{\mathrm{d}\theta_j}$$

$$\theta_j \leftarrow \theta_j - \alpha \cdot \nabla_{\theta_j} J_{\text{ADP}}$$

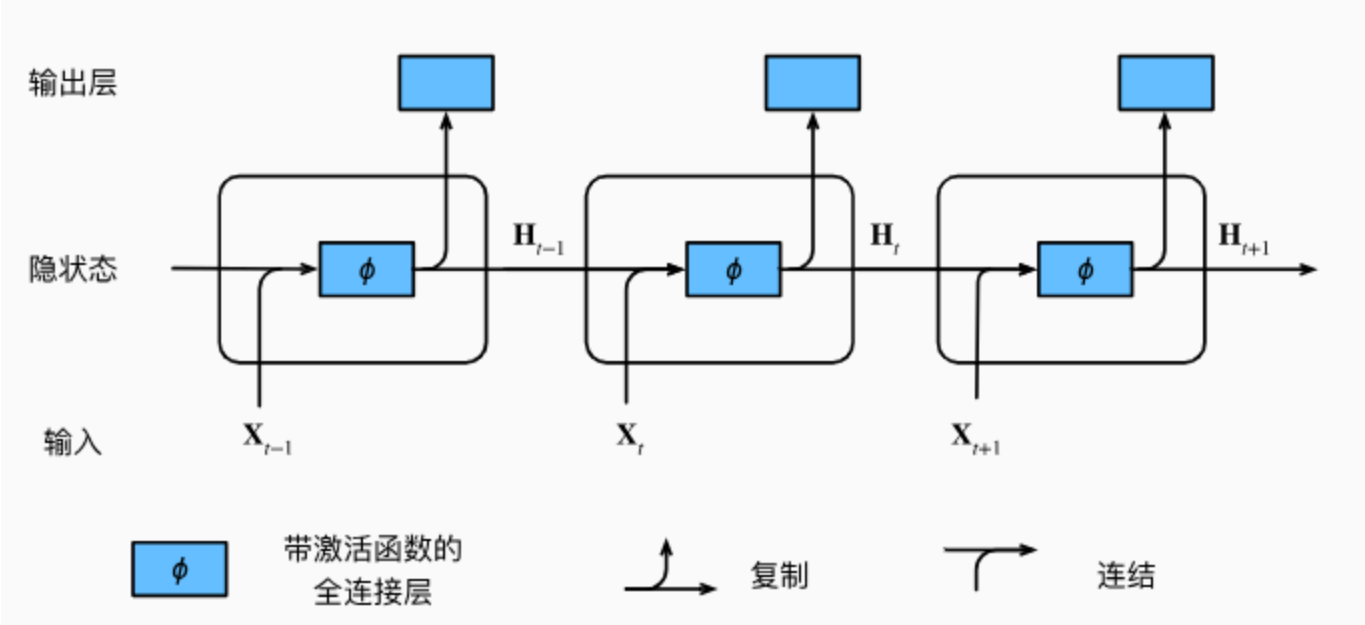
End

End

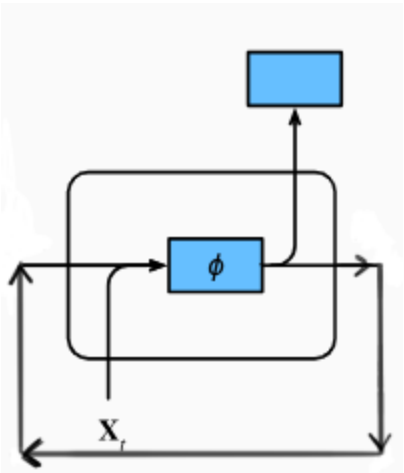
8.5.4 使用循环/递归（Recurrent）策略的最优Tracker的Finite-horizon ADP算法

一个有意思的事实是我们可以将上面的multi-stage的策略简化为一个使用recurrent函数的策略。与multi-stage的策略不同，recurrent策略只需要一套参数（那么自然其输入维度也是恒定的，而不是像之前讲过的multi-stage的策略那样输入维度随时间步变化）。那么我们应该怎样建模这样一种具有recurrent特性的函数呢？其实，可以使用RNN（Recurrent Neural Network）来实现这样的函数。

下面来简单介绍一下RNN的基本知识。一个RNN的架构可以这样表示：

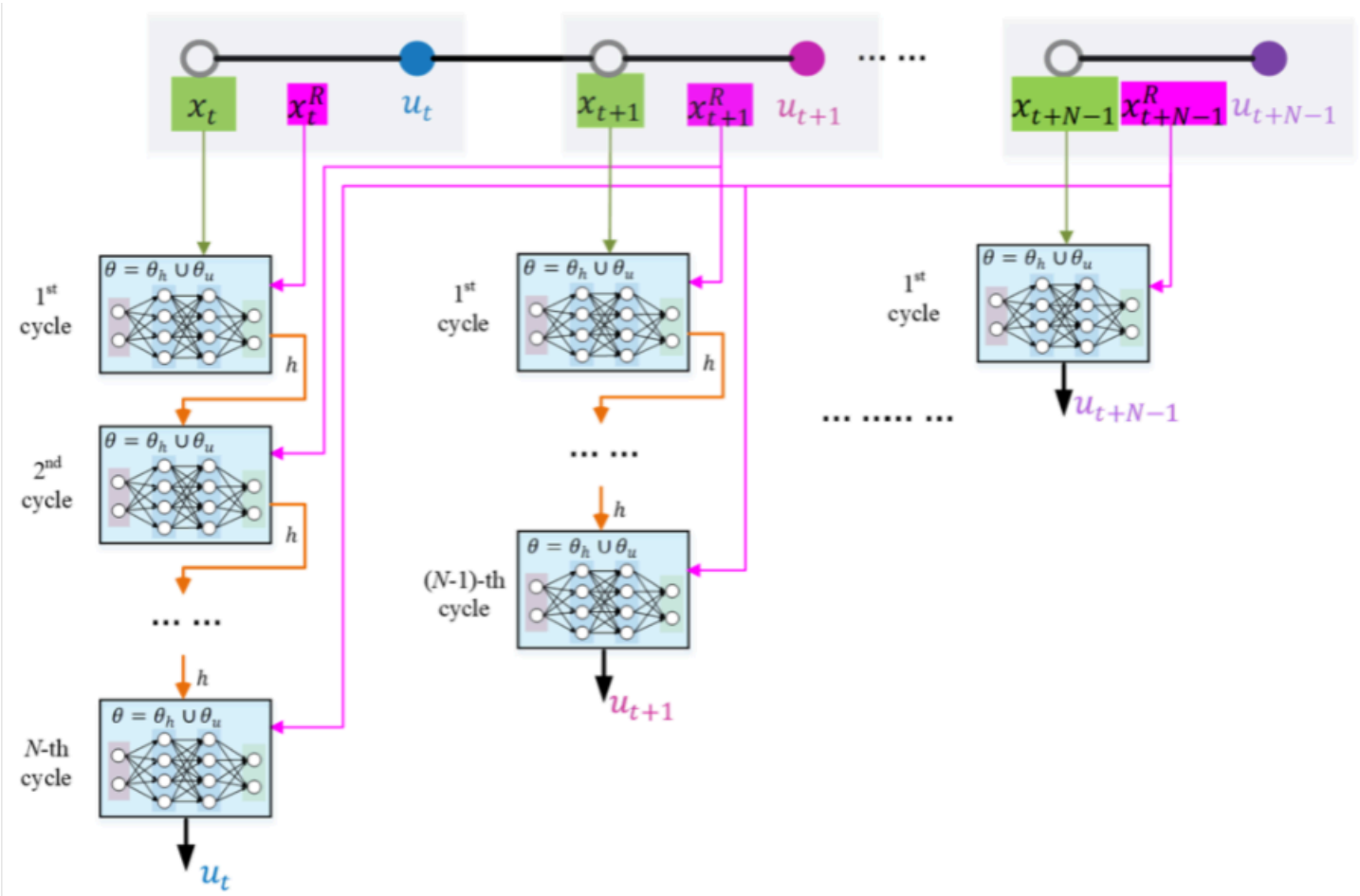


注意，上图中因为我们将其展平了，看起来似乎有很多个用于处理隐状态的模块（图中的矩形框），实际上那些模块都是同一个，只不过其上一个时间步的隐状态在下一个时间步输入到这个模块中。像下图所示：



RNN的训练需要使用BP的变种——BPTT（Back Propagation Through Time）。

那么，我们应该怎样用一套参数来对于multi-stage的策略进行建模呢？答案是对于我们的horizon中的不同时间步，采用不同的RNN循环次数，越靠前的时间步因为需要输入的参考信息好数目越多，因此循环次数也越多。如下图所示：



有了RNN作为策略函数，我们就可以将策略结构表述为如下形式：

$$u_{t+i} = \pi^{N-i}(x_{t+i}, x^R_{t+i:t+N-1}; \theta),\ i = 0, 1, ..., N-1.$$

更具体地来说，在每个时间步i，针对于其中的每个循环c，都可以表述如下：

$$\begin{gathered} h_c = \sigma_h(x_{t+i}, x^R_{t+i+c-1}, h_{c-1}; \theta_h), \\ \pi^c(x_{t+i}, x^R_{t+i:t+N-1}; \theta) = \sigma_u(h_c; \theta_u) \\ c = 0, 1, ..., N-i, \end{gathered}$$

上式中的σ_h和σ_u分别表示隐藏层和输出层的函数变换。而π^c表示在时间步i中第c个循环的策略（这只是一个中间策略，不作为时间步i的最终策略）。

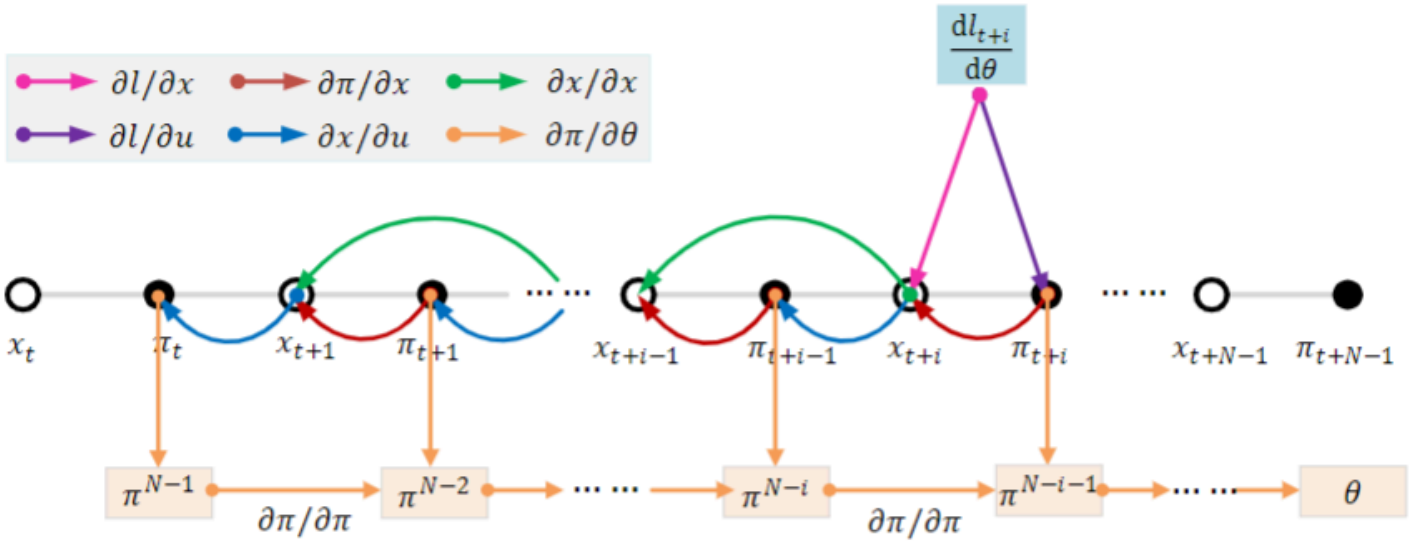
最后，我们来看看使用recurrent策略的policy gradient是怎么样的。与之前类似，我们有如下形式的policy gradient：

$$\begin{aligned}\frac{\mathrm{d}V(x,X^R)}{\mathrm{d}\theta}&=\sum_{i=0}^{N-1}\frac{\mathrm{d}l_{t+i}(x_{t+i}^R)}{\mathrm{d}\theta}\\&=\sum_{i=0}^{N-1}\left(\frac{\mathrm{d}x_{t+i}^{\mathrm{T}}}{\mathrm{d}\theta}\frac{\partial l_{t+i}(x_{t+i}^R)}{\partial x_{t+i}}+\frac{\mathrm{d}u_{t+i}^{\mathrm{T}}}{\mathrm{d}\theta}\frac{\partial l_{t+i}(x_{t+i}^R)}{\partial u_{t+i}}\right)\end{aligned}$$

上式中的$l_{t+i}(x_{t+i}^R)=l(x_{t+i},x_{t+i}^R,u_{t+i})$、$f_{t+i}=f(x_{t+i},u_{t+i})$、$\pi^{N-i}(\theta)=\pi^{N-i}(x_{t+i},x_{t+i:t+N-1}^R;\theta)$。仿照之前我们再引入一些记号：

$$\begin{aligned}\phi_{t+i}&=\frac{dx_{t+i}^{\mathrm{T}}}{d\theta}=\frac{df^{\mathrm{T}}(x_{t+i-1},u_{t+i-1})}{d\theta},\\\psi_{t+i}&=\frac{du_{t+i}^{\mathrm{T}}}{d\theta}=\frac{d[\pi^{N-i}(x_{t+i},x_{t+i:t+N-1}^R;\theta)]^{\mathrm{T}}}{d\theta}.\end{aligned}$$

那么，结合下图，就可以得到递归的计算各个ϕ和ψ的方法：



$$\begin{aligned}\phi_{t+i}&=\phi_{t+i-1}\frac{\partial f_{t+i-1}^{\mathrm{T}}}{\partial x_{t+i-1}}+\psi_{t+i-1}\frac{\partial f_{t+i-1}^{\mathrm{T}}}{\partial u_{t+i-1}},\\\psi_{t+i}&=\phi_{t+i}\frac{\partial[\pi^{N-i}(\theta)]^{\mathrm{T}}}{\partial x_{t+i}}+\frac{\partial[\pi^{N-i}(\theta)]^{\mathrm{T}}}{\partial\theta},\\\frac{\partial[\pi^{N-i}(\theta)]^{\mathrm{T}}}{\partial\theta}&=\frac{\partial[\pi^{1}(\theta)]^{\mathrm{T}}}{\partial\theta}\prod_{k=N-i}^{2}\frac{\partial[\pi^{k}(\theta)]^{\mathrm{T}}}{\partial\pi^{k-1}(\theta)},\end{aligned}$$

尤其注意，这里比起8.5.2和8.5.3节，多出了一个$\partial[\pi^{N-i}(\theta)]^{\mathrm{T}}/\partial\theta$的计算，这是因为我们的策略是recurrent的，所有策略都依赖于同一套参数θ，而且我们前面的决策会影响后面的状态进而影响到后面的决策（详见上图最下方$\partial\pi/\partial\pi$的那些线）。最后，我们的policy gradient就可以写成如下形式：

$$\frac{\partial V(x, x^R)}{\partial \theta} = \sum_{i=0}^{N-1} \left(\phi_{t+i} \frac{\partial l_{t+i}(x_{t+i}^R)}{\partial x_{t+i}} + \psi_{t+i} \frac{\partial l_{t+i}(x_{t+i}^R)}{\partial u_{t+i}} \right)$$

初始条件为$\phi_t = 0$。

最后，使用RNN作为策略函数还有一个有意思的地方，即一个perfectly-trained的RNN策略，在推理时，它的RNN循环次数可以再任意步停止，却仍能保存一些最优性。例如，如果RNN循环了两次就停止，那么它的输出对于horizon长为2的问题来说是最优的。

@[toc]

书籍链接：Reinforcement Learning for Sequential Decision and Optimal Control

本篇博客对应于原书的第九章，主要讲述了带有约束的强化学习。在实际的场景中，我们往往会遇到很多约束条件，包括对于输入的约束和对于状态的约束。对于输入的约束相对而言比较容易处理，而更困难的是对于状态的约束。主要有三种方法来处理状态约束：(1) 使用罚函数来惩罚打破约束的行为；(2) 使用基于对偶理论的拉格朗日乘子法来确定原问题的下界；(3) 使用feasible descent direction方法来找到一个既满足约束又能使目标函数下降的方向。

可以看出，处理带约束的强化学习问题与static optimization问题是类似的，但是前者因为带有约束而产生了保持每步学得的策略的递归可行性的难题。递归可行性被破坏通常是由于过于严格的状态约束。另一个困难是要在学得策略的同时确定可行域，而这两个问题是紧密耦合在一起的。因此本章中提出了一种新的学习架构——Actor-Critic-Scenery（ACS）架构，其中的Actor和Critic作用不变，而Scenery则负责确定可行域。

那么，对于这种带有hard state constraints的强化学习问题，应该在哪里训练呢？有两种常用的方法：(1) 先offline的在仿真环境中训练，然后再作为一个最优控制器部署在真实环境中；(2) 直接在真实环境中训练并部署。

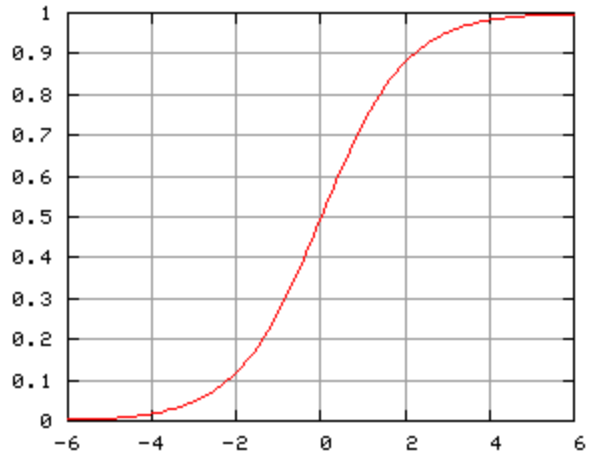
9.1 处理动作约束的方法

总的来说，有两种方法可以来处理动作约束：

- saturated policy function：saturation函数用于将一个未加限制的策略函数的输出约束到一定范围内。
- penalized utility function：向utility function中加入一个惩罚项，用于惩罚超出约束范围的动作。外点罚函数和内点罚函数都可以用来实现上述功能。

9.1.1 Saturated Policy Function

什么是saturation函数呢？其实，举个例子就知道了。一个在深度学习中使用的很广泛的激活函数Sigmoid函数就是一个saturation函数。其函数图像如下所示：



从上述图像也可以看出为什么我们把这类函数称为saturation函数了。因为在自变量趋于$+\infty$或$-\infty$时，函数值会趋近于一个常数而不会发散。

我们通常应用saturation函数的方式如下：

$$u = \phi(\pi(x;\theta))$$

这里的$\pi(x;\theta)$是我们原始的无限制的策略函数，而ϕ就是saturation函数。注意，这里的saturation函数**是一个逐元素的函数**，也就是说，它把输入的向量$\mathbf{x}$的每一个元素都约束到一个范围内。

下面，我们来看看一些常见的saturation函数：

- Sigmoid函数：

$$\phi(z) = \frac{1}{1+\exp(-z)}$$

- Tanh函数（双曲正切函数）：

$$\phi(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

- 反正切函数：

$$\phi(z) = \arctan(z)$$

- Softsign函数：

$$\phi(z) = \frac{z}{1+|z|}$$

加上saturation函数之后，我们需要使用链式法则来计算梯度：

$$\frac{\partial u}{\partial \theta}=\frac{\partial \phi(\pi)}{\partial \pi} \cdot \frac{\partial \pi}{\partial \theta}$$

Saturation函数的上述使用的一个典型例子就是在神经网络中的激活函数。这样就可以把输出压缩到一个有限的范围中（比如要输出一个概率值的时候）。但是，这样的saturation函数有一个缺点，就是在自变量的值很大或者很小的时候会出现梯度接近于0的现象，这样趋近于0的梯度再加上网络层数一深，就会导致梯度消失的问题。使用以0为中心的激活函数（比如tanH）可以缓解这个问题。

9.1.2 Penalized Utility Function

罚函数（penalty function）是一种广泛使用的可以把有约束问题转化为无约束问题的方法。罚函数主要分为两大类：外点罚函数和内点罚函数。下面分别介绍。

- **外点罚函数**：
  - **特点**：惩罚项只在输出的动作**违反约束**的时候才会生效。
  - **得到的策略是否一定满足约束，又是否一定最优**：通常不是，会有一点小的违反约束。这样我们得到的输出并不能严格的遵守约束，适合那些允许轻微违反约束的问题。
  - **对于初始点有无要求**：无
- **内点罚函数**：
  - **特点**：当输出的动作在接近约束边界（此时还未违反约束）时会趋向于无穷大。
  - **得到的策略是否一定满足约束，又是否一定最优**：一定满足约束，但通常会比真正的有约束最优策略略差一点。
  - **对于初始点有无要求**：有，需要在可行域内。

下面我们来介绍一下一种可以用于RL的内点罚函数法：在utility function中加入一个惩罚项：

$$l_{\mathrm{mod}}(x, u)=l(x, u)+\rho \cdot \varphi(u)$$

这里的$\varphi(u)$是一种给内点罚函数，而ρ是一个惩罚系数。这里，我们给出$\varphi(u)$的一个具体实例（这个例子可以把saturation函数和内点罚函数联系起来）：

$$\varphi(u)=\int_0^u\left(\phi^{-1}(v)-\rho^{-1} \frac{\partial l(x, v)}{\partial v}\right) \mathrm{d} v$$

这里的ϕ是一个element-wise的saturation函数，把输入从$(-\infty,+\infty)$映射到$[-1,1]$。ϕ^{-1}是ϕ的反函数，因此它能把$[-1,1]$映射回$(-\infty,+\infty)$。这样定义的$\varphi(u)$在输入的动作u趋于边界1或者-1（即马上就要违反约束的时候），就会趋向于无穷大。接下来我们来推导最优的策略。

首先，根据稳态时的最优性条件，我们有：

$$\frac{\partial V^*(x)}{\partial u}=0, \mathrm{when} u=u^*$$

那么，再结合引入的惩罚项的Bellman方程：

$$V^*(x) = \min_u \{l_{\mathrm{mod}}(x,u) + V^*(x')\}$$

那么，我们有：

$$\begin{aligned}\frac{\partial\Big(l_{\mathrm{mod}} + V^*(x')\Big)}{\partial u} &= \frac{\partial\Big(l(x,u) + \rho\cdot\big(\int_0^u \Big(\phi^{-1}(v) - \rho^{-1}\frac{\partial l(x,v)}{\partial v}\Big)\,\mathrm{d}v\big) + V^*(x')\Big)}{\partial u} \\ &= \frac{\partial l(x,u)}{\partial u} + \rho\cdot\left(\phi^{-1}(u) - \rho^{-1}\frac{\partial l(x,u)}{\partial u}\right) + \frac{\partial x'}{\partial u}\frac{\partial V^*(x')}{\partial x'} \\ &= \rho\phi^{-1}(u) + \frac{\partial f^{\mathrm{T}}}{\partial u}\frac{\partial V^*(x')}{\partial x'} \\ &= 0\end{aligned}$$

那么我们可以解得：

$$u^* = \phi\left(-\frac{1}{\rho}\frac{\partial f^{\mathrm{T}}}{\partial u}\frac{\partial V^*(x')}{\partial x'}\right).$$

可以看出，我们最终输出的策略被saturation函数约束在了$[-1,1]$之间。在这个例子中，我们将内点罚函数和saturation函数联系起来，从内点罚函数的推导入手，最后得到的最优策略的形式却是一个saturation函数。

9.2 状态约束和可行性的关系

本节我们先来讨论一下状态约束和可行性的关系，至于具体怎么在RL中处理状态约束，我们将会在下面几节中详细介绍。

与我们刚才说过的对于动作的约束相比，对于状态的约束之所以更加难以处理是因为它与不可行现象深度耦合在一起。状态约束可能来自于对于安全的考虑、物理规则的约束、系统performance的限制等。在任何可行的策略被找到之前，我们需要先回答两个问题：（1）怎么在原来的最优控制问题（OCP）的建模中加上状态约束；（2）怎么设计一个带有约束的RL算法。

我们先来看看第一个问题。它看似简单，但是其内在的困难在于我们在建模时需要从real-time的约束转化到virtual-time的约束。而对应的virtual-time的约束建模方法又不唯一，如果我们选择了不合适的建模方式，那么无论采取何种优化技巧都无法得到一个可行的策略。早期的关于带有约束的OCP的建模可以追溯到MPC（尽管没有被明确的指出）。在MPC中，我们可以通过只在prediction-horizon中的某些步施加约束而令其它步不受约束的方式来处理约束。另一个例子是Control Barrier Function（CBF）。CBF通常只需要去处理一些预测步，因此能降低在long-horizon限制中的计算复杂度。

接下来我们来看看第二个问题。怎么在完成建模之后把状态的限制纳入到RL的学习算法中去。现在主要有三种方法：

- **罚函数法**
- **拉格朗日乘子法**：这种方法通过使用对偶的上升更新方法来求解一个minimax优化问题。
- **feasible descent direction法**

大部分现实世界中的决策或者控制问题都是持续不断的（continuing），那么它们的状态限制也应该从当前状态一直持续到无穷远的未来都被满足。不失一般性，我们可以这样表述real-time域中的状态约束：

$$h(x_{t+i}) \leq 0, i = 1, 2, \ldots, \infty,$$

其中$h(\cdot) \in \mathbb{R}$是一个**标量函数**。可能大家会疑惑，在工程实际中，很多的状态约束是向量函数而不是标量函数，为什么我们这里要把$h(\cdot)$定义为一个标量函数呢？一方面，十一位内直接处理向量函数会带来很多计算上的负担，好比说在很多时候我们都需要求导，这时候如果$h(\cdot)$是一个向量函数，那么求一次导可能就得到了一个矩阵，再求导可能就得到了哟个张量，这样问题的计算复杂度会急剧上升。另一方面，我们也有很多手段把向量函数转化为标量函数，比如使用求和或者取最值的操作。接下来，我们来引入一个新的概念：**Risk Signal**，它被定义如下：

$$c_t = h(x_t)$$

可以这样理解，risk signal与标量的限制函数$h(\cdot)$的关系就与reward signal与utility function的关系一样。Risk signal是一个标量，它表示了一个状态与限制的边界距离多远。限制函数与risk signal构成了一个risk-aware系统，这与reward signal与utility function构成了一个reward系统是类似的。其实很多的代悦书的RL/ADP问题就是通过简单地把risk signal加入到reward signal中来处理状态约束的。这种思路可以极大地降低算法design的复杂度，但是其实仔细想想，这种相对naive的做法并不能保证约束一定会被满足。在这种情况下，因为要在optimal control（以reward signal反映）和约束满足（以risk signal反映）之间做权衡，因此总会看到一定程度的performance loss。另外还需要说明的是，约束函数$h(\cdot)$并不一定是一个已知量。如果这里的约束函数未知，那么就要求risk signal必须是可以观测的，并且我们需要通过不断地与环境交互来获得risk signal的信息。另一方面，如果约束函数已知（有显式解析形式），那么我们可以直接使用观察到的状态和动作来获得risk signal，这样有助于提升算法的效率和训练的稳定性。

9.2.1 从两个域的角度理解状态约束

对于带有状态约束的OCP问题，常见的解决问题的方法，比如变分法和庞特里亚金最小值原理都不再适用。而MPC虽然可以用，但是它的online计算的特性有天然制约了其解决复杂问题的能力。因此，一个offline训练、online应用的RL算法就成了不二选择。

为了说明怎么训这样的一个RL策略，我们需要使用在第八单元的博客中提到的Receding Horizon Control（RHC）的两个时间域的概念。我们需要在virtual-time域中训练策略，并在real-time域中应用策

略。事实上这是我们处理带有限制的RL/ADP问题时总是采用的方法。在virtual-time域中，通常这样定义约束：

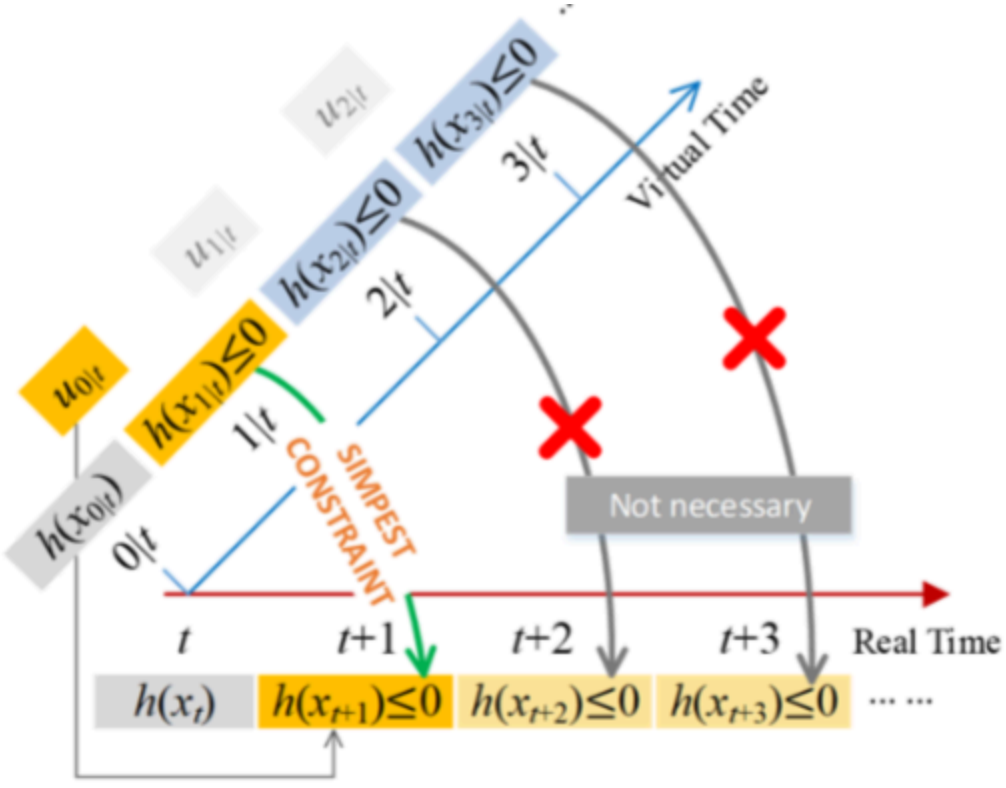
$$h(x_{i|t}) \leq 0, i = 1, 2, ..., \infty$$

注意到这个式子与我们刚刚提到的real-time域中的约束的定义$h(x_{t+i}) \leq 0, i = 1, 2, ..., \infty$的区别。那里的t+i表示在real-time域中的第i+t个时间步，而这里的i|t表示在virtual-time域中，从实际时间t开始的第i个时间点。在继续下去之前，我们需要指出一个有意思的观察：式子$h(x_{i|t}) \leq 0$中的约束并不一定需要在virtual-time中每个时间步都被满足。因此，virtual-time域中的约束和real-time域中的约束也不一定是一样的，这种可分离性在为我们在virtual-time域中构建OCP问题时提供了很大的灵活性。下面我们就来介绍一种用的最多的构建OCP问题的方法。

与MPC相似，这里我们在虚拟时间域中训练得到的策略预测出的最有动作序列中只有第一个动作会被真正在real-time域中执行。而执行之后引起的状态转移到的下一个状态（在虚拟时间域中是$x_{1|t}$，在实际时间域中是x_{t+1}）也必须满足状态限制：

$$h(x_{1|t}) = h(x_{t+1}) \leq 0$$

这也被称做**Simplest Constraint**。至于虚拟时间域中剩下的状态$x_{i|t}\ i = 2.3....\infty$是否满足$h(x_{i|t}) \leq 0.i = 2.3....\infty$并不重要。如下图所示：



那么，我们可以这样构建OCP：

$$\min_{u} V(x) = \sum_{i=0}^{N-1} l(x_{i|t}, u_{i|t}),$$
$$\text{s.t.}$$
$$x_{i+1|t} = f(x_{i|t}, u_{i|t}),$$
$$h(x_{1|t}) \leq 0,$$

其中，**初始状态**$x \overset{\text{def}}{=} x_{0|t} = x_t$**。并注意这里的环境模型**$f(\cdot)$**被假定是完美的，即在虚拟时间域和实际时间域中都是一样的**。根据N是否等于∞，可以将上述问题进一步细分为finite-horizon问题和infinite-horizon问题。上述这种在虚拟时间域中构建OCP的方法是最广为使用的，因为simplest constraint的存在，在满足约束的情况下计算量被压缩到了最小，也能够使得学到的策略最接近此约束下的真实策略（因为不必要的额外条件施加的最少）。

9.2.2 关于不可行和可行的定义与讨论

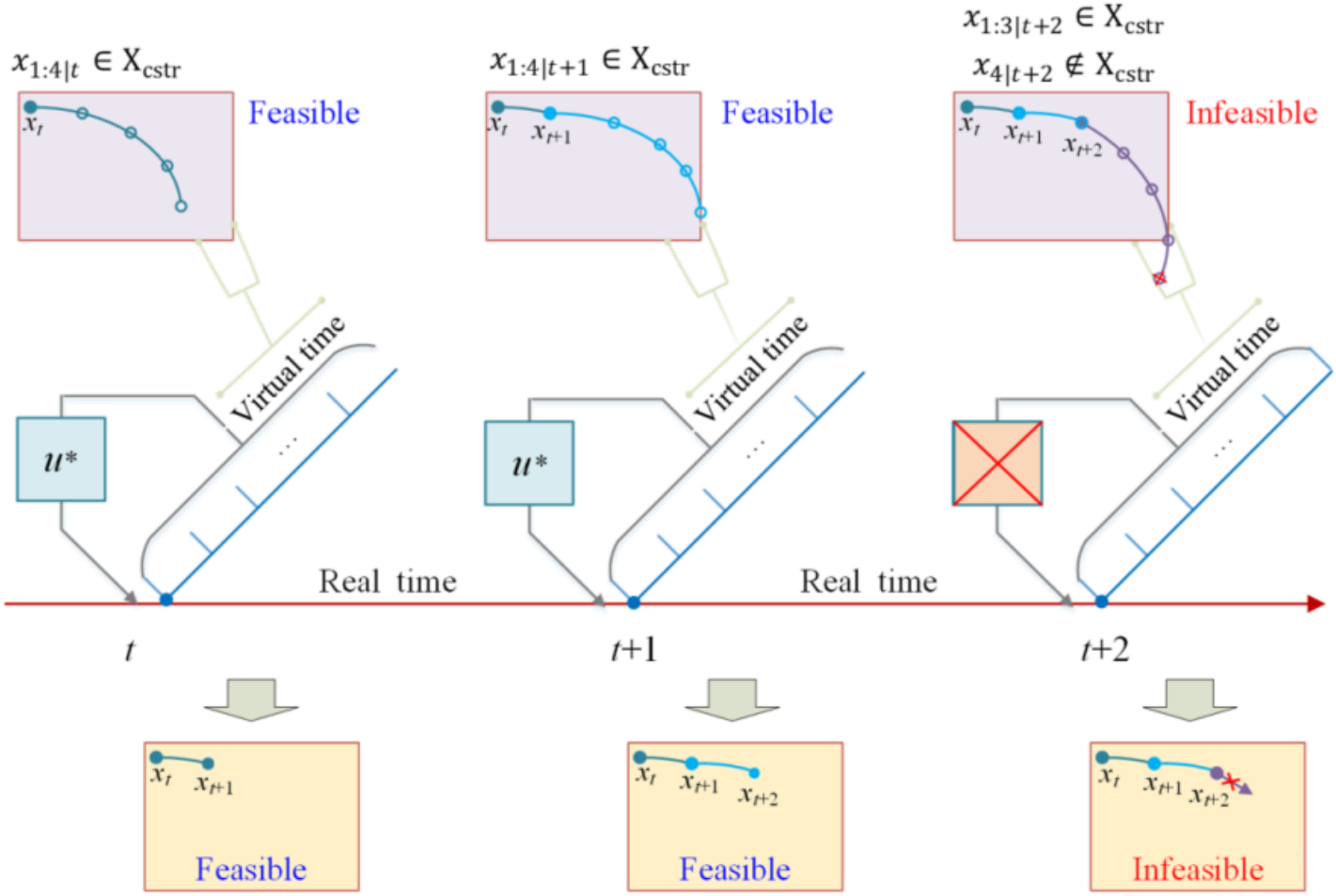
在上面我们讲了simplest constraint之后，可能大家就会以为这是一个绝佳的选择，就应该使用这种方式来在虚拟时间域中构建OCP问题。但是这种方法在实际中用到的却很少？这是为什么呢？为了回答这个问题，需要先来讨论一下什么是可行的和不可行的。

> **定义1：不可行（Infeasible）**：
> 我们使用不可行（infeasible）来描述一个定义在virtual-time域中的定义的OCP问题没有满足virtual-time域中的约束的可行解。

为了下面更好的说明不可行的现象是怎样发生的，我们可以定义如下集合：

$$\mathbf{X}_{\text{Cstr}} \overset{\text{def}}{=} \{x | h(x) \leq 0\}$$

集合$\mathbf{X}_{\text{Cstr}}$描述的时所有满足约束的状态的集合。现在我们可以来回答为什么simplest constraint在实际中用的很少了。因为这种约束只要去保证$x_{1|t}$满足约束就可以了，而不需要去保证$x_{i|t}, i = 2, 3, ...\infty$满足约束。换句话说，Simplest constraint只要求$x_{1|t}$在$\mathbf{X}_{\text{Cstr}}$中，而不要求$x_{i|t}, i = 2, 3, ...\infty$在$\mathbf{X}_{\text{Cstr}}$中。因此与其它在虚拟时间域中多步的约束相比，simplest constraint具有最差的保证长时间内feasibility的能力，很容易在之后的实际时间步求解virtual-time域时遇到找不到可行解的情况。而那些多步的约束能保证更多在virtual-time域中的状态点在$\mathbf{X}_{\text{Cstr}}$中。下面我们来看一个在虚拟时间域中使用4步constraint而遇到不可行的例子：



在上图中，我们可以看到，实际时间t时，实际的状态是x_t，它处于可行域之内，然后在时刻t的虚拟时间域中，它作为$x_{0|t}$。然后我们在虚拟时间域中向后预测了四步，得到了$x_{1|t}, x_{2|t}, x_{3|t}, x_{4|t}$，他们都在可行域中，即$x_{1|t}, \ldots, x_{4|t} \in X_{\text{Cstr}}$。然后我们选取此时在虚拟时间步中的第一个最优动作$u^*_{0|t}$作为控制器实际的动作$u_t$进行执行，此时实际状态转移到$x_{t+1}$。我们同样如法炮制，再让实际的状态转移到$x_{t+2}$。但是此时就发生了不可行问题，如上图所示，此时预测出的动作序列的最后一个超出了可行域，即$x_{1|t+2}, x_{2|t+2}, x_{3|t+2} \in X_{\text{Cstr}}$，$x_{4|t+2} \notin X_{\text{Cstr}}$。那么此时因为虚拟时间域中得不到可行解，因此也无法得到需要在实际时间域中要执行的动作，这时就会发生不可行的现象。从这里我们也可以看出，**不可行现象的发生不是因为再实际时间域中我们没有可以采用的动作来转移到下一个满足约束的状态（实际上仍然可能有），只是我们再虚拟时间域中构建的OCP问题没有找到可行解**。也就是说，并不是控制器没有可以采取的动作了，而是优化器找不到可行解了。

接下来我们来定性的分析一下状态约束和可行性的关系。对于在虚拟时间域中的OCP问题，如果我们把需要满足约束的horizon设置的越长，得到的策略的可行性就会越高。这是因为我们保证了long-term下满足约束。但是，增长horizon的代价是计算复杂度的增加以及策略最优性的降低。因此，需要恰当的在虚拟时间域中构建OCP问题来达到最优性和可行性的平衡。

实际上，不光是我们这里的讲到的带约束的RL/ADP问题，不可行（infeasible）的问题在很多的其它的带约束的最优控制问题里面页是一个由来已久的问题。比如在带约束的MPC中，就曾经提出了很多方法

来处理这个问题，如horizon enlargement、约束松弛等。下面来简要介绍一下这两种方法：

- **Horizon Enlargement**：
  - **特点**：通过增加horizon的长度来增加可行性。
  - **缺点**：增加horizon的长度会增加计算复杂度，同时也会降低策略的最优性。
- **松弛约束**：
  - **特点**：通过将一些硬性的状态约束转化为soft constraints。
  - **缺点**：
    - 很难确定哪些状态可以被松弛，要松弛的话又要松弛多少。
    - 某些状态由于实际的物理系统的约束是绝对不可以被松弛的。

上述的关于不可行的分析也为我们澄清一些长久以来finite-MPC中的模糊的看法提供了依据。广为人知，horizon很短的MPC问题往往难以保证闭环系统稳定性，只能采用一些stability conditions来修补，比如终止状态约束（terminal state constraints）和终止惩罚约束（terminal penalty constraints）。但是上述stable conditions的缺点和详尽的理论分析也一直未被讨论。其实上述的做法是基于RHC的可行性的假设，然而通过我们刚才的讨论可知这并不是恒成立的。实际情况是stablitity conditions只是通过增加更多的约束来达到效果的，但是这样自然会降低策略的最优性（因为约束多了）和可行性（因为增加了额外的约束，更容易违反了），如果增加的sta'bility conditions过于tight设置会找不到可行解。因此，在MPC中同通过上述tricks来解决闭环系统稳定性的做法不是free lunch，是以牺牲最优性和可行性为代价的。

9.2.3 Constraints的类型和可行性分析

Constraints的类型通常有两种：point-wise constraints和barrier constraints。它们的形式如下：

- **Point-wise constraints**：

$$h(x_{i|t}) \leq 0, \quad \forall i = 1, 2, \ldots, n$$

  这里的n表示需要施加的virtual-time域中的约束的个数。
- **Barrier constraints**：

$$\big(h(x_{i+1|t}) - h(x_{i|t})\big) + \lambda h(x_{i|t}) \leq 0, \quad \forall i = 1, 2, \ldots, n$$

注意，读者需要记住，**凡是带有约束的OCP都是定义在virtual-time域中的**。但是，正如同大多数的文献都不去区分，在下面的叙述中我们也会使用x_{t+i}来表示$x_{i|t}$。

9.2.3.1 类型I: Point-wise constraints

Point-wise constraints描述了在虚拟时间域中每个时间步都需要满足的约束。首先，为了下面描述的清晰，我们需要先厘清两个记号N和n之间的区别。N表示在real-time域中的真实的预测horizon的长度（即真实时间域中cost function中求和的序列的长度），而n表示在虚拟时间域中的预测horizon的长度（在虚

拟时间域中世家约束的不等式数量）。那么根据n域N之间的关系，可以对于point-wise constraints做出如下的分类：

- **Short-horizon Point-wise Constraints（$n < N$）**：
  易知，我们之前讲过的simplest constraint就是一种short-horizon point-wise constraints（$n = 1$）。
- **Full-horizon Point-wise Constraints（$n = N$）**：
  根据N是否等于∞，又可以将full-horizon point-wise constraints进一步细分：
  - **Finite pointwise constraint (full-horizon)**：

$$h(x_{i|t}) \leq 0, \quad \forall i = 1, 2, \ldots, N.$$

  - **Infinite pointwise constraint (full-horizon)**：

$$h(x_{i|t}) \leq 0, \quad \forall i = 1, 2, \ldots, \infty.$$

在下面的讨论中，我们主要讨论上述分类中的两种：$n = N =< \infty$和$n = N = \infty$。我们把前者称为finite-horizon OCP，把后者称为infinite-horizon OCP。注意到，这两种OCP都是full-horizon的。

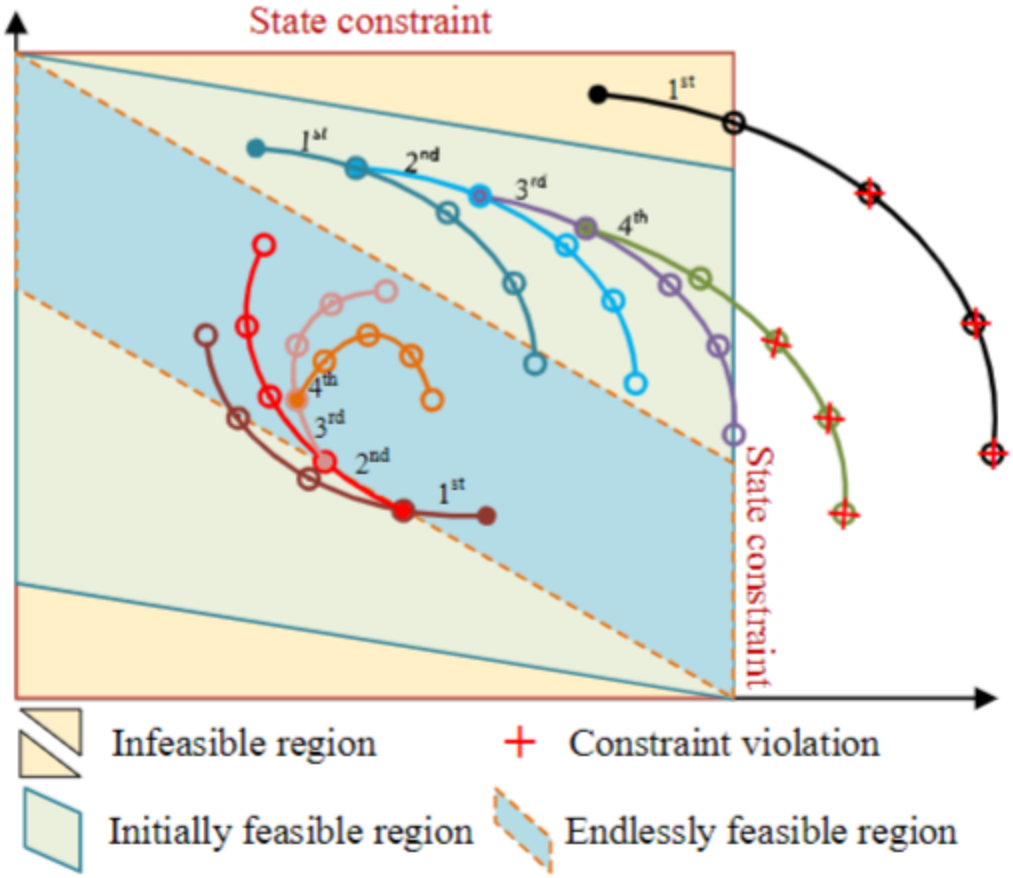
下面我们来对于状态空间$\mathcal{S}$进一步划分。首先，我们之前已经定义了$\mathbf{X}_{\mathrm{Cstr}}$，它表示了所有满足约束的状态的集合。但是，这个集合是否就是可行域呢（或者说，这个集合里的状态点是否全都是feasible的呢）？当然不是！因为在集合$\mathbf{X}_{\mathrm{Cstr}}$只是说明满足约束，但是可行域中每个点的“可行”不仅意味着满足约束，还意味着以该状态点作为输入的虚拟时间中的OCP问题是有解的。因此，**可行域是X_{Cstr}的一个子集**。我们可以进一步将可行域分为两类：

- **Initially Feasible Region**：
  > **定义2：Initially Feasible Region（IFR，X_{Init}）**：
  > 初始可行域是这样一些状态构成的集合：这些状态本身是可行的（对应于OCP有解），但是从它们转移到的后续状态不一定是可行的。

- **Endlessly Feasible Region**：
  > **定义3：Endlessly Feasible Region（EFR，X_{Edls}）**：
  > 永久可行域是这样一些状态构成的集合：这些状态本身是可行的（对应于OCP有解），并且存在一个策略，使得从这些初始状态转移到的后续状态也是可行的。

一个策略只有工作在EFR中才能被称为一个有用的策略并部署在真实环境中。

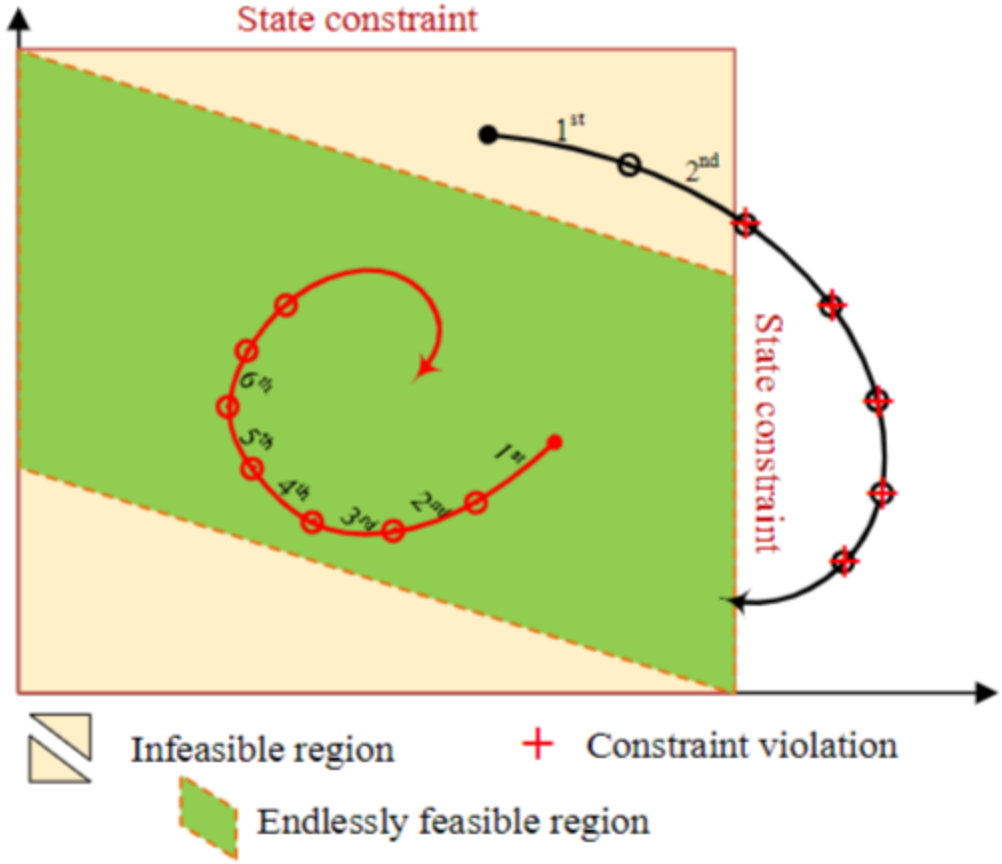
对于finite-horizon OCP问题，X_{Init}、X_{Edls}和X_{Cstr}之间的关系如下图所示：



上图中那个最大的正方形区域就是X_{Cstr}。它们之间的包含关系如下：

$$\mathrm{X_{Edls}} \subseteq \mathrm{X_{Init}} \subseteq \mathrm{X_{Cstr}}$$

而对于infinite-horizon OCP问题，X_{Init}、X_{Edls}和X_{Cstr}之间的关系如下图所示：



它们之间的包含关系如下：

$$\mathrm{X}_{\mathrm{Edls}} = \mathrm{X}_{\mathrm{Init}} \subseteq \mathrm{X}_{\mathrm{Cstr}}$$

这时EFR的等于IFR。这也很容易理解，因为此时我们的虚拟时间中的OCP问题是无限长的，只要输入的初始状态可行且OCP有解，那么必定能满足之后转移到的状态都是可行的（因为如果之后的状态不可行，那么无限长的OCP就没有解了）。因此，此时我们不区分IFR和EFR，将其简称为可行域（feasible region）。此时的可行域只由**环境模型**和**约束**决定，而与优化时采用的标准（criterion）无关。

我们的目标是找到最大的EFR，因为我们找到的EFR越大，可供算法工作在内的安全区域就越大。那么，大家可能会好奇哪种虚拟时间域中的约束可以带来最大的EFR呢？答案是full-horizon point-wise constraint。事实上，我们有如下的定理：

> **定理1**：对于一个**infinite-horizon** OCP问题，**full-horizon point-wise constraint**相比于其它所有可行的虚拟时间域中的约束，具有**最大的EFR**。

证明参考原书第9.2.3.1节。实际应用中，我们在构建OCP时不一定使用full-horizon point-wise constraint，因此得到的也不一定是最大的EFR，而是它的一个子集。在下面的叙述中，我们规定使用X_{Edls}来表示最大的EFR，而使用X来表示我们得到的EFR的一个子集。

其实IFR和EFR这个概念不是一个很新的概念。比如，在今天的MPC研究中，在一个MPC算法被应用之前，总是要计算它的EFR，因为RHC（Receding Horizon Control）的机制并不能保证算法的稳定性和

递归可行性。再者，对于一个RL/ADP算法，它学得的策略只有工作在EFR中才是物理上有意义的。当控制一个处于不可行区域或者IFR区域的状态时，总是会导致不可行（infeasibility）的问题，并可能带来严重的安全问题。因此，一个有用的RL/ADP算法总是应该同时输出一个EFR以及一个在EFR之内的最优策略。还应该注意的是，一个EFR不止对应一种可行的策略，只不过这其中有一个是最优的策略（可以最小化我们的criterion）罢了。

9.2.3.2 类型II: Barrier constraints

在具体讲Barrier constraints之前，我们先来想想一个理想的定义在虚拟时间域中的约束应该具有什么特点。显然，我们希望这个约束在保证系统能够严格保证约束不被破坏的前提下拥有尽可能大的EFR（Endlessly Feasible Region）。我们刚才已经证明过了，full-horizon的point-wise的约束拥有最大的EFR，但是它的不等式太多了；而最开始说的simplest constraint只有一个不等式，但是它的EFR很小。我们希望在二者之间区的一个平衡，那么Barrier constraints就是一个很好的选择。

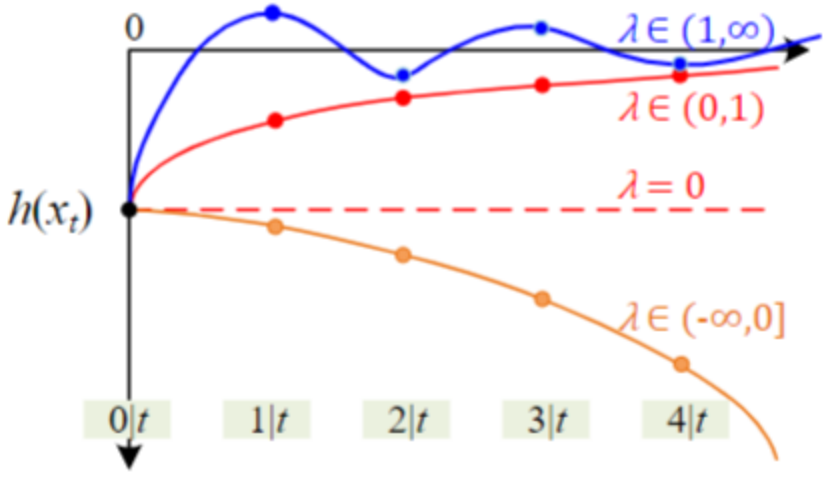
多步的Barrier constraints的形式如下：

$$B\left(x_{i|t}, x_{i+1|t}\right) \stackrel{\text{def}}{=} \left(h\left(x_{i+1|t}\right) - h\left(x_{i|t}\right)\right) + \lambda h(x_{i|t}) \leq 0$$
$$\forall i = 0, 1, ..., n-1,$$

这里的n是不等式约束的数量，通常我们选取时要使它远远小于N或者∞。可以看出，上述一系列不等式拥有cascading的结构，我们可以推出根据上述不等式组，每个虚拟时间步上的约束函数$h(x_{i|t})$之间满足下述的收敛关系：

$$h(x_{i|t}) \leq (1-\lambda)^i h(x_{0|t})$$

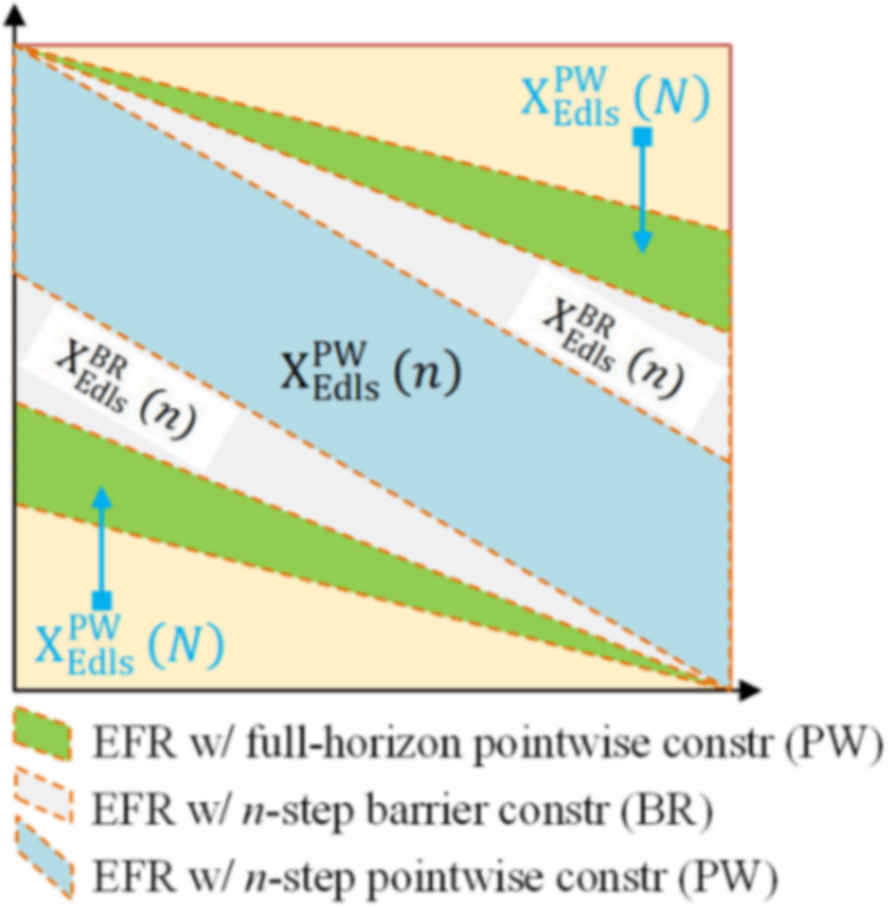
那么，从上式自然易得，γ的选取对于约束强度的影响：



- $\lambda \in (-\infty, 0]$：此时收缩因子$(1-\gamma)$大于1，约束力还不如point-wise的约束强。
- $\lambda \in (0, 1)$：此时收缩因子$(1-\gamma)$小于1，约束力比point-wise的约束更强。
- $\lambda = 1$：退化为标准的point-wise约束。

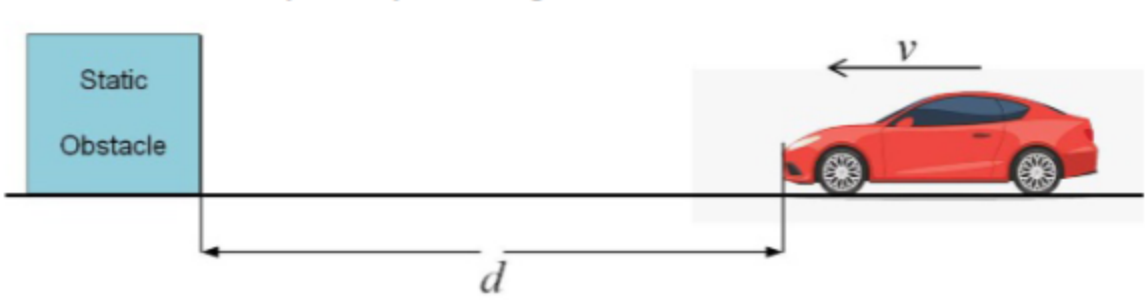
- $\lambda \in (1, +\infty)$：物理上无意义。

对于相同的horizon长度（即虚拟时间域中的不等式数量），barrier约束的约束力通常比point-wise约束更强。换句话说，要实现相同强度的约束，barrier约束所需的不等式数量更少。正因为barrier约束的约束力更强，所以对于同一个初始状态，barrier约束比point-wise约束更有可能保证该状态的长期的recursive feasibility。因此，**其EFR往往大于相同horizon长度的point-wise约束**。如下图所示：



不过，太强的barrier约束也可能会极大的缩小EFR，因为此时很难保证在状态的转移中长期满足约束。而且这种EFR的扩大是以最优性的丧失为代价的。合理的一种设计是只限制下一个状态，这样既有助于降低计算负担又有助于保持良好的recursive feasibility。

9.2.4 以紧急刹车问题为例来说明状态约束与可行性的关系



紧急刹车问题就是要在前方有障碍物时自动采取制动刹车措施。下面来看一下这个问题的建模：

状态	$s=[d,v]^{\mathrm{T}}$
动作	加速度a
环境模型	$\begin{bmatrix} d_{t+1} \\ v_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & -\Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} d_t \\ v_t \end{bmatrix} + \begin{bmatrix} 0 \\ \Delta t \end{bmatrix} a_t$
Performance Index	$\min_{\pi} J = \sum_{i=0}^{N-1} a_{t+i}^2$
约束	$a \in [a_{\mathrm{Brk}}, 0]$ $d_{t+i} \geq d_{\mathrm{safe}}, i = \{1, 2, \cdots, \infty\}$

上面的d和v分别表示车辆与前方障碍物的距离和速度，Δt表示采样时间间隔。Performance Index采用一个长为N的horizon中各个时刻的加速度的平方之和（表示能量）。约束有两类，第一类是加速度约束，其中a_{Brk}是制动的最大加速度；本处我们取为$-10m/s^2$。第二类是与障碍物的距离约束，其中d_{safe}是安全距离，本处我们取为0。

那么，根据我们之前的讲述，可以构建处两种不同的约束：

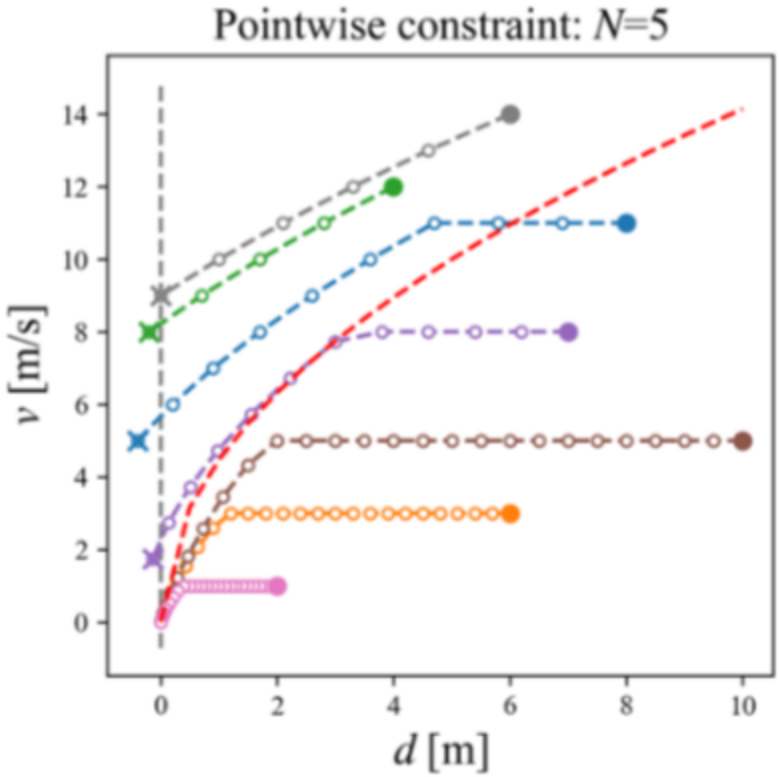
- **Point-wise constraints**：

$$d_{i|t} \geq d_{\mathrm{safe}}, i = \{1, 2, \cdots, N\}$$

- **Barrier constraints**：

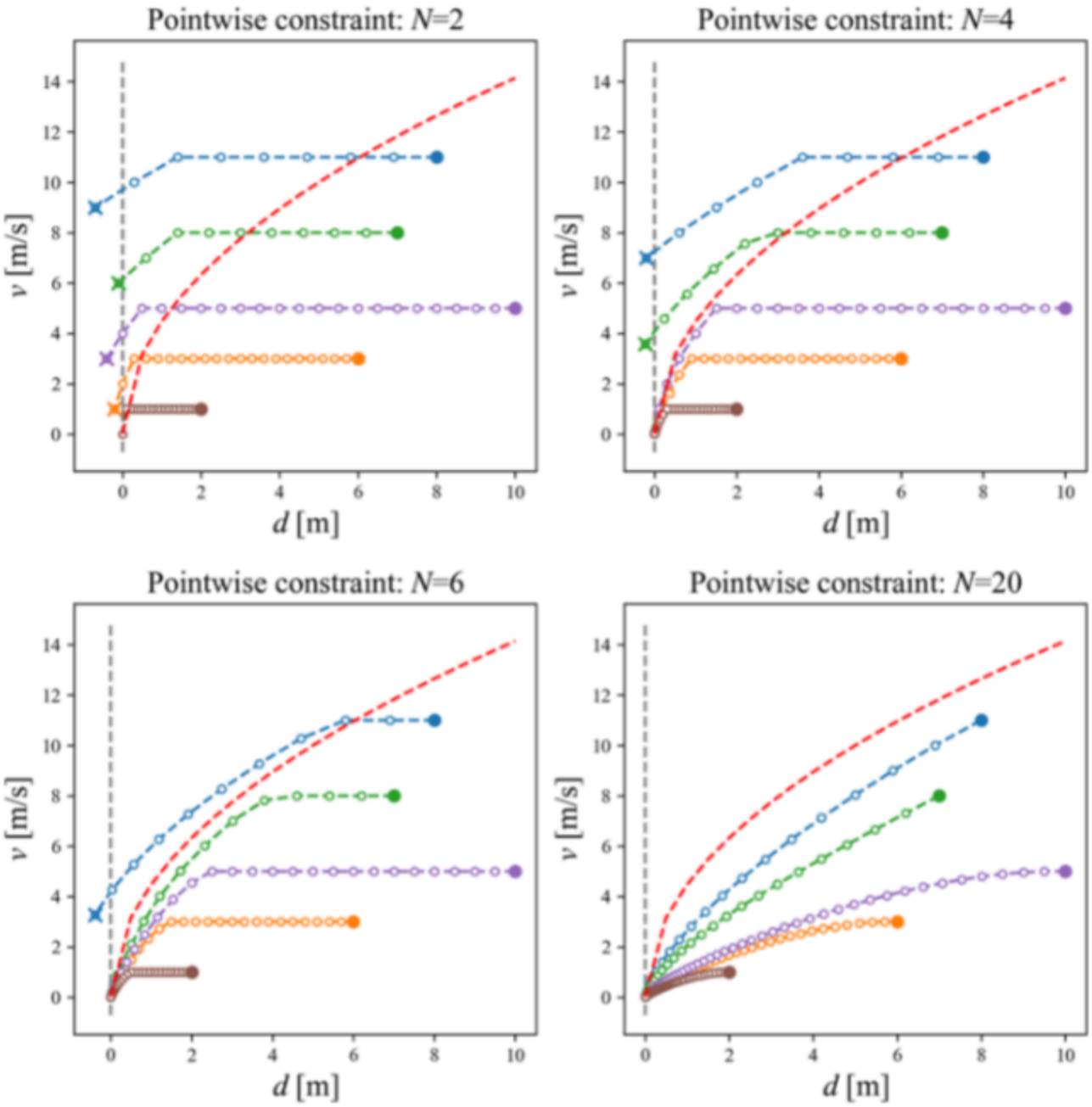
$$\left(d_{i|t} - d_{i+1|t}\right) + \lambda\left(d_{\mathrm{safe}} - d_{i|t}\right) \leq 0, i = \{1, 2, \cdots, N\}$$

下面我们使用上面的两种约束结合MPC进行分析（为什么这里不用RL算法？因为这里我们此时重点是要阐明约束与可行的关系，使用的算法只要能够套进RHC的分析框架即可）。那么，因为我们此时的状态只有两维，所以我们可以将状态之间转移形成的轨迹在二维平面上画出来，如下图所示：

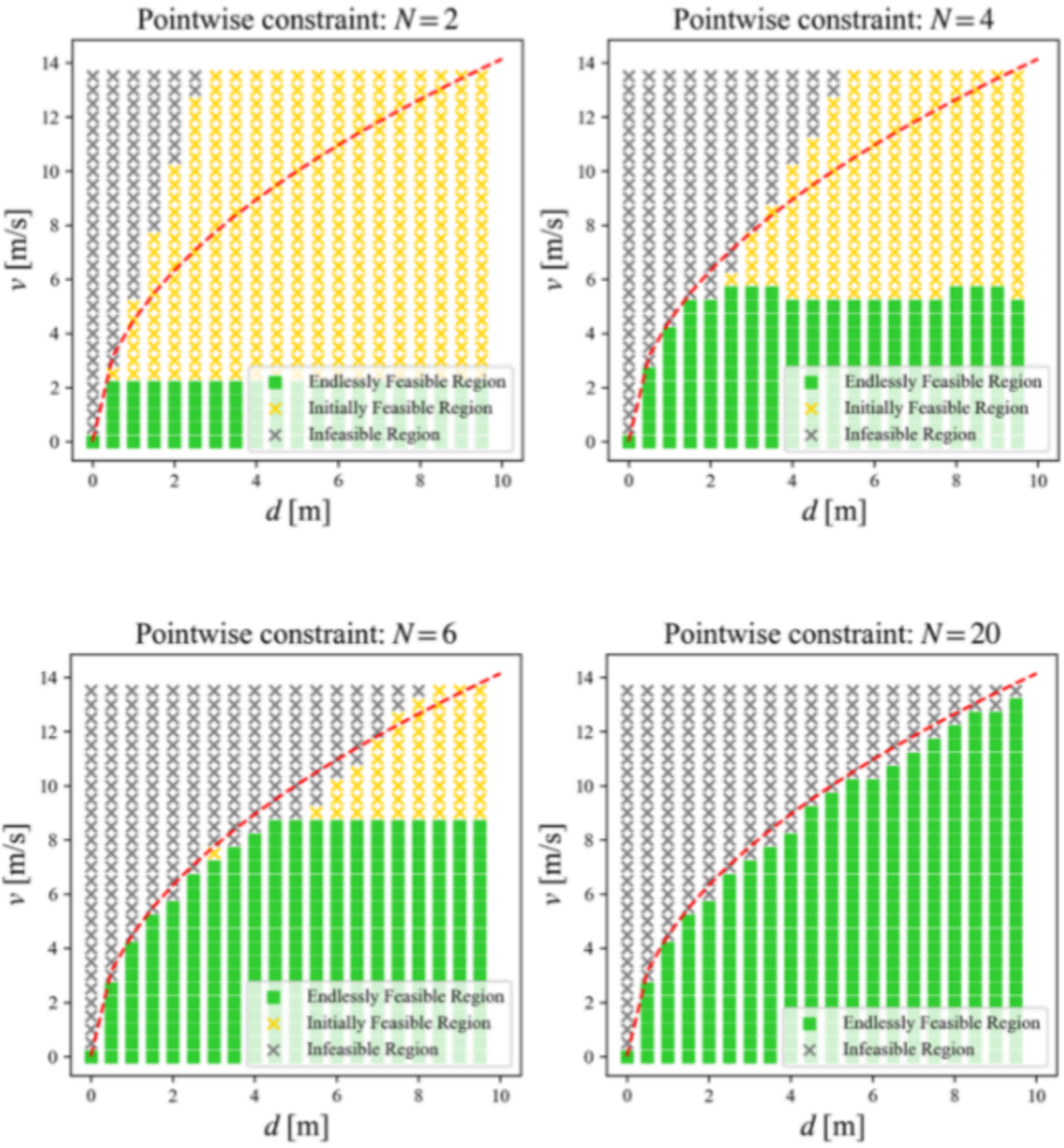


图中的垂直于横轴的那条虚线表示距离约束，而那条红色的曲线代表加速度约束（因为$v^2 = 2a \cdot d$，所以加速度约束可以表示为$d_{\min} = 0.5\frac{v_0^2}{|a_{\text{Brk}}|}$）。图的标题中的$N = 5$表示预测的horizon的长度。而图中所有哪些靠右的实心点表示初始状态，之后的状态则使用空心点表示。而破坏约束的点使用$\times$来表示。下面所有的同类图中含义均与此处相同。

首先我们来关注point-wise constraints的情况。下面一组图首先展示了不同的horizon长度（这里的N既是虚拟时间域中的约束数量，又是真实时间中的cost function中求和的长度，即$n = N$）下的情况：

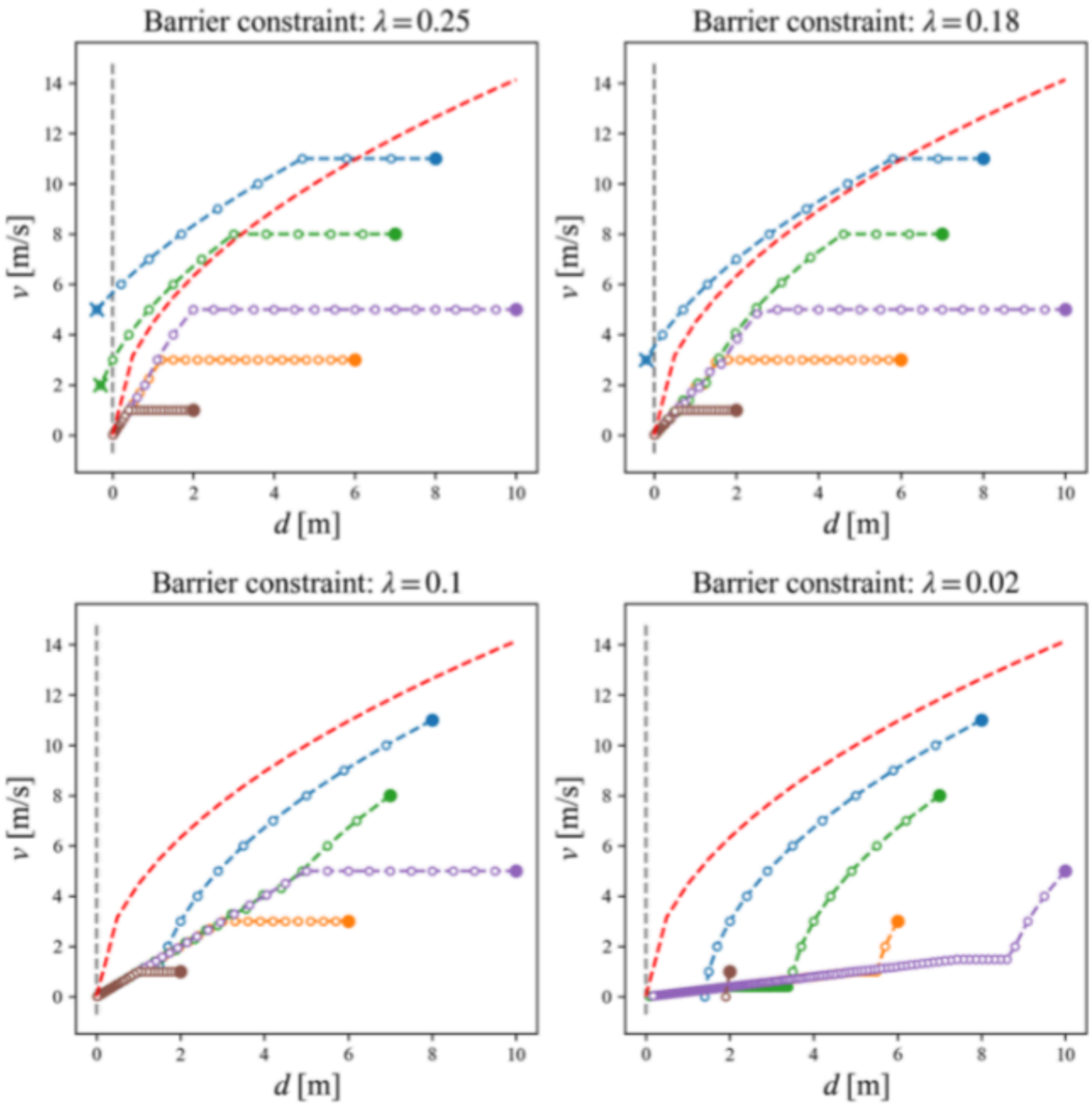


可以看出，随着horizon长度的增加，违反约束的轨迹越少。这里的道理很好理解，就是因为约束更加严格了，因此违反约束的可能性就越小。下图展示了不同horizon长度下不可行区域、初始可行区域（IFR）和永久可行区域（EFR）的情况：

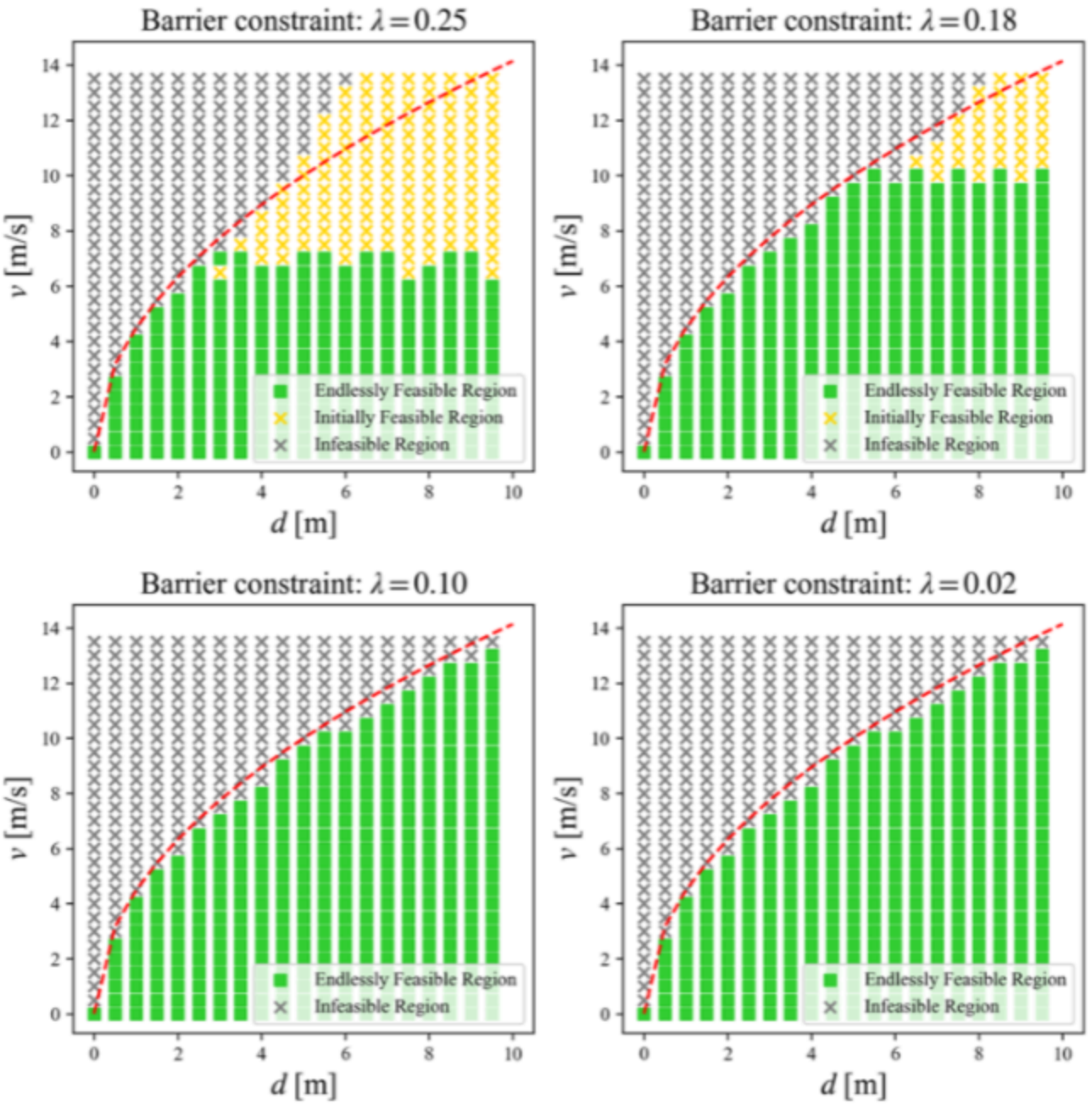


可以看出，随着horizon长度的增加，EFR面积也不断增加。当N=20的时候，此时的EFR已经与理论上的最大EFR（无限长horizon的point-wise constraints下的EFR，即图中红线下面的区域）非常接近了。同时，从N不断增大的过程中，我们也发现了初始可行区域也不一直在发生变化，不是一成不变的。这是因为IFR的定义是该集合中的状态使得OCP有解，那么随着N的增大，求解每个OCP时需要处理的约束数量也在增加，因此更难满足约束，因此IFR也在不断变小。从变化趋势也可以看出，当$N \to \infty$时，IFR也与理论上的那个红线下面的区域重合了。

接下来我们来看barrier constraints的情况。这里barrier constraints的取为两步的情况。先来看看不同的λ下的情况：



可以看出，随着λ的减小，违反约束的轨迹越来越少，这说明barrier constraints的约束力越来越强。当$\lambda = 0.1$的时候，约束效果就已经很好了。而当$\lambda = 0.02$的时候，约束力太强导致了有些轨迹甚至在离垂直于横轴的那条灰色虚线还有一定距离的地方就停下了。下面展示了不同λ下的不可行区域、初始可行区域（IFR）和永久可行区域（EFR）的情况：



可以看出，随着λ的减小，EFR面积也不断增加。当$\lambda = 0.1$的时候，此时的EFR已经与理论上的最大EFR很接近了。而当λ继续减小，此时的约束会越来越保守，可能导致EFR反而减小。

同时，可以看出，要达到同样的EFR近似于理论上最大的EFR，barrier constraints所需要的约束数量要远远小于point-wise constraints（2步对20步）。这也说明了barrier constraints的具有更强的约束力的特点。

9.2.5 带约束的RL/ADP算法的分类

正如之前所说，主要有三种方法来处理带约束的最优化问题（OCP问题）：

- **罚函数法**：将一个惩罚项加入cost function中，来惩罚违背约束的状态。

- **拉格朗日乘子法**：这种方法使用对偶理论来确定原始问题的下界。根据Slater条件，这样可以得到一个最优解。
- **Feasible Descent Direction（FDD）法**：在FDD方法中，更新的方向既可行又可以保证下降，并且原始的带约束的OCP问题被转化为了一系列的局部的凸优化问题。

与不带约束的RL/ADP问题相似，带约束的RL/ADP问题也可以分为两种方法：Direct Method（带约束的策略梯度）和Indirect Method（带约束的策略迭代和带约束的值迭代）。Direct RL/ADP方法中，带约束的OCP问题被转化为一个带约束的优化问题，最优解使用随机梯度下降来得到；而Indirect RL/ADP方法中，通过求解带约束的贝尔曼方程（通常同时是最优性的充分和必要条件）的解，来作为最优策略。

下表展示了带约束的RL/ADP算法的分类，其中黑色小圆圈表示对应的算法存在，而白色小圆圈表示对应的算法不存在，“#”表示会在下面的章节中讨论的算法。

	Direct methods			Indirect methods		
	Penalty	Lagrange	FDD	Penalty	Lagrange	FDD
Finite horizon	●	●	●	◌	◌	◌
Infinite horizon	●#	●	●	●	●#	●#

● Corresponding RL/ADP exists;

◌ Corresponding RL/ADP does not exist;

Constrained RL/ADP that is introduced in this chapter.

9.2.5.1 对于带约束的OCP问题的Direct RL/ADP方法

不失一般性，考虑之前说过的full-horizon point-wise constraints的情况。我们可以将带有状态约束的OCP问题表示为：

$$\min_{u} V(x) = \sum_{i=0}^{\infty} l(x_{t+i}, u_{t+i}),$$
$$s.t.$$
$$x_{t+i+1} = f(x_{t+i}, u_{t+i}),$$
$$x \overset{\text{def}}{=} x_t$$
$$\text{with the full-horizon pointwise constraint:}$$
$$h(x_{t+i}) \leq 0, \quad i = 1, 2, ..., \infty,$$

注意，这里的full-horizon问题的horizon取得是∞。另外，为了与其他的经典的文献里面的记号对齐，这里约束采用的记号看似（写成）定义在实际时间域中的，但需要牢记于心的是所有的约束都是定义在虚拟时间域中的。

前面介绍过的罚函数法、拉格朗日乘子法和FDD法都可以用来解决上述的带约束的OCP问题对应的RL/ADP问题。PDO（Primal-Dual Optimization）算法是一种前沿的拉格朗日乘子法类的方法。它使用对偶上升技术来进行策略更新。TRPO算法可以被看做一种带约束的model-free的强化学习算法。它使用二阶椭圆形的约束来限制策略更新的步长。局部的线性化允许TRPO算法使用一个共轭的梯度算子来更有效率的计算梯度。而CPO算法则是TRPO算法的一个扩展，它能保证每次策略更新不会违反约束。

9.2.5.2 对于带约束的OCP问题的Indirect RL/ADP方法

实际上，处理带约束的OCP问题的Indirect RL/ADP方法比不带约束的版本历史更长。罚函数法可以追溯到上世纪七十年代的risk-sensitive MDP。这类方法把约束作为奖励信号的一部分。由此也可以引出一系列risk-sensitive的TD算法，比如risk-sensitive Q-learning和risk-sensitive TD(0)。而为了应用拉格朗日或者FDD方法，需要使用一个一个**带约束的actor**和一个不带**约束的critic**来求解带约束的贝尔曼方程。带约束的actor需要去求解一个带约束的优化问题，而不管是拉格朗日乘子法还是FDD法，都可以在初始状态可行的条件下递归的保证后续状态的可行性。下面我们来看看怎么构建一个带约束的贝尔曼方程。

> **定义4：可行的策略（Feasible Policy**：
> 对于一个给定的区域$X \in X_{\mathrm{Edls}}$，我们称一个策略在这个区域内是可行的，如果对于任意的初始状态$x_0 \in X$，它后续转移到的状态$x_1, x_2, \cdots, x_\infty \in \mathrm{X}$。

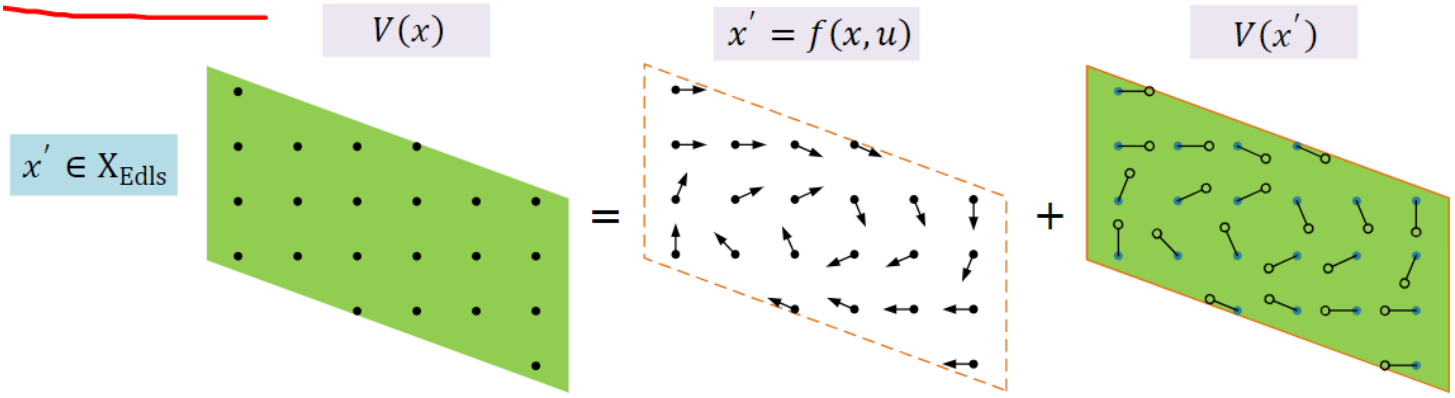
无论何时一个策略被称为是可行的，我们都必须事先指定它所工作的区域。那么当然，最完美的工作区域就是最大的endless feasible region：X_{Edls}。但这个最大的EFR不总是可以的，因此它的一些特殊的子集也可以被作为工作区域。换句话说，一个可行的策略要么被定义在X_{Edls}上，要么被定义在X_{Edls}的一个特殊的子集X上。这里的特殊指的是任何从X中开始的状态都可以保证后续的状态都待在X中。

带约束的Indirect RL/ADP方法的构建依赖于下面的贝尔曼方程：

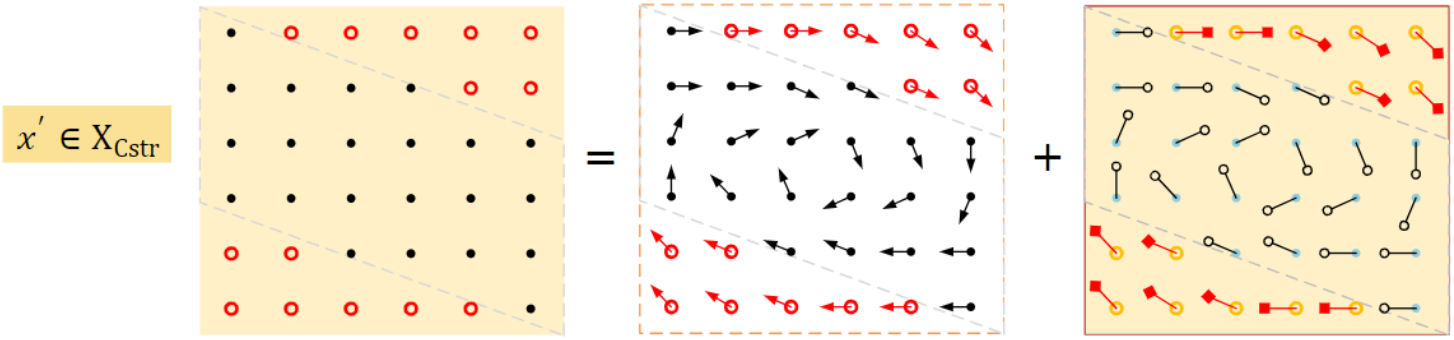
$$
\begin{aligned}
V^*(x) &= \min_u \{l(x,u) + V^*(x')\}, \\
&\text{s.t.} \\
x' &= f(x,u), \\
&\text{with one-step constraint:} \\
x' &\in \mathrm{X}_{\mathrm{Edls}}.
\end{aligned}
$$

可以看出，在带约束的贝尔曼方程中，约束只表现为一步的形式，即要求下一步转移到的状态x'停留在X_{Edls}中。为了保证递归可行性（recursive feasibility，即对于Policy Iteration来说，所有中间策略都是“可行”的），下一个状态x'必须定义在一个特殊的EFR中。我们自然会想，最好的选择就是X_{Edls}，它提供了关于约束的最明显信息。因此，我们在上面的带约束的贝尔曼方程中确实也是这样选的，这样可以最大化最终得到的策略的工作区域。可能大家会疑惑，为什么不能选最原始的$x' \in X_{\mathrm{Cstr}}$（即$h(x') \leq 0$）呢？因为这样选似乎不用提前计算任何EFR，算法的设计也会更显得简单。其实，这样选也是可以的，只不过这样是隐式的寻找一个满足约束的策略。我们可以使用贝尔曼方程的机制来解释一下为什么

这样选也是可行的。本质上来说，带约束的贝尔曼方程中的one-step约束表明了带约束的优化问题在哪些区域内是数学上可行的（可行换句话说就是状态值函数在定义的区域内是有限值）。因此选择$x' \in X_{\text{Cstr}}$也可以，因为这样在优化过程中那些违反约束的点会被排除在外。最终随着优化过程中对于带约束的环境动力学的探索，可行区域可以被确定下来。选择$x' \in X_{\text{Cstr}}$的坏处是这样没有显式的考虑到约束，需要进行大量的计算。关于选择$x' \in X_{\text{Edls}}$还是$x' \in X_{\text{Cstr}}$的一个简单图示如下：



(a) w/ the largest EFR



(b) w/ the original constraint

从上图中，可以看出，如果选择$x' \in X_{\text{Edls}}$，那么每次转移也都是在这样一个可行区域中进行转移，不会超出这个区域，探索到不可行的状态；而如果选择$x' \in X_{\text{Cstr}}$，，每次趋势会探索到一些不可行的状态（红色的点）。

下一个问题是怎么设计一个迭代算法来求解带约束的贝尔曼方程的解。我们可以选择带约束的策略迭代，那么关键问题就是怎么处理单步约束。一种看法是我们可以把约束看成是环境动力学的一部分。由此也可以知道，可行的约束应该具有马尔科夫性质（每个约束只与当前的状态，上一个状态以及动作有关），这样才能被视为马尔科夫性质的一部分。如果没有约束被违反，带约束的策略迭代就与不带约束的版本没有任何区别。因此，在EFR中，带约束的贝尔曼方程的解就是原始的OCP问题的最优解；而在EFR之外，讨论贝尔曼方程的解是没有意义的，因为在未来某个时间点约束总会被为违反。

现在，让我们来讨论一下怎么设计一个带约束的策略迭代算法。按之前的讨论，算法应该包含两部分：（1）PEV（Policy Evaluation）：即critic的更新。（2）PIM（Policy Improvement）：即actor的更新。在

PEV阶段，因为当前的策略是可行的，所以不需要考虑约束，所以说**PEC阶段与无约束的版本的PEV阶段是一样的**，self-consistency条件不变：

$$V^k(x) = l(x, \pi_k) + V^k(x').$$

在PIM阶段，则必须考虑约束，以便找到一个更好且可行的策略，否则在下一次迭代中self-consistency条件就会被破坏，因为此时状态值函数超出工作区域就不会有意义了。因此，PIM过程应该是下面这样的带约束优化问题：

$$\begin{gathered}\pi_{k+1}(x) = \arg\min_u \{l(x,u) + V^k(x')\}, \\ \text{with} \\ x' = f(x,u), \\ x' \in \mathrm{X}_{\mathrm{Edls}}.\end{gathered}$$

可以看出上述的优化目标说明，对于每一个状态x，都要找到一个u使得它的状态值函数最小（使用贝尔曼方程表示的）。通过限制x'在X_{Edls}中，可以保证每次都能是策略在保持可行的条件下变得更好。为了证明我们可以不断地重复应用PEV和PIM，我们需要证明策略π_k在X_{Edls}中是递归可行的（即每次通过π_k求得的π_{k+1}是可行的）。那么关键是证明π_{k+1}总是存在于X_{Edls}中。简单证明如下：

> **定理2**：对于一个带约束的策略迭代算法，如果初始策略π_0是可行的，那么对于任意的k，π_{k+1}都是可行的，只要它之前的策略π_k是可行的。

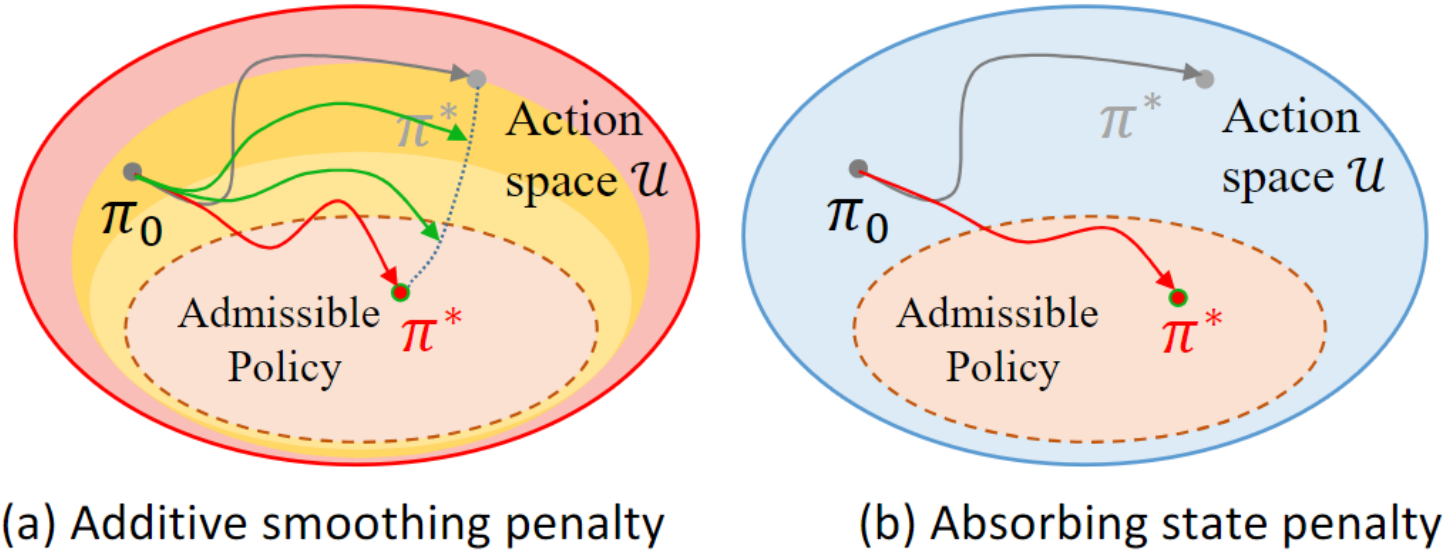
证明：首先，如果π_k是可行的，那么$V^k(x)$在X_{Edls}中是有限值。那么因为$V^k(x)$是有限值，那么每次PIM时构建的优化问题都是有有意义的。那么如果从这个优化问题求得的π_{k+1}是非平凡的，那么它必然是可行的。而如果它是平凡的，那么至少可以选择$\pi_{k+1} = \pi_k$。因此，总存在一个可行的π_{k+1}在X_{Edls}中。证毕。

除了递归可行性，收敛性也是一个重要的性质。在EFR中时，证明收敛性是比较容易的，可以像证明无约束的版本一样，主要分为两个部分：（1）值函数值不断下降；（2）最终可以收敛到最优值。这里就不再赘述了。另外值得指出的是，关于EFR和递归可行性的概念在一些MPC的文献中也被提到，只不过用的名词不一样而已。详情可参考Borreli在2017年的书（详见本文章对应的《Reinforcement Learning for Sequential Decision and Optimal Control》的参考文献[9-7]）。

9.3 罚函数法

这里我们使用Direct的基于模型的ADP来展示如何应用罚函数法。核心思路是在原本的代价函数中加入一个很高的cost来惩罚任何违反约束的状态。这样修改了代价函数后，优化这个目标就会使得学到的策略尽可能多的满足约束，便面违反约束的行为（虽然不一定能完全避免）。罚函数法可以分为两个主流

的类别：（1）**additive smoothing penalty**（加性平滑惩罚）；（2）**absorbing state penalty**（吸收状态惩罚）。下图简要展示了这两种罚函数法在策略更新时的区别：



(a) Additive smoothing penalty　　(b) Absorbing state penalty

Figure 9.15 Policy updates with penalty functions

在additive smoothing penalty中，学到的策略随着越来越多关于违反约束的证据的积累被缓慢拽回到可行区域。而absorbing state penalty则具有辨识可行的动作的one-shot能力。这是因为累积的Return变化的更快。

9.3.1 Additive Smoothing Penalty

通过将惩罚项以加法的形式与原有的cost function组合，就可以得到一个新的包含了约束信息的cost function。罚函数法中的外点罚函数和内点罚函数法都可以使用，但是外点罚函数法更常用，因为它放松了约束，更有利于找到可行解而不是陷入不可行（Infeasibility）的问题。下面以infinite-horizon的pointwise constraints为例来展示如何使用罚函数法来处理带约束的OCP问题。

首先，我们定义一个新的cost function：

$$J_{\text{mod}} = \sum_{i=0}^{\infty} l_{\text{mod}}(x_{t+i}, u_{t+i}),$$
$$\text{with}$$
$$l_{\text{mod}}(x,u) = l(x,u) + \rho \cdot \phi(x),$$

其中$\rho \geq 0$是一个惩罚系数，$\phi(x)$是一个惩罚函数。这里的惩罚函数的种类有很多，比如最常见的Rectified Linear Penalty（RLP）：`

$$\phi(x) = \begin{cases} 0 & h(x) \leq 0 \\ h(x) & \text{otherwise} \end{cases}$$

其他的惩罚函数还包括one-shot constant penalty、exponential risk penalty和rectified quadratic penalty。

Additive Smoothing Penalty容易学到一个局部最优策略。但是它也有自己的好处，即通过加入违反约束惩罚项学到的值函数的可以作为指示一个状态是否违反约束。如果penalised value function在某个状态下的值很大，那么这个状态就很可能是违反约束的。

9.3.2 Absorbing State Penalty

不同于Additive Smoothing Penalty在每个状态的cost function中都加入了惩罚项，Absorbing State Penalty则是将违反约束的状态直接赋值一个很大的常数作为惩罚而其他状态的代价不变：

$$l_{\mathrm{Mod}}(x,u)=\begin{cases}l_{\mathrm{absorb}} & h(x)>0\\ l(x,u) & h(x)\leq 0\end{cases}$$

这里的l_{absorb}是一个很大的常数。正是因为这个很大的常数，探索通常会结束的更早，因此在相同时间内可以收集到更多的experience，训练效率更好。

Additive Smoothing Penalty和Absorbing State Penalty两种方法兑换币，两者都是追求可行的策略更新。区别在于前者需要仔细平衡原有的cost function和惩罚项，否则算法很容易失败或停留在局部最优；而后者更像是一个one-shot策略，它的早停（违反约束时给一个很大的惩罚）使得它对于可能的违反约束的行为施加更强的惩罚，而且只要l_{absorb}足够大，就更有可能学到全局最优解。

9.4 拉格朗日乘子法

这里我们以带约束的PIM问题为例，来展示怎么使用拉格朗日乘子法。首先在理论上，我们希望每次PIM时构建的优化问题中的约束为$x'\in X_{\mathrm{Edls}}$。但是在实际中，X_{Edls}通常是未知的，因此我们需要使用使用已知的约束$h(x)\leq 0$来代替（即$x'\in X_{\mathrm{Cstr}}$）。正如我们之前说过的，这是一种放松的约束，但是也可通过不断的迭代试错最终找到一个可行的策略。关于更详细的合理性分析详见本章后续讲的Actor-Critic-Scenery架构。

拉格朗日对偶是拉格朗日乘子法的核心。它用来将带约束的PIM问题转化为一个拉格朗日对偶问题，这样当actor loss被最小化的时候至少可以找到原问题的一个下界。特别的，如果actor loss和状态约束都是凸的且满足Slater条件，那么拉格朗日对偶问题的解就是原问题的解（这就是满足了强对偶条件）。特别的，强对偶条件也解释了互补松弛条件的来源。这个条件也提供了一个检验可行性的方法。

9.4.1 Dual Ascent方法

为什么要把原问题（primal problem）转化为对偶问题（dual problem）呢？这是因为对偶问题比原问题好求解。即使原问题是一个非凸问题，对偶问题也是一个凹的最大化问题。

在duel ascent方法中，对偶问题可以通过交替求解最大化和最小化问题来求解。当强对偶条件满足时，就可以收敛到一个鞍点。首先，我们需要定义一个拉格朗日函数：

$$L(\theta,\lambda)\overset{\text{def}}{=}l(x,u)+V(x')+\lambda h(x'),$$

其中θ是actor的参数，λ是拉格朗日乘子。然后，我们可以定义一个对偶函数：

$$g(\lambda)\overset{\text{def}}{=}\min_{\theta}L(\theta,\lambda).$$

对偶函数的最大化问题就是对偶问题：

$$\max_{\lambda}g(\lambda).$$

也就是说，原问题的对偶问题是一个max-min问题：

$$\max_{\lambda\geq 0}\min_{\theta}\{L(\theta,\lambda)\}.$$

接下来我们就来看看怎么求解这个对偶问题。根据其max-min问题的本质，可以将其分为两步求解：

$$\begin{aligned}\theta&\leftarrow\arg\min_{\theta}L(\theta,\lambda),\\ \lambda&\leftarrow\lambda+\alpha_{\lambda}\cdot\nabla_{\lambda}L(\theta,\lambda).\end{aligned}$$

这里的λ必须非负。但是，这里第一步还是需要求解一个最小化问题，这不符合我们使用优化方法求解的本意。因此，我们要把这个最小化问题也转化为一个可以使用梯度下降求解的优化问题，转化后的两步更新如下：

$$\begin{aligned}\theta&\leftarrow\theta-\alpha_{\theta}\cdot\nabla_{\theta}L(\theta,\lambda),\\ \lambda&\leftarrow\lambda+\alpha_{\lambda}\cdot\nabla_{\lambda}L(\theta,\lambda),\end{aligned}$$

其中

$$\begin{cases}\alpha_{\theta}>0,\alpha_{\lambda}>0\\ \alpha_{\theta}\gg\alpha_{\lambda}\\ \nabla_{\theta}L(\theta,\lambda)=\frac{\partial u^{\mathrm{T}}}{\partial\theta}\frac{\partial L(\theta,\lambda)}{\partial u},\quad\nabla_{\lambda}L(\theta,\lambda)=h(x')\end{cases}$$

这里需要解释几点：

- 为何$\nabla_{\theta}L(\theta,\lambda)$写成这样？因为$\theta$是actor的参数，而$u$是actor(策略)的输出,因此在关于Actor的参数θ的梯度中，我们需要使用链式法则来求解。
- 为何$\alpha_{\theta}\gg\alpha_{\lambda}$？这是为了保证第一步通过梯度下降求得的结果接近于argmin运算符求得的最小值。

来总结一下,在Dual Ascent方法中,主循环包含了两个步骤:不带约束的PEV(原因已经在之前说过)和带约束的PIM。而每个PIM中我们也设置了内外双层，首先先通过梯度下降求解一个最小化问题，然后再通过梯度上升求解一个最大化问题。尽管这样做并没有理论保证，但是在实际中使用的效果却总是很好。

更深层的，拉格朗日乘子法可以被视为一种特殊的罚函数法，因此具有一定的自我调节能力来避免违反约束。当出现违反约束时，拉格朗日乘子会自动调节来增加对于拉格朗日函数惩罚的力度。自动调节会抑制约束函数h的力度，使其保持在不违反约束（0以下）的范围内。参考文献[9-34]更进一步提出了一种结合了固定的罚函数因子和拉格朗日乘子的方法，这种方法在实际中更快和更稳定的带约束的强化学习算法。

也可以通过检验一个状态对应的拉格朗日乘子法的值是否大于一个阈值来大概判断这个状态是否违反约束（不可行）。

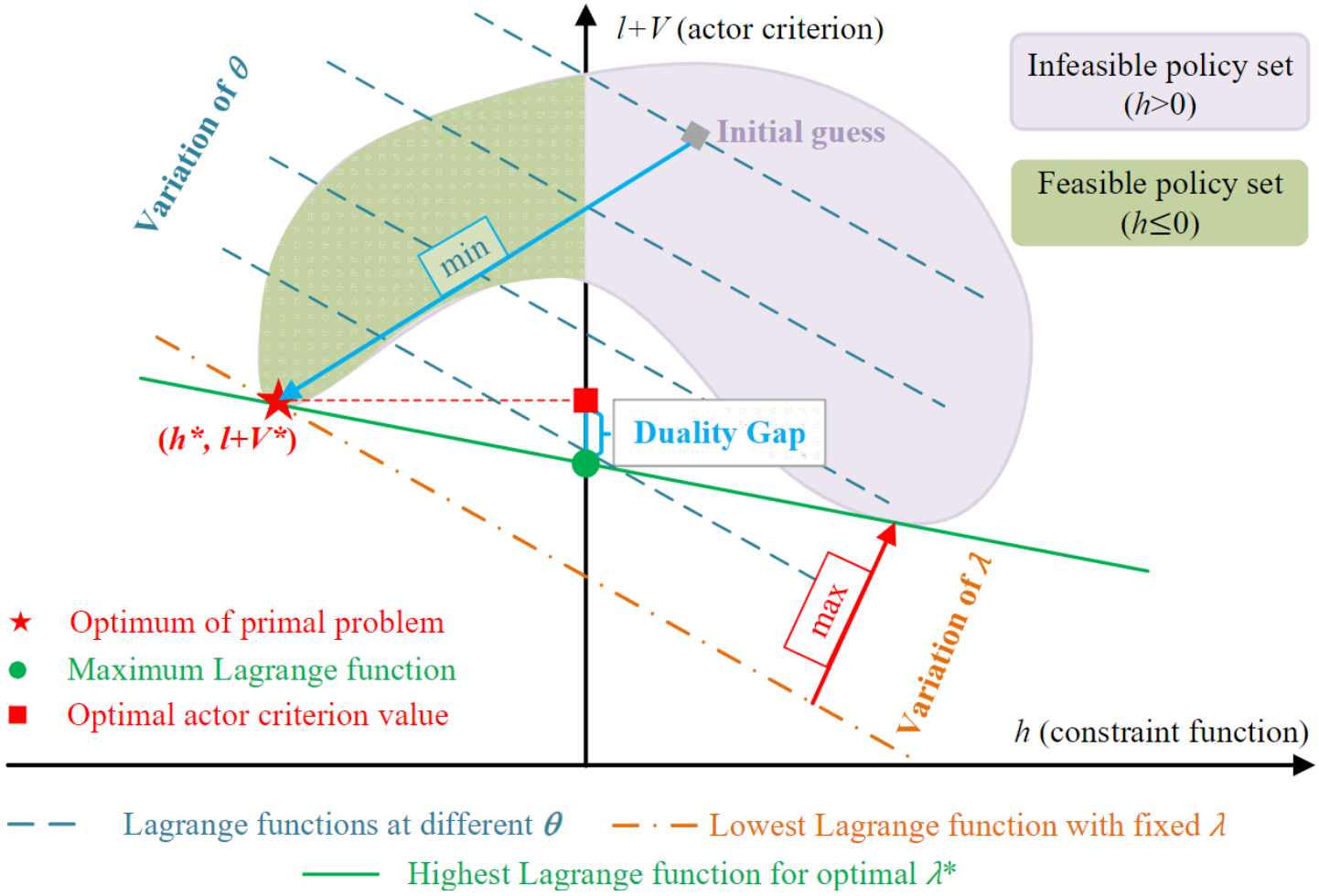
9.4.2 拉格朗日乘子法求解带约束的RL/ADP问题的几何解释

此处我们将用图解法来简单说明一下再拉格朗日乘子法求解带约束的PIM过程中可行性是怎样变化的。在带约束的PIM中，拉格朗日乘子不是一个固定的标量，而是一个定义在整个状态空间的函数。我们选择两个特殊的状态点来展示dual ascent方法的更新过程。第一个状态点来自EFR（Endless Feasible Region），但是它的初始策略是一个违反约束的策略（一个bad guess）；第二个状态点来自不可行的区域，这样自然就没有任何可行的策略存在。这两个状态点对应的拉格朗日乘子的更新过程呈现出非常不同的行为。下面将分别说明。

首先，我们来定义一个新的空间$\mathcal{G}$：

$$\mathcal{G} = \{(h(x), l(x,u) + V(x')) \mid x \in \mathcal{X}\},$$

这里h是约束（constraint function），在下面的图示中我们把它作为横轴；$l + V$是PIM的优化目标（actor criterion），在下面的图示中我们把它作为纵轴。这样，$\mathcal{G}$就是一个二维的平面（我的理解是对于每一个初始的状态都有这样一个平面）。我们先来看第一种情况，即状态点位于EFR中。那么从这个状态点出发的策略可以形成一个策略集合，在下图中表现为一个月牙形的区域：

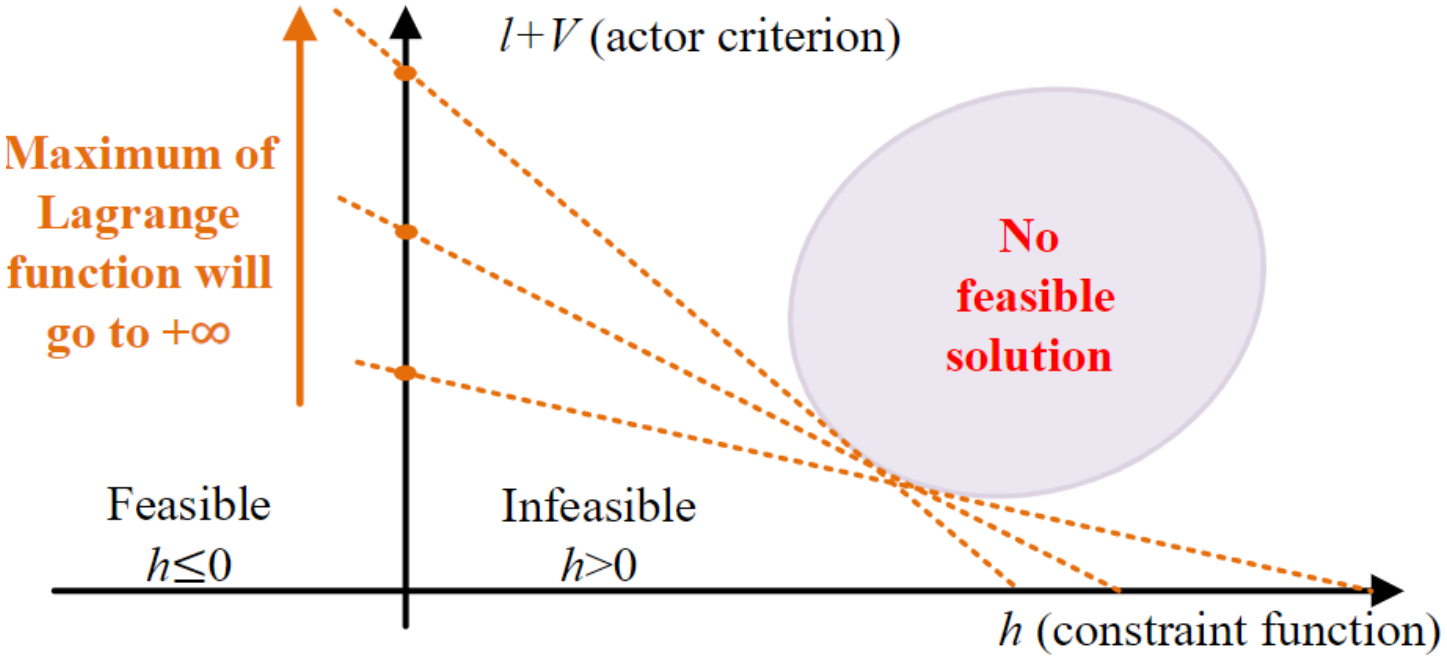


并注意到即使是从一个位于EFR的状态点出发的策略，也可能是违反约束的。因此这个策略集分为两个部分，左边那部分绿色的部分（$h \leq 0$）是可行的策略的集合，右边那部分粉色的部分（$h > 0$）是违反约束的策略的集合。在上图中，我们初始的猜测的策略是一个违反约束的策略（位于右边的粉色区域）回想刚才讲过的dual ascent算法的更新原理，可以把拉格朗日对偶问题写成下面的形式：

$$\max_{\lambda \geq 0} \min_{(h, l+V) \in \mathcal{G}} \{(l+V) + \lambda h\}.$$

这其实代表了PIM过程中双层循环：内层的最小化循环是在拉格朗日乘子λ固定的情况下，找到一个θ可以最小化拉格朗日函数$(l+V)+\lambda h$；那么根据上图的几何意义，易知图中每条虚线上的点拉格朗日函数值相同（$(l+V)+\lambda h = \text{const}$）同时虚线越靠下拉格朗日函数的值越小，而λ可以理解为图中虚线的斜率（实际不是斜率，差一个负号，不过理解意思就行）。因此，内循环的更新可以理解为不断平移图中虚线（保持斜率不变），直到找到一个与左侧可行域相切的且最靠下的虚线。再来看外层循环，它是要找到最大的拉格朗日函数的值。在图上就表示为在保持**与左侧可行的策略集合相切**的情况下不断改变斜率，使得拉格朗日函数的值（体现为与纵轴的截距）最大，并且不能进入右侧的违反约束的策略集合（即不能进入右侧的粉色区域）。这样，最终找到的直线就是那条绿色实线，它与纵轴的交点就是实际上通过dual ascent方法找到的最优的actor criterion的值，而实际上的最优值是图上的红色方块，绿色圆点与红色方块的距离就是所谓的duality gap。

下面来看看当初始化的状态点位于不可行的区域时的情况。



此时因为没有可行策略集合的约束，不断改变斜率会导致拉格朗日函数的值不断增大，直到无穷。因此，这再次印证了拉格朗日乘子法是否趋于无穷可以用来检验一个对于某个特定的状态是否能为其找到一个可行的策略。

9.4.3 增强的拉格朗日乘子法（Augmented Lagrangian Multiplier）

与普通的拉格朗日乘子法相比，增强的拉格朗日乘子法更加鲁棒，收敛速度更快。这种方法可以提视为罚函数法和拉格朗日乘子法的结合，其基本思路是在拉格朗日函数中加入一个二次惩罚项。为什么要做这种改进呢？因为虽然在理论上dual ascent方法可以收敛到带约束的最优值，但是对于非凸的问题收敛性常常不令人满意。而正如在罚函数法那里说的，罚函数法能改变带约束的策略梯度，因此能提升收敛的质量。增强的拉格朗日函数定义如下：

$$L_{\mathrm{Aug}}(\theta, \lambda, \zeta) = l(x, u) + V(x') + \lambda h(x') + \frac{\rho}{2}\|h(x') + \zeta\|_2^2,$$

其中$\rho > 0$是一个惩罚系数，$\zeta \geq 0$是一个松弛变量，用于将不等式约束转化为等式约束：

$$h(x') + \zeta = 0,$$

之后这个等式约束再转化为二次的形式。只要惩罚系数ρ足够大，这样的二次罚函数项的引入增强了拉格朗日函数的凸性，有助于非凸问题的快速收敛。考虑到新引入的松弛变量ζ，增强的拉格朗日乘子法有三个更新步骤：、

$$\theta^{\mathrm{new}} \leftarrow \arg\min_{\theta} L_{\mathrm{Aug}}(\theta, \lambda, \zeta),$$

$$\zeta^{\text{new}} \leftarrow \arg\min_{\zeta} \ln L_{\text{Aug}}(\theta, \lambda, \zeta),$$
$$\lambda^{\text{new}} \leftarrow \lambda + \alpha_\lambda \cdot \nabla_\lambda L_{\text{Aug}}(\theta, \lambda, \zeta).$$

增强的拉格朗日乘子法的几何解释与普通的拉格朗日乘子法类似，即拉格朗日乘子λ可以被视为一种为了避免违反约束的动态调整的惩罚力度。二者的区别在于增强的拉格朗日乘子法引入了一个二次惩罚项，由此带来的更强的收敛性有助于稳定训练。

9.5 Feasible Descent Direction（FDD）方法

罚函数法和拉格朗日法都关注于怎么修改原始的criterion。前者修改了overall objective function，后者修改了actor的loss function。但是在大规模优化问题中，由于不合适的惩罚项选择或者不稳定的dual ascent更新，收敛速度可能相对较慢。而作为另一种流行的求解带约束优化问题的方法，FDD方法旨在寻找一个既可行又下降的更新方向。FDD采用了多阶段的优化，通过把原来的优化问题分解为一系列简单的凸优化问题来求解，而策略则是沿着从这一些列简单的优化问题得来的既可行又下降的方向更新。

在静态优化领域，FDD的研究可以追溯到Zoutendiik。他率先将原问题线性化并通过box constraint来限制更新的幅度。之后，通过引入active constraint和增强的目标函数，FDD方法的表现得到了增强。具有线性化目标函数的FDD方法被称为Sequential Linear Programming（SLP）方法。在此基础上出现了一些使用二次近似的方法来改进的研究。在上世纪九十年代，二次目标函数预与约束线性化技术的结合催生了著名的Sequential Quadratic Programming（SQP）方法。SQP方法时至今日仍然是求解带约束优化问题的最有效的方法之一。

但是，不同于静态优化问题一次只需要求解一个固定点，RL/ADP问题需要在整个状态空间寻找一个最优策略，因此落实FDD、方法有一定难度，下面分别介绍。

9.5.1 可行方向和下降方向

这里我们也是通过带约束的PIM来研究怎么使用FDD来求解。并注意这里下一个状态$x' \in X_{\text{Edls}}$仍然被替换为了$h(x') \leq 0$来避免计算EFR。那么在FDD下，更新方向可以写成下面的形式：

$$\theta \leftarrow \theta + \Delta\theta,$$

其中$\Delta\theta$是一个可行的下降方向，必须满足下面两个不等式条件：、

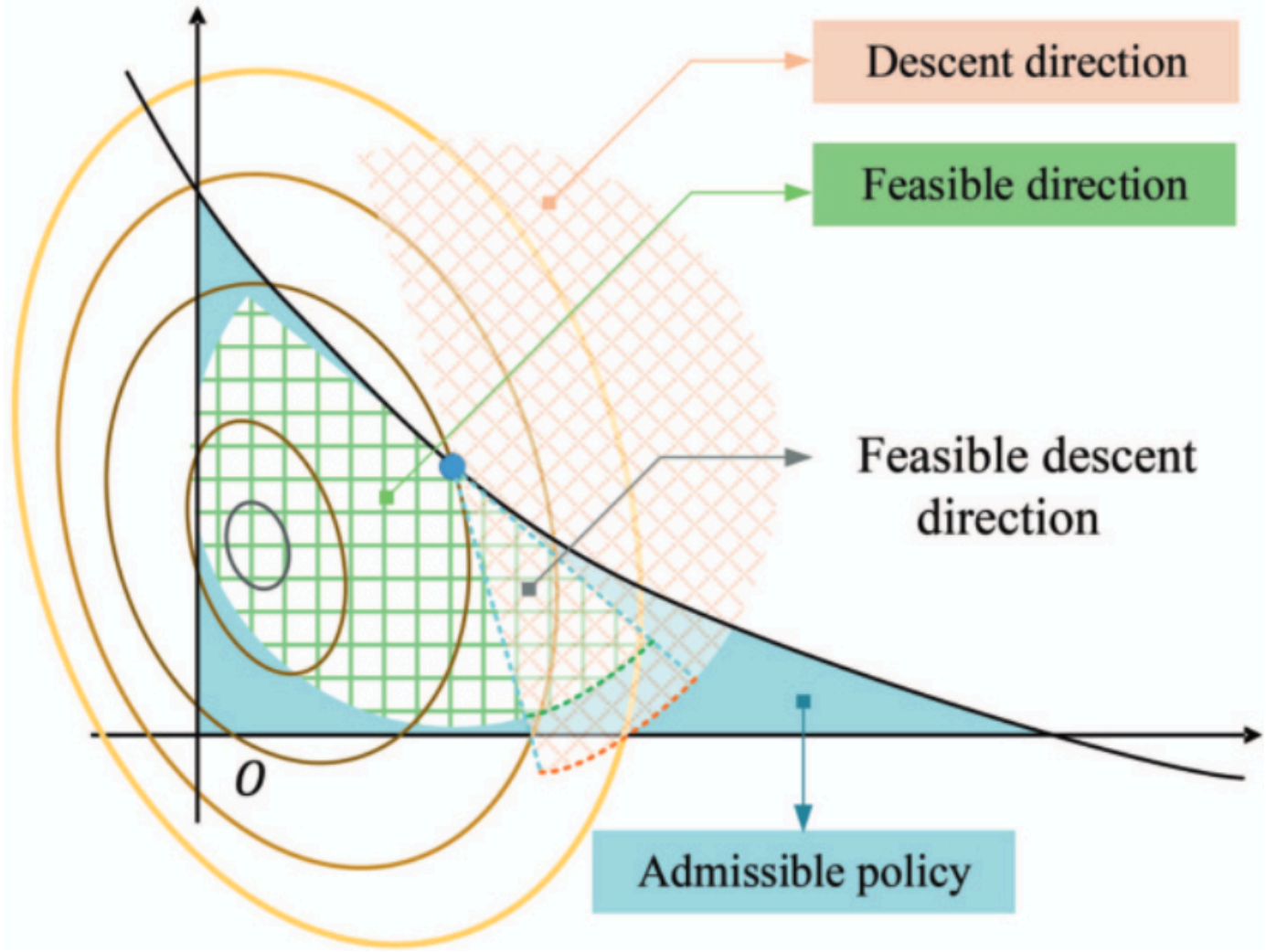
- **下降性条件**：

  $$J_{\text{Actor}}(\theta + \Delta\theta) \leq J_{\text{Actor}}(\theta)$$

  这个条件表示更新后的参数在标准J_{Actor}下降。
- **可行性条件**：

$$h(x'(\theta+\Delta\theta))\leq 0$$

这里$x'=f(x,u)$，$h(x')$是单步约束，而$h(x'(\theta+\Delta\theta))$是强调$x'$是通过更新后的策略选取$u$得到的。



上图中蓝色的区域表示可行的策略的集合，而潜橘黄色的半圆表示能够满足下降性条件的方向，绿色的半圆表示能够满足可行性条件的方向，而两个半圆的交集就是既可行又下降的方向。

但是在绝大多数情况下，解析的求解可行下降方向是不可能的，需要把大的优化问题转化为一系列的线性规划（LP）或者二次规划（QP）问题来更有效的计算。与静态优化问题不同，在每次带约束的PIM中序列化求解LP或者QP问题不是必需的。这是因为在PEV和PIM之间已经存在了迭代循环，所以序列化求解LP或者QP问题可以融入到这个主循环中。换句话说，与静态优化的SLP或SQP问题的不同，在RL/ADP问题中，每个带约束的PIM都只需要转化为一次LP或者QP问题。当然，带约束的PIM也可以多次转化并求解LP或者QP问题，这样可以获得更准确的可行下降方向。

还有一个问题，就是怎么检验可行性正成为以FDD为基础的优化问题中。一个启发式的想法是检验可行性条件$h(x'(\theta+\Delta\theta))\leq 0$是否有解。但是，当我们把这个问题转化为LP或者QP问题时，对这个条件的线性化会使得它失去了可行性的能力。更多关于带约束PIM可行性的evidence需要在给定的状态点反复求解LP或者QP问题来收集。

9.5.1.1 转化为LP问题

通过对下降性条件和可行性条件进行线性化，我们可以把带约束的PIM问题转化为一个LP问题：

$$\begin{aligned}&\min_{\Delta\theta} g^{\mathrm{T}}\Delta\theta,\\&\text{subject to}\\&h(x')|_{\theta}+\nabla_{\theta}h(x')^{\mathrm{T}}\Delta\theta\leq 0,\end{aligned}$$

其中$g=\frac{\partial J_{\text{Actor}}}{\partial\theta}=\frac{\partial\{l(x,u)+V(x')\}}{\partial\theta}$.是actor loss的一阶泰勒展开。下面来解释以下是怎么转化的。首先优化目标是通过对于$J_{\text{Actor}}(\theta+\Delta\theta)\leq J_{\text{Actor}}(\theta)$进行展开得到的。首先将所有项均移项到左边：

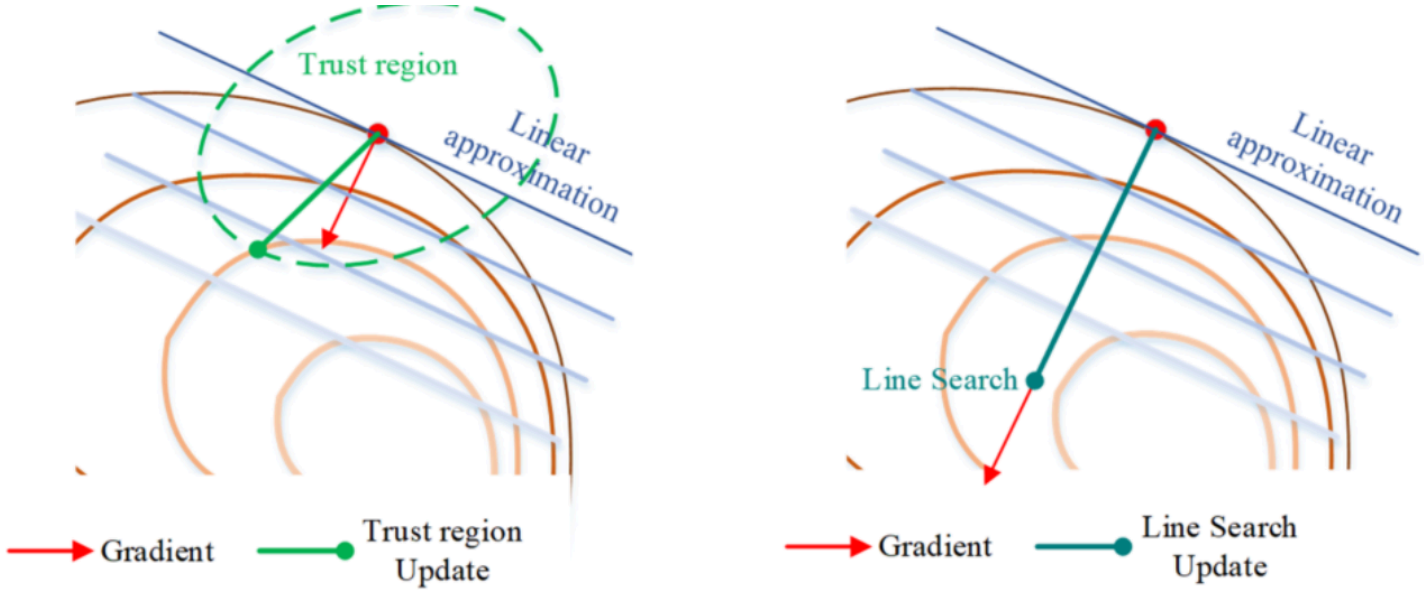
$$J_{\text{Actor}}(\theta+\Delta\theta)-J_{\text{Actor}}(\theta)\leq 0$$

再对左式进行泰勒展开，展开到一次项，可得：

$$J_{\text{Actor}}(\theta+\Delta\theta)-J_{\text{Actor}}(\theta)\approx(\frac{\partial J_{\text{Actor}}}{\partial\theta})^{\mathrm{T}}\Delta\theta\leq 0$$

$(\frac{\partial J_{\text{Actor}}}{\partial\theta})^{\mathrm{T}}\Delta\theta\leq 0$再接着转化为$\min_{\Delta\theta} g^{\mathrm{T}}\Delta\theta$即可。而对于可行性条件，也照样对于参数$\theta$进行一阶展开即可。

注意，上面只能确定更新的方向，而不能确定步长。确定一个合适的补偿有助于避免策略的快速变化和训练的不稳定。最naive的选择是一个固定步长。更高级的方法可以自动调节步长，比如信赖域或线搜索方法。



- **信赖域方法**：实际上是一种用来限制探索范围的不等式约束。最简单的形式如下：

  $$\Delta\theta^{\mathrm{T}}H\Delta\theta\leq\delta,$$

  其中$H\geq 0$是一个权重矩阵。Constrained policy optimization（CPO）实际上就是一种model-free的算法，它使用线性化的actor loss和信赖域形式的约束。

- **线搜索方法**：线搜索方法是在确定了更新方向滞后在这个方向上再求解一个优化问题来确定能够使得原始的actor loss的最优步长。其变式还包括：Armijo rule, limited line search and diminishing length。

9.5.1.2 转化为QP问题

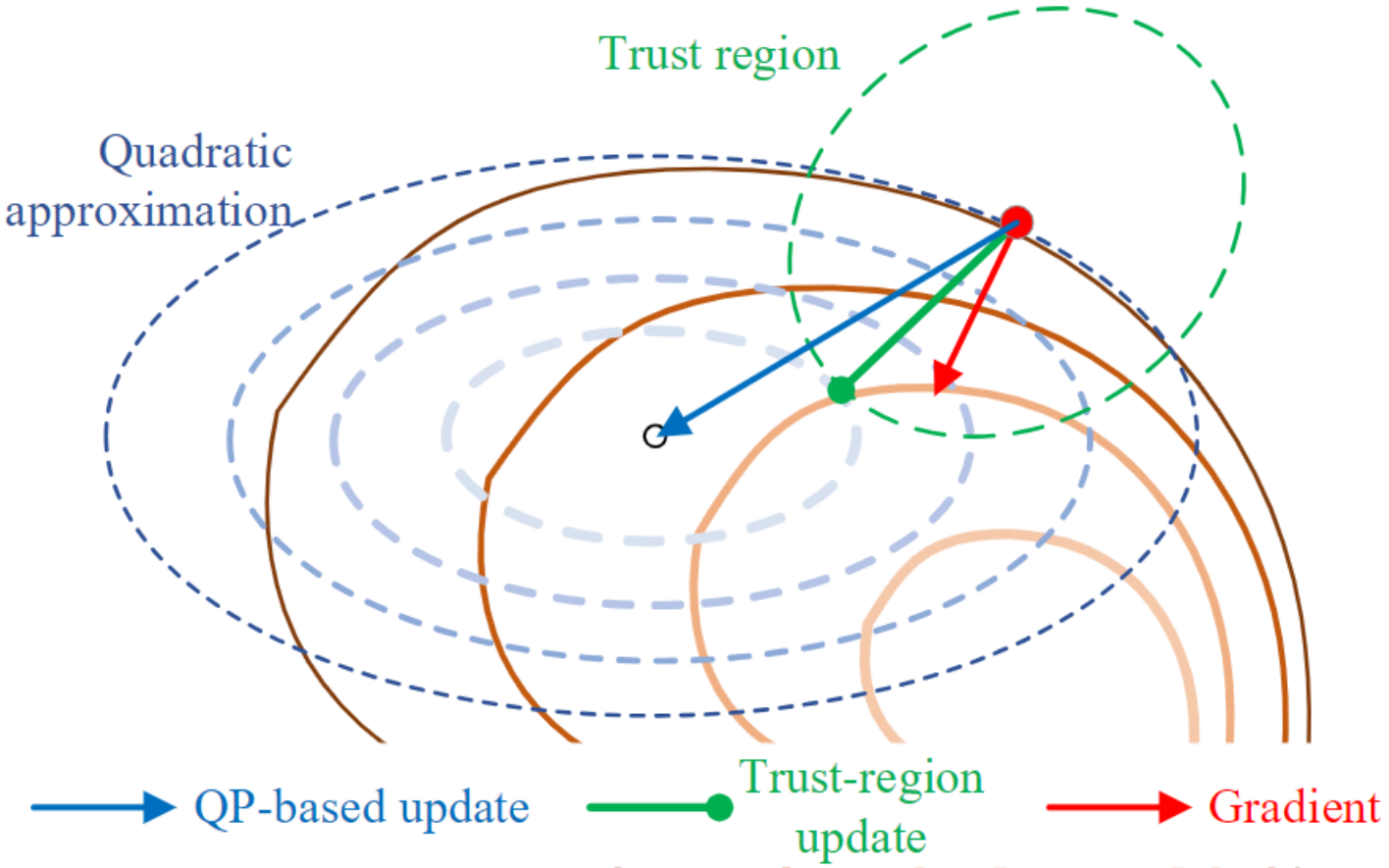
与一阶展开相比，二阶展开包含更多的梯度信息，能够加速带约束的RL/ADP问题的收敛。把带约束的PIM问题转化为一个QP问题的形式如下：

$$\begin{aligned}&\min_{\Delta\theta}\frac{1}{2}\Delta\theta^{\mathrm{T}}F\Delta\theta+g^{\mathrm{T}}\Delta\theta,\\&\text{subject to}\\&h(x')|_{\theta}+\nabla_{\theta}h(x')^{\mathrm{T}}\Delta\theta\leq 0,\end{aligned}$$

注意，这里只是对于下降性条件进行了二阶展开，可行性条件还是一阶展开。式中的g同LP那里的，F如下：

$$F=\frac{\partial^2 J_{\mathrm{Actor}}}{\partial\theta^2}=\frac{\partial^2\{l(x,u)+V(x')\}}{\partial\theta^2}$$

与LP那里不同，QP问题的解既可以确定更新方向又可确定更新步长。但是，直接使用QP求解的一个缺点是当步长太大时二阶近似时会有很大的误差。为此，可以使用信赖域方法来避免过大的策略更新。



另外，在基于QP的FDD中对于可行性的判断能力也失去了。这是因为这里也与LP那里进行了相同的线性化。

9.5.2 带约束的LP优化

本小节就来讲一讲具体怎样转化为LP问题。首先需要说明的是，FDD条件在每个带约束的PIM只被线性化一次，线性化与RL/ADP的主循环同步。这样做的好处是总的训练效率比多次转换的效率更高。在深度学习中，一阶优化方法更适合训练带约束的且使用神经网络的RL/ADP问题，比如著名的Adam算法。

为了更好地理解怎么使用LP来求解带约束的RL/ADP问题，下面介绍两种实用的带约束的LP算法：

- **Lagrange-based LP algorithm**
- **Gradient projection algorithm**

Table 9.3 Instances of RL with first-order optimization

Algorithm	Guidelines	
	Updating direction	Updating length
Lagrange-based LP	Lagrange duality	Trust region
Gradient projection	Gradient projection	Constant/Diminishing

9.5.2.1 Lagrange-based LP算法

这种方法的特点是使用拉格朗日对偶来求解LP问题和用于限制更新步长的信赖域。使用拉格朗日对偶的方法有一些好处，如对于初始的坏的猜测有较低的敏感性、以及在局部区域内较快的收敛性。而信赖域方法的引入则可以限制更新的幅度来避免可能的训练不稳定。通常使用KL散度来建模信赖域：

$$D_{\mathrm{KL}}(\pi(\theta), \pi(\theta^{\mathrm{new}})) \approx \frac{1}{2}\Delta\theta^{\mathrm{T}} H \Delta\theta \leq \delta,$$

这里的δ是一个限制步长的超参数，H是一个来自于KL散度的Hessian矩阵。

转化为LP问题并引入了信赖域约束的拉格朗日函数如下：

$$\max_{\mu,\lambda\geq 0} \min_{\Delta\theta} g^{\mathrm{T}}\Delta\theta + \lambda(h(x')|_\theta + \nabla_\theta h(x')^{\mathrm{T}}\Delta\theta) + \mu\left(\frac{1}{2}\Delta\theta^{\mathrm{T}} H \Delta\theta - \delta\right),$$

这里的$\lambda \geq 0$是线性化之后的约束的拉格朗日乘子，$\mu \geq 0$是信赖域的拉格朗日乘子。

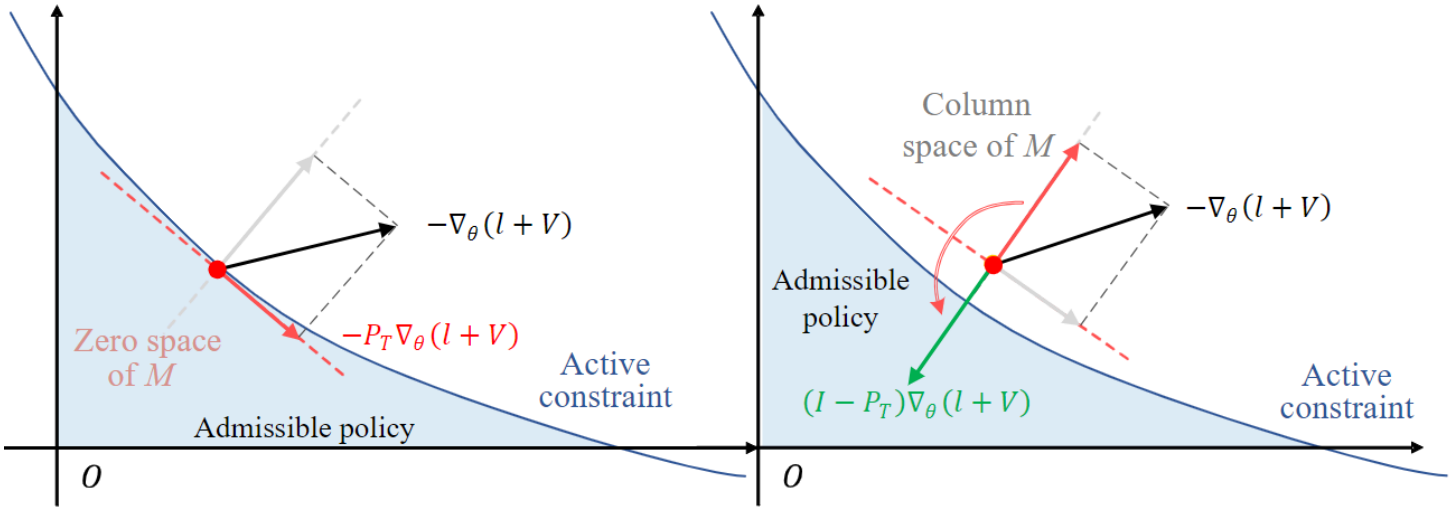
这个方法的一个好处是可以解析的给出其最优解：

$$\Delta\theta^* = -\frac{H^{-1}(g + \lambda^* \nabla_\theta h)}{\mu^*},$$

其中λ^*和μ^*是拉格朗日乘子的最优解。正如之前说的，传统的基于拉格朗日乘子法的优化需要引入一个计算上很低效的内循环（固定λ更新θ），而这里因为可以解析的给出解而避免这个问题。

9.5.2.2 Gradient projection算法

梯度投影算法是一个用于求解高维LP的有效方法。



(a) Projection of feasible policy　　(b) Projection of infeasible policy

正如上图中左图所示，梯度投影算法利用一个正定矩阵P_T来把一个未加约束的梯度投影到一个带约束的可行方向上来满足可行性条件。梯度投影在每次常规的策略梯度计算后单独计算，因此也可以被很容易的集成到其它的RL/ADP算法中。

一种经典的梯度投影算法是Rosen梯度投影。这个方法把梯度投影到约束的切平面（切线）上。梯度矩阵P_T可以通过下面的公式计算：

$$P_T = I - M^{\mathrm{T}}(MM^{\mathrm{T}})^{-1}M, \quad M = \nabla_\theta h^{\mathrm{T}},$$

这里的M是约束的梯度向量。这里投影矩阵P_T还有一个有趣的性质，即尽管M的维度与约束数量有关，但是P_T的维度确是固定的。

通常来说，梯度投影算法适合求解具有稀疏线性约束的大规模问题。还有一个问题，就是为什么Rosen梯度投影算法能够从一个初始不可行的策略开始。正如上图中右图所示，P_T可以给出一个向可行区域移动的投影后的梯度。矩阵$I - P_T$可以把任何梯度向量投影到约束的法线的方向上，这也是最快进入可行区域的方向。因此这种方法可以从一个不可行的策略开始，通过梯度投影逐渐进入可行区域。

9.5.3 带约束的QP优化

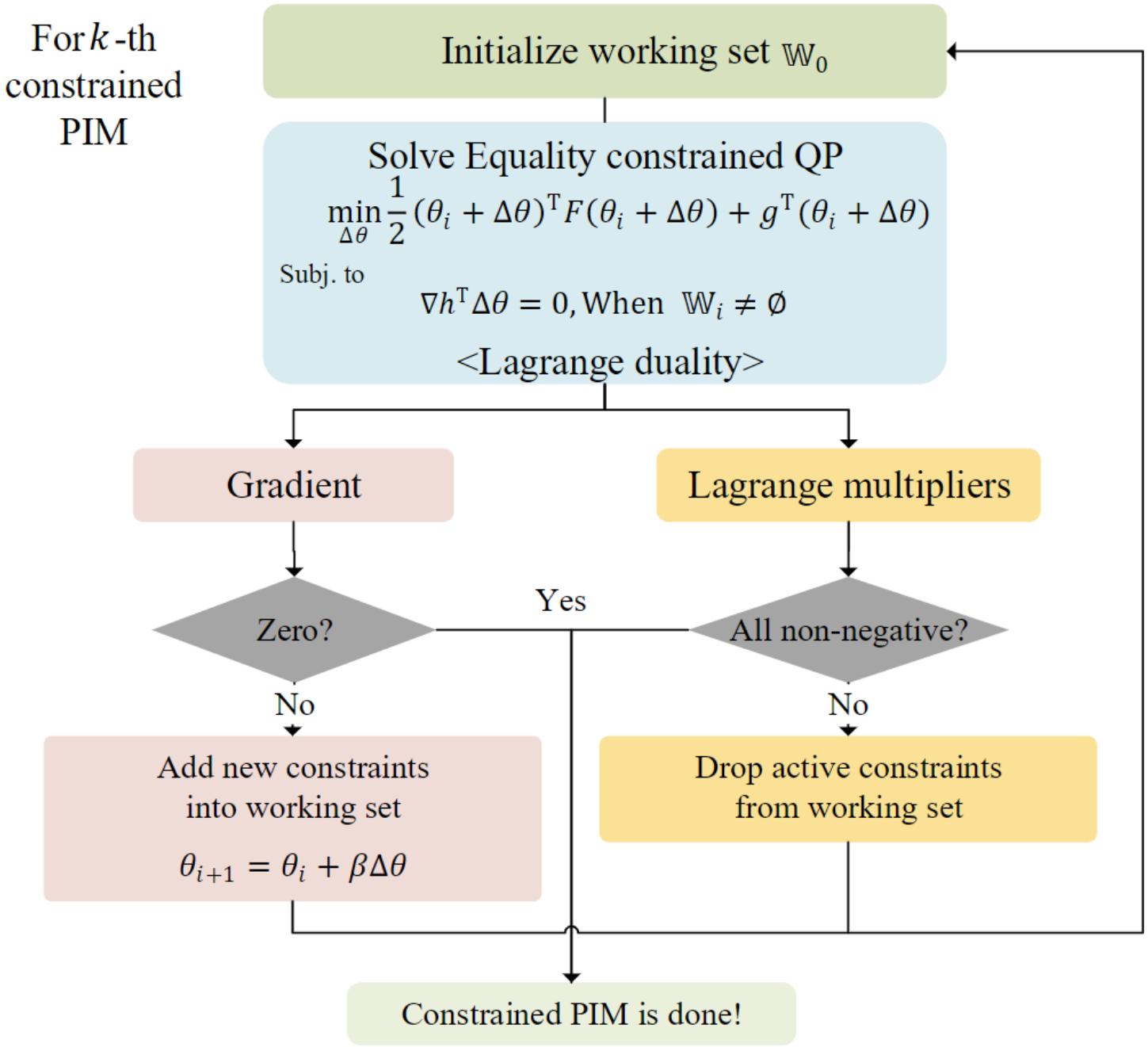
对于低维的任务，有两种带约束的QP优化方法：

- ** Active set QP算法**

- ** quasi-Newton算法**

当延伸到高维的任务时，Hessian矩阵的高昂的计算负担是一个主要的挑战。一些最近的研究使用近似的Hessian矩阵来减少计算负担。但是在大规模的策略搜索中，精确的Hessian矩阵估计被严重的限制了，尤其是当失去Hessian矩阵的稀疏性时。稀疏性的缺失使得矩阵求逆操作在基于神经网络的策略和值函数网络中计算效率很低甚至不可行。因此，带约束的QP优化在当今的大规模RL/ADP问题中并不是很流行。

9.5.3.1 Active set QP算法



如上图所示，Active set QP算法在每轮PIM迭代的时候主要分为两步：

- **求解一个等式约束的QP问题**：本步既给出了一个可行的下降方向，又给出了拉格朗日乘子。
- **通过不断添加和移除约束来refine工作集（working set）**：在本步，根据如下的经验规则来决定是否添加或移除约束：
  - 如果上一步给出的梯度非零，那么就把离梯度方向最近的约束添加到工作集中。
  - 如果存在负的拉格朗日乘子，那么就把对应的约束从工作集中移除。

  当第一步求出的梯度为零且且所有的拉格朗日乘子都是非负的时候，就可以停止迭代，找到了最优的策略。

很容易看出active set QP算法只能计算等式约束的QP问题，因为所有工作集中的不等式约束都必须被划分为active或者inactive的等式约束。这一性质也使得这种方法对于具有稀疏约束的低维约束计算很快。

9.5.3.2 quasi-Newton算法

因为计算Hessian矩阵的精确解计算量很大，因此quais-Newton算法通常使用近似的Hessian矩阵来代替以加速计算效率。而且这种方法对应的二阶优化器具有更鲁棒的学习行为，因此可以用来稳定强化学习算法。

在带约束的PIM中，需要计算的Hessian矩阵$\nabla_\theta^2 J_{\text{Actor}}$。但是我们用下式来近似：

$$\widehat{F}\Delta\theta = \nabla J_{\text{Actor}}|_{\theta+\Delta\theta} - \nabla J_{\text{Actor}}|_{\theta}$$

这里的$\widehat{F}$是近似的Hessian矩阵。

在一些递归的估计算法中，相邻两个Hessian矩阵之间的误差被强制保持在一个范围内来保持低秩特性。其它regularity手段也被用来保持Hessian矩阵的其它性质，比如symmetric rank-one（SR1）规则、Broyden-Fletcher-Goldfarb-Shanno（BFGS）规则。

9.6 Actor-Critic-Scenery（ACS）架构

之前章节介绍过的Actor-Critic框架作为当今最广为使用的强化学习框架，由Actor和Critic两部分构成。Actor即参数化的策略，Critic即参数化的值函数。这种架构有助于增强训练算法的稳定性并提升效率，被很多主流的算法所采用，如A2C、A3C、SAC、DSAC等。

而我们本节要介绍的ACS架构则是在Actor-Critic框架的基础上增加了一个Scenery模块，该模块用于计算训练的策略的可行的工作区域，即确定EFR。在这个架构中，Actor和Critic的功能不变，而Scenery模块则用阿里进行Region Identification。该架构的迭代分为三部分：PEV、PIM、RID（Region Identification）。

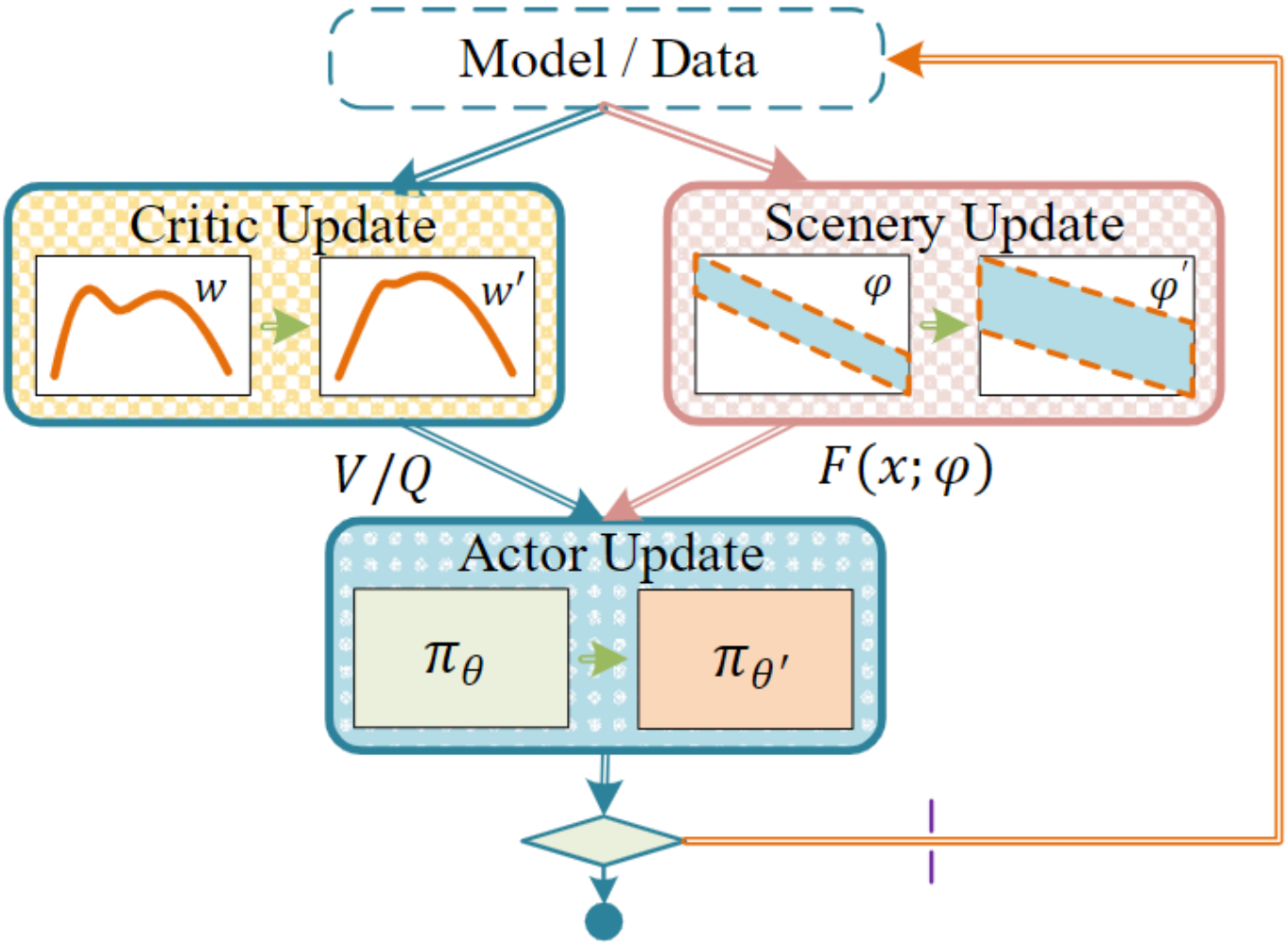
9.6.1 两种ACS架构

设计一个Scenery模块需要两个关键的步骤；

- **需要对于EFR有一个定量的描述**
- **根据怎样检验可行性设计一种Scenery优化算法**

构建完之后，Scenery更新就是去寻找一个完美的（perfect）的可行性函数，这个可行性函数对应的区域等于最大的EFR。 关于可行性函数的定义根据其检验可行性的机制的不同而大相径庭。其中一种叫做基于可达性的Scenery，另一种被称为基于可解性的Scenery。

- **基于可达性的Scenery**：

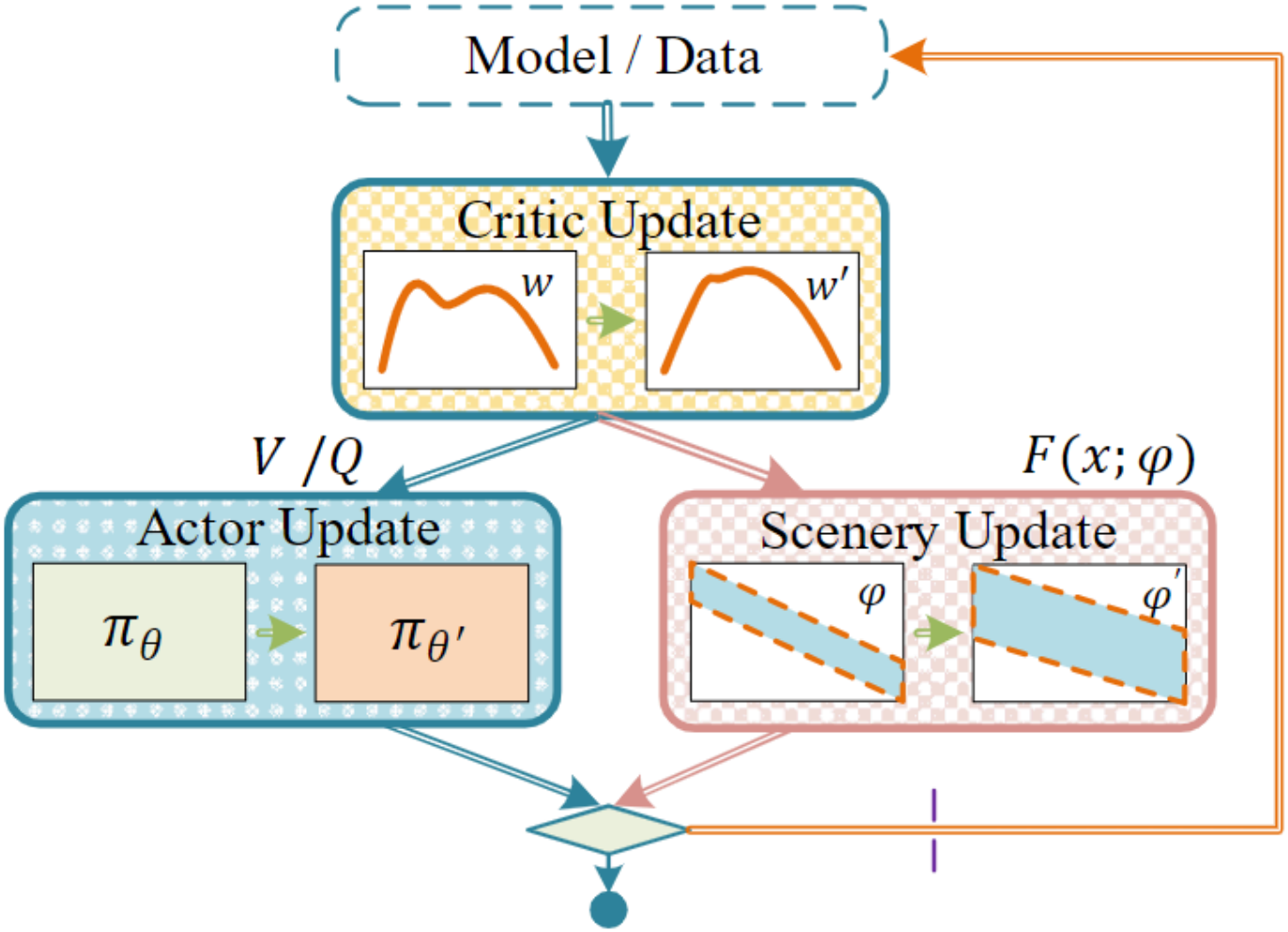


(a) Reachability-based scenery

基于可达性的Scenery可以被视为一种与风险评估有关的特殊的Critic，而它的可行性函数来源于Hamilton-Jacobi可达性分析，被称为基于可达性的可行性函数。它描述了从一个给定的状态出发最危险的约束值。一个状态的可达性值越大，意味着这个状态更危险，在未来更有可能发生约束违反。这样定义的好处是其完美的版本保持了最优性条件。它对应的可解性检验基于怎么评估对于每

个给定的状态的的风险等级。如这里的示意图所示，这里的Scenery组件与Critic组件是平行的，因为这里的风险评估与值函数评估是相似的。

- **基于可解性的Scenery**：



(b) Solvability-based scenery

基于可解性的Scenery则是一种类似于与Actor进行对抗的组件，在上图中表现为与Actor平行。它的可行性函数定义为每次的带约束的PIM是否具有一个可行解，因此也被称为基于可解性的可行性函数。该函数通常基于带约束优化问题的互补松弛条件。互补松弛条件的值是一个有效的区域是否可行的有效指示。这个值为0的时候就表示是一个可行解，如果是一个正实数的话就是不可行的。与它相关的可行性检验取决于对于可能的约束违反的惩罚程度。如这里的图所示，这里的Scenery组件实际上是与Actor组件一同更新的，因为他们实际上求解的是同一个优化问题。

实际上，这两种Scenery组件也可以工作在同一个ACS架构下，然而这可能会导致不可预测的策略表现，因此在大多数情况下，这两种Scenery组件我们只会选择其中一种。

EFR从数学上可定量定义如下：

$$\mathcal{X} = \{x | F(x) \leq 0\},$$

这里的$F(x)$是一个可行性函数，而$\mathcal{X}$是一个EFR。对应于最大的EFR的可行性函数被称为完美的可行性函数，任何策略在这个完美的可行性函数对应的区域内都必须是可行的。其定义如下：

$$\mathrm{X}_{\mathrm{Edls}}=\{x|F^{*}(x)\leq 0\},$$

这里的$F^{*}(x)$是一个完美的可行性函数。

9.6.2 基于可达性的Scenery

Hamilton-Jacobi可达性分析是一种用来描述带约束的动态系统的可扩展的安全验证方法。它具有兼容非线性系统、对于bounded disturbances的正式处理以及已经发展得很好的数值工具。给定任意一个约束函数，HJ可达性分析的基本是要计算一个可达集。

> **可达集**：由危险的状态组成，实际上是一个不可行区域。

由上述定义可以看出，可达集的补集就是一个EFR。

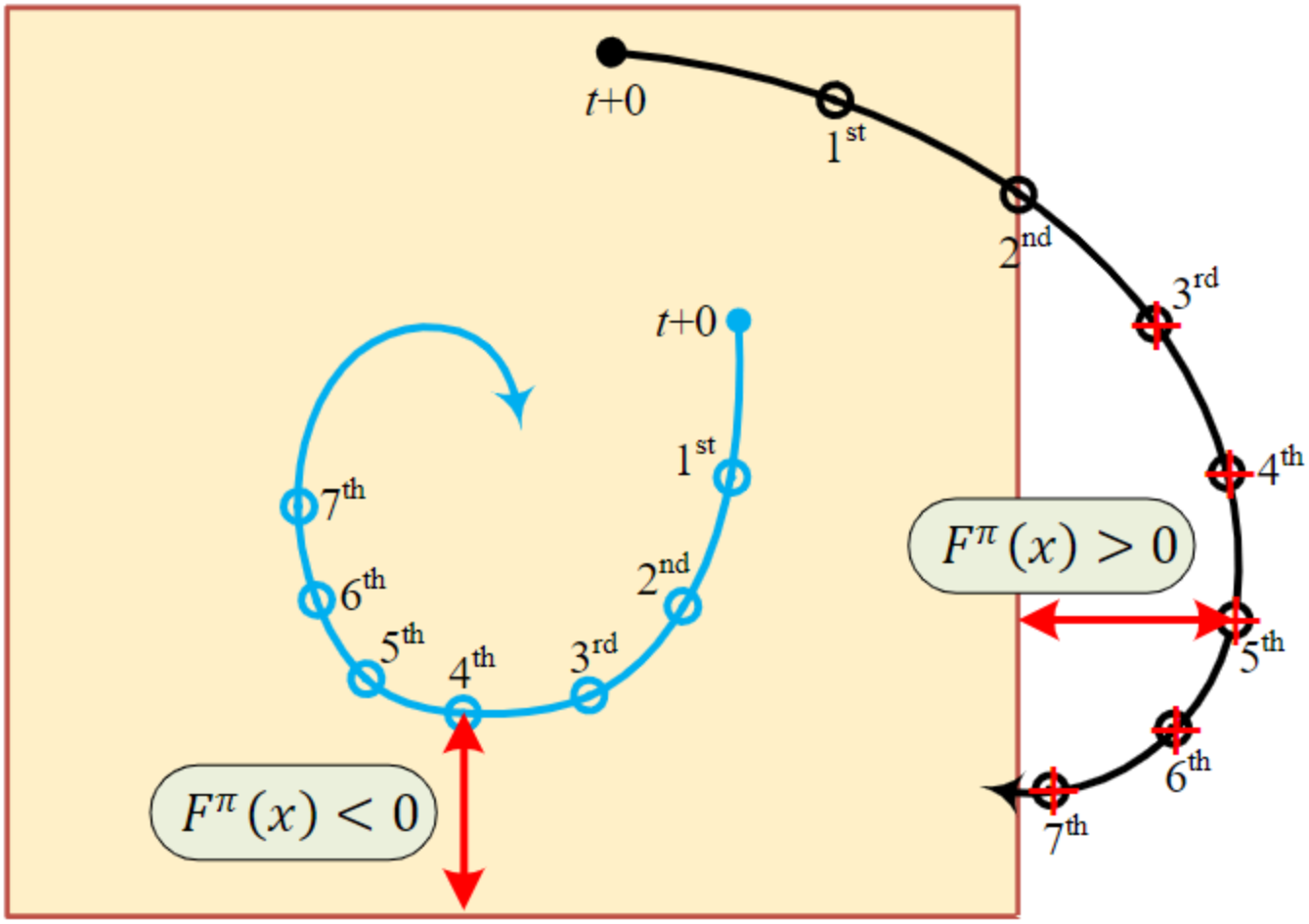
但是，传统的HJ可达性分析只检查带约束的动态系统的风险，而不关注怎样找到一个最优策略。作为对于RL的延伸，ACS架构被用于处理两个相互关联的要求：安全保证和追求最优性。在基于可达性的ACS中，尽管寻找最优性是第一追求，但是EFR也必须同时被确定，只有这样才能保证学得的策略的有效性。

9.6.2.1 基于可达性的可行性函数

让我们以一个确定性的离散时间的动态系统。系统是可控的，也就是说存在一个策略可以把一个状态转移到任何其它状态点。我们用记号$x_{t+i}^{\pi},i=0,1,2,\cdots,\infty$来描述在策略$\pi$下从任意状态$x_t=x$出发的状态轨迹。基于可达性的可行性函数$F(x)$定义为：

$$F^{\pi}(x)\overset{def}{=}\max_{i}h(x_{t+i}^{\pi}),i\in\{0,1,2,\cdots,\infty\},$$

这里的$F^{\pi}(x)$表示的是在某个具体策略π下的可行性函数。原本的约束$h(x)\leq 0$代表当前状态是安全的，而$F^{\pi}(x)$则描述了当前状态的风险等级，即在从当前点出发的状态轨迹上最危险的点的h值。当$F^{\pi}(x)\leq 0$时，表示当前状态是在未来绝对安全的，当$F^{\pi}(x)>0$时，表明在未来一定会发生约束违反。显然，$F^{\pi}(x)$越小，表示当前状态越安全。



因为可以描述任意策略的可行性函数，因此上述可行性函数提供了搜索一个风险更小的策略的方向指导。而在所有可能的策略中，有一个策略可以使得$F^\pi(x)$最小：

$$F^*(x) \overset{def}{=} \min_i \max_i h(x_{t+i}^\pi), i \in \{0, 1, 2, \cdots, \infty\},$$

此时的$F^*(x)$被称为完美的可行性函数。它是由内外两个相反的最值问题组成的，内部的最值问题是在所有可能的状态轨迹上找到最危险的点，而外部的最值问题是在所有可能的策略上找到最安全的策略。像任何多阶段优化一样，它拥有一个从贝尔曼最优性原理来的最优性条件。

9.6.2.2 风险贝尔曼方程

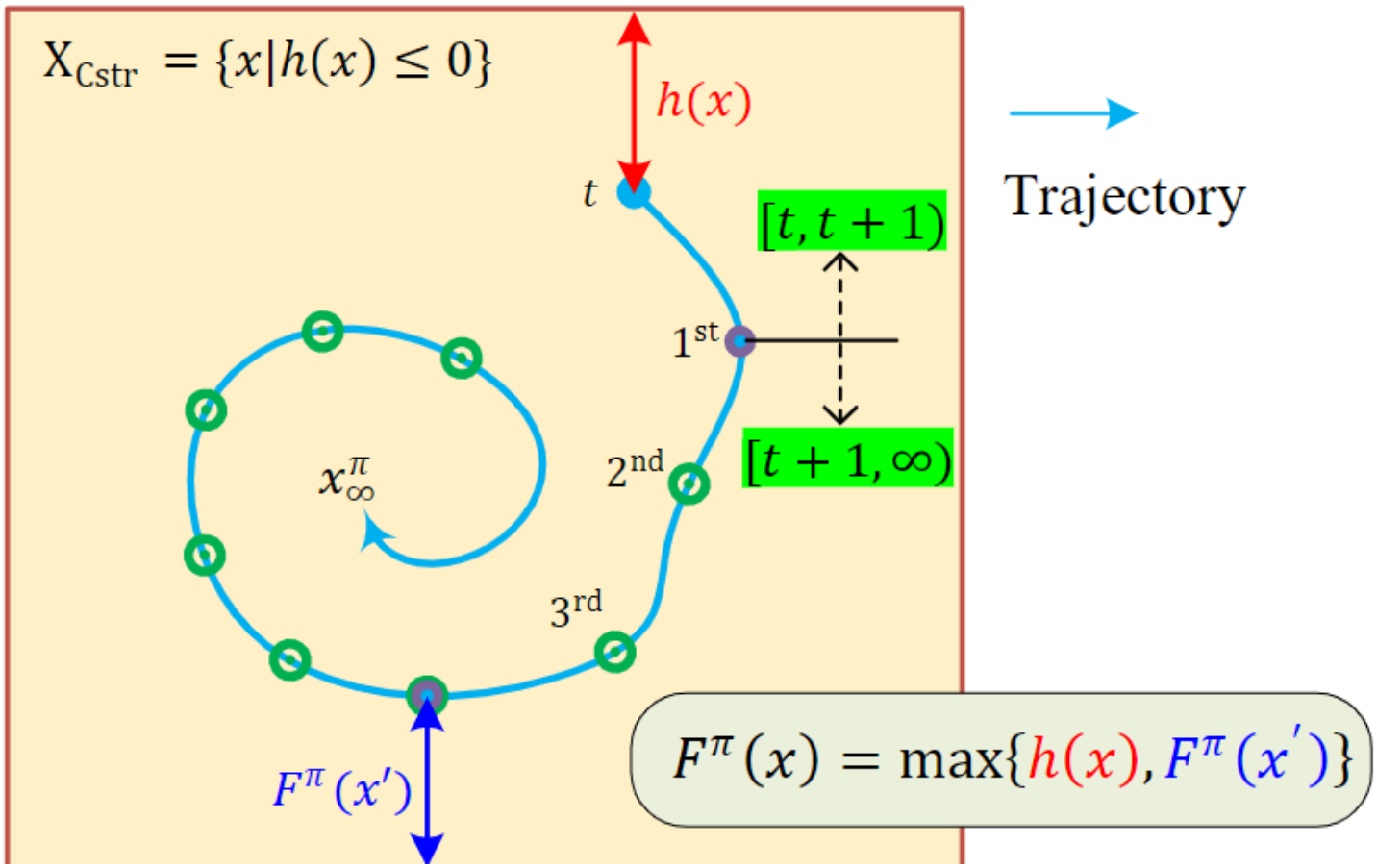
基于可达性的可行性函数有内在的递归结构，因此它也有一个与值函数的self-consistency条件类似的条件。因此，与之相关的Scenery优化问题也保有一个最优性条件，称为**风险贝尔曼方程**：

$$F^*(x) = \min_u \max\{h(x), F^*(x')\}.$$

简单证明如下：

$$
\begin{aligned}
F^*(x_t) &\overset{\text{def}}{=} \min_{\{u_t, u_{t+1}, \cdots\}} \max_i \{h(x_{t+i})\}, i \in \{0, 1, 2, \cdots, \infty\} \\
&= \min_{\{u_t, u_{t+1}, \cdots\}} \max \left\{ h(x_t), \max_i h(x_{t+i}) \right\}, i \in \{1, 2, \cdots, \infty\} \\
&= \min_{u_t} \max \left\{ h(x_t), \min_{\{u_{t+1}, u_{t+2}, \cdots\}} \max_i h(x_{t+i}) \right\} \\
&= \min_{u_t} \max \{ h(x_t), F^*(x_{t+1}) \}.
\end{aligned}
$$

证明过程的图解如下图所示：



正如之前讲过的，当不再追求最优性时，贝尔曼方程就退化为self-consistency条件。self-consistency条件实际上是一个可分离的多阶段优化控制问题的内在性质，类比值函数的self-consistency条件与值函数的贝尔曼方程，可得可行性函数的self-consistency条件：

$$
F^\pi(x) = \max\{h(x), F^\pi(x')\}.
$$

求解风险贝尔曼方程的方法与求解值函数的贝尔曼方程的方法是一样的，可以使用动态规划（表格形式or参数化形式）或其他方法。

还有一点值的指出，即根据贝尔曼最优性原理，最安全的状态轨迹上截取的任意一段也是最安全的。

9.6.2.3 基于模型的可达性ACS算法

首先，我们来给可行性函数进行参数化：

$$F(x;\varphi) \cong F(x),$$

这里的φ是可行性函数的参数。然后，通过风险贝尔曼方程和常规贝尔曼方程可以分别求得策略搜索和可行区域辨识。这里需要说明的是，不加上传统的贝尔曼方程，只凭借风险贝尔曼方程，是无法同时确定最优策略和其可行的工作区域的。这也是对于为什么基于可达性的ACS算法的Scenery更新与Critic更新是同时的，即把基于可达性的Scenery组件看成一种可以评估风险的特殊reward系统。可以把Scenery和Critic看成是两套独立的评估体系，一套负责策略评估（输出是策略的最优性程度），一套负责安全评估（输出是违反约束的风险）。在这样的观点下，Scenery的损失函数与Critic的损失函数非常相似，在这里我们选取L_2损失函数：

$$J_{\text{Scenery}} = \frac{1}{2}\left(\max\{h(x), \min_{u} F(x';\varphi)\} - F(x;\varphi) \right)^2,$$

上式中的$\max\{h(x), \min_u F(x';\varphi)\}$是根据风险贝尔曼方程来的，只不过这里加了$min$是因为此时还在探索的过程中。这个其实也可以类比Q-Learning中的TD-target。这样做的好处是**不需要在单独为Scenery组件再创建一个策略网络**，这样可以降低整体架构的复杂度。和Q-Learning还有一点相似，就是这样的设计只适用于离散的动作空间或者控制仿射系统，只有这样内层的最小化才有显式解，否则就需要使用数值优化的方式求解，而这样就会带来而外的计算复杂度。

下面来看看怎么计算上述损失的梯度。为了计算方便，我们可以使用之前章节讲过的semi-gradient方法来降低计算复杂度，不对target部分求导，只对于$F(x;\varphi)$求导：

$$\begin{gathered}
\nabla_{\varphi} J_{\text{Scenery}} = -\left(F^{\text{target}}(x) - F(x;\varphi)\right)\frac{\partial F(x;\varphi)}{\partial \varphi}, \\
F^{\text{target}}(x) \overset{\text{def}}{=} \max\{h(x), \min_{u} F(x';\varphi)\}.
\end{gathered}$$

接下来来修改PIM以适应Scenery组件。在传统的带约束的PIM中，我们给的约束都是$x' \in \mathcal{X}_{\text{Edls}}$，但是注意，我们实际上**并没有关于$\mathcal{X}_{\text{Edls}}$的信息**，但是等等，其实我们在第一步已经训练好了一个Scenery组件，所以我们可以用这个Scenery组件来代替$\mathcal{X}_{\text{Edls}}$：

$$\begin{gathered}
J_{\text{Actor}} = l(x,u) + V(x') \\
\text{s.t.} \\
F(x';\varphi) \leq 0\,.
\end{gathered}$$

这样修改的好处是这样可以与很多优化方法无缝衔接，如罚函数法、拉格朗日法、FDD法。在本节将以外点罚函数法来演示，为了达到良好的效果需要仔细选择里面的惩罚项参数，太弱则无法保证可行性，太强则会导致学到的策略不是最优的。具体算法如下：

Algorithm 9-1: Model-based ACS with reachability-based scenery

Hyperparameters: critic learning rate α, actor learning rate β, scenery learning rate ζ, maximum batch size M, penalty coefficient ρ

Initialization: state-value function $V(x;w)$, policy function $\pi(x;\theta)$, feasibility function $F(x;\varphi)$

Repeat

(1) Use environment model

Initialize memory buffer $\mathcal{D} \leftarrow \emptyset$

Repeat M times environment reset

$x \sim d_{\text{init}}(x)$

$u = \pi(x;\theta)$

$x' = f(x,u)$

Compute $\partial V/\partial w$, $\partial l/\partial u$, $\partial V/\partial x$, $\partial f^{\mathrm{T}}/\partial u$, $\partial h/\partial x$, $\partial \pi^{\mathrm{T}}/\partial \theta$, $\partial F/\partial \varphi$, $\partial F/\partial x$

$$\mathcal{D} \leftarrow \mathcal{D} \cup \left\{\left(l, V, \frac{\partial V}{\partial w}, \frac{\partial l}{\partial u}, \frac{\partial V}{\partial x}, \frac{\partial f^{\mathrm{T}}}{\partial u}, \frac{\partial h}{\partial x}, \frac{\partial \pi^{\mathrm{T}}}{\partial \theta}, \frac{\partial F}{\partial \varphi}, \frac{\partial F}{\partial x}\right)\right\}$$

End

(2) Critic update

$$\nabla_w J_{\text{Critic}} \leftarrow -\frac{1}{M}\sum_{\mathcal{D}}\big(l(x,u) + V(x';w) - V(x;w)\big)\frac{\partial V(x;w)}{\partial w}$$

$$w \leftarrow w - \alpha \cdot \nabla_w J_{\text{Critic}}$$

(3) Scenery update

$$\nabla_\varphi J_{\text{Scenery}} \leftarrow -\frac{1}{M}\sum_{\mathcal{D}}\left\{\max\left\{h(x), \min_u F(x';\varphi)\right\} - F(x;\varphi)\right\}\frac{\partial F(x;\varphi)}{\partial \varphi}$$

$$\varphi \leftarrow \varphi - \zeta \cdot \nabla_\varphi J_{\text{Scenery}}$$

(4) Actor update

$$\nabla_\theta J_{\text{Actor}} \leftarrow \frac{1}{M}\sum_{\mathcal{D}}\frac{\partial \pi^{\mathrm{T}}(x;\theta)}{\partial \theta}\left(\frac{\partial l}{\partial u} + \frac{\partial f^{\mathrm{T}}}{\partial u}\frac{\partial V(x')}{\partial x'} + \rho\frac{\partial f^{\mathrm{T}}}{\partial u}\frac{\partial \max\{F(x'),0\}}{\partial x'}\right)$$

$$\theta \leftarrow \theta - \beta \cdot \nabla_\theta J_{\text{Actor}}$$

End

- Actor Update中为什么有一个$\max\{F(x'),0\}$，按照上面的问题构建不是应该只是$F(x')$吗？我的理解是这样处理可以使得只有在违反约束（$F(x')>0$）的时候才会对Actor的更新产生影响，而在不违反约束的时候，即使$F(x')<0$也不会对Actor的更新产生影响。

总结一下，基于可达性的Scenery可以学到最大的EFR，而其他诸如Control Barrier Function（CBF）则不保证学到最大的EFR。而且基于可达性的ACS算法还对于Model Mismatch和外界扰动具有良好的鲁棒性。然而，其缺点也存在，即当动作空间不是离散或者控制仿射系统时，需要使用数值优化的方法来求解，实际上需要设置两个策略网络，一个常规的，另一个策略用于评估风险来追求安全。这样会导致整体架构的复杂度增加，且容易出现不稳定的情况。

9.6.3 基于可解性的Scenery

检查可解性主要是为了确定可能的约束违反，而这是通过检查带约束的PIM是否有可行解实现的。在基于可解性的ACS架构中，Scenery组件和Actor组件共用一个优化问题，它们的参数更新过程表现为两个互相对抗的过程：Actor追求更好的策略而Scenery追求更严格的约束满足。这个对抗的过程也表明了可行性的保证是以一定程度的策略最优性为代价的。现有的研究主要关注怎么在约束下找到一个最优策略，但是没有考虑到可行域有多大。但是正如之前所说的，一个有用的策略必须工作在一个已知的可行域内，因此准确的确定可行域是非常重要的与追求最优性同等重要。这节将介绍基于可解性的可行性函数的定义以及怎么设计一个基于可解性的ACS算法来确定最大的EFR。

9.6.3.1 基于可解性的Scenery的基础

检验可行性的关键是检验每个带约束的PIM是否有可行解。在大多数带约束的RL/ADP中这并不简单，因为这里面涉及的优化问题是高度非线性和非凸的，必须采用数值优化方法来找到最优解。因此，怎么检查可解性取决于使用的具体优化方法：

- **罚函数法**：选择惩罚项（惩罚因子乘约束函数）本身，因为惩罚项具有随着约束违法程度增长而单调增长的性质。可以通过将该值与一个阈值比较来给出结论。
- **拉格朗日乘子法**：有两种选择：
  - **拉格朗日乘子**：可以看成是一种特殊的惩罚因子，当约束违反将要发生时，会自动增大来惩罚那些将要超出边界的状态。因此可以根据拉格朗日乘子的大小来判定是否可解，与一个固定的阈值比较即可。
  - **互补松弛**：另一种更好的办法是通过检验互补松弛条件来检验。互补松弛条件是拉格朗日乘子与约束函数的乘积。其取值与判断如下：
    - **等于0**：说明拉格朗日乘子为0（找到了最优解且没有活跃的约束）或约束为0（特定的约束活跃且被满足）。
    - **不等于0（一个正数）**：不存在可行解。

9.6.3.2 基于可解性的可行性函数

之前构建基于可达性的PIM时，我们把$x' \in \mathcal{X}_{\text{Edls}}$转化为$F(x';\varphi) \leq 0$。那么在这里能不能也如法炮制呢？不太行。因为这里Actor和Scenery是共享一个优化问题的，如果这样做很容易导致在与Actor竞争的时候出现自激导致的不稳定。因此，在构建基于可解性的PIM问题时，需要依靠更加具有一致性且稳定的约束条件，比如之前讲过的one-step pointwise约束和one-step barrier约束，它们2提供了相对较松、固定的约束，计算负担也较小。相比来说，one-step barrier约束对于约束违反更加敏感，因此与one-step pointwise约束相比，使用其得到的可行域相对会扩大一点。但是，下面为了叙述方便，我们还是用one-step pointwise约束来构建基于可解性的PIM问题。

首先，我们使用函数近似来参数化拉格朗日乘子，并依次给出拉格朗日函数：

$$L(\theta,\varphi) = l(x,u) + V(x') + \lambda(x;\varphi)h(x'),$$

这里的$\lambda(x;\varphi)$是拉格朗日乘子的参数化形式。如前所述，这个损失函数同时用来更新Actor和Scenery，Actor旨在寻找最优策略，而Scenery则是寻找完美的可行性函数。再由拉格朗日对偶，可以得到如下的对偶优化问题：

$$\max_{\varphi} \min_{\theta} \{L(\theta,\varphi)\}.$$

在对偶问题中，拉格朗日乘子必须非负。这可以通过一些神经网络的技巧来实现，比如在网络的输出层接一些非负的激活函数。那么**如果上述的对偶问题在区域$\mathcal{X}$内可解，那么拉格朗日乘子对应的互补松弛条件应该为0**：

$$\lambda(x;\varphi)h(x') = 0, \forall x \in \mathcal{X}.$$

如果互补松弛条件不为0，则不管采取什么样的策略都无法避免违反约束。当互补松弛条件不为0时，对应的拉格朗日乘子也会趋于无穷大。综上，可以定义基于可解性的可行性函数如下：

$$F(x;\varphi) = \lambda(x;\varphi)h(x').$$

在这样的定义下，φ既是拉格朗日乘子的参数，也是可行性函数的参数。定义完之后就可以通过检查$F(x;\varphi)$是否为0来判断是否有可行解:

Region	Solvable feasibility function
x is in EFR	$F(x;\varphi) = 0$
x is not in EFR	$F(x;\varphi) > 0$

在实际中因为有误差，所以可以选择一个很小的经验阈值来代替0。

可以通过之前讲过的dual descent的方式来求解上述对偶问题：

$$\theta^{\text{new}} \leftarrow \theta - \alpha_\theta \cdot \nabla_\theta L(\theta, \varphi),$$
$$\varphi^{\text{new}} \leftarrow \varphi + \alpha_\varphi \cdot \nabla_\varphi L(\theta, \varphi),$$

其中的$\alpha_\theta \geq 0$和$\alpha_\varphi \geq 0$。式中的两梯度分别为：

$$\nabla_\theta L(\theta, \varphi) = \frac{\partial u^{\mathrm{T}}}{\partial \theta}\left(\frac{\partial l}{\partial u} + \frac{\partial f^{\mathrm{T}}}{\partial u}\frac{\partial V(x')}{\partial x'} + \lambda \frac{\partial f^{\mathrm{T}}}{\partial u}\frac{\partial h(x')}{\partial x'}\right),$$
$$\nabla_\varphi L(\theta, \varphi) = h(x')\frac{\partial \lambda}{\partial \varphi}.$$

使用互补松弛条件来构建基于可解性的ACS算法的难点在于ACS架构中含有两个不想对抗的组件，因此超参数（尤其是学习率）选择不恰当会导致训练发散的情况。可以采取一些方法来解决这个问题，如降低学习率，使用target网络，以及降低参数更新的频率等。

9.6.3.3 可行区域的单调扩展性质

基于可解性的Scenery与Actor公熊一个损失函数，通过dual ascent算法交替寻找一个更好的策略和一个更大的可行域。大家可能会疑惑为什么这样一种对抗的更新方式可以扩大可行区域而不是仅仅保持甚至缩小这个可行区域呢？这一性质也被称为单调的区域扩展性质：

> **定理3：单调的区域扩展性质**
> 在基于可解性的ACS架构中并使用拉格朗日法求解时，每次新学到的EFR X^{k+1}都至少是前一个EFR X^k的超集，即：
>
> $$X^k \subseteq X^{k+1}.$$

下面证明如下。首先我们需要假设ACS算法在使用函数近似以及minimax优化时没有数值误差，并且环境模型是准确的知道的。在第k次迭代中，我们已经有一个在EFR X^k内的可行策略π^k。根据我们上面基于可解性的可行性函数的的定义，X_k表述为：

$$X_k = \{x | F_k(x) = 0\},$$

即当$x \in X_k$时，$F_k(x) = \lambda_k(x)h(x') = 0$。另外，对于任意的$x \in X_k$，状态值函数$V_k(x)$是一个有限的值。通过拉格朗日乘子法，可以把Actor的损失函数定义为：

$$J_{\text{Actor}}(\pi, \lambda) \overset{\text{def}}{=} l(x, \pi) + V\left(x'(\pi)\right) + \lambda(x)h(x'(\pi)),$$

在第k+1步之后，Actor和Scenery的参数更新如下：

$$[\pi_{k+1}, \lambda_{k+1}] = \arg\max_{\lambda}\min_{\pi}\{J_{\text{Actor}}(\pi_k, \lambda_k)\}, x \in \mathcal{X}_k.$$

上述优化问题我们至少可以选择$\pi_{k+1}=\pi_k$且$\lambda_{k+1}=\lambda_k$（这样必然满足上述优化问题的约束条件且至少不会变差）。如果这样选择的话，可以由互补松弛条件得到：

$$F_{k+1}(x)=\lambda_{k+1}(x)h(x'(\pi_{k+1}))=\lambda_k(x)h(x'(\pi_k))=0, x\in\mathcal{X}_k.$$

即X_k中的所有元素都在X_{k+1}中。因此，$X_k\subseteq X_{k+1}$。证毕。

更进一步我们还可以得到下面的结论：

> **定理4**：
>
> $$\mathrm{X}_{\mathrm{Edls}}=\mathrm{X}_{\infty}\overset{def}{=}\lim_{k\to\infty}\mathrm{X}_k.$$

这个定理可以由反证法来证明。让我们假设存在一个X_{Edls}的子集$\mathrm{X}_{\mathrm{sub}}\subseteq\mathrm{X}_{\mathrm{Edls}}$ 但是这个子集与X_∞没有交集：

$$\mathrm{X}_{\mathrm{sub}}\cap\mathrm{X}_{\infty}=\emptyset.$$

那么对于任何状态点$x\in\mathrm{X}_{\mathrm{sub}}$，这个状态点都能提为策略更新提供一个新的更新方向（因为有关这个点的信息没有被充分利用）。基于此，说明Scenery的更新还没有达到它的最优值，所以区域扩展还会继续，这与之前的假设（在$k\to\infty$时，X_k收敛到X_∞）矛盾。因此，我们的假设是错误的，即$\mathrm{X}_{\mathrm{Edls}}=\mathrm{X}_{\infty}$。证毕。

根据上面两个定理，我们可以得出下面的结论：策略更新和EFR扩展是同时进行的，且二者最优值是同时达到的。

9.6.3.4 基于模型的可解性ACS算法

基于可解性的ACS算法与传统的AC架构的算法很相像（因为Actor与Scenery共享一个损失函数，二者在PIM的时候通过dual ascent算法一同更新），因此基于可解性的ACS算法既有on-policy的，又有off-policy的；既有model-free的，又有model-based的；既有确定性策略的，又有随机策略的。这里展示on-policy版本的model-based的基于可解性的ACS算法：

Hyperparameters: critic learning rate α, actor learning rate β, scenery learning rate ζ, times of environment reset M

Initialization: state-value function $V(x;w)$, policy function $\pi(x;\theta)$, Lagrange multiplier function $\lambda(x;\varphi)$.

Repeat (indexed by k)

(1) Use environment model

Initialize memory buffer $\mathcal{D} \leftarrow \emptyset$

Repeat M times environment reset

$x \sim d_{\text{init}}(x)$

$u = \pi(x;\theta)$

$x' = f(x,u)$

Compute $\partial V/\partial w, \partial l/\partial u, \partial V/\partial x, \partial f^{\mathrm{T}}/\partial u,$
$\partial h/\partial x, \partial \pi^{\mathrm{T}}/\partial \theta, \partial F/\partial \varphi$

$$\mathcal{D} \leftarrow \mathcal{D} \cup \left\{\left(l, V, \frac{\partial V}{\partial w}, \frac{\partial l}{\partial u}, \frac{\partial V}{\partial x}, \frac{\partial f^{\mathrm{T}}}{\partial u}, \frac{\partial h}{\partial x}, \frac{\partial \pi^{\mathrm{T}}}{\partial \theta}, \frac{\partial F}{\partial \varphi}\right)\right\}$$

End

(2) Critic update

$$\nabla_w J_{\text{Critic}} \leftarrow -\frac{1}{M}\sum_{\mathcal{D}}\left(l(x,u) + V(x';w) - V(x;w)\right)\frac{\partial V(x;w)}{\partial w}$$

$$w \leftarrow w - \alpha \cdot \nabla_w J_{\text{Critic}}$$

(3) Actor update

$$\nabla_\theta J_{\text{Actor}} \leftarrow \frac{1}{M}\sum_{\mathcal{D}}\frac{\partial \pi^{\mathrm{T}}(x;\theta)}{\partial \theta}\left(\frac{\partial l}{\partial u} + \frac{\partial f^{\mathrm{T}}}{\partial u}\frac{\partial V(x')}{\partial x'} + \lambda(x)\frac{\partial f^{\mathrm{T}}}{\partial u}\frac{\partial h(x')}{\partial x'}\right)$$

$$\theta \leftarrow \theta - \beta \cdot \nabla_\theta J_{\text{Actor}}$$

(4) Scenery update

$$\nabla_\varphi J_{\text{Scenery}} \leftarrow \frac{1}{M}\sum_{\mathcal{D}} h(x')\frac{\partial \lambda(x;\varphi)}{\partial \varphi}$$

$$\varphi \leftarrow \varphi + \zeta \cdot \nabla_\varphi J_{\text{Scenery}}$$

End

让我们来总结一下上述基于可解性的ACS算法。首先，相比于整体以参数化可行性函数，我们只对于拉格朗日乘子进行参数化，这样也方便后续构建可行性检验的方法（通过互补松弛条件）。其次，Actor和Scenery在更新的时候石象湖竞争的，理想情况下可以同时收敛到最优策略和完美的可行性函数。最后，基于可解性的ACS算法比基于可达性的版本在计算上通常效率更高。

9.7 现实世界中的安全考量

在实际的控制任务中，严格的的安全保证是不可或缺的。这里安全保证被定义为避免对于硬约束的严重违反的能力。硬约束被定义为在任何情况都不能被违反的约束，比如自动驾驶系统中关于避免碰撞的约束。通过将硬约束加入OCP中，就可以通过求解这个OCP来来找到能保证安全的控制策略（该策略从它对应的EFR开始，永不违反硬约束）。本节将会讨论如何在现实世界的控制任务考虑安全保证。本章将会讲到两种为搜索安全策略设计的训练模式，并介绍它们各自对应的ACS算法（会同时介绍model-based和model-free的版本）。

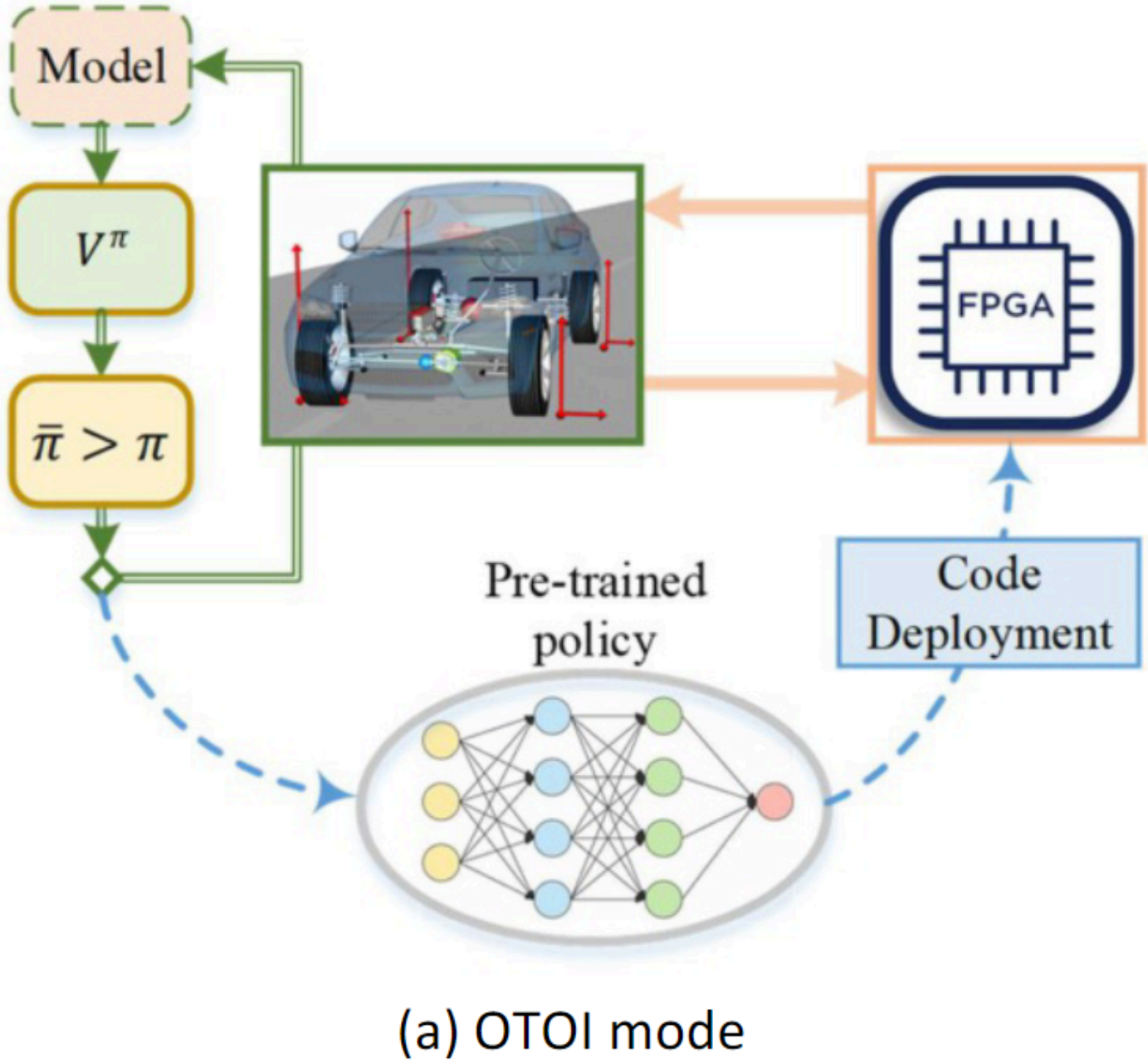
然而，在现实中，因为函数近似的误差、建模的不确定性、状态测量的噪声等问题，很难得到绝对安全的策略。为此，可设置安全盾（safety shield）机制作为最后的保险。

9.7.1 训练安全策略的两种基本模式

安全的策略既可以在真实环境又可以在虚拟环境中训练。这里所说的虚拟环境指的是一个高保真的的仿真模型。使用真实环境和虚拟环境的一大区别是在蓄念环境中安全不是必须的，即使违反了约束也没什么大不了的。这是因为在虚拟环境中违反了约束不会造成任何影响，把环境重置即可。

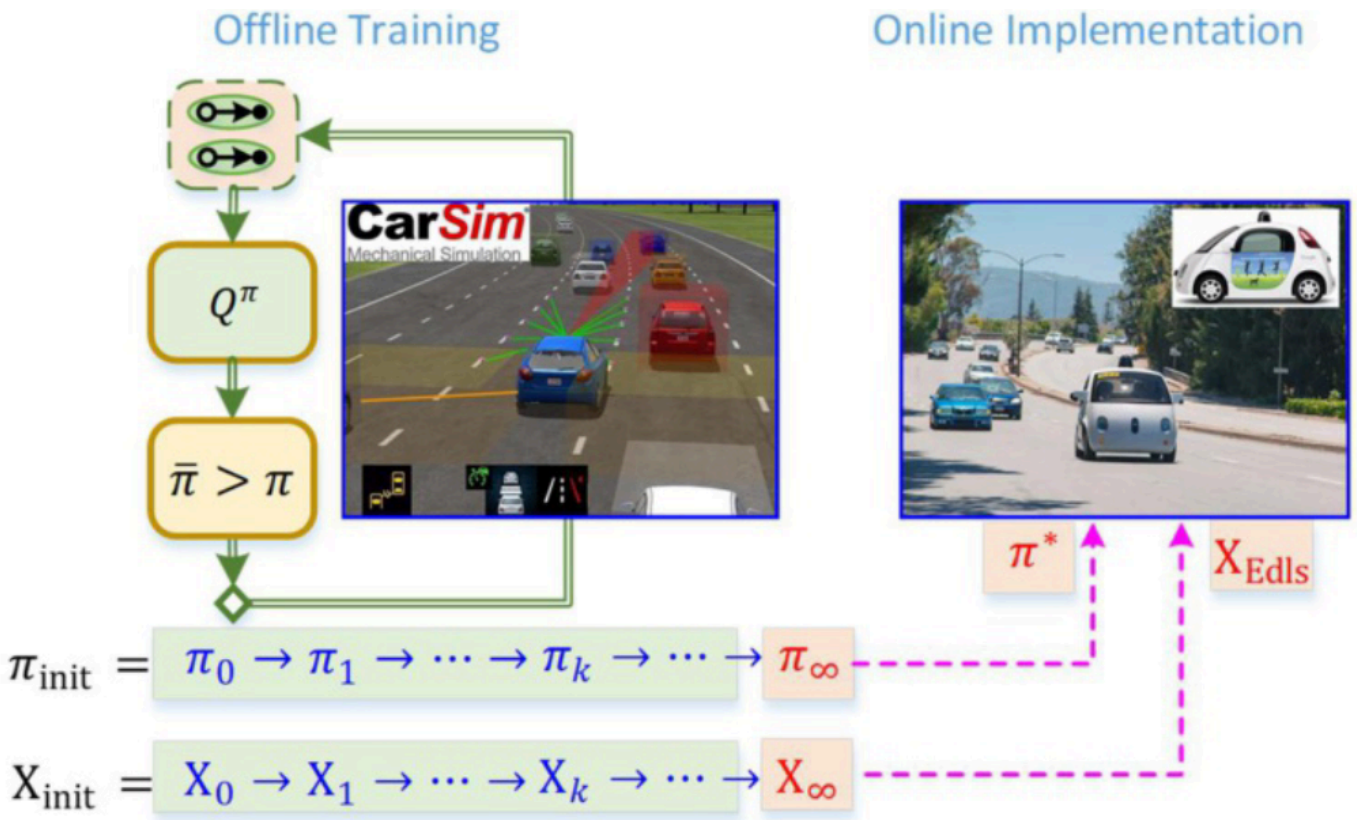
知道了训练的两种环境后，我们就可以来介绍下面的两种训练安全策略的模式。分别如下所示：

- **OTOI（Offline Training and Online Implementation）**：



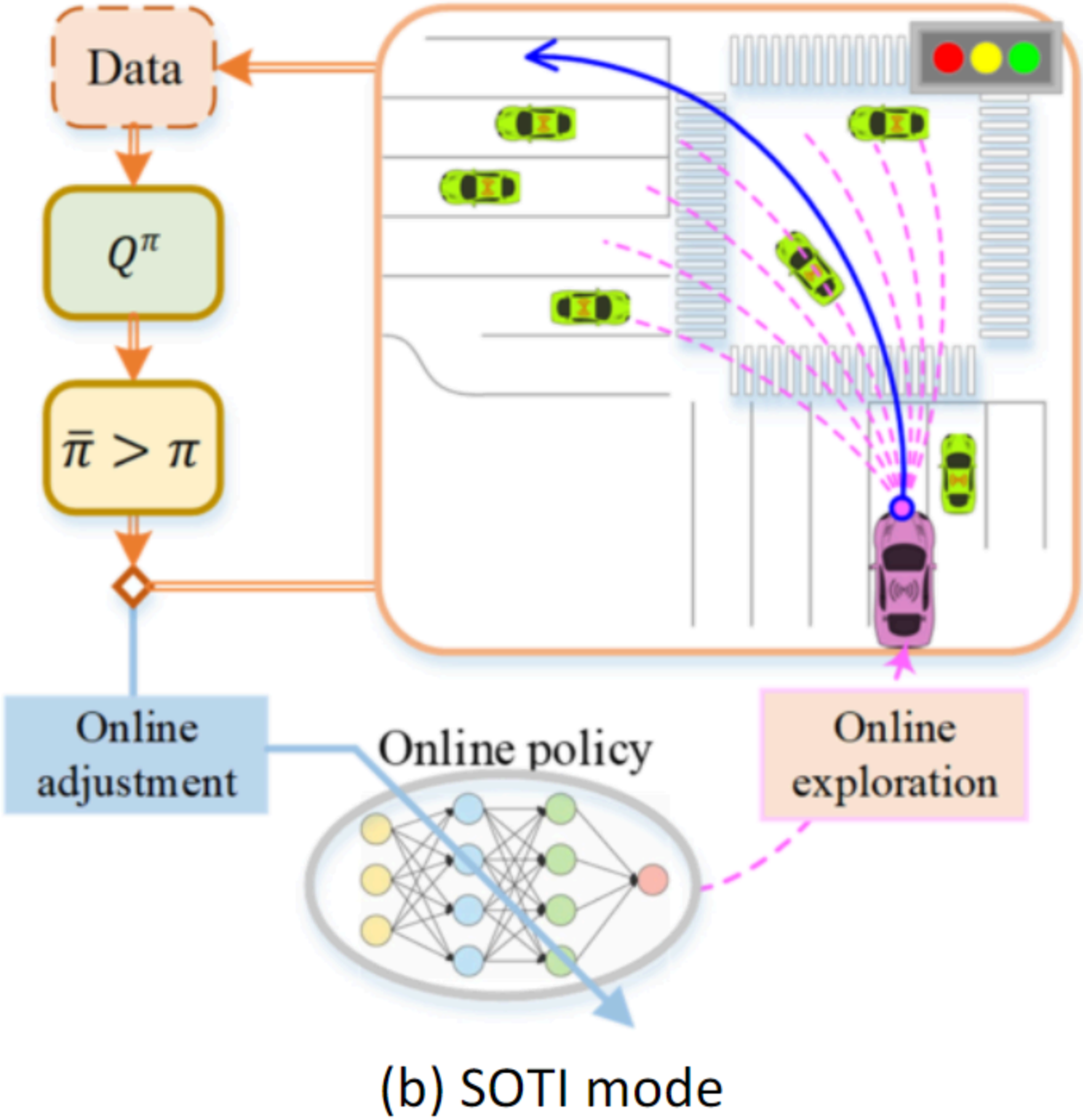
(a) OTOI mode

这种模式首先在虚拟环境中训练一个安全策略，然后将这个策略部署到真实环境中。这种方法与MPC一样可以进行Receding Horizon Control，但是又不必像MPC那样实时求解优化问题，因此计算效率较高。OTOI模式只有**在最后一步才获得一个安全的策略，中间的策略未必是安全的**。中间阶段对于策略的违反只是为了整体策略相更安全的方向移动提供了惩罚信号。



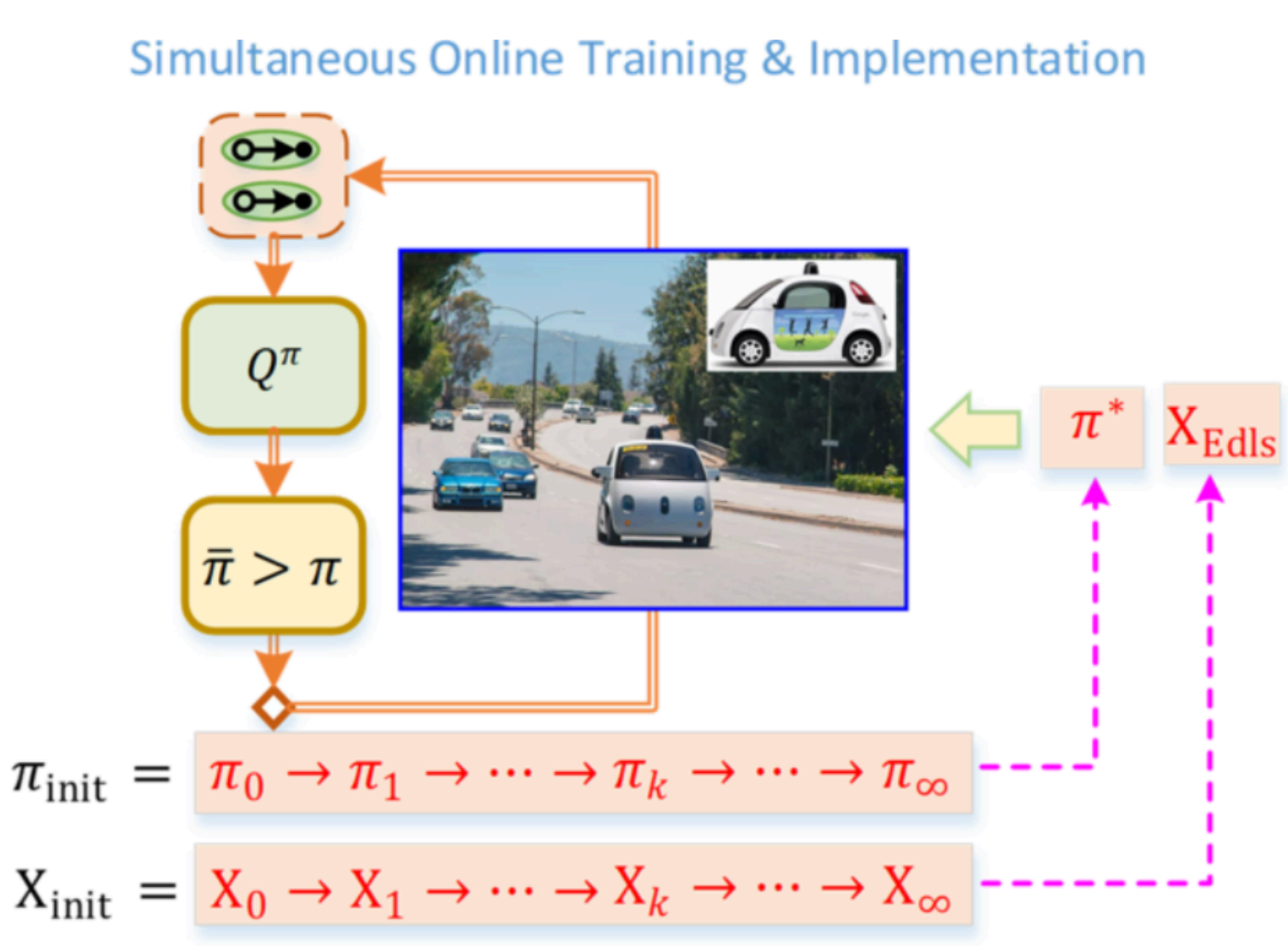
(a) OTOI mode

- **SOTI（Simultaneous Online Training and Implementation）**：



(b) SOTI mode

这种模式通过与真实环境交互学习一系列的安全策略。**每个中间策略都必须保证是安全的，且每个中间策略的工作区域都必须是已知的**。如本处的图示所示，一个使用这样的模式训练的自动驾驶车辆可以在真实道路上持续学习，同时保证安全，并不断提升自己的策略。



那么应该怎样选择该使用哪种策略呢？这取决于我们手边有的环境模型是不是完美的。完美的环境模型意味着仿真环境与真实环境之间没有误差（或者在很多控制任务中，它们使用的高保真仿真环境也可以被粗略的认为是一种完美的环境模型）。这时候就可以选择OTOI模式。否则，就必须选择SOTI模式。这时从环境中收集的经验可以被视为**对于不完美的环境模型的有力补充**。在这种情况下，需要设计一种混合的ACS Trainer来混合来自环境模型和收集到的经验。此时环境动力学一方面来自于模型另一方面来自于对真实环境的探索。具体可参见下表：

	OTOI	SOTI
Perfect model	Only the final policy needs to be safe	--
Imperfect model	--	Each intermediate policy must be safe for real environment interaction

还应该指出的是，即使对于SOTI模式，也需要一个环境模型，即使这个模型是不完美的。这是因为环境探索总需要从已知的区域延伸到未知的区域，如果没有一个事先的模型，纯靠试错的话很容易导致不可预测的风险行为。而如果由一致的模型，即使它不完美，也能够提供相大大的一部分指引，指导这种ACS Trainer来和环境安全的交互。

9.7.2 具有最终安全保证（OTOI模式下）的Model-free的ACS算法

在OTOI模式下，相比于model-based，model-free的ACS算法显然是一个更好的选择。这是因为model-free的ACS算法不需要一个准确的环境模型，而这通常是通过高保真度但不可微的仿真模型来实现的。因为我们没有确切的环境模型的表达式，隐刺1在model-based的方法中计算基于模型的梯度是非常困难的，应当取而代之的是使用model-free的方法中通过样本来计算的梯度。

首先，因为没有可解析的环境模型，所以需要训练一个风险评估函数（类似于动作值函数，是风险的累积“回报”，我们将其定义为状态和动作的函数）：

$$H(x,u;\eta)\cong h(x'),$$

其中的η是函数的参数。上式表明，输入一个状态和一个动作，风险评估函数的输出应该接近于下一个状态的约束函数的值（下文中记为风险信号c'）那么就可以通过优化下面的平方损失形式的目标函数来学习：

$$\min_{\eta} J_{\mathrm{Risk}}=\left(c'-H(x,u;\eta)\right)^2.$$

下面给出基于拉格朗日乘子法来设计dual ascent的model-free的ACS算法。下面给出的算法是基于可达性的：

Algorithm 9-3: Model-free ACS algorithm in the OTOI mode

Hyperparameters: critic learning rate α_{Critic}, risk learning rate α_{Risk}, actor learning rate β, scenery learning rate ζ, maximum batch size B.

Initialization: action-value function $Q(x,u;w)$, risk evaluation function $H(x,u;\eta)$, policy function $\pi(x;\theta)$, Lagrange multiplier function $\lambda(x;\varphi)$.

Repeat (indexed by k)

(1) Collect samples

$\mathcal{D} \leftarrow \emptyset$

$x_0 \sim d_{\text{init}}(x)$

For i in 0,1,2, … , $B-1$ or until termination

Apply $u_i = \pi(x_i;\theta)$ and then observe x_{i+1}, c_{i+1} and r_i

$\mathcal{D} \leftarrow \mathcal{D} \cup \{(x_i, u_i, r_i, x_{i+1}, c_{i+1})\}$

End

(2) Critic update

$$\nabla_w J_{\text{Critic}} \leftarrow -\frac{1}{|\mathcal{D}|}\sum_{\mathcal{D}}\big(r+\gamma Q(x',u';w)-Q(x,u;w)\big)\nabla_w Q(x,u;w)$$

$$\nabla_\eta J_{\text{Risk}} \leftarrow -\frac{1}{|\mathcal{D}|}\sum_{\mathcal{D}}\big(c'-H(x,u;\eta)\big)\nabla_\eta H(x,u;\eta)$$

$w \leftarrow w - \alpha_{\text{Critic}} \cdot \nabla_w J_{\text{Critic}}$

$\eta \leftarrow \eta - \alpha_{\text{Risk}} \cdot \nabla_\eta J_{\text{Risk}}$

(3) Actor update

$$\nabla_\theta J_{\text{Actor}} \leftarrow \frac{1}{|\mathcal{D}|}\sum_{\mathcal{D}}\nabla_\theta \pi(x;\theta)\big(\nabla_u Q(x,u)+\lambda(x)\nabla_u H(x,u)\big)$$

$\theta \leftarrow \theta - \beta \cdot \nabla_\theta J_{\text{Actor}}$

(4) Scenery update

$$\nabla_\varphi J_{\text{Scenery}} \leftarrow \frac{1}{|\mathcal{D}|}\sum_{\mathcal{D}} H(x,u)\nabla_\varphi \lambda(x;\varphi)$$

$\varphi \leftarrow \varphi + \zeta \cdot \nabla_\varphi J_{\text{Scenery}}$

End

- 在OTOI模式下，只需要保证最终策略是安全的，中间的策略不必是安全的。因此能够更广泛的探索虚拟环境，找到更多关于风险和奖励的信息。
- 与model-based的ACS算法相比，model-free的版本需要额外学习一个有关累积风险的风险评估函数$H(x,u;\eta)$，来取代model-based的环境模型中的$h(x')$。此外，这个函数应该与动作值函数具有一样的参数结构，输入为状态和动作，且二者为独立的变量。特别的，动作变量的引入可以植入环境信息这在model-free的Actor和Scenery更新中需要。
- 此处的的算法需要一个完美的虚拟环境，因为它需要提前碰到所有违反约束的情况来学习到最终的安全策略。

9.7.3 安全探索环境的（在SOTI模式下）的混合ACS算法

与OTOI模式不同，SOTI模式更具挑战。这是因为SOTI模式需要持续与真实环境进行交互以收集样本弥补模型的不完美，这就要求中间的每个策略都具有安全性。

不完美的环境模型可以提建模成下述形式：

$$x_{t+1} = f(x_t, u_t) + \delta_t,$$

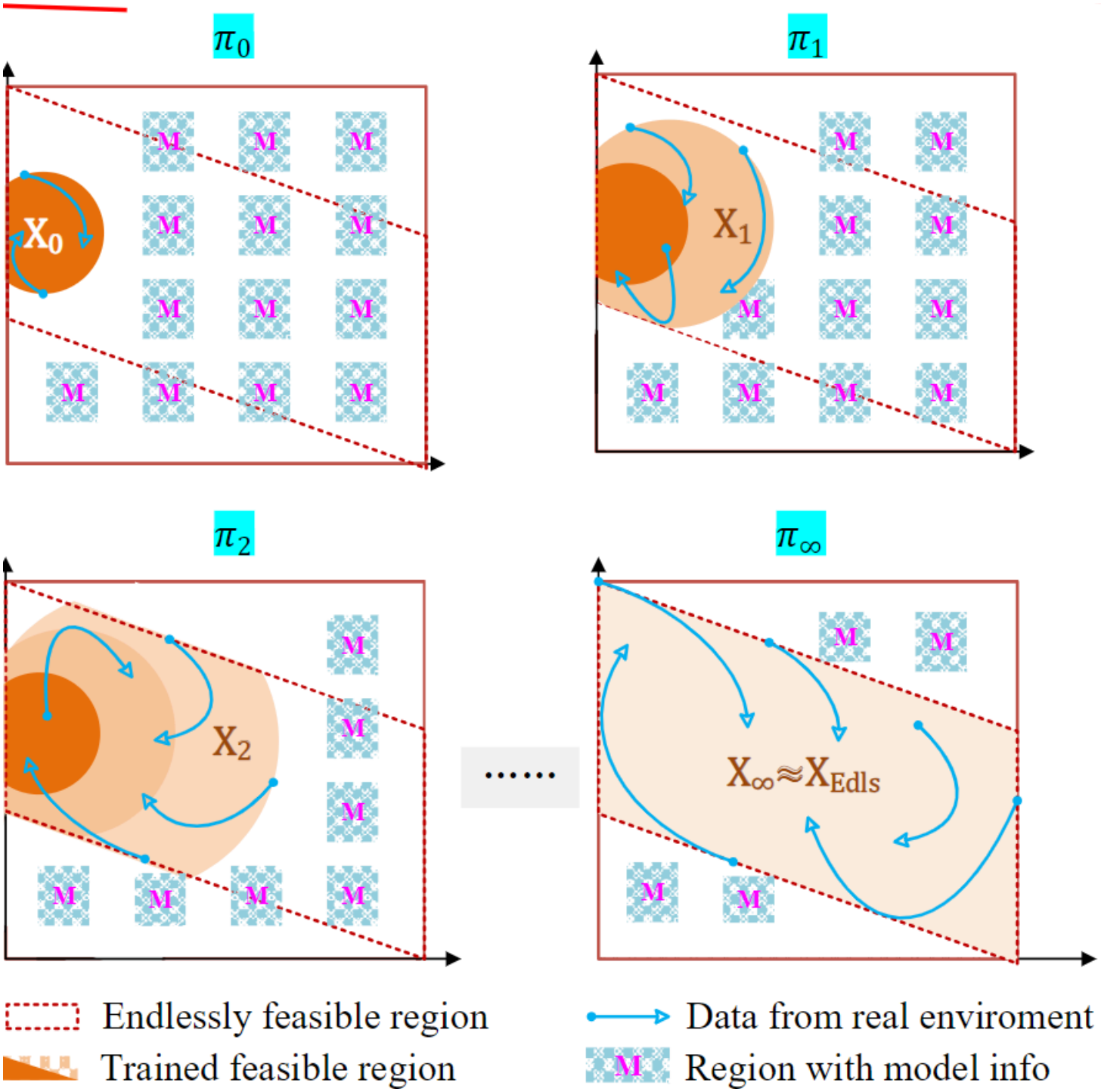
其中的δ_t是用来对于模型的不确定性进行建模的，通常把它限制在一个已知的范围内，如$||\delta_t|| \leq 1$。尽管环境模型是不完美的，但是它仍然可以在探索目前暂时未知的区域时提供先验知识来避免无目的的和危险的探索。如果只使用Trial-and-Error的方法来探索，那么在于真实环境交互的过程中危险的行为是不可避免的。

但是，仅仅有一个不完美的环境模型无法探索到更大的区域，需要额外的信息，这就需要搜集样本了。通过混合来自于模型和样本的信息就可以设计一个混合ACS算法。这里的混合指代的就是混合来自于模型和样本的信息。

9.7.3.1 安全环境交互的机制

在具有安全保证的混合ACS算法中，每次迭代都必须同时给出一个安全的策略π_k和一个安全的可行域X_k。在SOTI模式下，π_k必须在X_k内是安全的，相应的第k+1步的探索也被限制在X_k内（这里讨论的是只使用环境模型的情况，使用样本的情况会在下一节讨论）。而第k+1步探索也会输出一个新的策略-区域对(π_{k+1}, X_{k+1})，其中策略π_{k+1}是在X_{k+1}内是安全的。这样就递归的保证了安全。

然而，只是这样有一个缺陷，就是安全探索会被局限在每个局部区域内，耳聪局部区域获取的信息不足以训练一个在整个状态空间的策略。为了解决这个问题，混合的ACS Trainer首先根据局部探索得到的数据来执行区域的PIM，之后再转向使用在局部区域之外的模型信息。混合的ACS迭代过程如下图所示：



在上图中，通过每次获取可行区域之外的与真实环境交互得到的信息（蓝色箭头）来更新模型，这样可行区域越来越大，最终收敛到EFR。

另外，一个实际的ACS Trainer必须参数化它的策略函数、值函数和可行性函数。这样做有两个好处：

- 可以减小搜索空间，避免维度爆炸。
- 函数的连续性可以将局部区域内准确的环境信息扩展到每个局部区域的领域，这对于加速可行性检验的过程是非常重要的。如果合适的选取函数的连续性，那么学到的可行区域的大小就会不断增长，直至收敛到EFR（但是实际上这是一个不可能的，因为在缺乏对于不可行区域准确信息的情况下无法再保证安全的情况下确定可行与不可行区域的边界）。

9.7.3.2 使用区域加权梯度的混合ACS算法

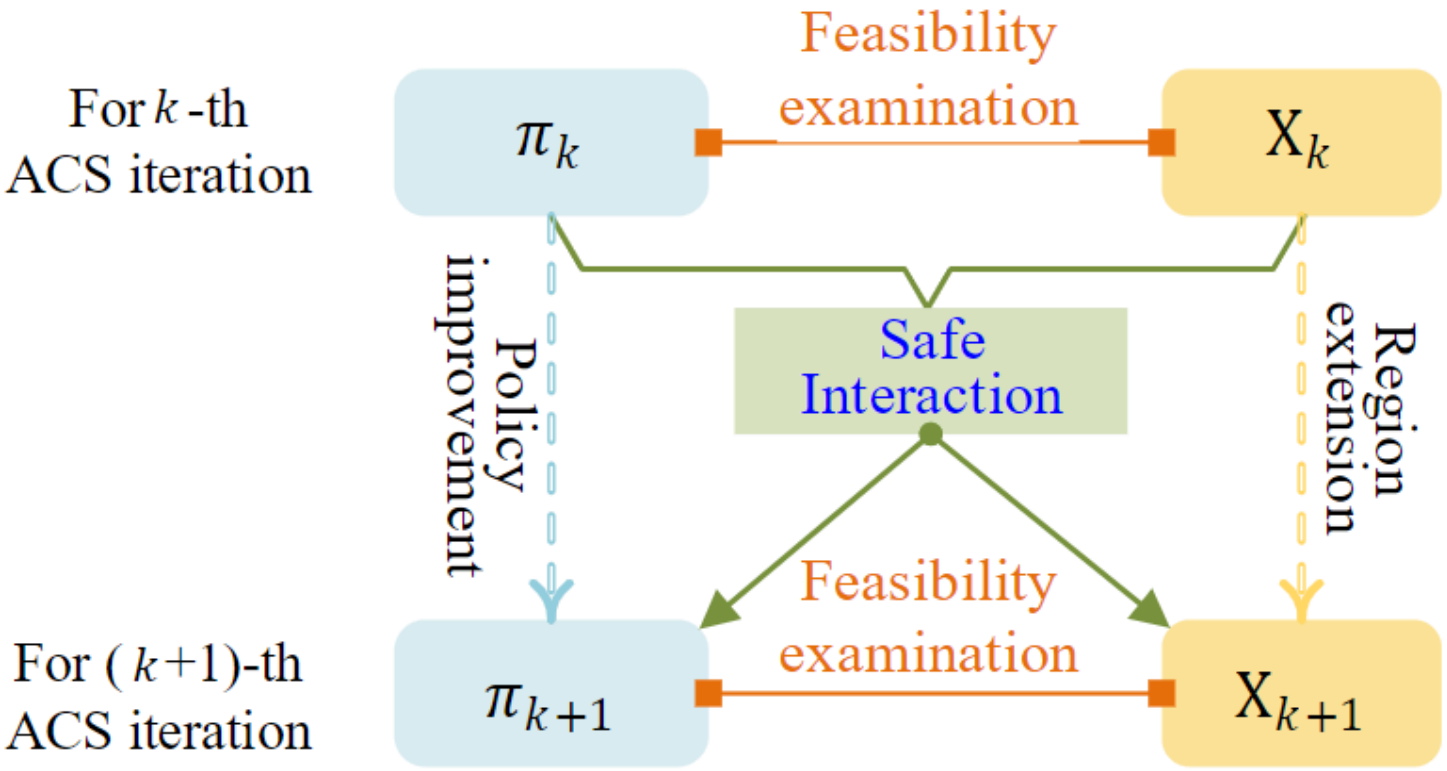
在混合的ACS算法中一个核心问题是怎么处理不完美的环境模型（提供的关于环境动力学的信息中包含一定程度的不确定性）。解决的思路如下：**在最坏情况下搜寻一个次优的策略和次完美的可行区域**。最坏情况保证了保有安全探索的能力（即使这会牺牲一定程度的最优性）。从这个角度来看，可以把环境模型的不确定性看成是一个对抗的组件，该组件的目的是搜索在最坏情况下的不确定性。那么，在第9.6.3.2节转化得到的拉格朗日对偶优化问题$\max_{\varphi}\min_{\theta}\{L(\theta,\varphi)\}$的基础上，进一步在最内层加入参数化的不确定性来构建一个新的优化问题：

$$\max_{\theta}\min_{\delta}\max_{\delta}\{L(\theta,\varphi,\delta)\}$$
$$\text{with}$$
$$L(\theta,\varphi,\delta)=Q(x,u)+\lambda(x;\varphi)h(x').$$

这里的θ是Actor的参数，φ是Scenery的参数，δ是不确定性的参数。最内层针对δ的最大化问题意味着寻找最坏情况下的扰动，而中间层和最外层则分别表示在最坏情况下寻找此时的最优策略和最优可行区域。

接下来，我们要解决怎么混合来自于真实环境采集的数据的信息和来自于不完美模型的信息。方法是加权。将使用真实数据计算得到的梯度记为∇J^{D}，使用模型计算得到的梯度记为∇J^{M}，那么最终的梯度为：
$\kappa=\mathrm{Area}(\mathrm{X}_k)/\mathrm{Area}(\mathcal{X})$
$\nabla J_{\#}^{\mathrm{Mix}}=\kappa\nabla J_{\#}^{\mathrm{D}}+(1-\kappa)\nabla J_{\#}^{\mathrm{M}}$
$\#=\mathrm{Actor},\mathrm{Critic},\mathrm{Scenery}$, 其中的$\kappa$是一个权重，$\mathrm{Area}(\mathrm{X}_k)$是当前的可行区域的大小，$\mathrm{Area}(\mathcal{X})$是整个状态空间的大小。但是实际上定义这样一个面积函数是很困难的，因此可以采取抽样若干个Sample来看有多少落在可行区域内来估算这个比值。

总结一下，混合ACS算法应该具有一个安全的Sampler来实现在线的安全采样以及一个混合的Learner来实现策略改进和可行区域扩展。



下面给出基于可解性函数、在SOTI模式下的混合ACS算法的伪代码：

Hyperparameters: critic learning rate α, actor learning rate β, scenery learning rate ζ, uncertainty learning rate ξ, maximum batch size B, critic updating frequency n_c, and actor updating frequency n_a

Initialization: action-value function $Q(x,u;w)$, policy function $\pi(x;\theta)$, adversarial uncertainty function $\delta(x;\psi)$, Lagrange multiplier function $\lambda(x;\varphi)$

//Note that initial policy guess π_0 must be safe in X_0.

Repeat (indexed by k)

(1) Collect data in the local region

$\mathcal{D}_{\mathrm{D}} \leftarrow \emptyset, \mathcal{D}_{\mathrm{M}} \leftarrow \emptyset$

$x_0 \sim d_{\mathrm{init}}(x)$

If $x_0 \in \mathrm{X}_k$

//Safe environment interaction

For i in 0,1,2, … , $B-1$ or until termination

Apply $u = \pi_k$ in the real-world environment, observe x' and r

$\mathcal{D}_{\mathrm{D}} \leftarrow \mathcal{D}_{\mathrm{D}} \cup \{(x,u,r,x')\}$

End

Else

//Use imperfect model

For i in 0,1,2, … , $B-1$ or until termination

$u = \pi(x;\theta), \delta = \delta(x;\psi)$

$x' = f(x,u) + \delta$

Compute $r, h, \partial Q/\partial u\,, \partial h/\partial x\,, \partial \pi^{\mathrm{T}}/\partial \theta\,, \partial f^{\mathrm{T}}/\partial u\,, \partial \lambda/\partial \varphi$

$$\mathcal{D}_{\mathrm{M}} \leftarrow \mathcal{D}_{\mathrm{M}} \cup \left\{\left(r, h, Q, \frac{\partial Q}{\partial u}, \frac{\partial h}{\partial x}, \frac{\partial \pi^{\mathrm{T}}}{\partial \theta}, \frac{\partial f^{\mathrm{T}}}{\partial u}, \frac{\partial \lambda}{\partial \varphi}\right)\right\}$$

End

End

$\kappa = \mathrm{Area}(\mathrm{X}_k)/\mathrm{Area}(\mathcal{X})$

(2) Mixed critic update

Repeat n_c times

$$\nabla_w J^{\mathrm{D}}_{\mathrm{Critic}} \leftarrow -\frac{1}{|\mathcal{D}_{\mathrm{D}}|}\sum_{\mathcal{D}_{\mathrm{D}}}\big(r + \gamma Q(x',u') - Q(x,u)\big)\nabla_w Q(x,u;w)$$

$$\nabla_w J^{\mathrm{M}}_{\mathrm{Critic}} \leftarrow -\frac{1}{|\mathcal{D}_{\mathrm{M}}|}\sum_{\mathcal{D}_{\mathrm{M}}}\big(r + \gamma Q(x',u') - Q(x,u)\big)\nabla_w Q(x,u;w)$$

$$w \leftarrow w - \alpha\big(\kappa \nabla_w J^{\mathrm{D}}_{\mathrm{Critic}} + (1-\kappa)\nabla_w J^{\mathrm{M}}_{\mathrm{Critic}}\big)$$

End

(3) Robust actor and scenery updates

Repeat n_a times

//Actor policy parameter

$$\nabla_\theta J^{\mathrm{D}}_{\mathrm{Actor}} \leftarrow \frac{1}{|\mathcal{D}_{\mathrm{D}}|}\sum_{\mathcal{D}_{\mathrm{D}}} \nabla_\theta \pi(x;\theta)\left(\nabla_u Q(x,u) + \lambda(x)\frac{\partial f^{\mathrm{T}}}{\partial u}\frac{\partial h(x')}{\partial x'}\right)$$

$$\nabla_\theta J^{\mathrm{M}}_{\mathrm{Actor}} \leftarrow \frac{1}{|\mathcal{D}_{\mathrm{M}}|}\sum_{\mathcal{D}_{\mathrm{M}}} \nabla_\theta \pi(x;\theta)\left(\nabla_u Q(x,u) + \lambda(x)\frac{\partial f^{\mathrm{T}}}{\partial u}\frac{\partial h(x')}{\partial x'}\right)$$

$$\theta \leftarrow \theta - \beta\left(\kappa\nabla_\theta J^{\mathrm{D}}_{\mathrm{Actor}} + (1-\kappa)\nabla_\theta J^{\mathrm{M}}_{\mathrm{Actor}}\right)$$

//Adversarial uncertainty parameter & model-based update

$$\nabla_\varphi J^{\mathrm{M}}_{\mathrm{Adv}} \leftarrow \frac{1}{|\mathcal{D}_{\mathrm{M}}|}\sum_{\mathcal{D}_{\mathrm{M}}} \nabla_\psi \delta(x;\psi)\frac{\partial h(x')}{\partial x'}\lambda(x)$$

$$\psi \leftarrow \psi + \xi J^{\mathrm{M}}_{\mathrm{Adv}}$$

//Scenery parameter

$$\nabla_\varphi J^{\mathrm{D}}_{\mathrm{Scenery}} \leftarrow \frac{1}{|\mathcal{D}_{\mathrm{D}}|}\sum_{\mathcal{D}_{\mathrm{D}}} h(x')\nabla_\varphi \lambda(x;\varphi)$$

$$\nabla_\varphi J^{\mathrm{M}}_{\mathrm{Scenery}} \leftarrow \frac{1}{|\mathcal{D}_{\mathrm{M}}|}\sum_{\mathcal{D}_{\mathrm{M}}} h(x')\frac{\partial \lambda(x;\varphi)}{\partial \varphi}$$

$$\varphi \leftarrow \varphi + \zeta\left(\kappa\nabla_\varphi J^{\mathrm{D}}_{\mathrm{Scenery}} + (1-\kappa)\nabla_\varphi J^{\mathrm{M}}_{\mathrm{Scenery}}\right)$$

End

(5) Update policy and feasible region

$$\pi_{k+1} = \pi(x;\theta)$$

$$\delta_{k+1} = \delta(x;\psi)$$

$$\mathrm{X}_{k+1} = \{x|\lambda(x;\varphi)h(x') \le \varepsilon\}$$

End

- 与真实世界交互收集到的数据必须在可行区域内进行收集（因为必须保证安全），而不完美的环境模型可以提供一些关于未知区域的信息（虽然可能不准但是起码能提供一个方向性指导）来扩大可行区域。

- 在每次迭代时，每个组件（Actor、Critic、Scenery）都会更新若干次来防止过早停止造成的数值误差。这样能够找到更安全的策略和更大的可行区域。
- 上面代码里的(1) Collect data in the local region部分数据分为两个来源，来自于真实环境交互的数据必须保证其来自于当前的可行区域，而来自于模型的数据则来自于当前可行区域之外。
- 更新J_{Adv}的时候梯度为什么这样算呢？因为是关于ϕ求梯度且$x' = f(x,u) + \delta$，而δ又是ϕ的函数，所以要用链式法则。
- 新的可行区域X_{k+1}为什么是通过上面那个式子计算的呢？就是在9.6.3.2节讲过的通过互补松弛条件来确定新的可行区域。

9.7.4 安全盾机制

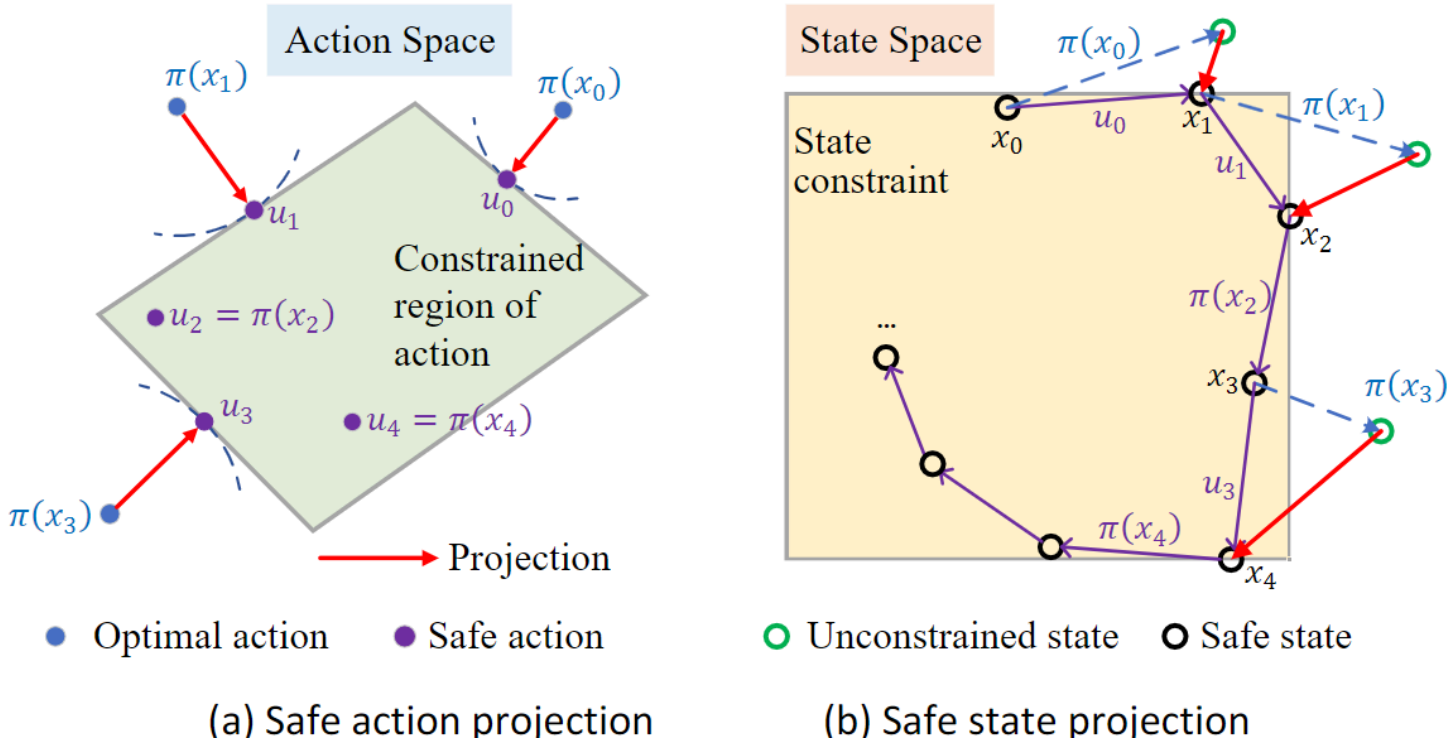
因为不充足的探索、函数近似误差或者过早停止算法迭代或者压根就没有考虑到安全，我们都需要引入安全盾机制作为最后一道屏障来保证安全。安全盾的原理是持续监控RL Agent的行为，如果发现有不安全的动作就用一个安全的动作来替换。最简单的替换方式是将不安全的动作投影到到一个新的动作，使得通过这个新的动作转移到的下一个状态满足约束。投影实际上可以看成一个优化问题：

$$
\begin{gathered}
u_{\mathrm{Safe}} = \arg\min_{u} \max_{\|\delta\| \leq 1} \|u - \pi(x)\|^2, \\
\text{subject to} \\
x' = f(x,u) + \delta, \\
h(x') \leq 0,
\end{gathered}
$$

其中的$\pi(x)$是当前的策略，x是当前的状态，x'是下一个状态，$h(x')$是下一个状态的约束函数，这里的环境模型考虑了不确定性。还需做几点解释：

- 上面的minimax问题是什么意思？为什么这样构建：内层的max代表了在模型不完美存在不确定性的情况下的最坏情况，外层的min代表了在最坏情况下找到一个投影，那么这个投影必定是能保证安全的。为什么这样构建一是因为极难获取到完美的环境模型，且如果有了完美的环境模型，你把么直接使用OTOI模式就只需要保证最后的策略是安全的，也就不需要安全盾了。

下图展示了安全盾的工作原理：



(a) Safe action projection (b) Safe state projection

值得注意的是，如上图中的右图所示，安全盾机制无法保证递归可行性，因此也无法快速把靠近不可行区域的状态拉回到可行区域。一个可能的解决方案是在安全盾机制中使用one-step barrier约束来替代one-step pointwise约束。

还应指出，安全盾机制并不总是能保证安全，特别是当因为某些原因进入了一个不可行区域时并不能把agent拉回到可行区域。因此，安全盾机制的应用应该满足下面的条件：

- 尽可能多的满足安全约束
- 尽可能少的应用安全盾机制（防止由使用导致的最优性损失以及计算量增大）

@[toc]

书籍链接：Reinforcement Learning for Sequential Decision and Optimal Control

本篇博客对应于原书的第十单元，重点讲述了深度强化学习（Deep Reinforcement Learning，DRL）的基础知识。深度强化学习时神经网络与强化学习的结合，需要进行大量的交互并消耗巨量的计算资源。在早起因为硬件计算资源的限制，并未受到太多重视。但是近些年来随着计算能力的不断提高，将深度学习与强化学习结合的研究取得了巨大的进展。AlphaGo及其后继者不断战胜人类顶尖高手标志着人工智能在围棋领域已经通过强化学习取得了超越人类的成就。后续的深度强化学习算法又陆续攻克了Atari、StarCraft等领域。

早期将强化学生与神经网络结合的尝试未取得成功，这可以归结为若干原因，如非独立同分布的（IID）的序列数据、算法发散、过估计和采样效率低下。首次将二者结合的成功尝试是Deepmind的DQN算法。在该算法中，Q-Learning和一个卷积神经网络进行结合，而这个卷积神经网络被用来在连续状态空间中学习一个对于最优Q函数的估计。除了算法本身，DQN的论文还提出了两个对于后续算法影响深远的技巧来稳定训练过程：**经验回放（Experience Replay）和目标网络（Target Network）**。之后提出的Double DQN采用了两个Q网络，从而将动作选择和动作评估分离。作为DQN的一个重要的变体，DDPG（Deep Deterministic Policy Gradient）用于同时学习一个 Q 函数和一个用于连续控制任务的确定性策略。

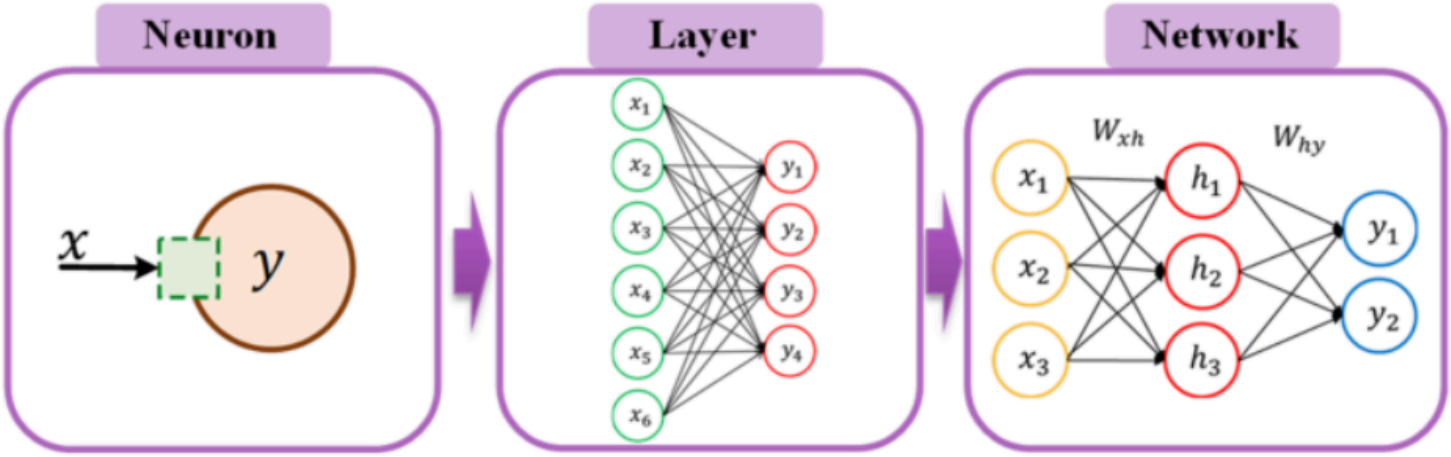
Q 值的高估是 DDPG等算法中的一个常见问题。在DDPG中这导致了脆弱的策略，即使添加了一些例如双 Q 函数技巧。为了解决这个问题，Fujimoto提出了Twin-Delayed DDPG（TD3），其中包含一些新提出的技巧，包括**有界双 Q 函数**和**延迟策略更新**。 Soft Actor-Critic （SAC）算法依赖于最大熵 RL 框架，在随机探索和确定性策略搜索之间建立了一座桥梁。在这里，“soft” 意味着传将策略熵加入原本的奖励中，这样可以平衡探索（Exploration）和利用（Exploitation）。而Distributional Soft Actor-Critic（DSAC）是 SAC 的增强版本，通过学习回报（Return）的分布而不是直接学习Return的期望（值函数）来克服高估问题并提高学习效率。

深度强化学习中另一个常见的问题时训练的不稳定性，这还通常伴随着严重的低效率。一种常见的应对之道是在每步更新的时候避免过大的改变策略。但是怎么定义改变的大小呢？又该怎么决定每次的更新步长呢？太小的步长导致收敛过慢，但是太大的步长则会导致不稳定性。TRPO（Trust Region Policy Optimization）算法通过构造一个信赖域约束来量化旧策略和新策略之间的距离。TRPO的好处是可以保证策略优化时的单调改进（即一直变好），但它的缺点在于每次优化都要求解一个二阶优化问题。为了解决这个问题，PPO（Proximal Policy Optimization）算法提出了一种一阶的剪裁方法来简化计算。令人惊奇的是，在大大提升计算效率的同时仍能保证策略的单调改进。

另外，online的强化学习算法通常效率很低。为此，A3C（Asynchronous Advantage Actor-Critic）算法提出了一种并行化的方法，通过多个actor同时与平行的环境交互来提高效率。A3C同时还使用了熵正则化来平衡探索和利用。

10.1 神经网络入门

神经网络的基础单元是神经元，神经元在宽度方向上排列组成层，层在深度方向上排列组成网络。



一个网络如果超过5层就可以被称为深度网络。

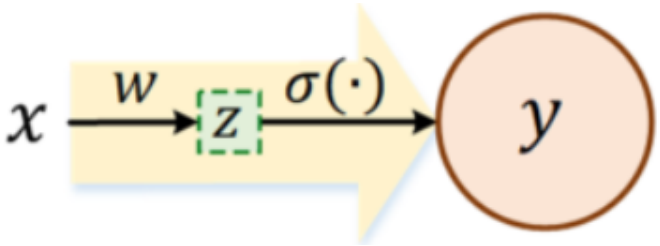
神经网络的一大优点在于所谓的全局逼近能力。现已证明即使是一个简单的只具有单一隐层的前馈网络，在一些轻微的假设下，也可以逼近任意连续函数。这个定理被称为万能逼近定理（Universal Approximation Theorem）。在1989年，Cybenko证明了一个使用logistic激活函数（sigmoid）的单隐层网络可以逼近任意连续函数。后续研究证明了只要激活函数不是线性的多项式函数就可以逼近任意连续函数。

10.1.1 神经元

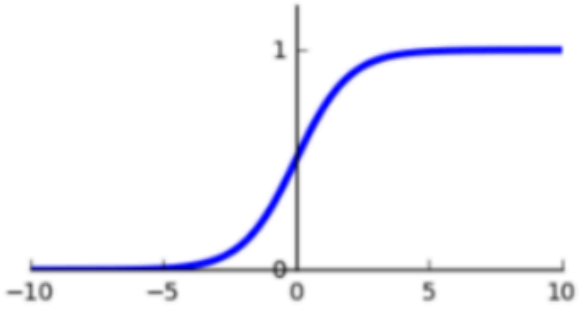
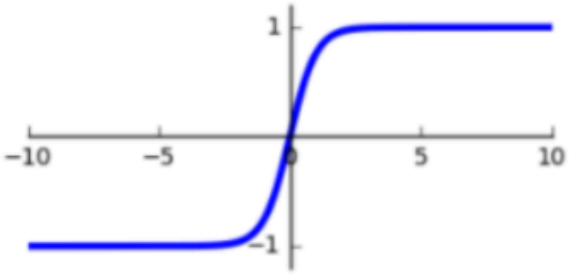
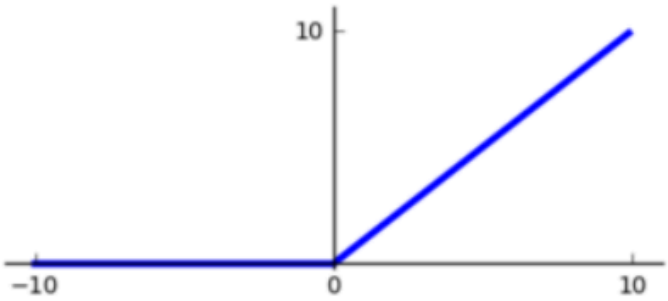
人工神经元作为对于生物神经元的一种模仿，也具有生物神经元接受来自其它神经元不同强度的电信号并在超过一定阈值之后被激活的特性。这种模式的数学表述如下：

$$z = w^{\mathrm{T}} x + b,$$
$$y = \sigma(z).$$

这里，x是输入向量，$ww \in \mathbb{R}^n$是权重向量，$b \in \mathbb{R}$是偏置，z是加权之后的和，σ是激活函数，最终输出为y。一个神经元的典型图示如下：



常见的激活函数由如下三种：

logistic $\sigma(z) = \frac{1}{1+\mathrm{e}^{-z}}$	
tanh $\sigma(z) = \frac{\mathrm{e}^{z}-\mathrm{e}^{-z}}{\mathrm{e}^{z}+\mathrm{e}^{-z}}$	
ReLU $\sigma(z) = \begin{cases} 0, z \leq 0 \\ z, z > 0 \end{cases}$	

- Sigmoid函数（Logistic函数）：

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

  输出为0到1之间的值，适合做二分类问题。
- 双曲正切函数（Tanh函数）：

$$\sigma(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$

  Sigmoid函数的变体，关于原点对称，输出为-1到1之间的值。
- ReLU函数：

$$\sigma(z) = \max(0, z)$$

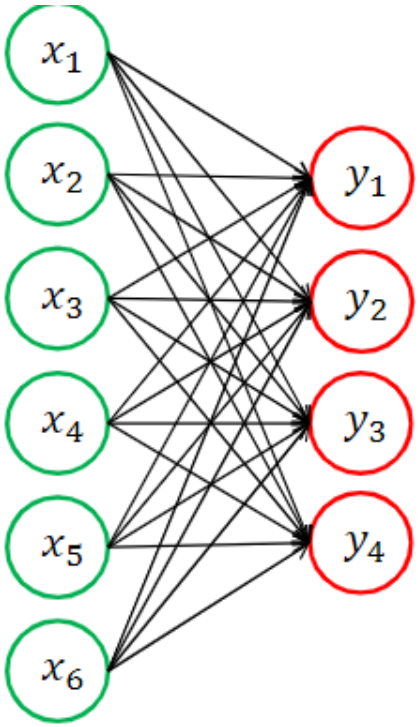
  上述两种激活函数都有严重的梯度消失问题（从上图也可以看出，即在两端梯度接近于0，这种现象在深层网络的反向传播过程中影响尤其显著），因此提出了ReLU函数。ReLU函数在$z < 0$时梯度为0，而在$z > 0$时梯度为常数。而且由于ReLU函数能够输出0值，因此它具有所谓的“稀疏表征”的特性，即某些神经元如果没用的话就会自动被“关闭”，这有助于提高网络的泛化能力。

10.1.2 典型的神经网络的层

层可以分为输入层、隐层和输出层。一般来说，一个神经网络具有一个输入层、一个输出层和若干个隐层。通常认为，隐层越多，网络的近似能力就越强。一般来说，一个隐层只与它前后的两层相连，但是在DenseNet中，一个隐层可以与所有前面的层相连，且它的输出会传给之后的所有层。因此可以用更少的参数实现更高的准确率。

10.1.2.1 全连接层

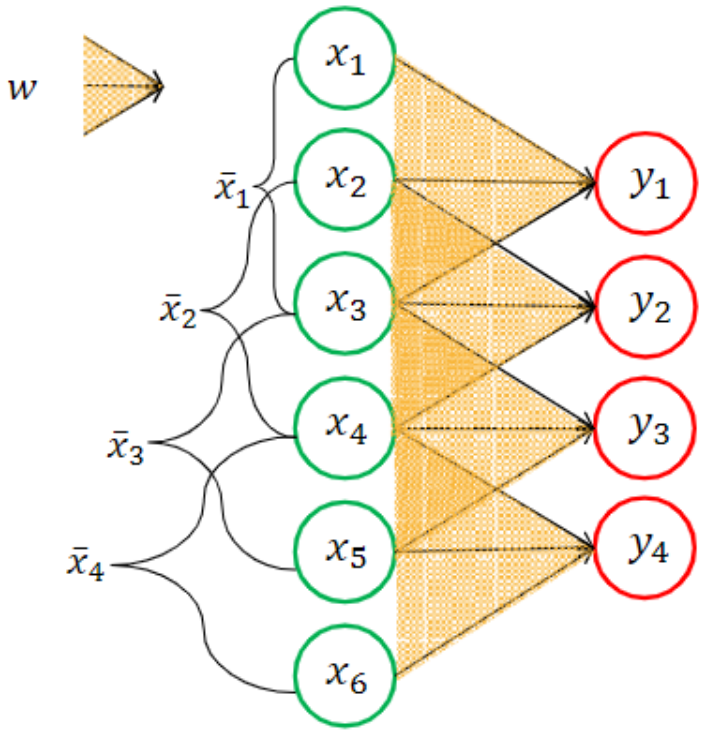
全连接层是最简单的一种层，每个神经元均接受前一层的所有神经元的输入，通过权重矩阵W和偏置向量b进行线性变换，然后通过激活函数进行非线性变换，之后输出给下一层。



但是，这种层的缺点在于计算量大，参数多，且容易过拟合，因此泛化性能较差。为了解决这个问题，常使用Dropout技术，即手动或自动的去处神经元之间的一些连接，从而在不损失准确率的情况下减少过拟合。

10.1.2.2 卷积层

卷积层是全连接层的一种特殊变体。它的基础是一种叫做卷积的数学操作。一个一维卷积的例子如下：



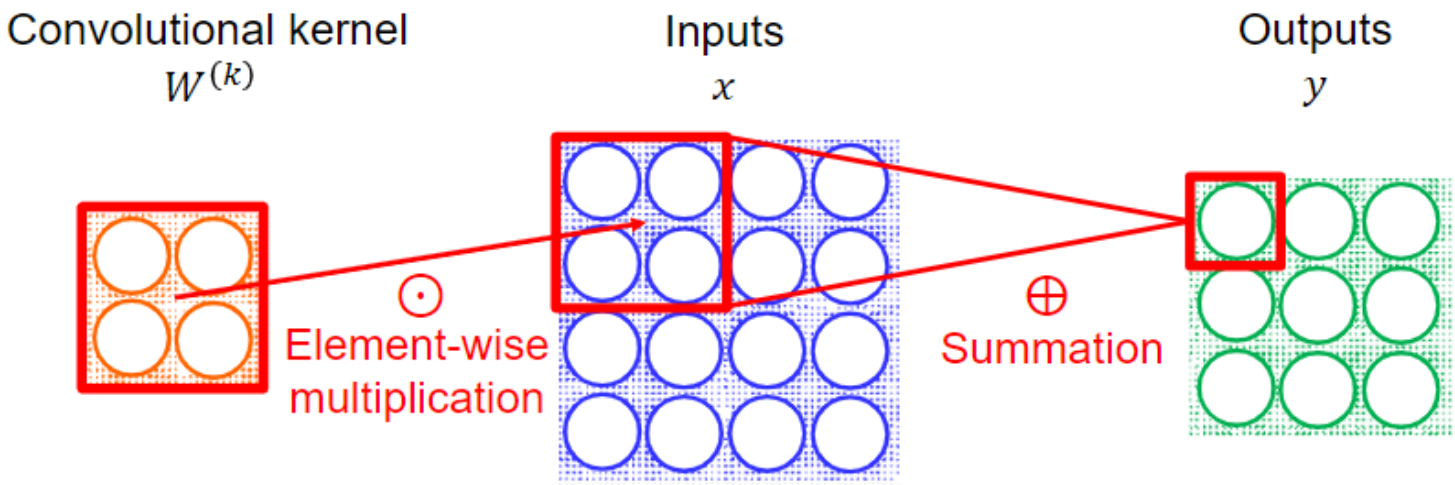
其数学公式为：

$$y = \sigma(w * x),$$
$$y_i = \sigma(w^{\mathrm{T}}\bar{x}_i) = \sigma\left(\sum_{j=1}^{N} w_j \cdot x_{i+j-1}\right), i = 1, 2, \cdots, m,$$

其中，w被称为卷积核，长度为N，x是输入向量，包含n个元素，y是输出向量，包含m个元素，σ是激活函数。如上图所示，卷积操作可以看做卷积核在输入向量上的滑动（上图中每次滑动一格，实机不一定，这个滑动的步长称为stride），每次卷积核与其覆盖区域的对应元素先相乘最后再相加即可得到输出向量的一个元素。从上图可以看出，卷积相比于全连接好在两点：

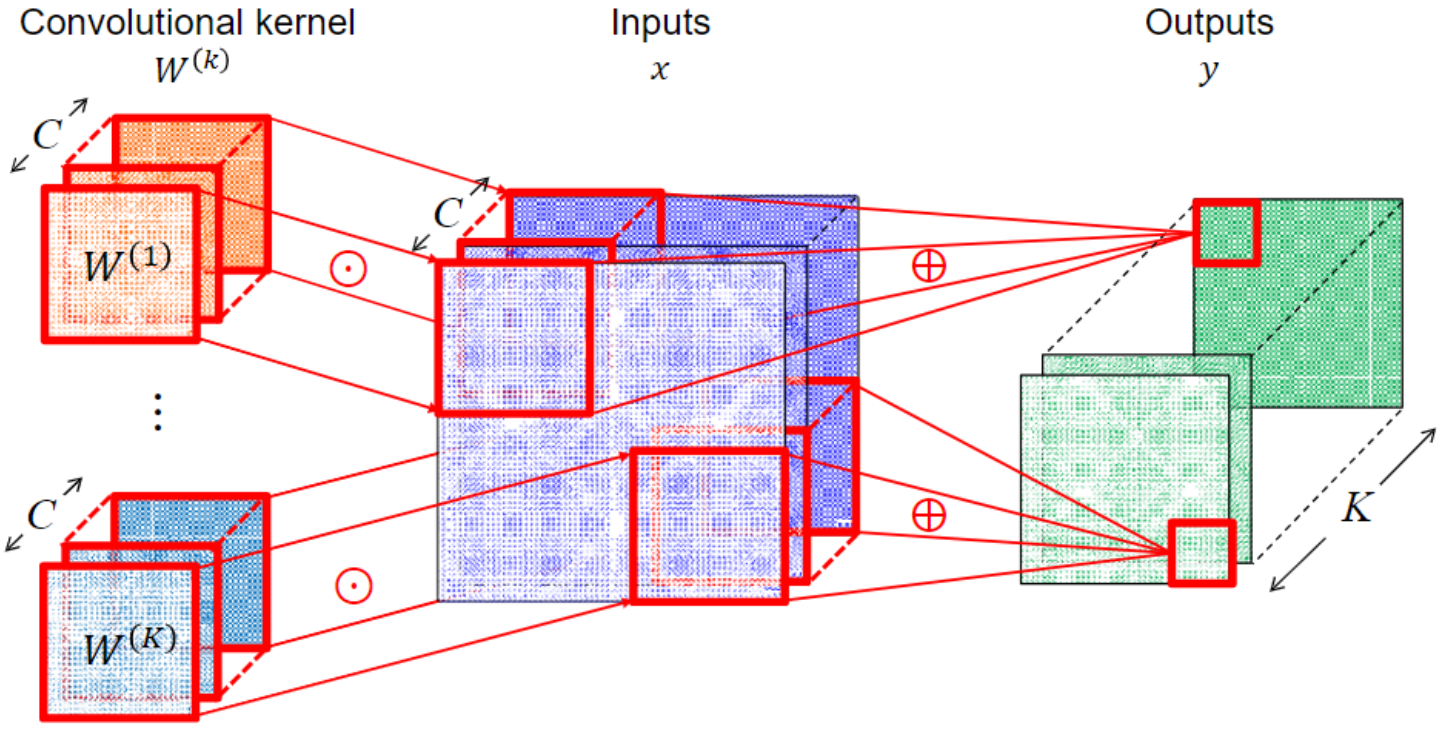
- **参数共享**：卷积核在整个输入上滑动，因此每个位置上的参数都是一样的，这样可以大大减少参数数量。
- **局部连接**：卷积核每次计算的时候仅与输入的一部分相连，这样可以保留输入的空间结构，从而提高网络的泛化能力。

卷积可分为一维卷积、二维卷积和三维卷积，分别对应于一维、二维和三维的输入。一维卷积刚才已经举过例子，下面来看一个二维卷积：



(a) 2D convolution

上图的例子中，卷积核大小为2×2，步长为1，输入大小为4×4，那么根据卷积的原理，可以自己是试一试，易得输出大小为3×3。三维卷积则是在二维卷积的基础上再增加一个维度，即深度（或者称为通道数）。三维卷积的例子如下：

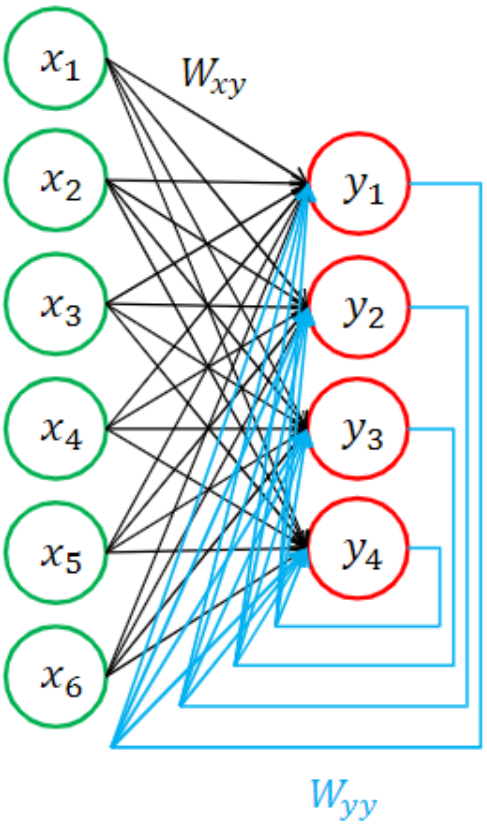


在三维卷积中，**输入的通道数等于卷积核的厚度C**，而**输出的通道数等于卷积核的个数K**。可以看出，每个卷积核在于三维输入做卷积的时候实际上是卷积核C层中的每一层都在于输入进行运算，最后加在一起，而着在输出中只表现为K个通道中的一个。

然而，卷积的一个缺点是它对于输入的每个位置都是敏感的，因此对于输入的变化不够鲁棒。为了解决这个问题，可以使用降采样技术，如池化（Pooling）技术。池化将输出划分为一系列不重合的矩形区域，每个区域的输出则根据最大化、最小化或者平均化等操作得到。池化技术使得网络对于输入的变化更加鲁棒，这也被称为平移不变性。

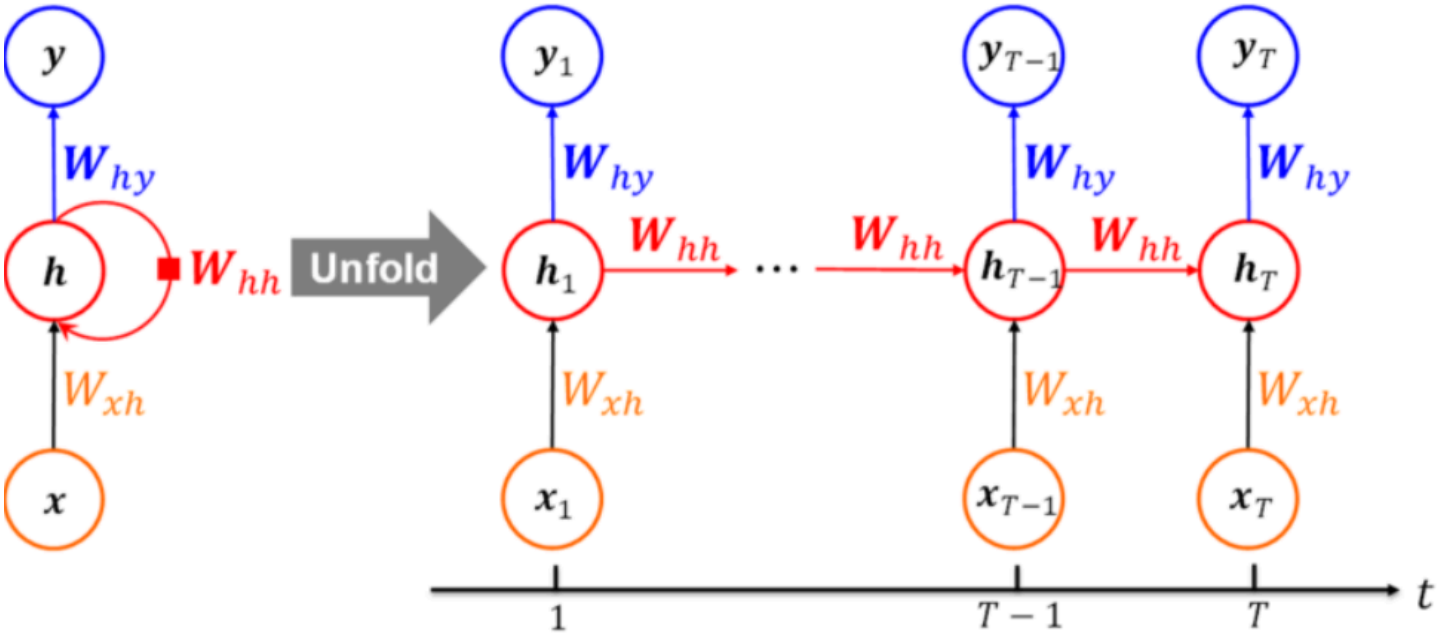
10.1.2.3 循环层

循环层是一种特殊的具有时序上的反馈机制的层。上一个时间步的输出会作为当前时间步的输入的一部分。这种循环机制要求网络具有记忆能力，这通常通过隐状态（Hidden State）来实现。循环层的图示和公式如下：

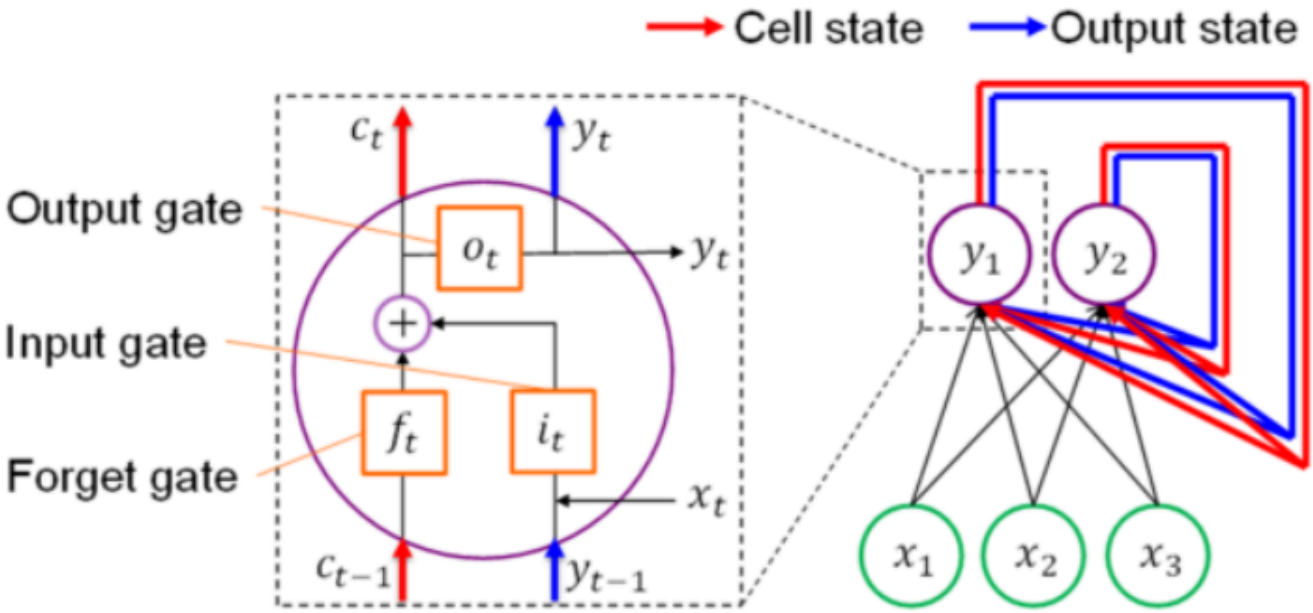


$$y_t = \sigma(W_{xy}x_t + W_{yy}y_{t-1}),$$

其中，x_t是当前时刻的输入，y_t是当前时刻的输出，y_{t-1}是上一个时刻的输出，W_{xy}和W_{yy}分别是针对当前输入和上一个输出的权重矩阵，σ是激活函数。当在时间维度上展开后循环层就变成了一个前馈层：



但是，训练一个循环神经网络是非常困难的，因为它的梯度通常会爆炸或者消失，即使只经过了几个时间步。为了解决这个问题，可以添加一些额外的连接来避免反向传播的时候对于梯度的指数操作。这种想法产生了两种主要的循环神经网络的变体：长短时记忆网络（LSTM）和门控循环单元（GRU）。

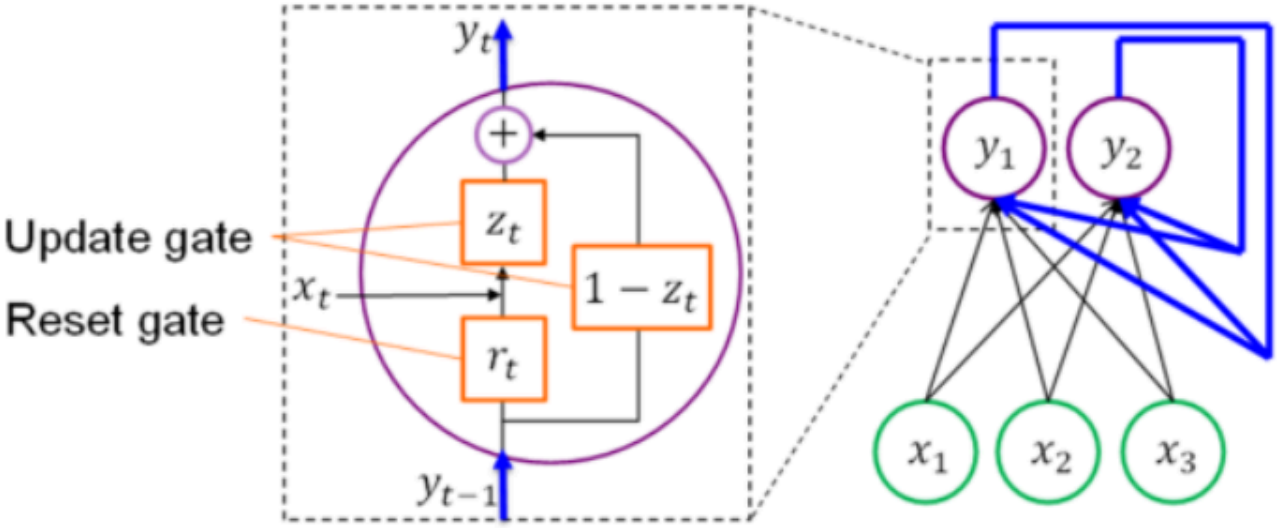


(a) Long short-term memory (LSTM)

首先是LSTM，它引入了三个门：**遗忘门、输入门和输出门**。遗忘门决定了上一个时刻的隐状态中哪些信息需要被遗忘，输入门决定了当前时刻的输入中哪些信息需要被记忆，输出门决定了当前时刻的隐状态中哪些信息需要被输出。这些门通常使用sigmoid函数来产生0到1之间的值，“0”表示完全关闭，“1”表示完全打开。LSTM的公式如下：

$$
\begin{aligned}
f_t &= \sigma(W_{xf}x_t + W_{yf}y_{t-1} + b_f),\\
i_t &= \sigma(W_{xi}x_t + W_{yi}y_{t-1} + b_i),\\
o_t &= \sigma(W_{xo}x_t + W_{yo}y_{t-1} + b_o),\\
c_t &= f_t \odot c_{t-1} + i_t \odot \sigma(W_{xc}x_t + W_{yc}y_{t-1} + b_c),\\
y_t &= o_t \odot \sigma(c_t).
\end{aligned}
$$

其中，f_t是遗忘门，i_t是输入门，o_t是输出门，c_t是细胞状态，h_t是隐状态，$\odot$表示逐元素相乘。LSTM的一个缺点是它的参数量较大，因此GRU被提出来减少参数量。



(b) Gated recurrent unit (GRU)

GRU只有两个门：**更新门和重置门**。重置门决定了之前的信息有多少需要被遗忘，更新门决定了新的信息有多少需要被记忆，又有多少是上一时刻信息的copy。GRU的公式如下：

$$
\begin{aligned}
r_t &= \sigma(W_{xr}x_t + W_{yr}y_{t-1} + b_r),\\
z_t &= \sigma(W_{xz}x_t + W_{yz}y_{t-1} + b_z),\\
h_t &= (1 - z_t) \odot y_{t-1} + z_t \odot \sigma(W_{xh}x_t + W_{yh}(r_t \odot y_{t-1}) + b_h).
\end{aligned}
$$

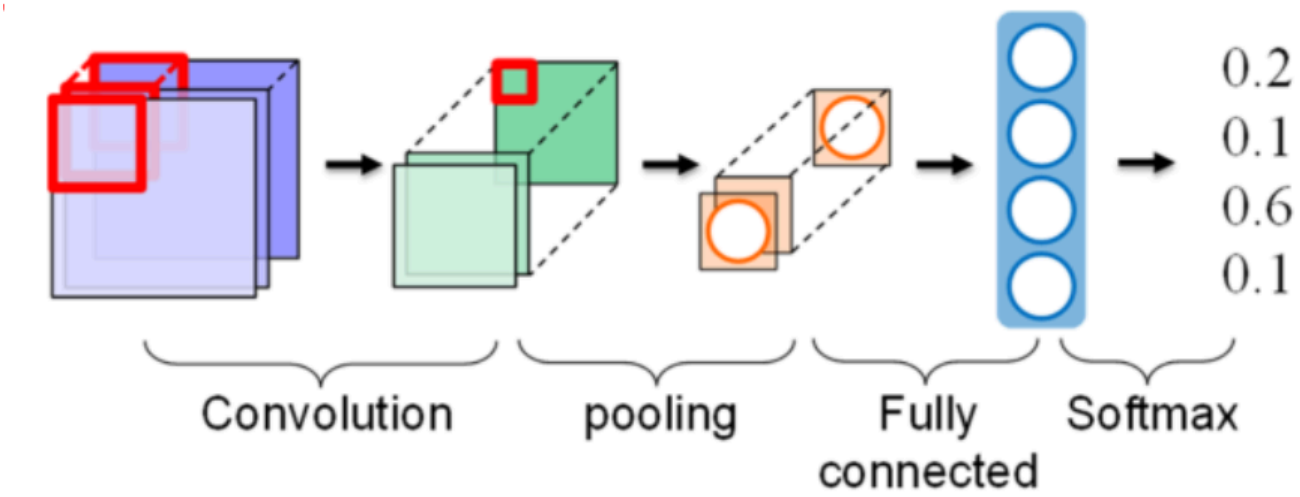
其中，r_t是重置门，z_t是更新门，h_t是隐状态。

10.1.3 典型的神经网络

简单的来说，神经网络可以视为把一些层堆叠在一起。按照是否具有循环结构，神经网络可以分为前馈神经网络和循环神经网络。前馈神经网络是最简单的一种神经网络，每一层的输出都只与前一层的输出有关，没有循环结构，构成有向无环图；而循环神经网络则具有循环结构。

时至今日，已经提出了很多神经网络。其中最著名的有卷积神经网络（CNN）和循环神经网络（RNN）：

- **卷积神经网络（CNN）**：
  - **特点**：卷积神经网络是一种专门用于处理具有空间结构的数据的神经网络。它的特点在于参数共享和局部连接，这使得它对于图像等数据具有很强的表达能力。
  - **结构**：卷积神经网络通常由若干个卷积层、池化层和全连接层组成。



- **代表模型**：LeNet、AlexNet、VGG、GoogLeNet、ResNet等。

	Number of convolutional layers	Size of kernels	Number of kernels	Number of channels	Sliding stride
LeNet	2	5	20,50	1,20	1
AlexNet	5	3,5,11	96~384	3~256	1,4
GoogLeNet	21	1,3,5,7	16~384	3~832	1,2
VGG-16	13	3	64~512	3~512	1
ResNet-151	151	1,3,7	64~2048	3~2048	1,2
MobileNet	27	1,3	32~1024	3~1024	1,2

- **循环神经网络（RNN）**：
  - **特点**：循环神经网络是一种专门用于处理具有时间结构的数据的神经网络。它的特点在于具有循环结构，这使得它对于序列数据具有很强的表达能力，可以编码序列关系。
  - **代表模型**：stacked RNN、bidirectional RNNs、structural RNNs、 recursive NNs。

10.2 神经网络的训练

10.2.1 损失函数

10.2.1.1 交叉熵（CE）损失和均方误差（MSE）损失

- **交叉熵损失**：
  - **定义**：

$$J \overset{def}{=} \mathcal{H}(p^{\text{target}}, p) = -\sum_i p_i^{\text{target}} \log(p_i),$$

其中，p^{target}是来自于真实分布的概率值，p是来自于模型的概率值。

交叉熵损失可以用来衡量两个分布之间的差异。

- **适用于**：交叉熵损失通常用于分类问题，特别是多分类问题。

- **均方误差损失**：
  - **定义**：

$$J \overset{\text{def}}{=} \text{MSE}(y^{\text{target}}, y) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i^{\text{target}} - y_i\right)^2,$$

其中，y^{target}是真实值，y是预测值。

  - **适用于**：均方误差损失通常用于回归问题。

然而，在一些特殊的条件下，交叉熵损失和均方误差损失等价，比如当目标分布是高斯分布时。

10.2.1.2 偏差-方差权衡

先来看两组概念：

- **偏差与方差**
  - **偏差**：偏差是模型的预测值与真实值之间的差异，即模型的拟合能力。偏差越大，模型的拟合能力越差。**高偏差表示模型对于训练数据没有充分拟合，训练的误差很大**。
  - **方差**：方差是模型在不同数据集上的预测值之间的差异，即模型的泛化能力。高方差说明魔性对于噪声很敏感，输入的细小变化都会导致输出的巨大变化。**高方差表示模型对于训练数据过拟合，训练的误差很小，但是验证的误差很大。这表示模型的泛化能力很差**。
- **欠拟合与过拟合**
  - **欠拟合**：欠拟合是指模型的偏差很大，即模型的拟合能力很差。这通常是因为模型的复杂度不够，或者模型的训练数据不够多或者未充分训练。
  - **过拟合**：过拟合是指模型的方差很大，即模型的泛化能力很差。这通常是因为模型的复杂度太高，或者模型的训练数据太少。

在深度学习中，由于神经网络强大的拟合能力，过拟合通常比欠拟合更常见。为了解决过拟合问题，可以采用正则化技术。该技术通过对于模型的复杂度施加惩罚来降低过拟合：

$$J_{\text{Reg}}(w) = J(w) + \rho\Omega(w),$$

其中，$J(w)$是损失函数，$\Omega(w)$是正则化项，用于度量模型复杂度。ρ是正则化系数（$\rho \in [0, \infty)$）。常见的正则化项有L_1正则化和L_2正则化：

- **L_1正则化**：

$$\Omega(w) = \|w\|_1 = \sum_i |w_i|.$$

这里，w是权重向量。

- **L_2正则化**：

$$\Omega(w) = \|w\|_2^2 = \sum_i w_i^2.$$

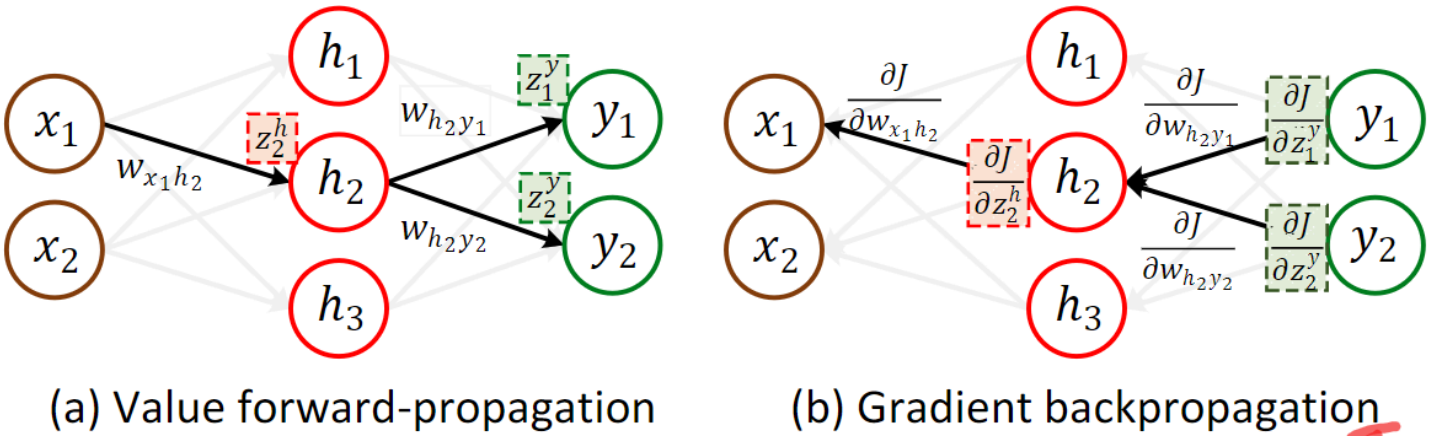
有趣的是，研究发现，L_1和L_2正则化对于模型的影响方式是不同的。L_1会通过将不重要的权重缩减至0来降低过拟合（这可能会导致一些我们关注的特征消失），而L_2则会通过减小权重的大小来降低过拟合（尽可能的减小，然而并不到0）。

10.2.2 训练算法

神经网络的训练过程可以被看做一个随机优化过程，而在随机优化的诸多方法中，一阶优化因其计算效率高而被广泛应用。一阶优化使得可训练的权重可以沿着梯度的反方向进行更新，直到达到损失函数的一个局部最小值。

10.2.2.1 反向传播算法（Backpropagation，BP）

BP算法从最后一层开始使得损失函数所表征的误差信息沿着网络向前传播来更新权重，这与网络输出时的信息流动方向正好是相反的。如下图所示：



现在我们以上图所示的单隐层全连接前馈网络为例，来推导BP算法的具体过程。先从后面往前看，假设最后一层的输出为$y_m(m=1,2)$，最后一层的激活函数统一记为$\sigma(\cdot)$，在经过激活函数激活之前输入最后一层的输入为$z_m(m=1,2)$，那么对于隐层与输出层之间的权重$w_{h_jy_m}$，我们可以写出损失函数关于这个权重的梯度：

$$\frac{\partial J}{\partial w_{h_jy_m}} = \frac{\partial J}{\partial z_m^y}\frac{\partial z_m^y}{\partial w_{h_jy_m}} = \frac{\partial J}{\partial y_m}\frac{\partial y_m}{\partial z_m^y}\frac{\partial z_m^y}{\partial w_{h_jy_m}} = \frac{\partial J}{\partial y_m}\sigma'\left(z_m^y\right)h_j,$$

这里的σ'表示激活函数的导数，而$\frac{\partial z_m^y}{\partial w_{h_jy_m}}$之所以等于$h_j$是因为$z_m^y = \sum h_j w_{h_jy_m}$。因为$J$关于$y_m$以及$\sigma$都有显式的表达式，因此上式是容易计算的。同理，可以计算输入层与隐层之间的权重$w_{x_ih_j}$的梯度：

$$\frac{\partial J}{\partial w_{x_ih_j}} = \frac{\partial J}{\partial z_j^h}\frac{\partial z_j^h}{\partial w_{x_ih_j}} = \frac{\partial J}{\partial h_j}\frac{\partial h_j}{\partial z_j^h}\frac{\partial z_j^h}{\partial w_{x_ih_j}} = \frac{\partial J}{\partial h_j}\sigma'\left(z_j^h\right)x_i = \left(\sum_m w_{h_jy_m}\frac{\partial J}{\partial z_m^y}\right)\sigma'\left(z_j^h\right)x_i,$$

这里的z_j^h是隐层的输入，h_j是隐层的输出。特别需要注意一下为什么$\frac{\partial J}{\partial z_j^h}$为什么等于$\sum_m w_{h_jy_m}\frac{\partial J}{\partial z_m^y}$。另外，$\frac{\partial J}{\partial z_m^y}$在上一步已经计算过了。

需要说明的是BP并不是唯一的计算方向，也可以从前向后计算，但是BP在计算较靠前的梯度时可以利用较靠后的许多中间计算结果，如$\frac{\partial J}{\partial z_m^y}$和$\frac{\partial J}{\partial z_j^h}$，这样可以减少计算量而不用重复计算。这也是BP算法被广泛应用的原因之一。

另外需要注意的是，BP并不等价于训练神经网络算法的全部，而只是其中关于按什么顺序怎么计算梯度的一种规则。真正用于更新的是梯度下降法，其中较为常用的是mini-batch SGD：

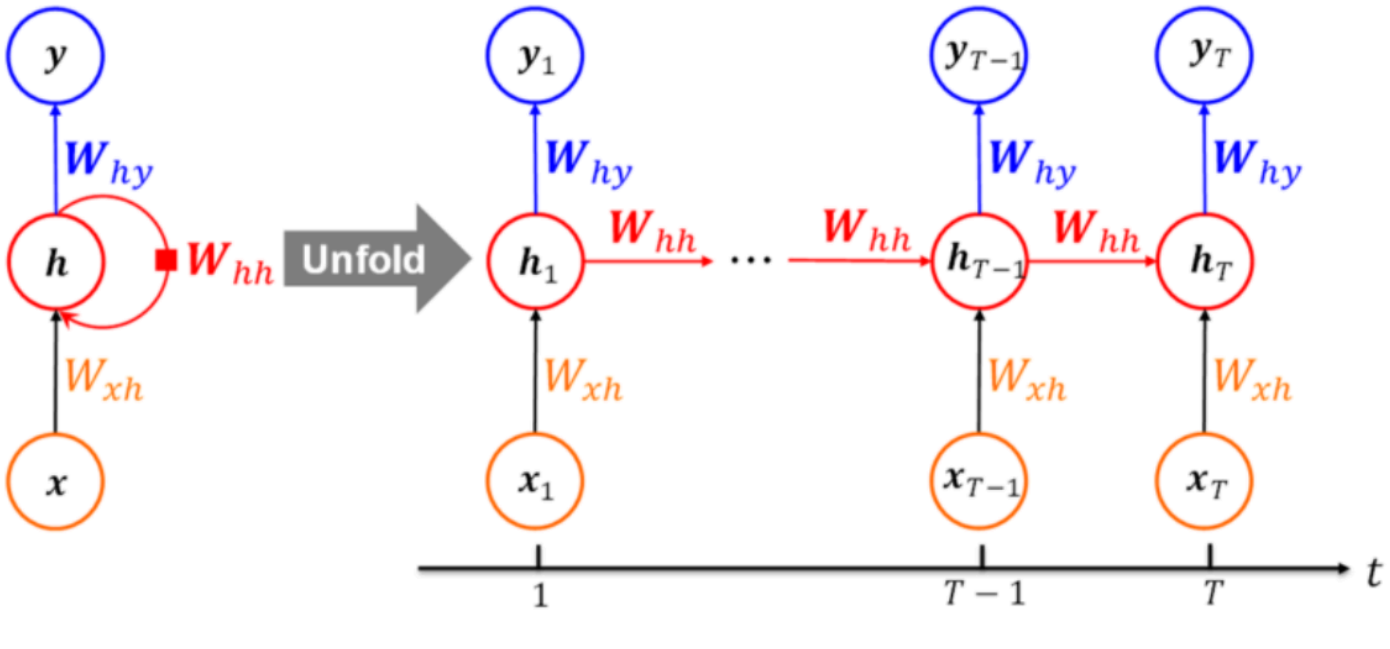
$$w \leftarrow w - \eta \mathbb{E}_{\mathcal{D}} \left\{ \frac{\partial J}{\partial w} \right\},$$

在上例中，共有12个可训练的参数：

$$\frac{\partial J}{\partial w} = \left[\underbrace{\frac{\partial J}{\partial w_{x_i h_j}}, \frac{\partial J}{\partial w_{h_j y_m}}}_{12=2\times 3+3\times 2} \right]^{\mathrm{T}} \in \mathbb{R}^{12},$$

这里的η是学习率，$\mathcal{D}$是用于更新的mini-batch数据集。原始的SGD只使用一个样本来计算梯度，这样可能会导致梯度的方差很大，这可能导致训练时严重的震荡，降低训练稳定性。因此mini-batch SGD使用一个小的数据集来计算梯度。另外，还可通过动量法来降低训练的震荡，该方法既考虑了当前的梯度，又考虑了历史的梯度信息。

10.2.2.2 通过时间的反向传播算法（Backpropagation Through Time，BPTT）



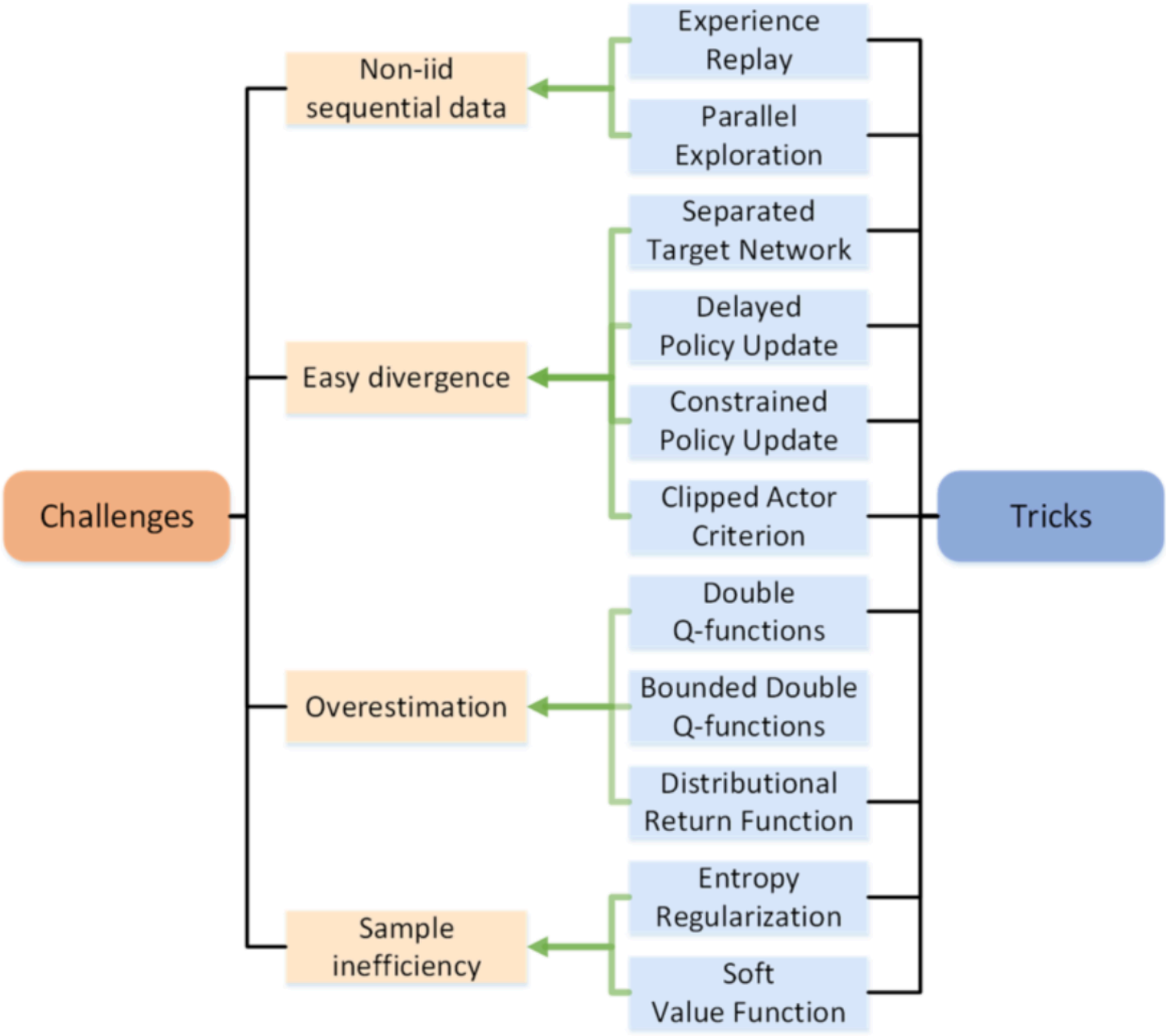
通过将循环层沿着时间展开，就可以得到一个**沿着时间维度，所有隐藏层的权重都相等的前馈网络**。这也被称为**沿着时间维度的反向传播算法（BPTT）**。注意，这里展开的时间步数是有限的，其大小等于**输入序列的长度**。这样就可以通过类似于标准BP算法的方式来计算梯度。

然而，因为所有的权重在时间维度上都是相同的，因此反向传播时对于某个相同参数的梯度会发生很多连乘（沿着时间步分享累积），因此会导致梯度爆炸或者消失。为了解决这个问题，LSTM和GRU通过引入额外的直接连接来降低这种情况的影响。

10.3 深度强化学习的挑战与解决办法

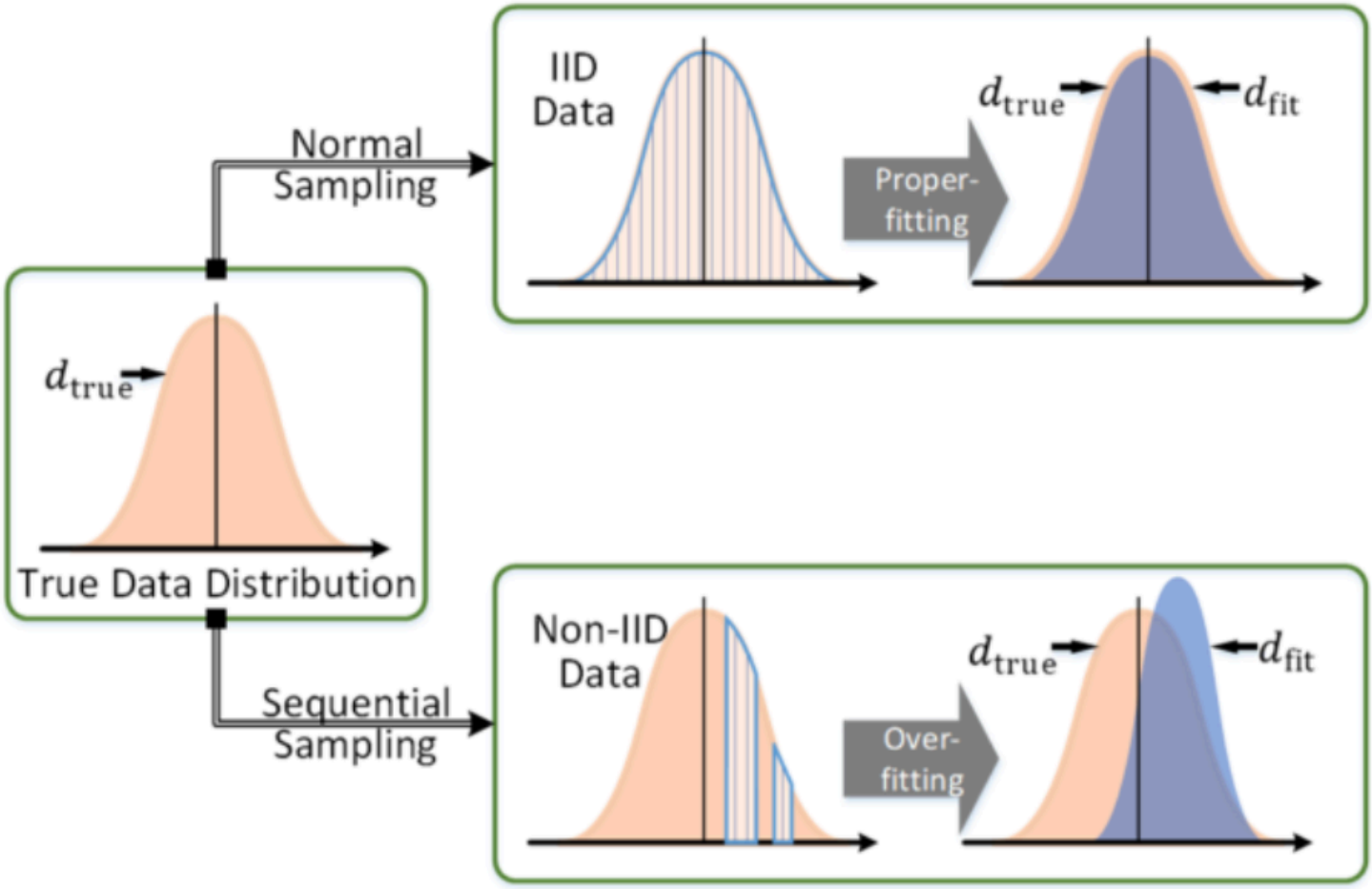
与其他用于近似的函数（如多项式函数或者径向基函数）相比，神经网络提供了一种更为强大的近似方式。将RL与神经网络结合使得强化学习算法可以直接从来自于高位的复杂任务中的原始的图像或者视频数据中学习，而且不必借助于任何手工设计的特征或者领域的先验知识，最终还能获得比人类专家更好的效果。

然而，深度强化学习可不仅仅是将深度学习与强化学习结合这么简单，因为深度强化学习从二者继承了一些棘手的挑战亟待解决。而从2015年开始，一些empirical的tricks被提出来解决这些挑战。有关于这些挑战和对应的tirck的汇总如下：



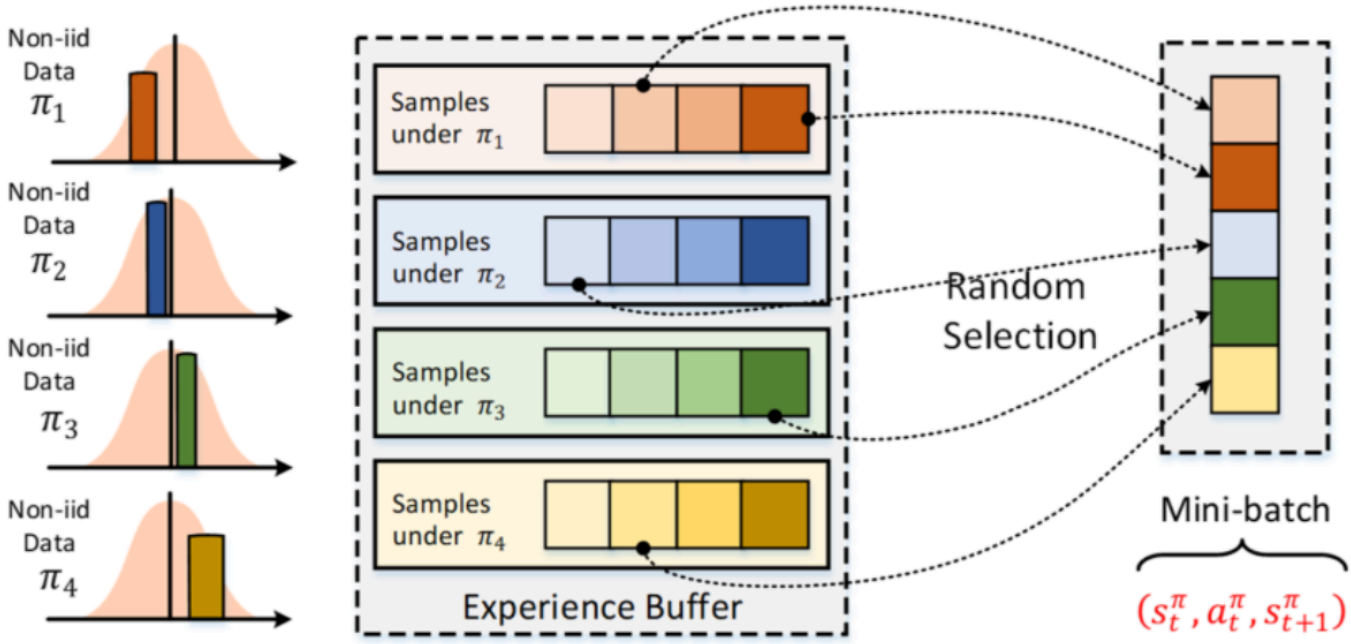
10.3.1 挑战一：非独立同分布数据（Non-IID Data）及其解决办法

对于大部分统计学习算法，收集到的数据被假设为独立同分布（IID）的。然而，在深度强化学习中，由于局部探索、有限的采样时间等原因，当从一个动态的环境中每次以序列的形式采集数据时，这种假设便不再成立。非独立同分布数据会导致学到的数据分布与实际的分布有较大差异，从而导致了不准确的值函数估计，进而导致在策略的更新方向上造成较大的误差，从而最终导致了很差的策略。关于非独立同分布数据造成的分布估计的偏差可由下图形象的表示：



10.3.1.1 Trick1：经验回放（Experience Replay，ExR）

- **要解决的挑战**：非独立同分布数据（Non-IID Data）。
- **适用于**：off-policy的强化学习算法。
- **原理**



对于最优的动作值函数来说，它对于各个状态-动作对来说，都满足贝尔曼方程。因此，Q-Learning和它的变体可以重复利用历史的sample，而不需要显式的IS Ratio。因此，每次在不同的behavior策略下采集到的各个sample (s_t, a_t, s_{t+1})都被储存在Replay Buffer中。之后，每次每次更新Q函数时可以从Buffer中随机抽取一个mini-batch的sample（可能来自于不同的behavior策略），然后用这个mini-batch来更新Q函数。随机抽样**不仅能够减少样本之间的相关性，还能平均训练数据的分布**，从而减少了非独立同分布数据带来的问题。

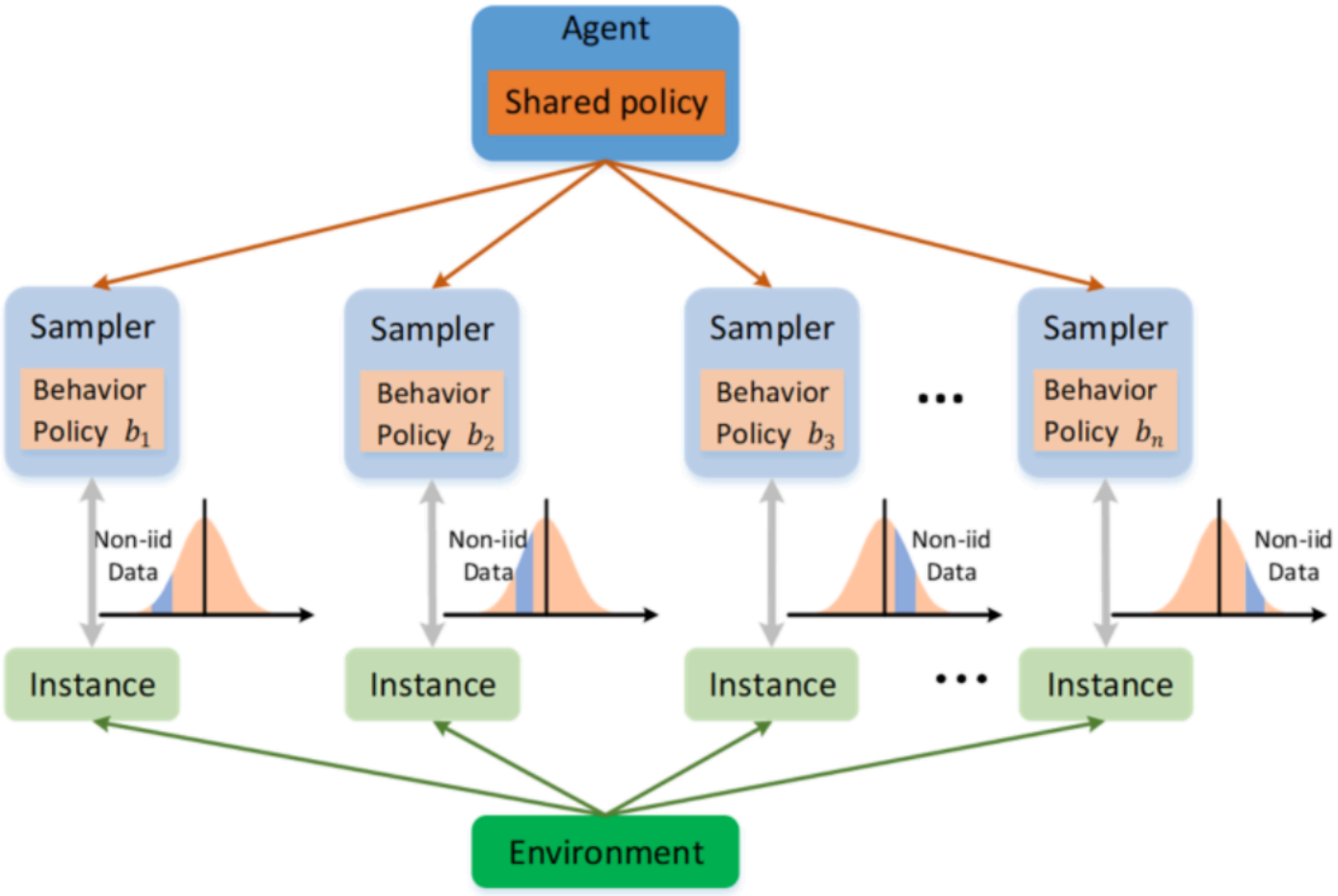
- **好处**
  - **减少样本之间的相关性**
  - **平均训练数据的分布，使分布变化地更平滑**
  - **提升样本效率**：

    > **定义1：样本效率**
    > 达到特定的策略表现需要的新样本的数量。

    显然，经验回放可以提升样本效率，加快策略更新。
- **变式**
  - **优先级经验回放（Prioritized Experience Replay，PExR）**：在经验回放的基础上，根据TD Error的大小来对样本进行优先级采样，从而提高样本效率。TD Error可以用来衡量样本中包含多少有用的信息，可以作为样本重要性的一个指标。

10.3.1.2 Trick2：平行探索（Parallel Exploration，PEx）

- **要解决的挑战**：非独立同分布数据（Non-IID Data）。
- **适用于**：on-policy、off-policy的强化学习算法。
- **原理**

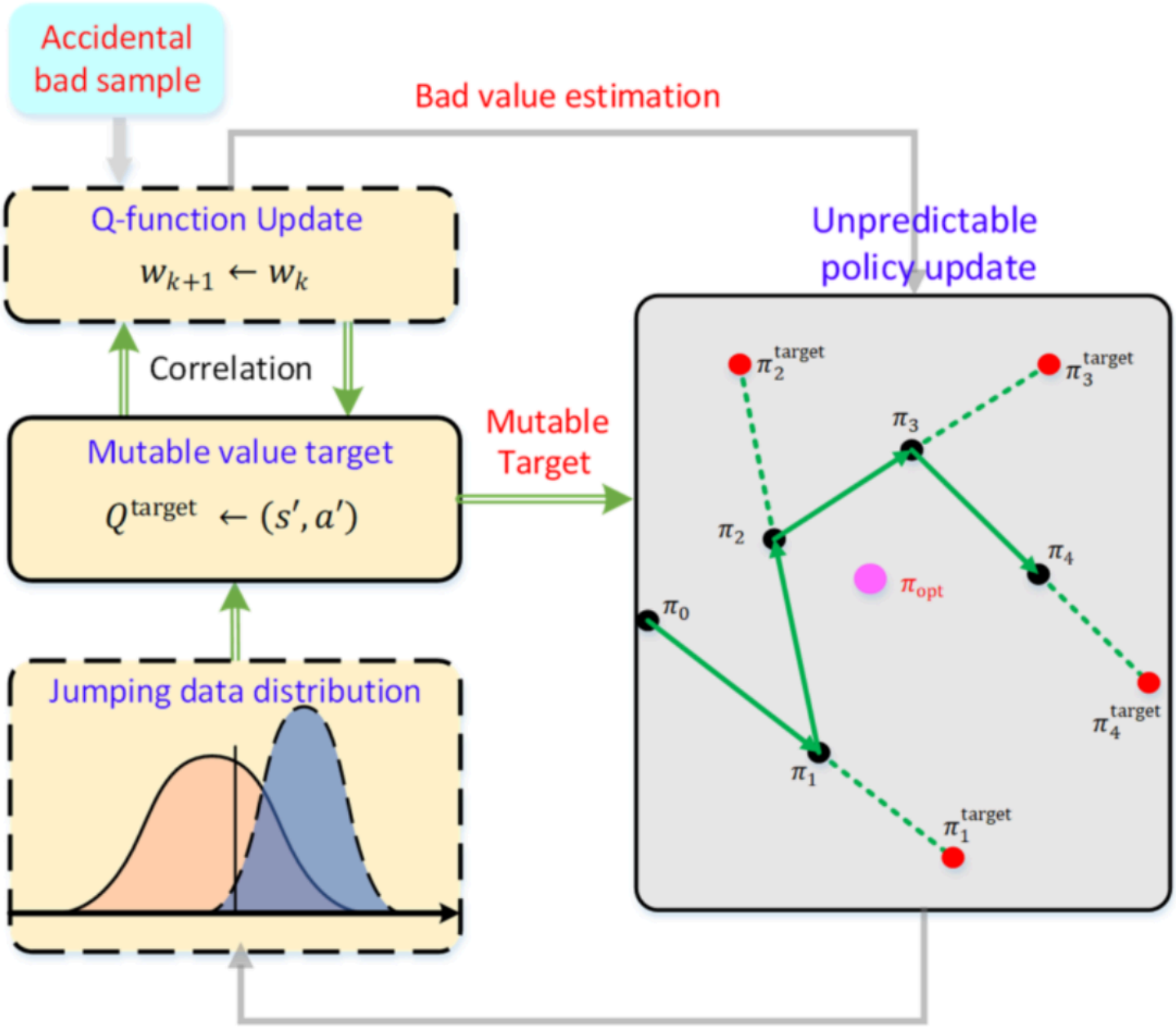


初始化多个环境实例和Sampler，每个Sampler都在各自对应的环境实例上进行探索。这样，采到的总的数据的分布就更接近于真实数据的分布，因为多个Sampler更衣覆盖到整个环境动力学的各个部分，且在off-policy算法中不同的Sampler可以采用不同的behavior策略，从而进一步扩大了对于环境的探索，减少了非独立同分布数据带来的问题。

10.3.2 挑战二：很容易发散（Easy Divergence）及其解决办法

当使用深度神经网络对于强化学习的值函数和策略网络进行近似的时候，很容易发生训练不稳定甚至发散的情况（此时目标值会发生震荡，策略更新也变得不稳定）。对于深度强化学习中经典的Actor-Critic架构来说，造成这种现象的首要原因是**耦合的Critic和Actor的更新时的自激现象**：对于值函数的很差的估计导致了策略更新方向的严重偏离，而策略的更新又会进一步加剧Critic（值函数）的不准确性，从而形成了一个恶性循环。

至于为什么一开始对于值函数的估计会很差，这有多方面原因，如偶然的低质量样本、函数近似的误差、以及不充足的Critic更新等。这里以偶然的低质量样本为例，来看看为什么会引起发散。



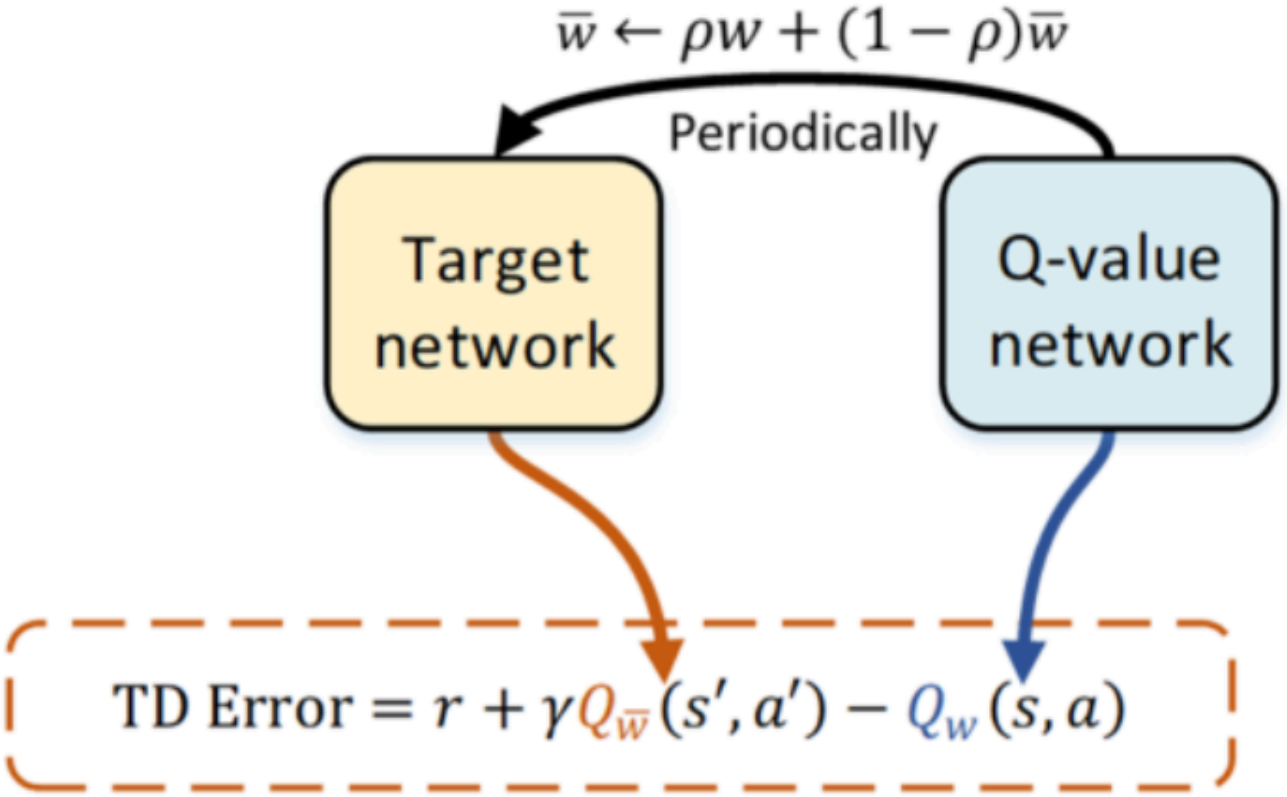
在上图中，首先，由于偏重于探索的策略或者环境的随机性等原因，Sampler采到了一个很差的样本，这个样本可能会导致值函数的估计质量恶化。而质量变低的值函数又为策略更新提供了一个错误的方向，从而导致了策略的更新的不可预测。而使用这样的策略对于环境进行探索又会导致采到的样本的分布出现跳变，进而导致了采到的样本质量更差，这样就形成了一个恶性循环。

造成发散的另一个原因是Actor更新的步长太大，这样会导致策略的更新来回震荡。

为了解决这个问题，可以采用稳定的值函数估计、降低值函数近似误差以及控制过快的策略更新等方法。

10.3.2.1 Trick3：目标网络（Separated Target Network，STN）

- **要解决的挑战**：很容易发散（Easy Divergence）。
- **原理**：



从Q网络中分离出一个目标网络，用于计算Target Q值，来降低Q值和Target Q值之间的相关性。具体来说，就是我们有两个Q网络，一个online Q网络和一个target Q网络。Target Q网络负责在计算TD Error的时候给出Target Q值：

$$Q_{\overline{w}}^{\text{target}} = r + \gamma Q_{\overline{W}}(s', a'),$$

其中，$Q_{\overline{w}}^{\text{target}}$是Target Q值，$Q_{\overline{W}}(s', a')$是Target Q网络的输出，$\overline{w}$是Target Q网络的权重。这样，Target Q值就不再依赖于online Q值。而Target Q网络也不是像online Q网络一样每次都依赖BP更新，而是**每隔一段时间才通过从在线网络复制权重的方式更新一次**：

$$\overline{w} \leftarrow \rho w + (1-\rho)\overline{w},$$

其中，ρ是一个权重，用于平衡更新时Target Q网络的历史权重和来自于online Q网络的新权重之间的关系。特别的，当$\rho = 1$时，Target Q网络每次更新时直接从online Q网络中复制权重。

需要注意的是，尽管可以降低发散的风险，但是Target Q网络的低频更新也会增加训练时间。因此找到一个合适的更新间隔n是非常重要的。

10.3.2.2 Trick4：延迟策略更新（Delayed Policy Update，DPU）

- **要解决的挑战**：很容易发散（Easy Divergence）。
- **原理**：在Actor-Critic架构中，值函数和策略函数在每轮迭代的时候都会更新，这也是造成发散的一个原因。但是其实两者对于更新频率的需求并不一样。Critic（值函数）需要高频的更新来获得准确的值函数估计，而Actor（策略函数）则不需要这么频繁的更新。因此一种简单的办法是，**每次在若干次Critic更新之**

后再更新Actor。这背后的原理是，Critic的多次更新保证了近似误差足够小，只哟这样才能保证高质量的策略更新。

10.3.2.3 Trick5：受约束的策略更新（Constrained Policy Update，CPU）

- **要解决的挑战**：很容易发散（Easy Divergence）。
- **原理**：避免策略变化太快从而引起不稳定甚至震荡。具体来说，可以通过某种标准来衡量相邻两次策略的变化幅度，限制其不超过一个阈值：

$$\left\|\pi_{k+1}-\pi_k\right\| \leq \delta_\pi,$$

这里的$\left\|\cdot\right\|$是某种衡量标准，可以取为向量的范数、交叉熵、KL散度等。

10.3.2.4 Trick6：裁剪的Actor目标（Clipped Actor Criterion，CAC）

- **要解决的挑战**：很容易发散（Easy Divergence）。
- **原理**：与直接限制相邻两个策略之间的“距离”类似，也可以通过限制相邻两个策略在其Actor Cost Function的值之间的差值来限制策略的变化幅度：

$$0 \leq J_{\text{Actor}}(\pi_{k+1}) - J_{\text{Actor}}(\pi_k) \leq \delta_J,$$

这里的J_{Actor}是Actor的Cost Function，δ_J是一个阈值，不超过这个阈值从而保证策略不会变化的太快，而大于等于0则保证了策略单调变好。

10.3.3 挑战三：过估计（Overestimation）及其解决办法

深度强化学习中的过估计问题指的是，算法会在某些状态-动作对时学到不切实际的过高的Q值，造成了对于Q值估计的严重失真和学到的策略的不可用。对于Q值过估计的原因在于Q-Learning系列算法中的max操作，只要使用了这个max操作，不管TD Error是正还是负，都会导致Q值的过估计。

其实，应该澄清的是，Q值的过估计本身并不是一个问题，只要对于所有的状态-动作对都均匀地高估，那么动作被选择的相对优先级并不会被破坏。这里坏就坏在，在实际的情况中，只有在少数的状态-动作对上才会被高估，这就导致了动作的相对优先级顺序被打乱，从而导致了学到的策略完全不可用。

下面就来看看过估计问题是如何造成的，这里的讲解参考了DSAC算法里的证明过程，详细的证明过程可以参考DSAC论文。首先，对于Q-Learning类算法，其参数更新可以写成如下形式：

$$w' \leftarrow w + \alpha \left(Q_w^{\text{target}}(s,a) - Q_w(s,a)\right) \nabla Q_w(s,a),$$

但是在实际中，不管是$Q_w(s,a)$还是$\nabla Q_w(s,a)$都存在着由状态测量不准或者函数近似精度等导致的近似误差。我们可以把当前权重为w的Q函数表示为其真实值与随机误差的和：

$$Q_w(s,a) = Q_{\hat{w}}(s,a) + \epsilon(s,a),$$

其中，$\hat{w}$是真实的权重，$\epsilon(s,a)$是随机误差。接下俩，我们把$\hat{w}'$记为从当前的w更新得到的下一个真实权重，那么可证明关于Q值的估计值与真实值的误差$\Delta(s,a)$可以写成如下形式：

$$\Delta(s,a) = \alpha\gamma \cdot \delta(s,a) \cdot \|\nabla Q_{\hat{w}'}(s,a)\|_2,$$
$$\delta(s,a) = \mathbb{E}_{s'\sim\mathcal{P}}\left\{\mathrm{E}_{\epsilon}\left\{\max Q_w(s',a')\right\} - \max \mathrm{E}_{\epsilon}\{Q_w(s',a')\}\right\},$$

可证明$\delta(s,a) \geq 0$，因此$\Delta(s,a) \geq 0$，即Q值的估计值总是高于真实值。又因为不管是Target Error还是Q函数的梯度都在不同的状态-动作对上都是高度不可预测的，因此这种过估计不是均匀的。在实际中，这种过估计还会随着计算Target的时候bootstrapping机制而传播累加，加剧了过估计的问题。

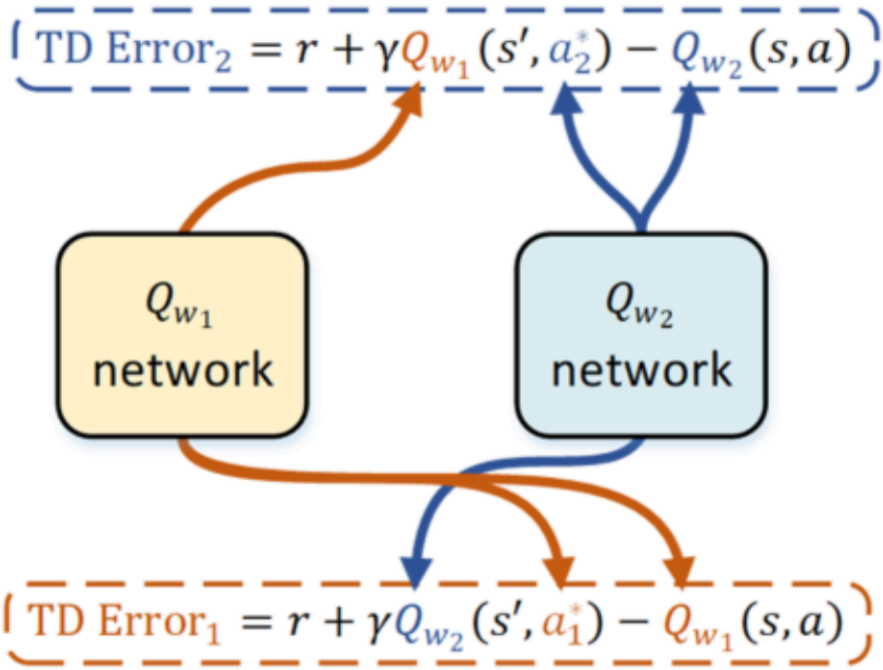
10.3.3.1 Trick7：双Q函数（Double Q-Functions，DQF）

- **要解决的挑战**：过估计（Overestimation）。
- **原理**：将计算Q-Target的过程解耦为两个独立的步骤：动作选择和动作评估。具体来说，我们需要维护两个Q函数Q_1和Q_2，这两个Q函数是独立训练的。在每次更新时，对于其中一个Q函数，先使用它自己来给出动作选择（以Q_1为例）：

  $$a_1^* = \arg\max_{a'} Q_{w_1}(s',a'),$$

  然后使用另一个Q函数来给出动作评估以更新Q函数：

  $$Q_{w_1}(s,a) \leftarrow Q_{w_1}(s,a) + \alpha\left(r + \gamma Q_{w_2}(s',a_1^*) - Q_{w_1}(s,a)\right).$$



  为了实现上述过程，需要两个Q函数足够的相互独立。可以通过使用两个平行采样的包含不同样本的数据集来实现。

最后来解释一下为什么双Q函数可以降低过估计。这是因为这种方法可以提供一定程度的低估（underestimation），从而一定程度上抵消了过估计的影响。来简单证明一下。首先，假设Q_{w_1}和Q_{w_2}均是无偏估计，即：

$$\mathbb{E}\left\{Q_{w_1}(s,a)\right\}=\mathbb{E}\left\{Q_{w_2}(s,a)\right\}=q^{\pi}(s,a).$$

之后，定义集合$\mathcal{M}$为各个状态下最优的动作选择集合：

$$\mathcal{M}\overset{\text{def}}{=}\left\{a_{\text{opt}}|a_{\text{opt}}=\arg\max_{a'}q^{\pi}(s',a')\right\}.$$

注意，这里的a_{opt}和a^*的区别在于前者是能够最大化特定状态s'下的q的真值，而后者是能够最大化特定状态s'下用来拟合的Q函数网络。这样，对于状态s'时最佳动作a_1^*的Q求期望，可得：

$$\begin{aligned}
&\mathbb{E}\{Q_{w_2}(s',a_1^*)\}\\
&=\Pr\{a_1^*\in\mathcal{M}\}\mathbb{E}\{Q_{w_2}(s',a_1^*)|a_1^*\in\mathcal{M}\}+\Pr\{a_1^*\notin M\}\mathbb{E}\{Q_{w_2}(s',a_1^*)|a_1^*\notin\mathcal{M}\}\\
&=\Pr\{a_1^*\in\mathcal{M}\}\max_{a'}q^{\pi}(s',a')+\Pr\{a_1^*\notin\mathcal{M}\}\mathbb{E}\{Q_{w_2}^*(s',a_1^*\notin\mathcal{M}\}\\
&\leq\Pr\{a_1^*\in\mathcal{M}\}\max_{a'}q^{\pi}(s',a')+\Pr\{a_1^*\notin\mathcal{M}\}\max_{a'}q^{\pi}(s',a')\\
&=\max_{a'}q^{\pi}(s',a').
\end{aligned}$$

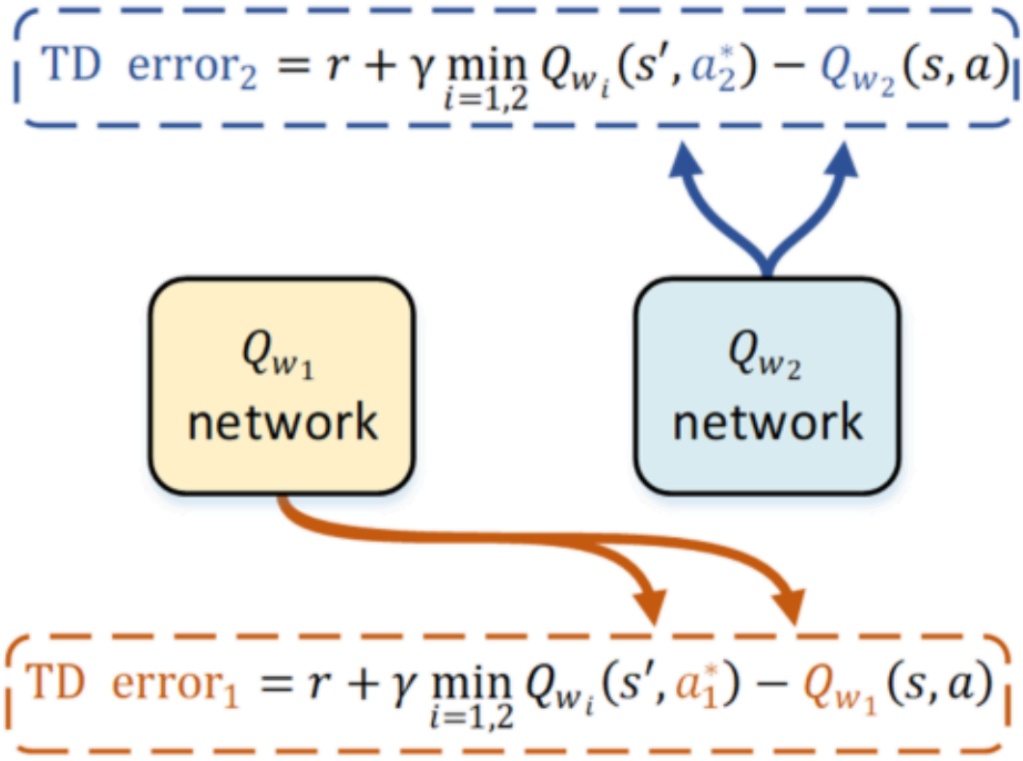
这里第一步主要是根据a_1^*是否属于该状态下a_{opt}的集合$\mathcal{M}$来划分的。最后，即可证明Q函数在特定状态s'下的最优动作的估计值的期望小于等于该状态下的真值的最大值，即发生了低估。

另外注意，这里的双生Q函数技巧（DQF）与之前的目标网络（STN）技巧可以融合在一起（也是现在的常用做法），即在线网络和目标网络可分别看做两个Q函数网络。

10.3.3.2 Trick8：双Q函数取最小值（Bounded Double Q-Functions，BDQ）

- **要解决的挑战**：过估计（Overestimation）。
- **原理**：进一步认为构造出低估。具体来说没在第一步通过各自的Q网络选择动作后，不使用另一个Q网络来构造目标，而是使用两个Q网络的最小值来构造目标：

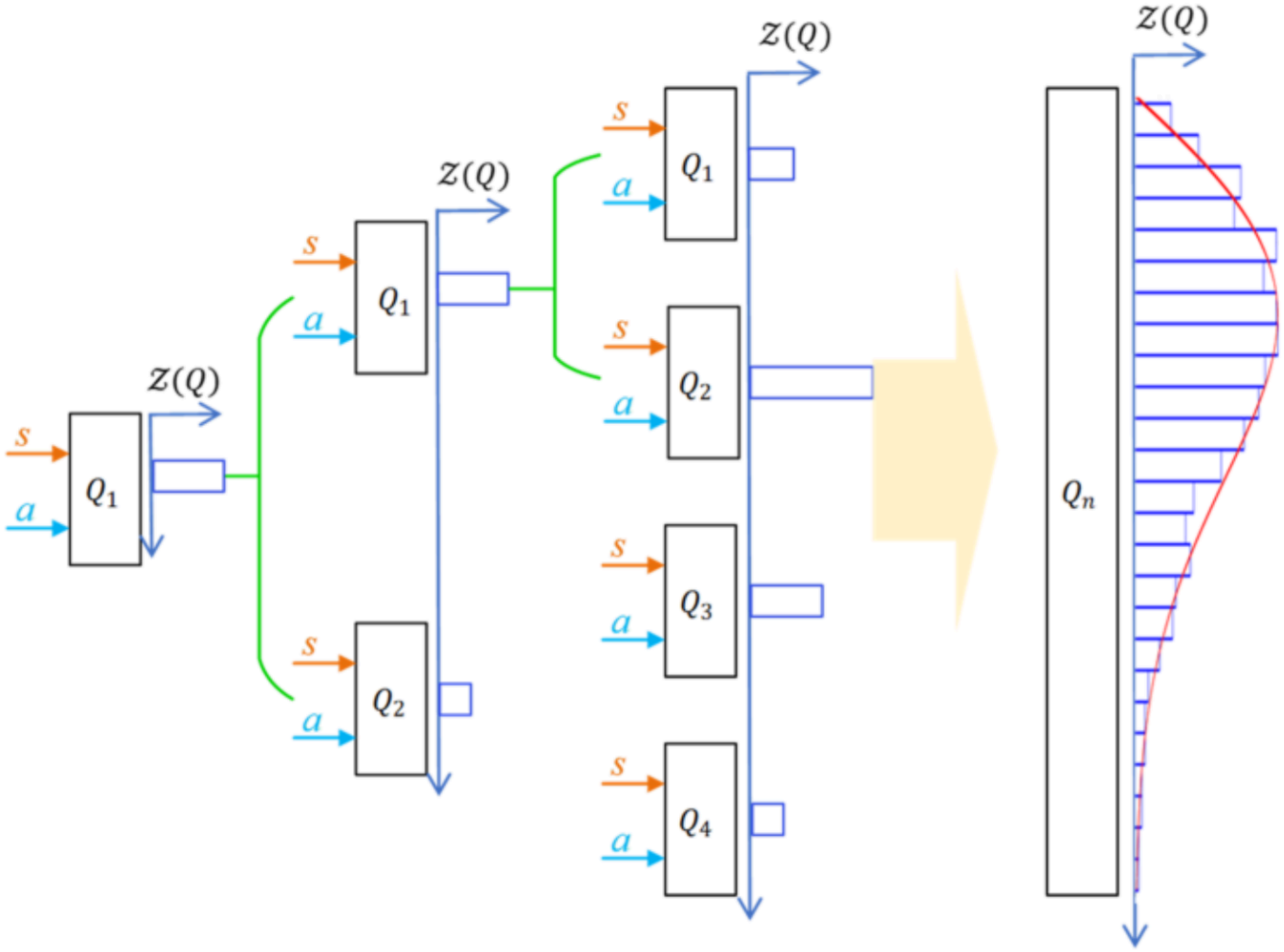
$$\begin{aligned}
Q_{W_1}^{\text{target}}(s,a)&=r+\gamma\min_{i=1,2}Q_{w_i}(s',a_1^*),\\
Q_{W_2}^{\text{target}}(s,a)&=r+\gamma\min_{i=1,2}Q_{w_i}(s',a_2^*).
\end{aligned}$$



这样只要两个Q网络提供的估计值中至少有一个低于真值就可以提供低估的目标值。这样就可以进一步降低过估计的问题。

10.3.3.3 Trick9：分布式回报函数（Distributed Return Function，DRF）

- **要解决的挑战**：过估计（Overestimation）。
- **原理**：对两个独立Q函数的自然延伸。既然能够学习两个Q函数来降低过估计，为什么不能学习无穷多个呢？而无穷多个Q函数就构成了一个分布。



具体地，把状态-动作对的回报定义为：

$$Z(s,a) = r_{ss'}^{a} + \gamma G_{t+1},$$

这里的G_{t+1}是从s'开始的长期回报。由于环境转移模型、策略等蕴含的随机性，这个回报是一个随机变量，因此$Z(s,a)$也是一个随机变量。并且可证明，Q函数是$Z(s,a)$的期望：

$$Q(s,a) = \mathbb{E}_{Z\sim\mathcal{Z}^{\pi}}\{Z(s,a)\}.$$

其中$\mathcal{Z}^{\pi}$是$Z(s,a)$的分布。这样的分布相当于学习得到了无穷个Q函数，这被证明可以有效的降低过估计。假如我们使用高斯分布来近似$Z(s,a)$的分布，而高斯分布的均值和标准差分别记作$q(s,a)$和$\sigma(s,a)$，那么可证明使用分布式回报函数前后的过估计误差$\Delta(s,a)$和$\bar{\Delta}(s,a)$之间的关系为：

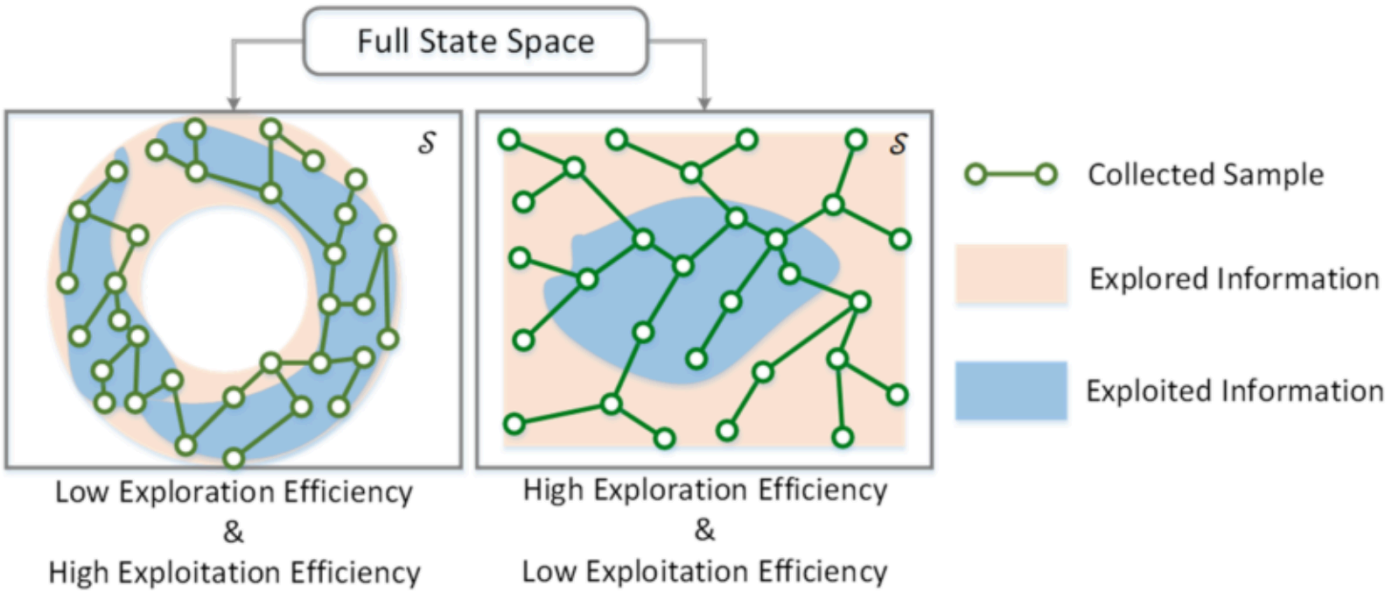
$$\overline{\Delta}(s,a) = \frac{\Delta(s,a)}{\sigma(s,a)^2}.$$

并且可证明在值估计不准确和未来状态不确定的区域，$\sigma(s,a)$更大，因此相对于原始的过估计误差，新的过估计误差能有效减小。

10.3.4 挑战四：样本效率低（Sample Inefficiency）及其解决办法

样本效率被定义为**达到特定的策略表现需要的新样本的数量**。强化学习中的样本指通过与环境交互得到的转移三元组(s_t, a_t, s_{t+1})。由定义可知重复使用旧的样本不会影响样本效率，只有与环境交互得到的新样本才会影响。

样本效率取决于两个方面：探索效率（Exploration Efficiency）和利用效率（Exploitation Efficiency）。前者指RL Agent是否可以通过更少的样本来覆盖整个环境。后者则指怎么利用收集到的样本更有效率的来学习特定区域。



实现更高的探索效率的方式包含使用更多初始状态随机的Agent来进行探索、使用熵正则项和软值函数等方式来鼓励Agent进行探索等。

实现更高的利用效率的方式包含经验回放等、优先级经验回放等方式来更高效的利用收集到的样本。

10.3.4.1 Trick10：熵正则化（Entropy Regularization，EnR）

- **要解决的挑战**：样本效率低（Sample Inefficiency）。
- **原理**：在强化学习总的目标函数里面添加一个熵正则项来鼓励探索：

$$J(\theta) = \mathbb{E}_{\pi_\theta}\{v^{\pi_\theta}(s) + \alpha\mathcal{H}\left(\pi_\theta(\cdot|s)\right)\},$$

这里的$\mathcal{H}(\cdot)$是熵函数，用于衡量策略的随机性，α是温度系数，用于平衡熵正则项和原始的目标函数之间的关系。

熵正则项技术有助于鼓励探索，捕捉更广泛的潜在动作模态病防治过早收敛到次优解。

10.3.4.2 Trick11：软值函数（Soft Value Function，SVF）

- **要解决的挑战**：样本效率低（Sample Inefficiency）。
- **原理**：与仅仅在优化目标中添加熵相比，把奖励信号直接使用熵项来进行增强是一个更好的方法。这也就是最大熵强化学习的思想。新的奖励信号为：

$$r_{\text{aug}}(s,a,s') = r(s,a,s') + \alpha\mathcal{H}\left(\pi(\cdot|s)\right).$$

那么对应的值函数如下：

$$v^{\pi}(s) = \mathbb{E}_{\pi}\left\{\sum_{i=0}^{\infty}\gamma^{i}\left(r_{t+i} + \alpha\mathcal{H}\left(\pi(\cdot|s_{t+i})\right)\right)\middle| s_t = s\right\},$$

$$q^{\pi}(s,a) = \mathbb{E}_{\pi}\left\{\sum_{i=0}^{\infty}\gamma^{i}r_{t+i} + \alpha\sum_{i=1}^{\infty}\gamma^{i}\mathcal{H}\left(\pi(\cdot|s_{t+i})\right)\middle| s_t = s, a_t = a\right\}.$$

进一步的，亦有下述self-consistency的关系：

$$v^{\pi}(s) = \mathbb{E}_{a\sim\pi}\{q^{\pi}(s,a)\} + \alpha\mathcal{H}\left(\pi(\cdot|s)\right).$$

在探索较少的区域，原本的reward信号比后面的熵正则项小得多，因此策略的方差会变大来增加熵项以促进探索。而在探索较多的区域，则会倾向于利用。这样就可以实现对于未充分探索的区域进行更多的探索的目的。

10.4 深度强化学习算法

深度强化学习算法（Deep Reinforcement Learning，DRL）将深度神经网络引入强化学习中。主流的深度强化学习算法及其利用的前述技巧如下表所示：

Alg.	ExR	PEx	STN	DPU	CPU	CAC	DQF	BDQ	DRF	EnR	SVF	π
DQN	★		★									off
Dueling DQN	●		●									off
DDQN	●		●			★						off
TRPO					★							on
PPO					★							on
A3C		★								★		on
DDPG	●		●			●						off
TD3	●		●	●		●	★					off
SAC	●		●			●	●				★	off
DSAC	●	●	●	●		●		★		●		off

★ Formally proposed for the first time
● Inherited tricks from previous DRLs

其中，上述各个Trick的缩写含义如下表所示：

Trick	Full Name
ExR	Experience Replay
PEx	Parallel Exploration
STN	Separated Target Network
DPU	Delayed Policy Update
CPU	Constrained Policy Update
CAC	Clipped Actor Criterion
DQF	Double Q-Functions
BDQ	Bounded Double Q-Functions
DRF	Distributed Return Function
EnR	Entropy Regularization
SVF	Soft Value Function

10.4.1 深度Q网络（Deep Q-Network，DQN）

DQN是第一个成功的深度强化学习算法。它的原始版本在2013年提出，但是存在容易发散的问题。2015年的改进版**通过引入经验回放（ExR）和目标网络（STN）解决了不稳定的问题**。

在DQN中，每条经验e_t被定义为：

$$e_t = \left(s_t, a_t, r_t, s_{t+1}\right).$$

而负责储存经验的Replay Buffer被定义为：

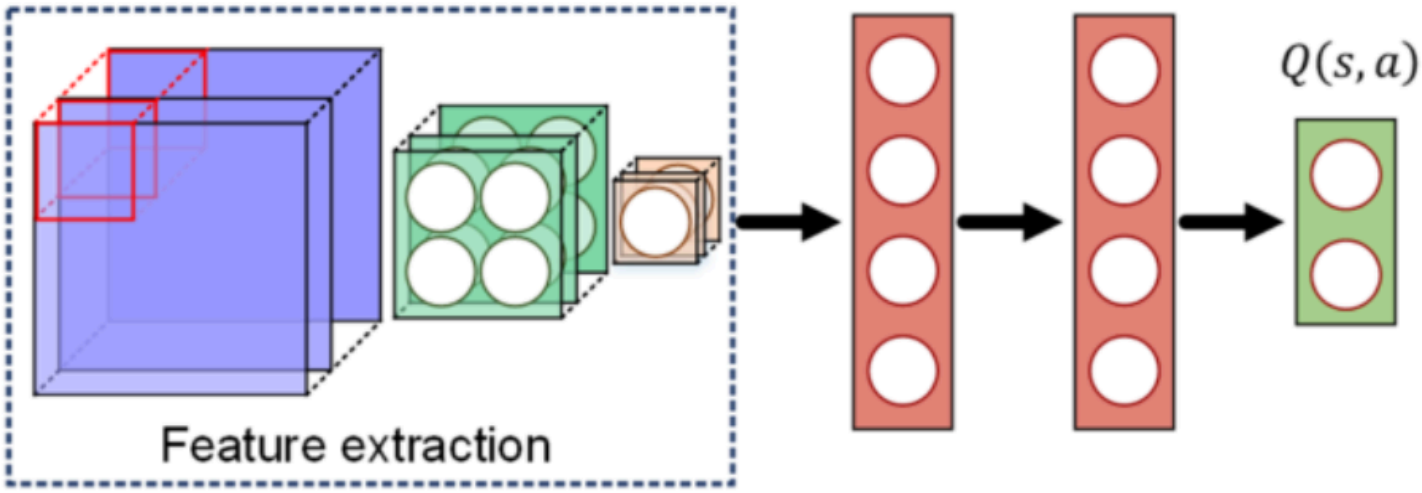
$$\mathcal{D} = \left\{e_0, \ldots, e_t, \ldots, e_N\right\}.$$

DQN每次更新使用从Replay Buffer中随机抽取的mini-batch来进行。更新方式为最小化均方贝尔曼误差（单步TD Target与当前Q值之间的均方差）：

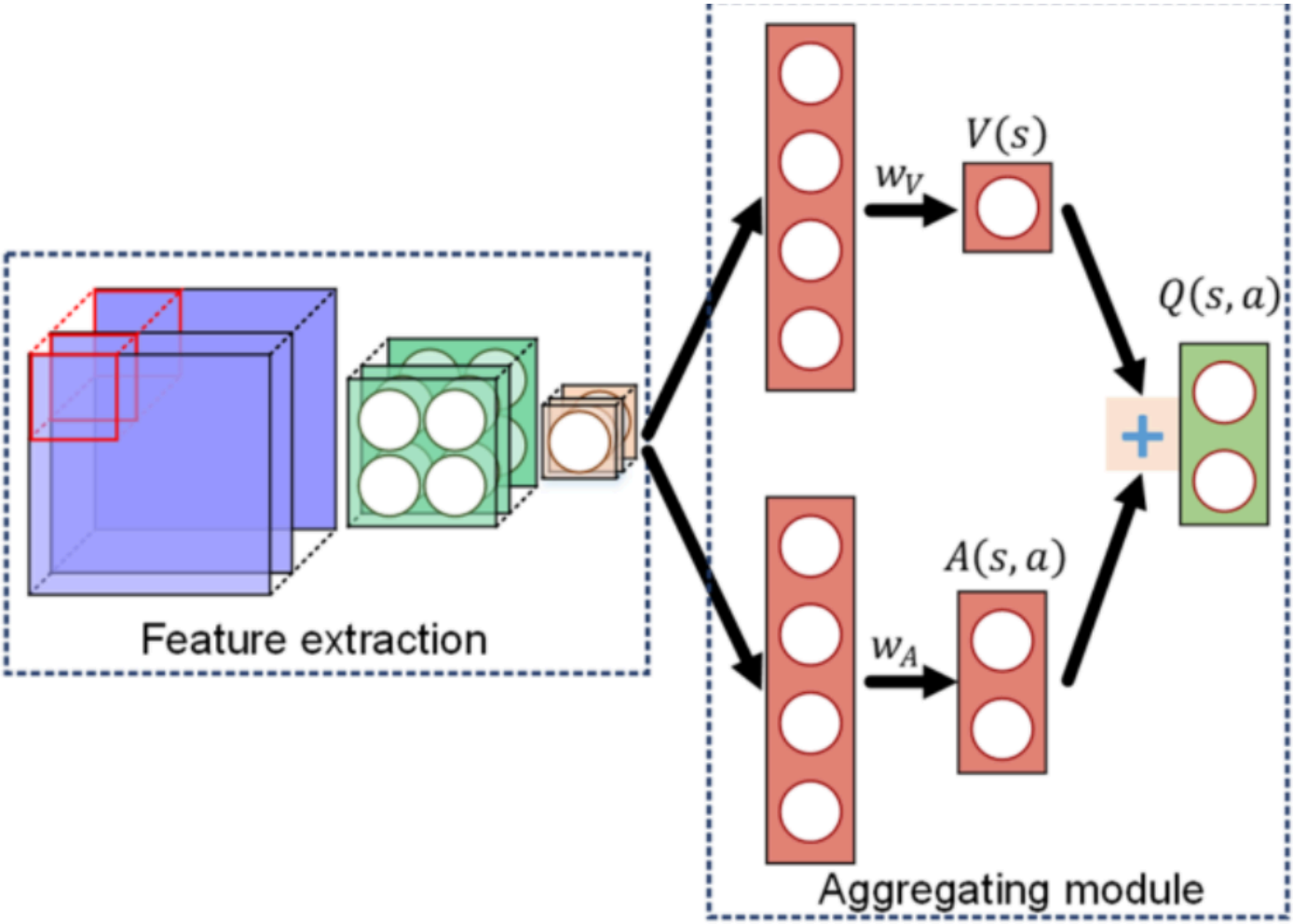
$$J(w) = \mathbb{E}_{s,a,s' \sim \mathcal{D}_{\text{Replay}}} \left\{ \left(r + \gamma \max_{a'} Q_{\overline{W}}(s', a') - Q_w(s, a) \right)^2 \right\}.$$

值得指出的是，经验回放技巧是DQN成功的关键。直接学习连续的、来自同一条轨迹上的样本会导致训练的不稳定性。使用经验回放技巧后可以打破样本之间的相关性。另外，任何不依赖重要性比例（Importance Sampling Ratio，ISR）的off-policy算法都可以使用经验回放技巧来提升样本效率。因此，大部分具有动作值函数的算法的可使用这一技巧，而大部分基于状态值函数的算法则不能使用这一技巧。

10.4.2 Dueling DQN



(a) Single-stream Q-network (i.e., normal network)



(b) Two-stream Q-network (i.e., dueling network)

Dueling DQN**将DQN中的动作值函数的估计分解为两个并行的网络：状态值函数网络和优势函数网络**。状态值函数网络只关注状态本身的价值而不在乎具体的动作的影响；优势函数网络则关注在指定状态下，选择特定动作的优势。之后，再通过这两者可以合成出动作值函数。这种设计有助于利用同一个状态下不同动作的Q值来学习V值（显然直接对于Q值进行学习的话就无法利用同状态下其它Q值的信息）。另外，同一状态下不同动作的Q值之间的差异比它们的幅值小得多，这样的学习方式也有助于算法对噪声更加鲁棒。下面来看看具体的原理。

我们把V值函数和优势函数分别记作$V(s;w_V)$、$A(s,a;w_A)$，那么如何获得动作值函数呢？一个简单的想法是直接相加：

$$Q(s,a;w_V,w_A)=V(s;w_V)+A(s,a;w_A).$$

但是这里就存在一个问题，即可辨识性问题，具体来说，这样直接相加造成即使对于同一个状态-动作对(s,a)的动作值函数，其分解也不是唯一的（只需要V值函数和优势函数分别加上和减去一个常数就可以了）。这样做的坏处有很多，比如网络可能退化为传统DQN，V值函数被学习为0，仅由优势函数表示Q值。
状态价值的估计失去意义，无法体现状态本身的“整体好坏”，导致模型无法有效泛化到相似状态。为了解决这个问题，主要有两种方法：

- **减去均值**
  - **做法**

$$Q(s,a)=V(s)+\left(A(s,a)-\frac{1}{|\mathcal{A}|}\sum_{a'}A(s,a')\right).$$

  - **约束条件**：优势函数的均值为零，即：

$$\frac{1}{|\mathcal{A}|}\sum_{a'}A(s,a')=0.$$

  - **唯一性证明**：在这种情况下，对于$Q(s,a)$的分解是唯一的（$V(s)$唯一），证明如下：

$$\frac{1}{|\mathcal{A}|}\sum_{a}Q(s,a)=V(s)+\frac{1}{|\mathcal{A}|}\sum_{a}\left(A(s,a)-\frac{1}{|\mathcal{A}|}\sum_{a'}A(s,a')\right)=V(s).$$

    此时$V(s)$等于状态s下所有动作的Q值的均值，$A(s,a)$则表示动作a相对于均值的优势。
- **减去最大值**
  - **做法**

$$Q(s,a)=V(s)+\left(A(s,a)-\max_{a'}A(s,a')\right).$$

  - **约束条件**：优势函数的最大值为零，即：

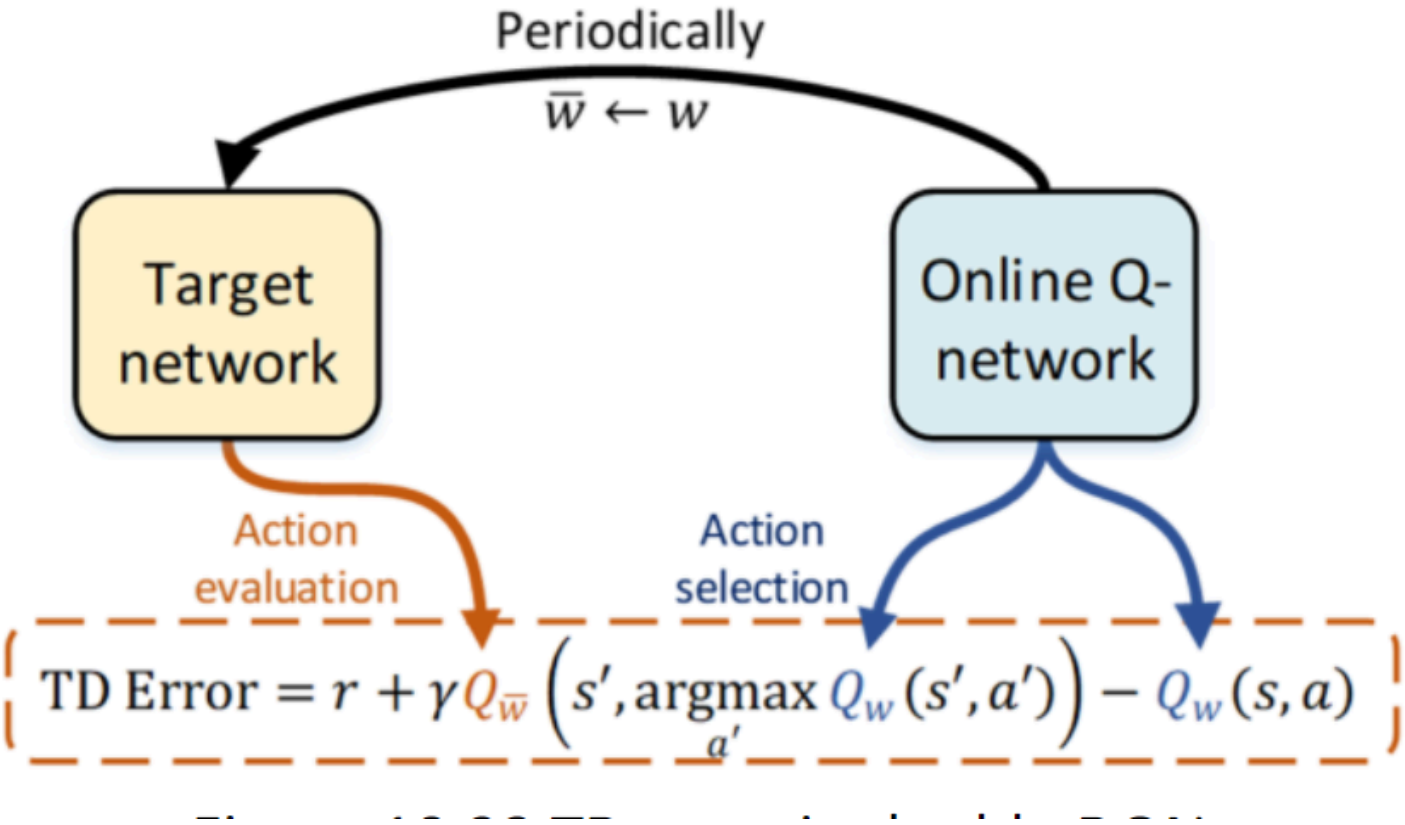
$$\max_{a'}A(s,a')=0.$$

  - **唯一性证明**：在这种情况下，对于$Q(s,a)$的分解是唯一的（$V(s)$唯一），证明如下：

$$\max_{a}Q(s,a)=V(s)+\max_{a}\left(A(s,a)-\max_{a'}A(s,a')\right)=V(s).$$

    此时$V(s)$等于状态s下所有动作的Q值的最大值，$A(s,a)$则表示动作a距离最大值的差距。

10.4.3 Double DQN（DDQN）



DDQN与DQN几乎一模一样，只不过额外使用了双Q函数（DQF）来降低过估计的问题。它也是有两个网络，在线网络（Online Network）和目标网络（Target Network），但是在构造TD Error的时候与DQN不同，如下表所示：

DQN	DDQN
$Q_{\overline{w}}^{\text{target}} = r + \gamma \max_{a'} Q_{\overline{w}}(s', a') - Q_w(s, a)$	$r + \gamma Q_{\overline{w}}\left(s', \underset{a'}{\operatorname{argmax}} Q_w(s', a')\right) - Q_w(s, a)$

可以看出，虽然二者在构造TD Error的时候都使用了目标网络，但是前者在构造TD Target的时候的最佳动作是通过目标网络来选择的（即$a^* = \arg\max_{a'} Q_{\overline{w}}(s', a')$），而后者则是通过在线网络来选择的（即$a^* = \arg\max_{a'} Q_w(s', a')$）。这就相当于把STN和DQF结合在了一起。而除此之外DDQN的网络结构和训练目标与DQN完全一样。

10.4.4 TRPO

TRPO算法旨在解决原始的策略梯度算法在更新时参数变化过快的导致策略不稳定的问题。通过引入minorize-maximization (MM) 优化，TRPO算法可以解决策略变化过快的问题并从理论上保证策略单调变好。它使用了受约束的策略更新（CPU）技巧来限制策略的变化幅度，具有一个状态值函数网络（V函数）以及一个策略网络。

MM优化本质上是找到原目标函数的一个下界（lower bound）作为替代函数，然后优化这个下界函数。TRPO算法的优化过程如下：

$$\theta \leftarrow \arg\max_{\theta} \mathbb{E}_{s\sim d_{\pi_{\theta_{\text{old}}}}, a\sim\pi_{\text{old}}} \left\{ \frac{\pi_\theta(a|s)}{\pi_{\text{old}}(a|s)} A^{\pi_{\text{old}}}(s,a) \right\}$$
$$\text{s.t.}$$
$$\overline{D}_{\text{KL}}(\pi_{\text{old}}, \pi_\theta) \leq \delta_\pi,$$

这里的$A^{\pi_{\text{old}}}(s,a)$是优势函数：$A^{\pi_{\text{old}}}(s,a) = r + \gamma V^{\pi_{\text{old}}}(s') - V^{\pi_{\text{old}}}(s)$。而$\overline{D}_{\text{KL}}(\pi_{\text{old}}, \pi_\theta)$是新旧策略之间的KL散度：$\overline{D}_{\text{KL}}(\pi_{\text{old}}, \pi_\theta) = \mathbb{E}_{s\sim d_{\pi_{\text{old}}}} \{D_{\text{KL}}(\pi_{\text{old}}(\cdot|s), \pi_\theta(\cdot|s))\}$。

TRPO算法在使用时需要调整的超参数比较少，因此具有一定实用性。且其单调改进的性质保证了算法的鲁棒性。但是，TRPO算法是on-policy算法，不能使用经验回放技巧，样本效率较低。且无法充分探索环境导致了算法容易陷入局部最优。

10.4.5 PPO

TRPO算法的计算负担很大。原始版本在求梯度时需要计算Hessian矩阵的逆，而即使改进后的版本仍然需要通过共轭梯度法来求解。因此，PPO算法被提出来解决上述问题。它的架构与TRPO完全一样（一个V函数网络和一个策略网络），但是将TRPO每次优化时需要求解的带约束优化问题转化为直接在优化目标中通过设置上下界来进行裁剪从而保证策略不会变化的太快。PPO算法使用了裁剪的Actor准则（CAC）技巧。

具体来说，PPO算法的优化目标为：

$$\theta \leftarrow \arg\max_{\theta} \mathbb{E}_{s\sim d_{\pi_{\text{old}}}, a\sim\pi_{\text{old}}} \left\{ \min\left(\rho_{t:t} A^{\pi_{\text{old}}}(s,a), \rho_{\text{clip}} A^{\pi_{\text{old}}}(s,a) \right) \right\},$$
$$\rho_{\text{clip}} \overset{\text{def}}{=} \text{clip}(\rho_{t:t}, 1-\epsilon, 1+\epsilon),$$

这里的$\rho_{t:t} = \frac{\pi_\theta(a|s)}{\pi_{\text{old}}(a|s)}$为IS Ratio，$\rho_{\text{clip}}$为裁剪后的IS Ratio，$\epsilon$为裁剪的范围。优化目标中min的括号里面的前面那项就是TRPO算法里原始的优化目标，而后面那项则是裁剪后的优化目标。

值得注意的是，$\min\left(\rho_{t:t} A^{\pi_{\text{old}}}(s,a), \rho_{\text{clip}} A^{\pi_{\text{old}}}(s,a) \right)$仍然是原始的优化目标的下界，因此单调改进的性质仍然成立。由此可知PPO算法是一个计算上更高效的一阶优化算法。

10.4.6 A3C（Asynchronous Actor-Critic）

A3C算法是一个**on-policy算法**，其最主要的贡献在于提出了异步平行探索和更新的思想，这启发了后续的分布式强化学习算法。A3C算法使用了平行探索（PEx）和熵正则化（EnR）技巧。

A3C算法的Sampler和Learner都可以设计成并行的都需要独立的更新自己的参数。同时A3C算法还有一个全局的Learner（Global Learner）和若干局部的Learner（Local Learner），它们与全局Learner的结构相同（一个V函数网络和一个策略网络）。**局部Learner独立地更新自己的参数，并定期将从全局Learner那里获取最新的参数版本；全局Learner则一旦发现有局部Learner发生了参数更新就会那个局部Learner那里获取参数**。

A3C的优化目标为：

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi \left\{ \nabla_\theta \log \pi_\theta \left(a|s \right) A_w(s,a) + \alpha \nabla_\theta \mathcal{H}\big(\pi_\theta(\cdot|s)\big) \right\},$$

这里加入了熵正则项来鼓励探索。里面的$A_w(s,a)$是优势函数：

$$A_w(s_t,a_t) = \sum_{i=0}^{n-1} \gamma^i r_{t+i} + \gamma^n V_w(s_{t+n}) - V_w(s_t).$$

值得注意的是，因为A3C是**on-policy算法**，因此它不能利用历史样本，因此也无法使用经验回放技巧。它主要通过异步平行机制来取得较高的样本效率和较好的表现的。这种异步并行的方式尤其适合具有多个CPU核心的机器。

10.4.7 DDPG（Deep Deterministic Policy Gradient）

DQN和DDQN算法均**不具有显式的策略函数**，直接通过greedy search的方法，最大化动作值函数来选择动作。但是这样带来的坏处是这样*__动作空间只能是离散的__，而无法扩展到连续的动作空间。DDPG是对于DQN的扩展，可以处理连续的动作空间。它具有两个Q函数网络（Q^{online}和Q^{target}）和两个策略网络（Q^{online}和Q^{target}）。Target网络通过定期从Online网络那里获取来更新。具体地，DDPG算法使用了经验回放（ExR）、分离目标网络（STN）、双Q函数（DQF）技巧。

首先，因为DDPG是对于DQN的扩展，因此它自然也是off-policy算法，就可以使用经验回放技巧。

其二，DDPG算法使用了Slow Update的方法来更新目标网络：

$$\begin{aligned} \overline{w} &\leftarrow \rho_w w + (1-\rho_w)\overline{w}, \\ \bar{\theta} &\leftarrow \rho_\theta \theta + (1-\rho_\theta)\bar{\theta}, \end{aligned}$$

这里的$\overline{w}$和$\overline{\theta}$分别是Q函数网络和策略网络的目标网络，w和θ分别是Q函数网络和策略网络的在线网络，ρ_w和ρ_θ分别是Q函数网络和策略网络的更新速率。这样做的好处是可以避免目标网络的参数变化过快导致的不稳定性。

其三，不同于DQN和DDQN构造TD Error时使用最大化online或者target Q函数的方式来选取动作，DDPG算法则是直接使用目标策略网络给出的动作$\pi_{\overline{\theta}}(s')$来近似最优动作。这种**动作选择与动作评估解耦的方式实际上就相当于双Q函数技巧（本质目的是一样的，即解耦）**。其值函数网络（Critic）的优化目标为：

$$J(w) = \mathbb{E}_{s,a,s' \sim \mathcal{D}_{\text{Replay}}} \left\{ \left(r + \gamma Q'_{\overline{w}}, \pi_{\overline{\theta}}(s') \right) - Q_w(s,a) \right)^2 \Big\}.$$

并且与DQN和DDQN不同，DDPG算法还有显式的策略网络（Actor）需要更新。其策略网络的优化目标为：

$$\max_\theta \mathbb{E}_{s \sim \mathcal{D}_{\text{Replay}}} \{ Q_w(s, \pi_\theta(s)) \}.$$

另外，因为DDPG的策略是确定性的，为了增强探索能力，在选择动作时会加入均值为0的高斯噪声来鼓励探索。并且在每次与环境交互时的初始状态随机从一个状态分布中采样得到。

10.4.8 TD3（Twin Delayed DDPG）

DDPG的一个问题是会对于值函数出现显著的高估，同时因为在PIM中又会用到值函数，从而导致了学到的策略的性能下降以及算法对于超参数设置十分敏感。TD3算法通过在DDPG使用的三个Trick基础上引入两个额外的技巧来缓解过估计问题。TD3算法具有四个Q函数网络和两个策略网络。

首先，TD3额外使用双Q函数取最小值（BDQ）技巧来降低过估计的问题。在构造TD Target的时候，使用两个Target Q函数网络的较小值来构造TD Target：

$$Q_{\overline{w}}^{\text{target}} = r + \gamma \min_{i=1,2} Q_{\overline{W}_i}(s', \pi_{\overline{\theta}}(s')).$$

后续更新两个online Q函数网络时均使用这个相同的TD Target。两个online Q函数网络的更新方式为：

$$\begin{aligned} J(w_1) &= \mathbb{E}_{s,a,s' \sim \mathcal{D}_{\text{Replay}}} \left\{ \left(Q_{\overline{W}}^{\text{target}} - Q_{w_1}(s,a) \right)^2 \right\}, \\ J(w_2) &= \mathbb{E}_{s,a,s' \sim \mathcal{D}_{\text{Replay}}} \left\{ \left(Q_{\overline{W}}^{\text{target}} - Q_{w_2}(s,\alpha) \right)^2 \right\}. \end{aligned}$$

其次，TD3算法使用了延迟策略更新（DPU）技巧。在DDPG算法中，Q函数网络和策略网络在每轮迭代都需要更新一次，而TD3算法则降低了策略网络的更新频率，每隔几轮才更新一次策略网络参数。防止过于频繁的更新导致的Target不稳定。

另外，TD3还有一些额外的处理，比如使用了目标策略平滑性（Target Policy Smoothing）技巧。在构建TD Target的时候，里面的最优动作在DDPG通过策略网络直接给出动作的做法的基础上，引入如下操作作为目标动作：

$$\begin{aligned} \alpha^* &= \text{clip}\Big(\pi_{\overline{\theta}}(s') + \text{clip}(\epsilon, -\epsilon_{\text{bnd}}, \epsilon_{\text{bnd}}), a_{\text{Low}}, a_{\text{High}}\Big), \\ \epsilon &\sim \mathcal{N}(0, \sigma), \end{aligned}$$

这里的ϵ是均值为0的高斯噪声，ϵ_{bnd}是动作噪声的上下界，a_{Low}和a_{High}是动作的上下界范围。这种技巧可以看做是一种特殊的策略正则化手段，可以平滑目标动作分布，来避免在Q函数的估计中出现不正确的异常尖峰，从而使Q函数的估计更加平滑。

10.4.9 SAC（Soft Actor-Critic）

SAC算法是一个off-policy算法且其策略为随机性的。它使用了随机策略来平衡累计回报和策略熵。SAC算法包含4个Q函数网络和一个随机策略网络（两个online Q函数网络均有对应的Target网络，而策略网络没有对应的Target网络）。它使用了经验回放（ExR）、分离目标网络（STN）、双Q函数（DQF）、双Q函数取最小值（BDQ）、软值函数（SVF）技巧。

根据SVF那里讲过的结合熵之后的self-consistency的关系$v^{\pi}(s) = \mathbb{E}_{a \sim \pi}\{q^{\pi}(s,a)\} + \alpha \mathcal{H}\left(\pi(\cdot|s)\right)$，SAC算法构建了一个隐式的状态值函数：

$$V_{\overline{w}}(s')=\mathbb{E}_{a'\sim\pi_\theta}\left\{\min_{i=1,2}Q_{\overline{W}_i}(s',a')-\alpha\log\pi_\theta(a'|s')\right\}.$$

之后，根据这个隐式的状态值函数来构建TD Target并由此得出Q函数（Critic）的优化目标：

$$J_{\mathrm{Critic}}(w_i)=\mathbb{E}_{s,a,s'\sim\mathcal{D}_{\mathrm{Replay}}}\left\{\left(r+\gamma V_{\overline{w}}(s')-Q_{w_i}(s,a)\right)^2\right\},\forall i=1,2.$$

而策略网络（Actor）则是通过最大化隐式的状态值函数的期望来进行更新：

$$J_{\mathrm{Actor}}(\theta)=\mathbb{E}_{s\sim\mathcal{D}_{\mathrm{Replay}},a\sim\pi_\theta}\{Q^{\pi_\theta}(s,a)-\alpha\log\pi_\theta(a|s)\}.$$

另外，SAC的熵项前面的温度系数α是自动调节的，通过构建一个优化问题来求解。该优化问题在保证策略熵大于等于一个目标值的前提下（保证策略具有一定的随机性）来最大化累计回报（保证策略性能）：

$$\begin{aligned}&\max_{\boldsymbol{\pi}_0,\cdots,\boldsymbol{\pi}_N}\mathbb{E}_{s_{t+i}\sim d_{\pi_i},a_{t+i}\sim\pi_i}\left\{\sum_{i=0}^{N}r(s_{t+i},a_{t+i})\right\},\\&\mathrm{s.t.}\\&\mathcal{H}\left(\pi_i(\cdot|s_t)\right)\geq\mathcal{H}_{\mathrm{des}},\end{aligned}$$

这里的$\mathcal{H}_{\mathrm{des}}$是目标熵值，通常可取为动作空间大小的负值$-|\mathcal{A}|$。这个优化问题可以通过对偶梯度下降的方式来求解。

总结一下，SAC算法具有高样本效率，能很好地平衡探索和利用，并且对于不同人wide普适性很强，不需要过多的任务/环境相关的超参数调节。

10.4.10 DSAC（Distributional Soft Actor-Critic）

DSAC算法是对SAC算法的扩展，对于累计回报的分布而不是累计回报的期望（即Q函数）进行建模，从而降低过估计。与之前一些值分布强化学习算法里离散的分布建模不同，DSAC使用连续的高斯分布对于累计回报分布进行建模。DSAC包含两个值分布函数网络（一个online网络和一个target网络）和两个策略网络（也是一个online网络和一个target网络）。它使用了经验回放（ExR）、平行探索（PEx）、分离目标网络（STN）、延迟策略更新（DPU）、双Q函数（DQF）、软值函数（SVF）、分布式回报函数（DRF）技巧。

DSAC借鉴了A3C算法的平行探索思想，具有多个Sampler和Learner来平行探索。同时，它的在线值分布函数网络和策略网络都具有对应的目标网络。同时，DSAC算法也与SAC相同，使用了最大熵强化学习框架，通过软值函数（SVF）来平衡探索与利用。DSAC对于策略网络及温度系数α均采用了延迟更新技术，其更新频率低于Critic（值分布函数网络）的更新频率。

DSAC算法的Critic（值分布函数网络）的优化目标为：

$$J_{\mathrm{Critic}}(w)=\mathbb{E}_{\mathrm{s},a\sim\mathcal{D}_{\mathrm{Replay}}}\left\{D_{\mathrm{KL}}\left(\mathcal{Z}_{\overline{w}}^{\mathrm{target}}(\cdot|s,a),\mathcal{Z}_w(\cdot|s,a)\right)\right\},$$

这里的$\mathcal{Z}_{\overline{w}}^{\mathrm{target}}(\cdot|s,a)$是目标分布，$\mathcal{Z}_w(\cdot|s,a)$是当前网络拟合出的分布。但是上述目标不方便直接优化，因为我们对于目标分布的信息并不清楚。，直接对于上述目标求梯度$\nabla_w J_{\mathrm{Critic}}(w)$不可行。可证明优化目标等价于：

$$\nabla J_{\text{Critic}} = -\mathbb{E}_{s,a,s' \sim \mathcal{D}_{\text{Replay}}, a' \sim \pi_{\overline{\theta}}} \{ \nabla_W \log \mathcal{Z}_W \left(Z_{\overline{w}}^{\text{target}}(s,a) | s,a \right) \},$$

这个式子的意思是在当前网络拟合出来的在线分布$\mathcal{Z}_W(\cdot|s,a)$下取到目标值$Z_{\overline{w}}^{\text{target}}(s,a)$的概率。而这个目标值符合目标分布：

$$Z_{\overline{w}}^{\text{target}}(s,a) \sim \mathcal{Z}_{\overline{w}}^{\text{target}}(\cdot|s,a),$$

e而这个值本身又可以通过下述单步TD Target的形式来近似：

$$Z_{\overline{w}}^{\text{target}}(s,a) = r + \gamma Z_{\overline{w}}(s', \pi_{\overline{\theta}}(a'|s')),$$

DSAC输出的策略与SAC一样，都是随机策略。其优化目标为：

$$J_{\text{Actor}} = \mathbb{E}_{s \sim \mathcal{D}_{\text{Replay}}, a \sim \boldsymbol{\pi}_\theta} \{ \text{Perc}(\mathcal{Z}_w(\cdot|\text{s}, \alpha)) - \alpha \log \pi_\theta(\alpha|s) \},$$

这里的$\text{Perc}(\mathcal{Z})$是分布的分位数，具体是几分位数可以自行指定。这就为算法提供了更加灵活多变的选择。例如，若选择50%分位数（均值）则自动退化为SAC算法的Actor的优化目标：

$$J_{\text{Actor}} = \mathbb{E}_{s \sim \mathcal{D}_{\text{Replay}}, a \sim \boldsymbol{\pi}_\theta} \{ Q_w(s,a) - \alpha \log \pi_\theta(a|s) \}.$$

除此之外，DSAC算法还使用了其它一些技巧来保证训练稳定，如裁剪Target和当前回报分布来避免梯度爆炸等。DSAC算法与之前的强化学习算法相比取得了更好的训练稳定性、值函数估计的准确度（降低了过估计）以及策略的表现，是新一代强化学习SOTA算法。

@[toc]

书籍链接：Reinforcement Learning for Sequential Decision and Optimal Control

本篇博客对应于原书的第11章，主要介绍了强化学习的前沿方向和各种问题的讨论，作为本系列的收尾再合适不过了。使用强化学习的挑战主要来源于以下两个方面：

- **如何更高效地与环境交互**：解决这个问题的思路包括on-policy/off-policy算法、随机探索、稀疏奖励等增强、offline RL等。
- **怎么在一定量数据的情况下学习到最优策略**：解决这个问题的思路包括混合表示（Mixed Representation）、minimax优化等。

在本章中，将会讨论以下几个方面的内容：

- **如何解决模型不确定性**
- **如何解决部分可观性（partially observable）的问题**
- **如何通过更少的样本学习**
- **如何从专家（expert）那里学习奖励**
- **如何解决多智能体（multi-agent）问题**
- **如何从离线数据中学习**
- **SOTA的强化学习算法、强化学习库及仿真平台**

首先，将介绍如何应对模型不确定性的问题。鲁棒强化学习（Robust Reinforcement Learning）是一种方案。它将模型不确定性视为一个对抗的agent。在这个视角下，鲁棒强化学习呗建模为一个minimax问题，控制策略和对抗策略被分别视为一个两玩家零和博弈问题的的参与者。前者试图提升表现而后者试图使其恶化。这类问题的最优性条件是Hamilton-Jacobi-Isaacs（HJI）方程。

现实世界的数据还总是含有噪声。含噪声的测量为马尔克夫环境带来了额外的不确定性。此时对应的马尔克夫决策过程（MDP）被称为部分可观测马尔克夫决策过程（POMDP）。常见的解决POMDP的方法是构建一个两阶段的训练过程。首先，我们为每个状态测量引入一个隐式的状态表示，称为belief state。所谓的belief state是真实state的后验分布，可通过贝叶斯估计获得，我们的任务就是获得一个belief state估计器。其次，我们以belief state作为策略输入，训练一个策略。

元强化学习（Meta Reinforcement Learning）是为了解决强化学习泛化能力不足的问题，提升强化学习算法的样本效率。在元强化学习中，在训练任务上学到的知识被迁移至其它未见但相关的任务上。元强化学习的核心思想是确定可迁移的知识的种类，是经验（experience）、策略（policy）还是损失函数（loss function）。

多智能体强化学习正受到广泛注意。与单智能体系统相比，多智能体系统面临维度爆炸（联合动作/状态空间维度随着智能体数量的增加而指数级增长）和容易发散的问题。各个agent的最优策略与其它agent密切相关。另外，agent之间的联合动态是非平稳的，这加剧了学习的困难。因此，准确预测其它agent的行为就对于稳定策略更新至关重要。在随机多智能体游戏中，其它agent的行为在转移函数和值函数设计中通常被考虑。

Inverse Reinforcement Learning（IRL）是从专家演示中学习奖励函数的过程。IRL的困难是专家演示可能对应于多中奖励设计方案。因此关键是找到一种能区分想要的和不想要的策略的奖励函数。一种方法被称为最大间隔逆强化学习（Maximum Margin Inverse Reinforcement Learning），它的目的是找到一种奖励函数能够最大化专家策略和次优策略之间的间隔。另一种方法是最大熵逆强化学习（Maximum Entropy Inverse Reinforcement Learning）。此时最好的奖励函数应该能够最大化观测到相同专家轨迹的概率。

现在大多数强化学习算法需要在线地与环境交互，这通常十分昂贵耗时，甚至危险，而构建仿真环境在里面进行在线交互的做法也不总是可行的，因为构建高保真的仿真环境十分困难。而另一个问题是我们已经有大量预先收集的数据。因此，离线强化学习（Offline Reinforcement Learning）成为了一个重要的研究方向。它使用事先离线收集好的数据，而不需要与环境交互。然而，因为策略和值函数是在离线的数据分布上训练的，而验证则是在另一个分布（由学到的策略决定）上进行的，因此离线强化学习面临着分布转移（distribution shift）的问题，从而导致了离线强化学习算法不稳定，很难训练。

11.1 鲁棒强化学习

模型误差和外界干扰在反馈系统中很常见，因此十分需要一个鲁棒的控制律。从上世纪八十年代开始，鲁棒控制理论开始受到广泛关注。基于H_∞范数的系统分析和控制器合成取得了很大进展。但H_∞控制通常局限于线性系统及二次的效用函数（utility function，可以理解为类似奖励函数的东西）。作为拓展，在强化学习中，环境动力学可以是仿射非线性的，效用函数也可以是非二次的。一个典型的鲁棒最优控制问题建模如下：

$$\dot{x} = f(x,u) + g(x,w),$$

这里w是未知的扰动。这个环境模型包含一个标称部分$f(x,u)$和一个不确定部分$g(x,w)$。然后，无穷时域的代价函数被定义为效用函数的积分：

$$J(x(t),u(\tau),w(\tau))|_{\tau=[t,\infty)}=\int_t^\infty l(x,u,w)\mathrm{d}\tau,$$

这个代价函数是初始状态$x(t)$、控制序列$u(\tau)|_{\tau=[t,\infty)}$和扰动序列$w(\tau)|_{\tau=[t,\infty)}$的函数。当状态和动作空间很大或$g(x,w)$不具有易处理的结构时，求解这个问题是非常困难的。当u和w可以写成一个稳态马尔克夫策略时，状态值函数可以简单地写为初始状态$x(t)$的函数：

$$V\big(x(t)\big)=\int_t^\infty l(x,u,w)\big|_{u=u(x),w=w(x)}\mathrm{d}\tau.$$

这里，**控制策略是$u=u(x)$，对抗策略是$w=w(x)$**。当状态值函数是有限值的，在边界条件$V(0)=0$下，可以得出一个偏微分方程（PDE）：

$$H\left(x,u,w,\frac{\partial V(x)}{\partial x}\right)=l(x,u,w)+\frac{\partial V(x)}{\partial x^{\mathrm{T}}}\left(f(x,u)+g(x,w)\right)=0,$$

这里的H是Hamiltonian函数。最优控制律$u^*(x)$和对抗律$w^*(x)$自然就构成了微分博弈问题的一个鞍点(u^*,w^*)。这个纳什均衡可以通过求解对应的Hamilton-Jacobi-Isaacs（HJI）方程来获得。HJI方程是一个非线性的偏微分方程，其数值解很难获得（尤其对于非线性系统）。而鲁棒RL作为一种数值求解方法，通过一种类似于策略迭代的方式来求解，它包含两个交替的过程：策略评估（PEV）和策略改进（PIM）。

11.1.1 H_∞控制和零和博弈

首先先来介绍一下H_∞控制和零和博弈。大部分H_∞控制问题都使用传递函数矩阵来设计优化目标，因此局限于线性时不变系统。一个带有不确定性的线性系统如下：

$$\dot{x}=(A+\Delta A)x+(B+\Delta B)u,$$

这里的A和B是标称模型，而ΔA和ΔB是模型误差。那么可以把扰动w写成下面的形式：

$$w=\Delta Ax+\Delta Bu.$$

与此同时，我们把目标输出z定义为：

$$z^{\mathrm{T}}=\begin{bmatrix}\left(\sqrt{Q}x\right)^{\mathrm{T}} & \left(\sqrt{R}u\right)^{\mathrm{T}}\end{bmatrix},$$

这里$Q=Q^{\mathrm{T}}>0$ and $R=R^{\mathrm{T}}>0$是正定的权重矩阵。那么就可以把扰动w使用z及Q和R来表示（展开即可证明）：

$$w=\begin{bmatrix}\Delta A\cdot\sqrt{Q^{-1}} & \Delta B\cdot\sqrt{R^{-1}}\end{bmatrix}\begin{bmatrix}\sqrt{Q} & 0\\ 0 & \sqrt{R}\end{bmatrix}\begin{bmatrix}x\\ u\end{bmatrix}=\Delta\cdot z,$$

这里的$\Delta=\begin{bmatrix}\Delta A\cdot\sqrt{Q^{-1}} & \Delta B\cdot\sqrt{R^{-1}}\end{bmatrix}$。从小增益定理可知闭环系统是鲁棒稳定的，如果：

$$\|T_{ZW}\|_\infty\|\Delta\|_\infty<1,$$

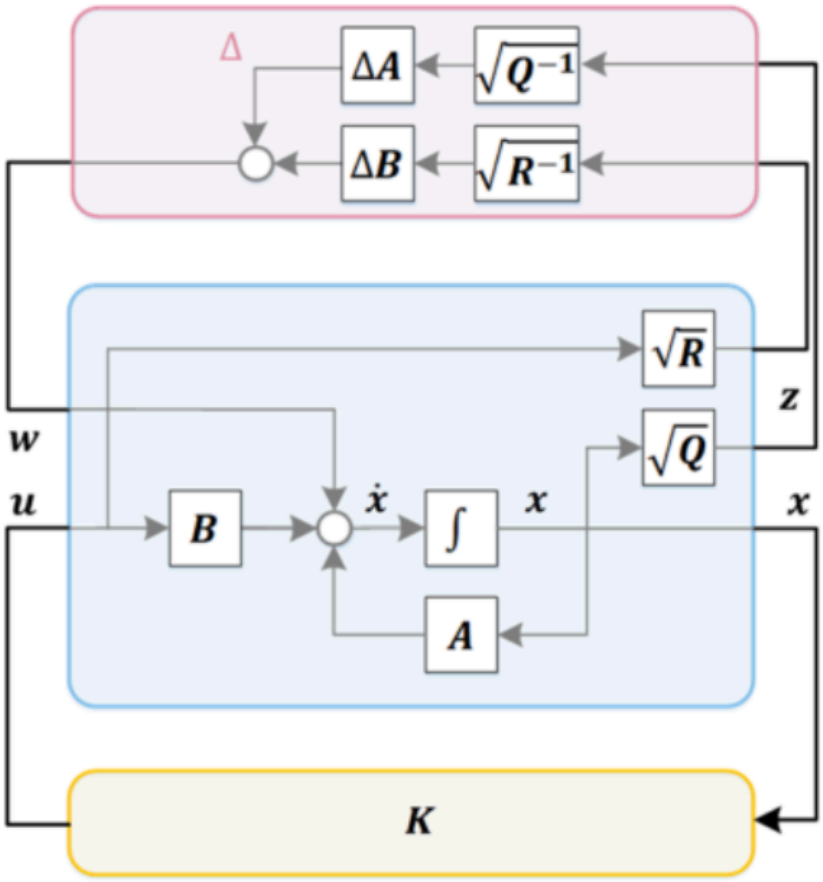
这里的T_{ZW}是从w到z的传递函数矩阵。这里Δ是未知的，但是$||\Delta||_\infty$被$1/\gamma$所约束。因此，上述不等式可以改写为：

$$\|T_{ZW}\|_\infty<\gamma,$$

进一步改写为：

$$\|T_{zw}\|_\infty^2=\sup_w\frac{\|z\|_2^2}{\|w\|_2^2}=\sup_w\frac{\int_t^\infty z^{\mathrm{T}}z\,\mathrm{d}\tau}{\int_t^\infty w^{\mathrm{T}}w\,\mathrm{d}\tau}\leq\gamma^2,$$

式中sup表示上确界。从上式中第一项到第二项使用了传递函数的定义。这个式子的含义是把未知的扰动抑制在一个扰动衰减水平之内。



上图展示了H_∞控制的标准建模。这个框图中包含两个Loop，下面的那个是标准的反馈控制Loop，上面那个则是扰动衰减Loop。

通常来说，上述鲁棒控制问题会被转化为一个线性矩阵不等式形式（Linear Matrix Inequalities，LMIs）的凸优化问题。因此，LMIs的发展就与鲁棒控制息息相关，也有很多研究关注如何开发高效的LMI求解器。然而，尽管转化为LMI的求解是一种可行的方法，但是对于很多实际问题来说转化成的LMI问题规模太大，现有的求解器无法求解。一种新的方法是将H_∞控制问题转化为一个minimax优化问题。首先，是要找到一个满足下式的次优控制器：

$$J(x,u,w)=\sup_w\int_t^\infty(z^{\mathrm{T}}z-\gamma^2w^{\mathrm{T}}w)\mathrm{d}\tau\leq 0.$$

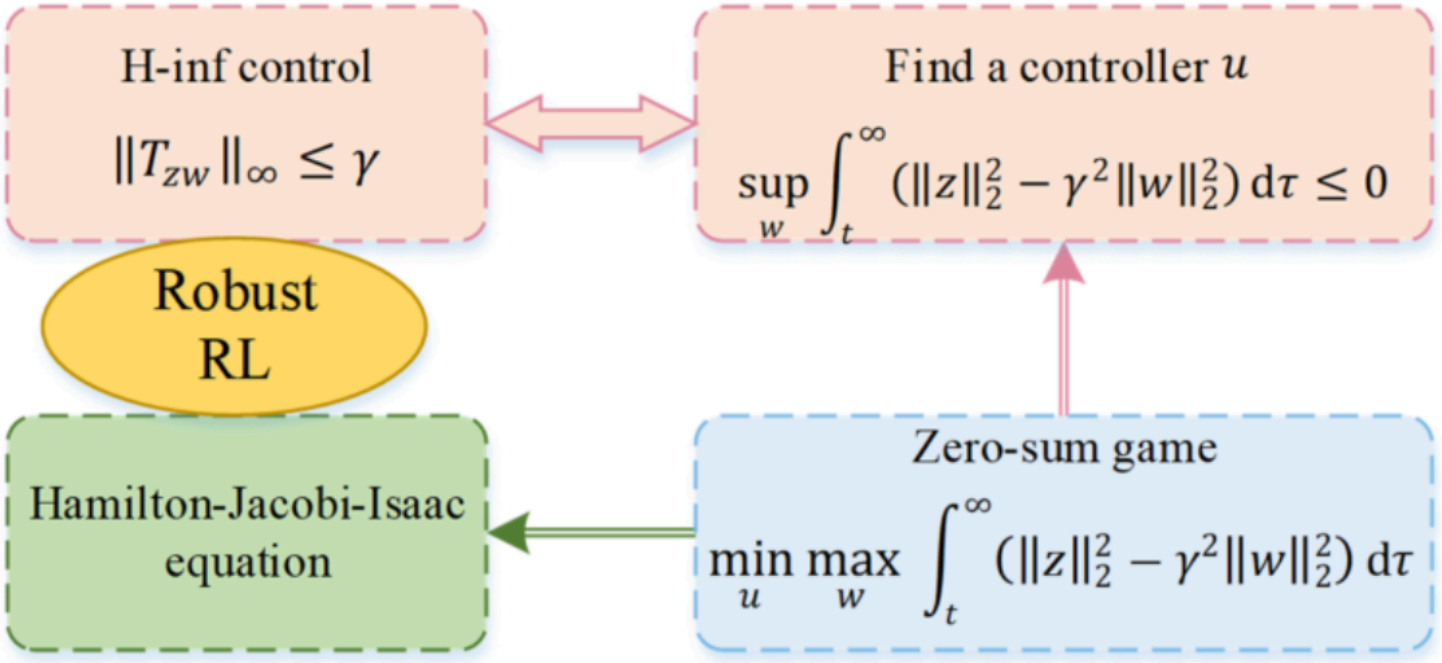
那么效用函数自然就可以写成：

$$l(x,u,w)=z^{\mathrm{T}}z-\gamma^2w^{\mathrm{T}}w=x^{\mathrm{T}}Qx+u^{\mathrm{T}}Ru-\gamma^2w^{\mathrm{T}}w.$$

因此，可以通过最小化上述表现指标$J(x,u,w)$的上确界的方式来获得一个可行的控制律。那么，最后问题就被转化为：

$$V^*(x)=J(x,u^*,w^*)=\min_u\max_w J(x,u,w),$$

这里的$V^*(x)$是最优值函数。这个问题可以被视为一个两玩家微分博弈问题，其中一个玩家是控制器$u=u(x)$，另一个玩家是对抗者$w=w(x)$。控制器的目标是最小化代价函数，而对抗者的目标是最大化代价函数。因为一方的获益自然会造成另一方的损失，因此这个博弈是一个零和博弈。这也即鲁棒RL的标准问题建模方式。



上图展示了H_∞控制的零和博弈的关联，也说明了如何从minimax优化的角度来求解鲁棒控制问题。

11.1.2 线性版本的鲁棒强化学习

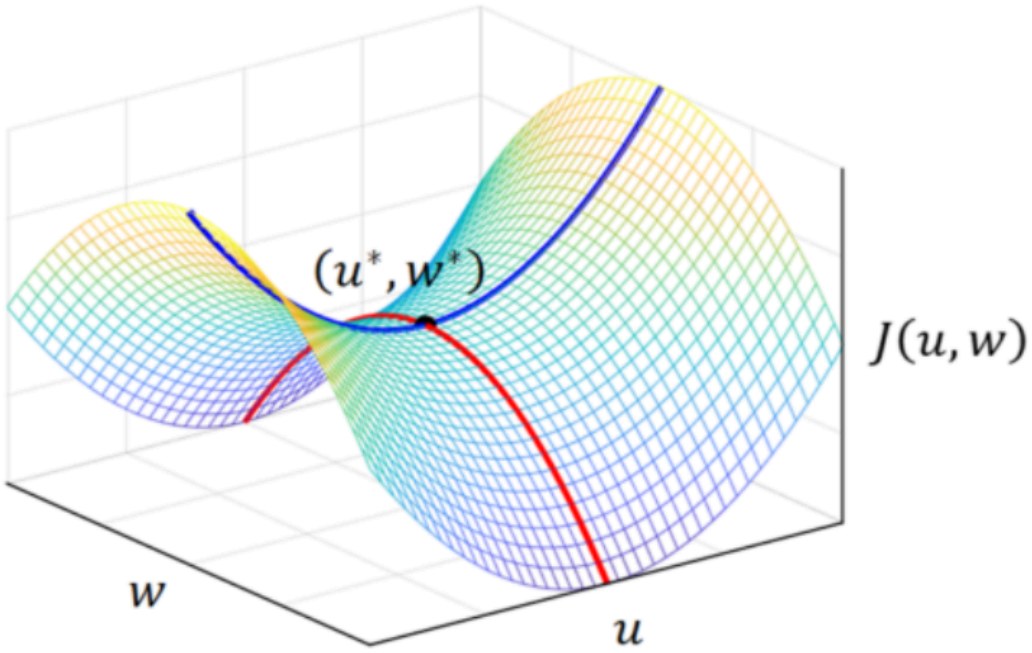
线性版本的鲁棒强化学习不管在理论上还是实践上都很重要，因此，已有许多研究关注。研究表明，当衰减率γ足够大时，解总是存在的。此外，在线性系统中不好的初始化会导致会极大影响收敛性，因此一些鲁棒的可行初始化方式被提出来稳定训练过程。一些研究使用多项式来近似值函数。另有一些研究通过将系统的非线性看作线性化后的近似误差来处理非线性系统，尽管这样会导致稳定的区域面积缩小。

这里，我们沿用前面H_∞控制和零和博弈那里的推导，将哈密顿函数H改写为：

$$\begin{aligned} H\left(x, u, w, \frac{\partial V(x)}{\partial x}\right) &= l(x,u,w) + \frac{\partial V(x)}{\partial x^{\mathrm{T}}}\left(f(x,u) + g(x,w)\right) \\ &= x^{\mathrm{T}} Q x + u^{\mathrm{T}} R u - \gamma^2 w^{\mathrm{T}} w + \frac{\partial V(x)}{\partial x^{\mathrm{T}}}\left(Ax + Bu + w\right) \\ &= 0, \end{aligned}$$

那么，式子$V^*(x) = J(x, u^*, w^*) = \min_w \max_w J(x,u,w)$在下述纳什条件下有唯一解：

$$\min_u \max_w J(x(t), u, w) = \max_w \min_u J(x(t), u, w).$$



纳什条件的如上图所示。在鞍点(u^*, w^*)时，两个玩家u和w都没有单方面改变策略的动力，因为此时u移动的话会导致代价函数J增加，这与它的最小化目标相悖；同理，w移动的话会导致代价函数J减少，这与它的最大化目标相悖。

纳什条件的一个必要条件是Isaacs条件（可被视为庞特里亚金最小值原理的推广）：

$$\min_u \max_w H\left(x, u, w, \frac{\partial V(x)}{\partial x}\right) = \max_w \min_u H\left(x, u, w, \frac{\partial V(x)}{\partial x}\right).$$

这个式子与上面纳什条件的形式类似，只是将J替换成了H。当动态系统是零状态可观（zero-state observable）时，Isaacs条件也是一个充分条件。Isaacs条件的优势在于它的max和min操作可以交换，这样在求得最优值函数$V^*(x)$后就可以根据驻点条件$\partial H/\partial u = 0$和$\partial H/\partial w = 0$来求得驻点$(u^*, w^*)$：

$$u^* = -\frac{1}{2} R^{-1} B^{\mathrm{T}} \frac{\partial V^*(x)}{\partial x},$$
$$w^* = \frac{1}{2\gamma^2} \frac{\partial V^*(x)}{\partial x}.$$

注意到$\partial^2 H/\partial u^2 = 2R > 0$及$\partial^2 H/\partial w^2 = -2\gamma^2 < 0$，可得哈密顿函数确实在$u^*$和$w^*$处取得极小值和极大值，所以驻点$(u^*, w^*)$确实是一个鞍点。把最优的$(u^*, w^*)$及$V^*(x)$代入哈密顿量$H$中，可得：

$$H\left(x, u^*, w^*, \frac{\partial V^*(x)}{\partial x}\right) = 0, V^*(0) = 0.$$

这就是HJI方程。可证明如果HJI方程有解，那么上述解析形式的u^*和w^*不但可保证存在一个衰减率γ，也能够鲁棒地稳定系统。

对于线性系统来说，HJI方程可以化简为代数Riccati方程的形式，其中最优值函数是一个二次的形式：

$$V^*(x) = x^{\mathrm{T}} P x,$$

这里$P = P^{\mathrm{T}} > 0$是Ricacati矩阵。而通过驻点的条件，可计算出u^*和w^*的解析形式并回代回HJI方程中，得到一个代数Riccati方程：

$$A^{\mathrm{T}} P + PA - P\left(BR^{-1}B^{\mathrm{T}} - \frac{1}{\gamma^2} I\right) P + Q = 0.$$

Riccati方程的求解已经有很长时间的研究历史，有很多成熟的求解方法，即使上述代数Riccati方程与LQR、卡尔曼滤波等经典控制问题里的Riccati方程有一些不同，但是那些求解方法仍然适用。

11.1.3 非线性版本的鲁棒强化学习

尽管非线性系统没有传递函数的概念，也没有之前那种线性的放射形式的模型，更不用说准确的解析模型了。但是我们仍然可以使用闭环系统的L_2增益来设计优化目标。上述Issaecs条件和两个稳态时的条件（$\partial H/\partial u=0$和$\partial H/\partial w=0$）仍然适用。得到最优的值函数之后，可以得出最优的控制律和对抗律：

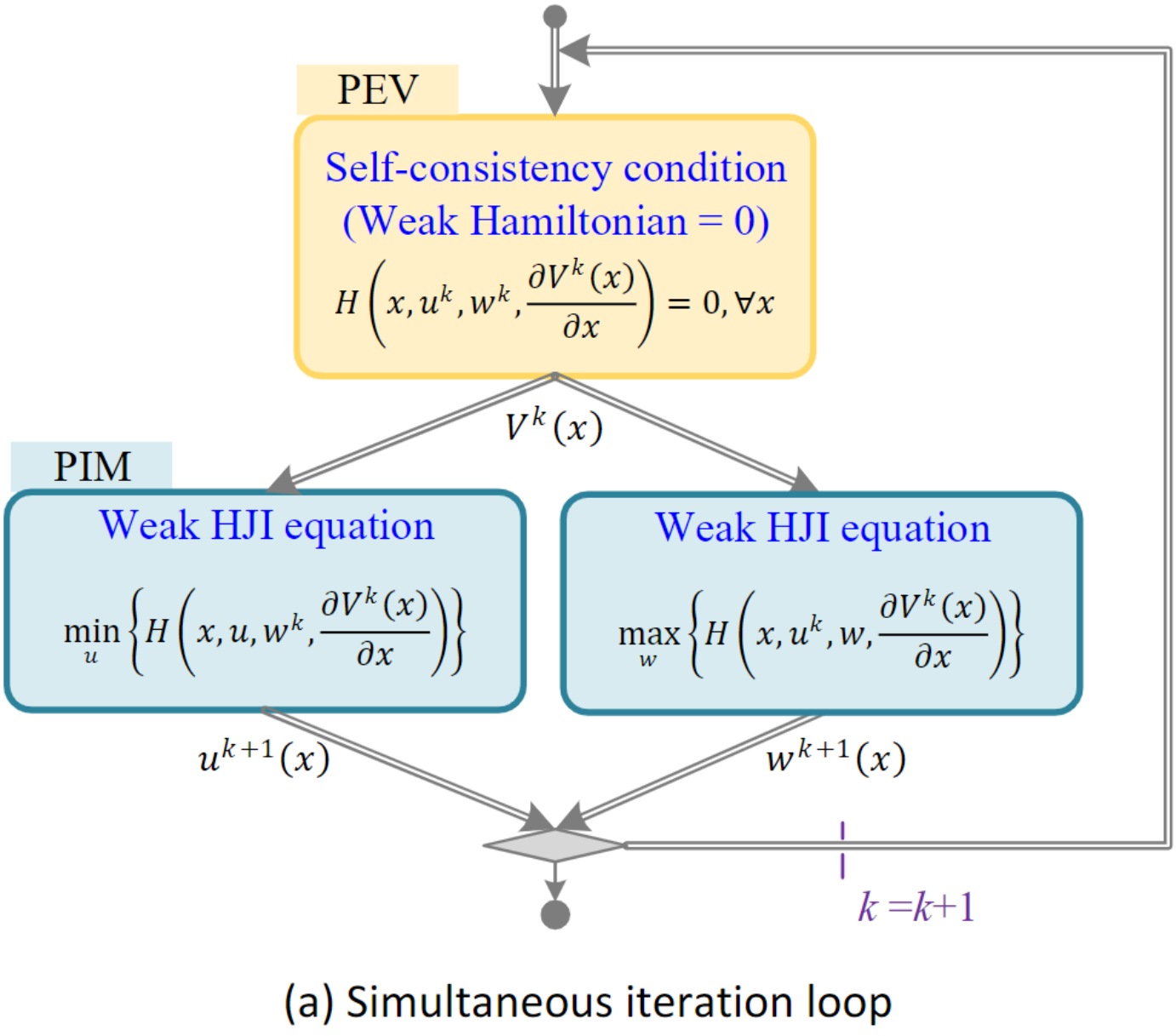
$$u^*=-\frac{1}{2}R^{-1}\frac{\partial f(x,u)}{\partial u}\bigg|_{u=u^*}\frac{\partial V^*(x)}{\partial x},$$
$$w^*=\frac{1}{2\gamma^2}\frac{\partial g(x,w)}{\partial w}\bigg|_{w=w^*}\frac{\partial V^*(x)}{\partial x}.$$

这里$f(x,u)$和$g(x,w)$就没有解析的形式了，可能是通过神经网络等方式来近似的。对应的HJI方程如下：

$$\min_u\max_w\left[l(x,u,w)+\frac{\partial V^*(x)}{\partial x^{\mathrm{T}}}\left(f(x,u)+g(x,w)\right)\right]=0.$$

可以使用策略迭代的方式来求解HJI方程。可以分为两种子方法：

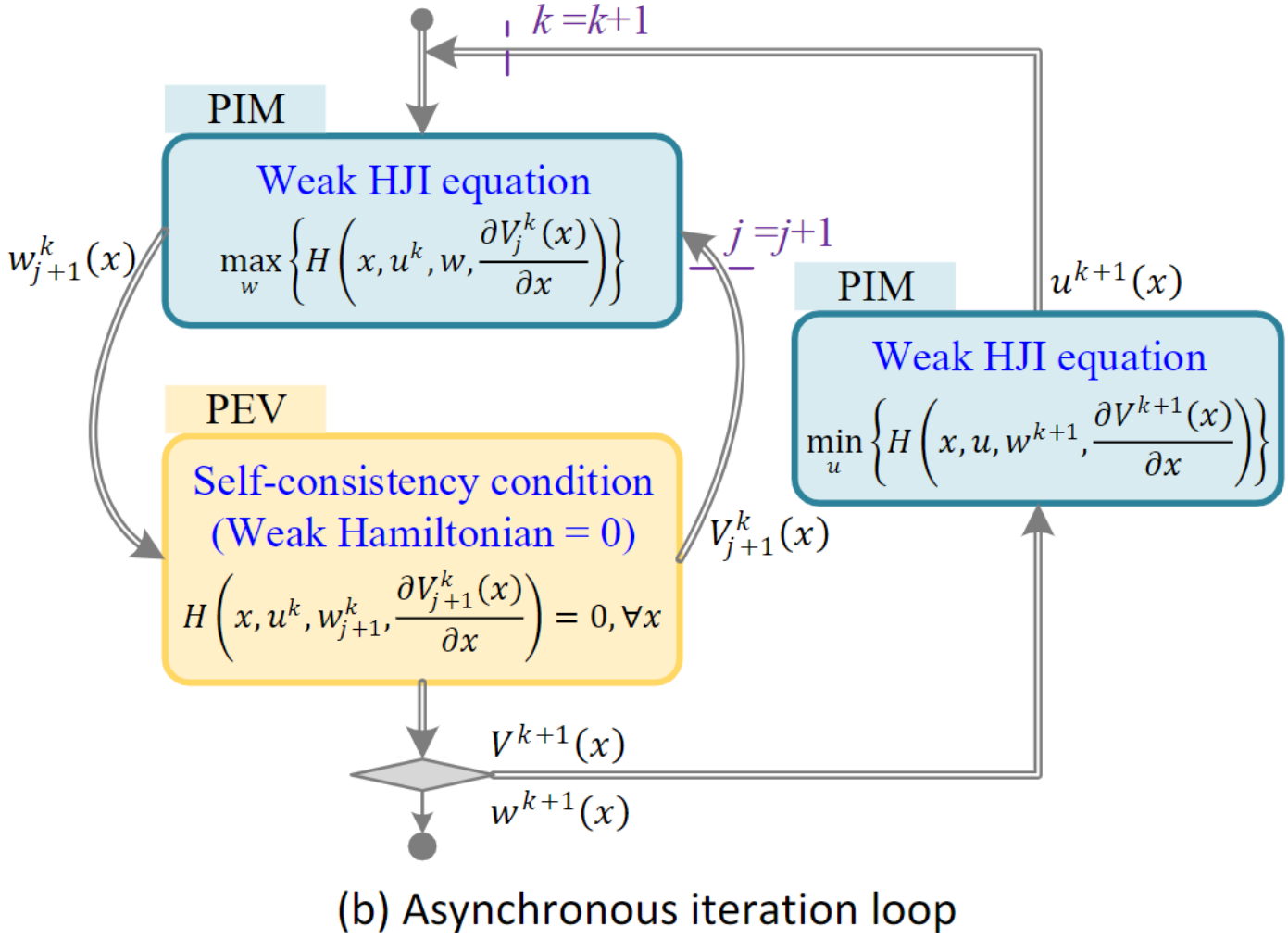
- **同时迭代更新u和w**



(a) Simultaneous iteration loop

这里只有一层循环，每次先进性PEV，再同时进行两个策略的PIM。

- **异步的内外双循环迭代更新w和u**



(b) Asynchronous iteration loop

这里有两层循环，外层循环是u的PIM，内层循环是w的PIM和PEV。

除此之外，此时的效用函数$l(x,u,w)$就不再是二次的形式了，而是可以根据实际的问题来设计（即一般强化学习中的奖励函数设计）。

最后，在使用深度神经网络来近似时，近似误差等也可被纳入位置但有界的扰动中去。而鲁棒RL抵抗各种不确定性的能力就可发挥作用，得到可在最坏的扰动下的最优策略。

11.2 部分可观测马尔克夫决策过程（Partially Observable Markov Decision Process，POMDP）

不完美的环境建模和观测影响了强化学习在现实中的应用。例如在自动驾驶中，车辆和周围环境总有一些不确定性，如车辆参数、周围交通参与者的随机行为等。另外，传感器等也存在噪声。

总的来说，不确定性可以分为两种：

- **过程不确定性（process uncertainty）**：指未被建模的高阶动力学、参数或结构误差及不确定的外部扰动等。另外，执行器执行动作时的不准确（即不能完全执行收到的指令）也可以视为过程不确定性（把收到的指令当成动作即可）。
- **观测不确定性（observation uncertainty）**：指传感器在测量状态变量时的不准确/不完美。

而这些不确定性可以使用随机噪声来建模，它们服从某种分布（如高斯分布）。

11.2.1 部分可观测马尔克夫决策过程问题定义（POMDP）

具有不完美观测的离散时间随机系统由以下两个方程描述：

$$x_{t+1} = f(x_t, u_t) + \xi_t \ ,$$
$$y_t = g(x_t) + \zeta_t,$$

方程中各个变量的含义如下表所示：

符号	含义
$x_t \in \mathbb{R}^n$	系统状态（state）
$u_t \in \mathbb{R}^m$	动作（action）
$xi_t \in \mathbb{R}^n$	过程噪声（process noise）
$y_t \in \mathbb{R}^l$	测量值（measurement）
$zeta_t \in \mathbb{R}^l$	观测噪声（observation noise）
$f(\cdot, \cdot)$	环境动力学的确定性部分
$g(\cdot)$	测量函数（measurement function）

这里我们假设过程噪声序列$\{\xi_0, \xi_1, \ldots, \xi_\infty\}$和观测噪声序列$\{\zeta_0, \zeta_1, \ldots, \zeta_\infty\}$是**独立同分布的（i.i.d.）且与初始状态$x_0$独立**。因此，下一个状态$x_{t+1}$只与三元组$(x_t, u_t, \xi_t)$有关，即在带有不确定性的情况下环境**仍具有马尔可夫性质**。但是事情没有那么简单，因为在带有不确定的设定下，当前状态x_t没法由当前观测值y_t确定，因此也就无法利用上述马尔可夫性质。而且，正是因为真实状态是不可观的，因此策略也就无法被定义为当前状态的函数。一个可行的替代方案是使用包含历史信息的状态h_t来替换当前状态x_t：

$$h_t \overset{def}{=} \{y_{1:t}, u_{0:t-1}\}.$$

这个信息包含了从初始开始的所有动作和观测。因此，对应的策略就是：

$$u_t = \pi(h_t).$$

PMDP的目标是最小化累计代价：

$$\min_{u=\pi(h)} V_h^\pi(h_t) = \mathbb{E}_\pi \left\{ \sum_{i=t}^{\infty} \gamma^{i-t} l(x_i, u_i) | h_t \right\},$$

这里的$V_h^\pi(h_t)$是累计代价的期望，即状态值函数。$l(x_i, u_i)$是效用函数，下标h则表示值函数的参数是历史信息h。

可以证明，历史信息h_t近似具有马尔可夫性质：

$$p(h_{t+1}|h_t) = p(h_{t+1}|h_1, h_2, ..., h_t).$$

可通过如下方式获得：

$$h_{t+1} = \mathrm{Update}\{h_t, u_t, y_{t+1}\}.$$

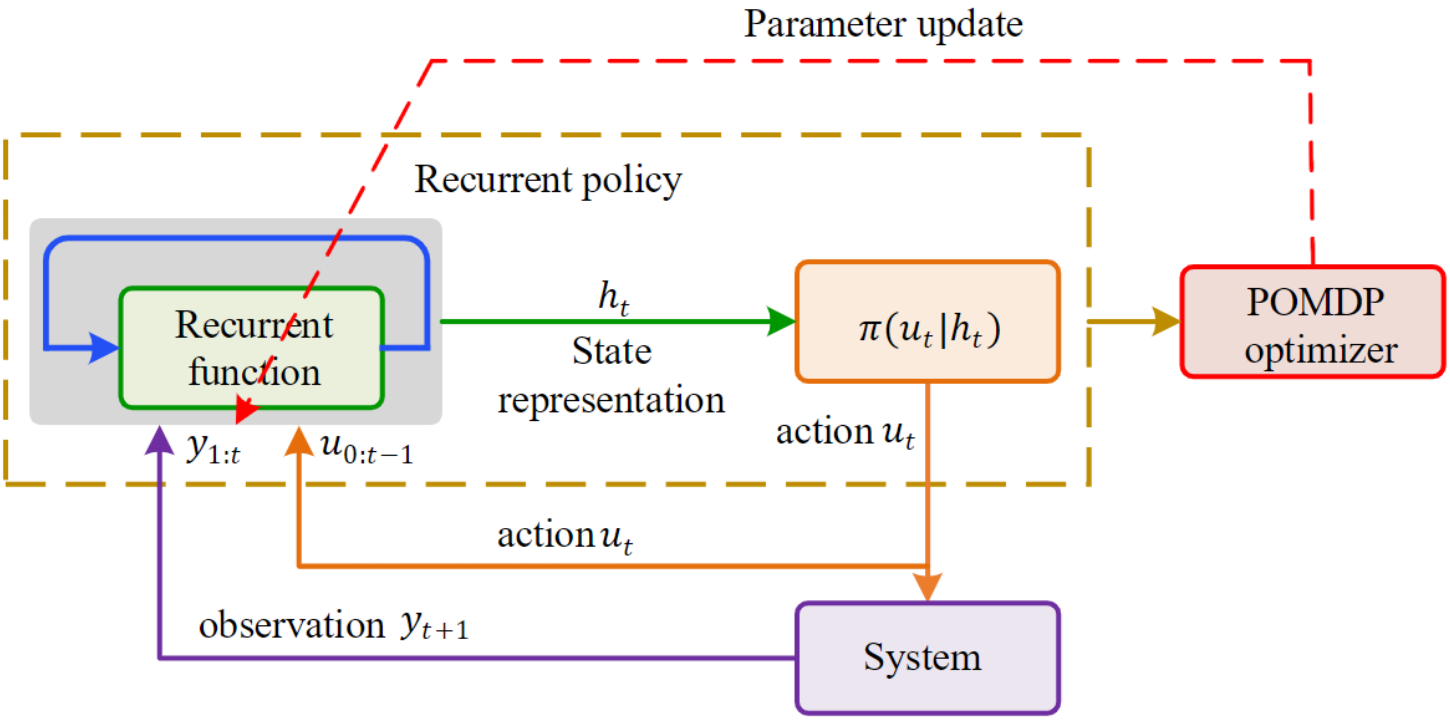
关于h_t具有马尔可夫性质，可以这样理解：h_t本身就包含了从开始到当前的全部观测和动作信息，并且每步的新信息可以通过更新来加入。因此，这种建模下的自洽条件为：

$$\begin{aligned}
&V_h^\pi(h_t) \\
&= \mathbb{E}_\pi\{l_t|h_t\} + \mathbb{E}_\pi\{\sum_{i=1}^{\infty} \gamma^i l(x_{t+i}, u_{t+i}) | h_t\} \\
&= \mathbb{E}_\pi\{l_t|h_t\} + \gamma \sum_{h_{t+1}} p(h_{t+1}|h_t) \mathbb{E}_\pi\{\sum_{i=1}^{\infty} \gamma^{i-1} l(x_{t+i}, u_{t+i}) | h_{t+1}\} \\
&= \mathbb{E}_\pi\{l_t|h_t\} + \gamma \sum_{h_{t+1}} p(h_{t+1}|h_t) V_h^\pi(h_{t+1}).
\end{aligned}$$

而贝尔曼方程则是：

$$V_h^*(h_t) = \min_{u_t} \left\{ \mathbb{E}\{l_t|h_t\} + \gamma \sum_{h_{t+1}} p(h_{t+1}|h_t, u_t) V_h^*(h_{t+1}) \right\}.$$

将历史积累下来的信息当做“状态”的做法面临的挑战在于历史信息的长度不固定。为此，可采用深度学习中一个简单的想法，即通过一个recurrent函数来讲历史信息编码为一个固定长度的隐状态。如下图所示：



最常用的实现recurrent函数的方法就是使用循环神经网络（RNN）。在这种情况下，需要同时解决两个学习问题：**隐状态表示的学习**和**策略的学习**。而这两个过程是强烈耦合在一起的，因此假如不仔细设计这两个过程，可能会导致学习过程不稳定甚至失败。实际上，POMDP被观察到存在隐状态表征学习的瓶颈，需要投入大量学习时间才能学到一个好的隐状态表征。这也是POMDP的一个尚未解决的开放问题。

11.2.2 信念状态（Belief State）和分离原理

如刚才所述，因为观测带有噪声，因此POMDP的状态本身不具有马尔可夫性质。为此在上一小节中通过将历史信息作为新的状态并证明了其具有马尔可夫性质。但是如果直接这样做会导致不固定长度的状态表示以及随着时间增长而导致的状态空间维度爆炸。上一小节中提到的recurrent函数虽然可以解决不固定长度的问题，但是其学习过程耗时且可能导致训练不稳定。为此，本小节将介绍一种高效且有理论保证的方法来处理POMDP问题：信念状态（Belief State）。

信念状态（Belief State）是在历史信息的基础上真实状态的概率分布。也就是说，信念状态是一个**概率分布**，因此可以避免处理随时间增长而区域无穷的历史信息。信念状态b_t是历史信息h_t的一个**充分统计量**。这里“充分”的意思是从实际的分布中采样得到的样本不能比这个（些）统计量给出更多的信息。比如对于高斯分布，充分统计量就是均值和标准差。只要确定了这两个就能唯一确定一个高斯分布。或者如果均值已知，那么标准差本身就是一个充分统计量。具体到我们这里，历史信息h_t的充分统计量b_t的定义如下：

> **充分统计量**：如果下式成立，那么称b_t是h_t的一个充分统计量：
>
> $$p(x_t|h_t) = p(x_t|b_t).$$

充分统计量可取代原本的历史信息h_t作为策略或值函数的输入。值得一提的是，像这种充分统计量并不是唯一的。可以根据实际问题的需要选择一个有实际意义且形式简洁的表示。一个常见的选择是以历史信息为条件的当前状态的概率分布。这种充分统计量被称为**信念状态（Belief State）**，记作b_t，其定义如下：

> **信念状态**：信念状态b_t是当前状态x_t的以历史信息h_t为条件的概率分布：
>
> $$b_t(x_t) = p(x_t|h_t).$$

并且，可证明信念状态b_t具有马尔可夫性质（参考原书11.2.2节第415页证明）。其可由下面的方式更新：

$$b_t = \text{Update}(u_{t-1}, b_{t-1}, y_t).$$

由证明过程及上述更新公式可知，信念状态的更新与贝叶斯估计等价，都是将新收到的信息与之前的信息相结合。在POMDP中，信念状态可由任意计算上高效的贝叶斯算法来更新，比如卡尔曼滤波（Kalman Filter）、扩展卡尔曼滤波（Extended Kalman Filter）、变分推断（Variational Inference）等。信念状态具有下述性质：

- **期望奖励/效用的等价性**：

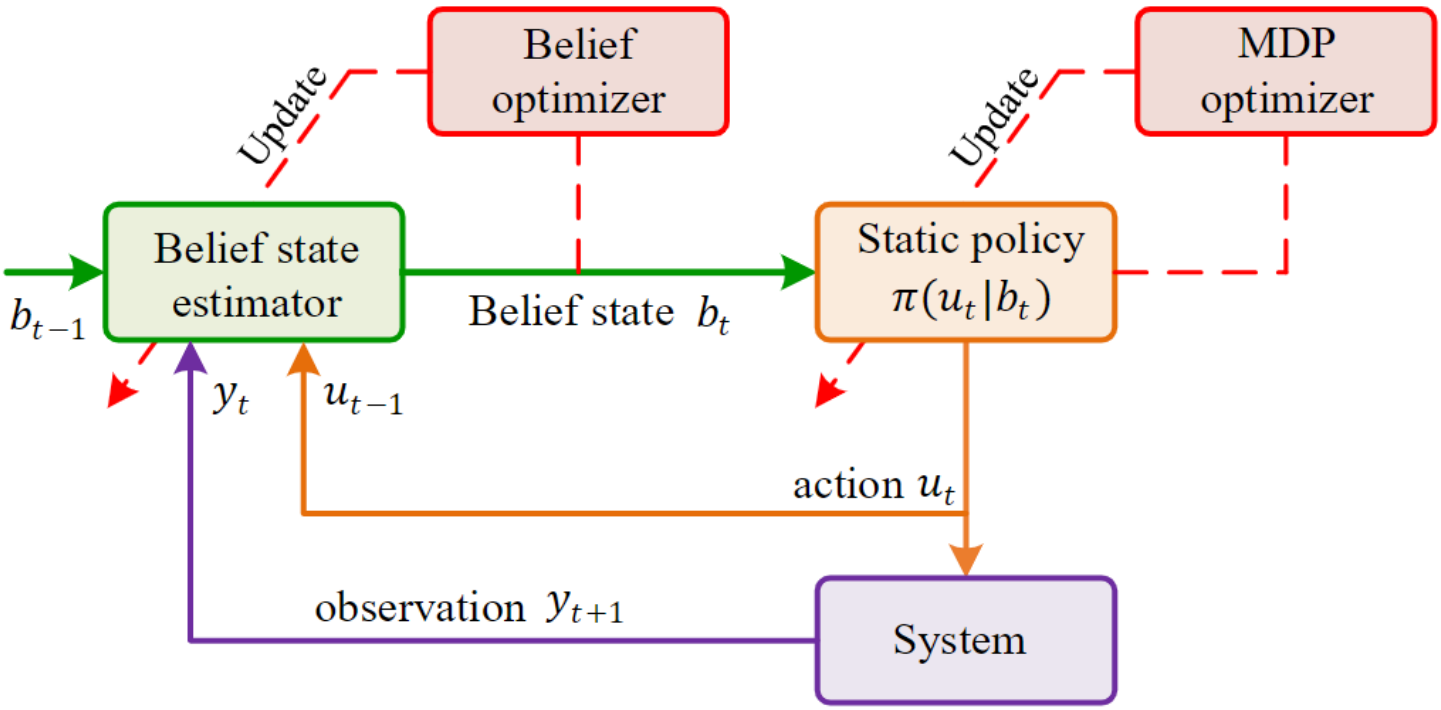
$$\mathbb{E}\{l_t|h_t, u_t\} = \mathbb{E}\{l_t|b_t, u_t\}.$$

- **下一个观测y_{t+1}的条件概率分布的等价性**：

$$p(y_{t+1}|h_t, u_t) = p(y_{t+1}|b_t, u_t).$$

除了马尔可夫性质，信念状态还具有维度固定的优点。比如，假设我们知道信念状态既有高斯分布的结构，那么每次只需要两个参数（均值和标准差）就可以确定下来了。这极大降低了了状态表示的复杂度。

事实上，有了信念状态之后，POMDP问题就可以分为两步求解：**最优的信念状态估计器**和**最优的确定性策略（控制器）**。这是由**分离原理**保证的。以LQG问题举例，其解包含一个卡尔曼滤波器（Kalman Filter）和一个LQR控制器。LQR控制器的输入就不是将整个高斯分布作为输入，而是使用高斯分布的均值，而最后的结果仍然是最优的。这种分两部分求解POMDP问题的示意图如下所示：



第一部分是对于信念状态的估计器，实际上就是一个贝叶斯估计器；第二部分与普通的RL问题类似，就是对于转化后的、以信念状态为输入的MDP问题的学习：

$$\min_{u=\pi(b)} V_b^\pi(b_t) = \mathbb{E}\left\{\sum_{i=t}^{\infty} \gamma^{i-t} l(x_i, u_i)|b_t\right\},$$

这里$V_b^\pi(\cdot)$是以信念状态为输入的值函数，而$\pi(b)$是以信念状态为输入的策略。分离原理的美妙之处在于原始POMDP和转化后的MDP问题的最优策略的等价性。分离原理如下：

> **分离原理**：对于POMDP问题，使用历史信息h_t的最优值函数$V_h^*(h_t)$和使用信念状态b_t的最优值函数$V_b^*(b_t)$是等价的，即：
>
> $$V_h^*(h_t) = V_b^*(b_t).$$

一般的情形的分离原理证明见原书11.2.2节第418页（就是利用上述信念状态的两条性质使用贝尔曼方程递归地展开值函数表达式）。

11.2.3 POMDP的一个例子：LQG问题

LQG问题是POMDP的一个典型例子。在这里，process noise和observation noise都服从高斯分布：

$$
\begin{aligned}
x_{t+1} &= Ax_t + Bu_t + \xi_t \ , \\
y_t &= Cx_t + \zeta_t,
\end{aligned}
$$

这里的$A \in \mathbb{R}^{n\times n}$是转移矩阵，$B \in \mathbb{R}^{n\times m}$是控制矩阵，$C \in \mathbb{R}^{l\times n}$是观测矩阵。$\xi_t$和$\zeta_t$是过程噪声和观测噪声，均服从均值为0的高斯分布。效用函数为二次形式：

$$
l(x_t, u_t) = x_t^{\mathrm{T}} Q x_t + u_t^{\mathrm{T}} R u_t,
$$

那么，总的代价函数就是：

$$
J = \mathbb{E}\{\sum_{i=t}^{\infty} \gamma^{i-t}(x_i^{\mathrm{T}} Q x_i + u_i^{\mathrm{T}} R u_i)|h_t\}.
$$

在带有高斯噪声的线性系统中，状态的条件均值等于它的最小方差估计：

$$
\mathbb{E}\{x_t|b_t\} = \hat{x}_t.
$$

那么，我们就可以把信念状态b_t定义为$\hat{x}_t$（根据卡尔曼滤波器的性质可知，最小方差估计就是真实状态的充分统计量）。那么，信念状态估计器部分就是一个估计最小方差的卡尔曼滤波器：

$$
\begin{aligned}
\hat{x}_{t+1} &= \mathrm{Kalman}(y_{t+1}, u_t, \hat{x}_t), \\
\hat{x}_{t+1} &= A\hat{x}_t + Bu_t + L\big(y_{t+1} - C(A\hat{x}_t + Bu_t)\big),
\end{aligned}
$$

这里的L是稳态卡尔曼增益。

另外，最优的代价函数可以分成两个部分，包含线性二次代价（包含动作和状态分布的均值）和估计代价（状态分布的方差）。只根据前者就可以得到策略。这也就是为什么LQG只使用均值作为策略输入。最优代价函数的形式如下：

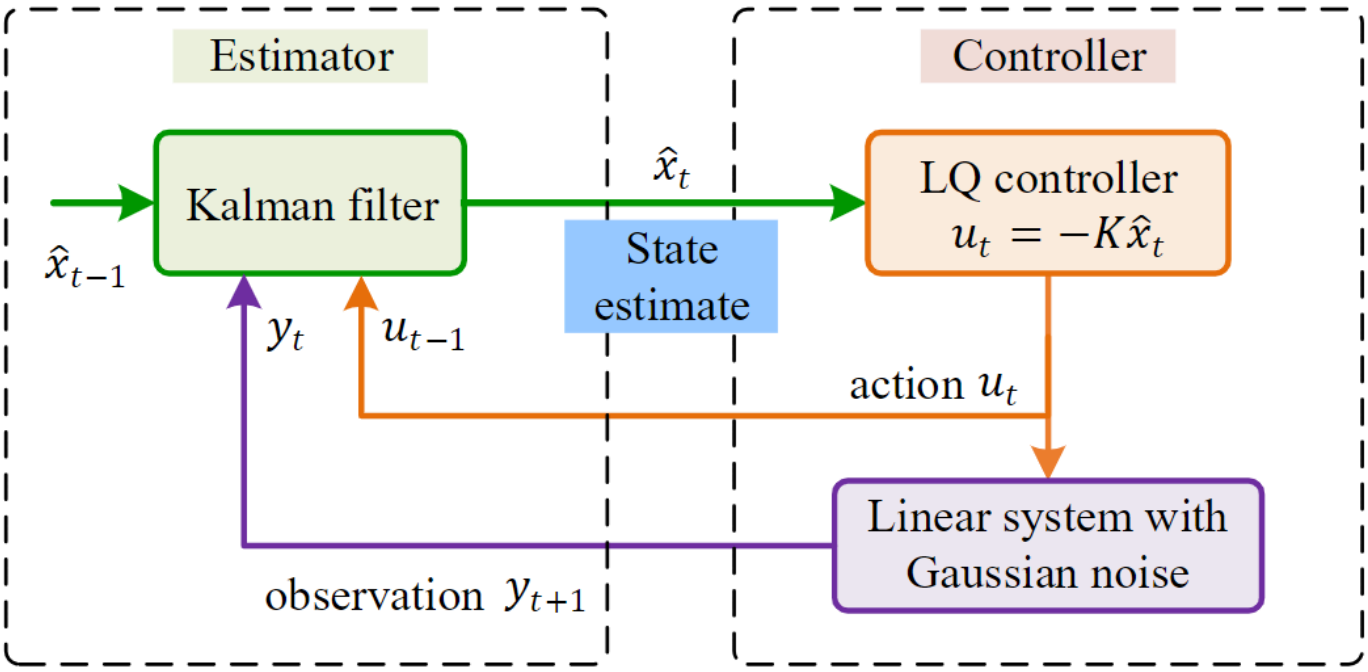
$$
J^*(x_t) = \mathbb{E}\{x_t^{\mathrm{T}} P^* x_t|b_{t-1}\},
$$

这里的$P^* = P^{*\mathrm{T}} > 0$是代数Riccati方程的解。最优策略是：

$$
\begin{aligned}
u_t &= -K_\gamma \mathbb{E}\{x_t|b_t\}, \\
K_\gamma &= \gamma(\gamma B^{\mathrm{T}} P^* B + R)^{-1} B^{\mathrm{T}} P^* A.
\end{aligned}
$$

证明见原书11.2.3节第419页。

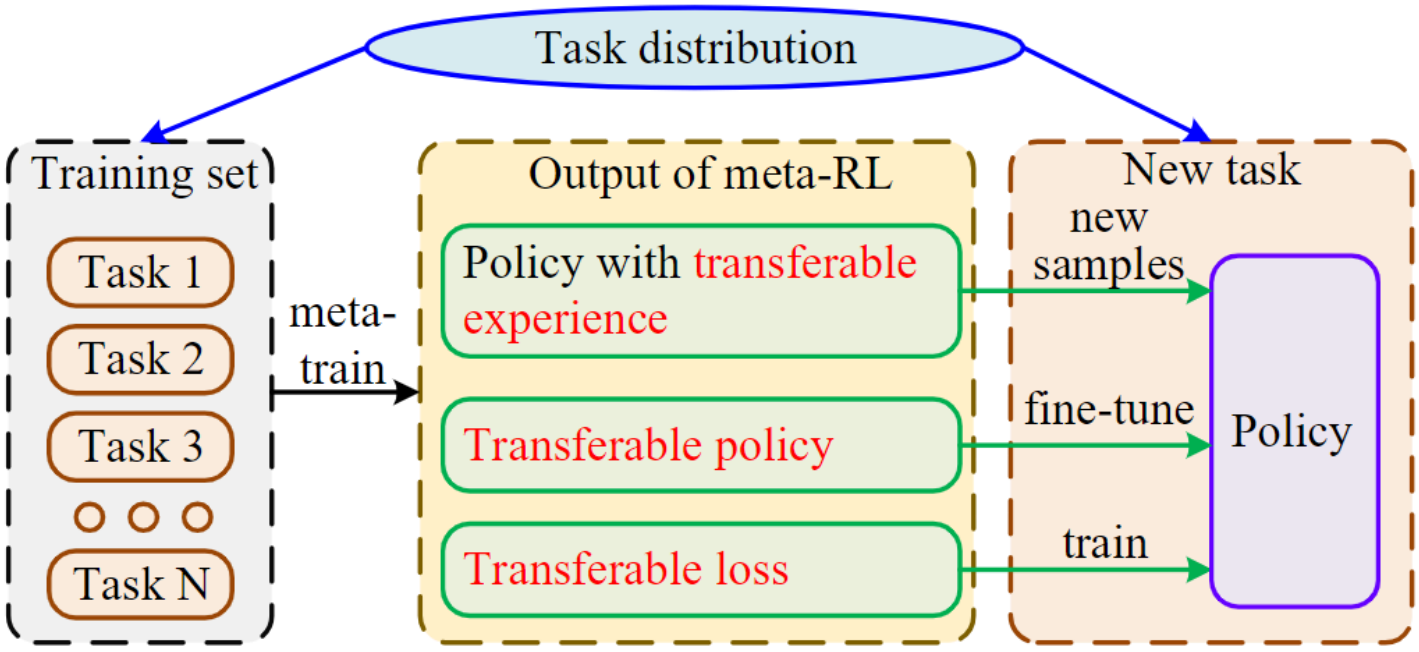
LQG的两阶段求解过程如下图所示：



在传统的LQG中，当系统状态的维度过大时难以找到最优解。通过限制控制状态的的维度到一个有限的范围，可以得到降阶的LQG问题。另外，为了提升LQG的鲁棒性，可以通过精细地调节超参数或在部分系统参数带有随机性或未知的情况下来设计和测试LQG控制器。

11.3 元强化学习（Meta Reinforcement Learning）

强化学习的一大痛点是需要大量的交互数据来训练一个好的策略，这与生物学习的方式截然不同。能否在已有知识的基础上通过少量样本和尝试快速学习新技能、适应新环境？这就是元学习（Meta Learning）研究的问题。元学习有很多别名，如学习如何学习（Learning to Learn）、多任务学习（Multi-task Learning）、few-shot学习（Few-shot Learning）、终身学习（Lifelong Learning）及迁移学习（Transfer Learning）等。



在元强化学习中，我们考虑一个任务分布$\mathcal{T}_i \sim \rho(\mathcal{T})$，每个任务$\mathcal{T}_i$都是一个不同的MDP问题，其奖励函数$R_i$和状态转移函数$P_i$各不相同。元强化学习的目标是通过在$\rho(\mathcal{T})$中的一些任务上学到可迁移的知识后再结合少量的样本或尝试来快速适应新任务。

根据可迁移知识的类型不同，元强化学习可以分为：

- **基于可迁移经验的元强化学习**
- **基于可迁移策略的元强化学习**
- **基于可迁移损失函数的元强化学习**

可迁移经验的元强化学习通过在其它相关任务上学习来获得可迁移的经验。这类算法不探索新的任务或在新的任务上微调。这类算法通常是由一个循环神经网络（RNN）来实现的，因此也被成为recurrent model-based meta reinforcement learning。尽管这类方法可以通过一次前向传播就快速适应，但是其渐进地改进算法表现的能力有限。因为其学到的策略通常不对应于一个收敛的优化过程，也不保证能持续改进。

而后两种方法则是通过通过学习可迁移的函数来实现的。前者学习一个可迁移的策略函数，后者则学习一个可迁移的损失函数。这两类算法可以迅速通过梯度下降来适应新任务，因此也被称为gradient-based meta reinforcement learning。与基于可迁移经验的元强化学习相比，这两类方法允许通过梯度下降来渐进地改进算法表现。

11.3.1 基于可迁移经验的元强化学习

11.3.1.1 RL^2算法

RL^2算法在任务分布$\rho(\mathcal{T})$上学习。它使用循环神经网络等模型作为策略，并且相较于传统的强化学习以状态s_t或动作状态对(s_t, a_t)作为输入，RL^2则**使用四元组(s_t, a_t, r_t, d_t)作为输入**，其中d_t是终止标志，表示当前episode是否结束。在同一个任务的不同episode之间，RNN的隐状态是保留的，这样就可以在不同episode之间共享经验（不过在不同任务之间隐状态是不保留的）。RL^2的目标是**最大化单一任务的累计折扣奖励，而不仅仅是一个episode内的累计折扣奖励（RL通常的做法）**。

RL^2适应能力来源于以下几点：

- **使用包含更多信息的四元组输入取代状态或状态动作对输入**
- **新任务和老任务来自于同一个任务分布，具有一定的相似性**
- **同一个任务的不同episode之间共享经验（RNN的隐状态）**：这样能够保存更长时域的经验，能够更好地适应新任务。

11.3.1.2 SNAIL

经典的RNN的线性时域依赖限制了其在输入流上进行复杂计算的能力。SNAIL（Simple Neural Attentive Meta-Learner）算法则引入了两种改进来解决上述问题：

- **时域卷积（Temporal Convolution，TC）**：TC层能够更直接、高带宽地处理固定大小的输入序列。但是其缺点在于需要的TC层数随着输入序列长度的增加而对数级增长。
- **软注意力（Soft Attention）**：软注意力机制能够在一个可能非常大的输入序列上定位到关键的信息。

SNAIL算法通过交替使用TC层和软注意力层来处理输入序列，前者负责聚合信息，后者则定位到关键的信息。通过在端到端模型里使用注意力机制，SNAIL能够从收集的经验中提取出关键信息，并且比传统的基于RNN的方法更好训练，并且可以在一次前向传播中处理整个序列。

11.3.2 基于可迁移策略的元强化学习

基于可迁移策略的元强化学习的目标是学习一个策略，使得其可以通过梯度下降迅速适应任务分布$\rho(\mathcal{T})$中的新任务。这类方法允许渐进地改进算法表现。这类方法可分为三列，下面将具体介绍。

11.3.2.1 预训练（Pre-training）

预训练是三种里面最简答的一种，首先在一系列任务上训练，之后再特定任务上使用梯度下降来微调。

但是，简单地使用预训练方法在一些情形下甚至不如随机初始化。这是因为在不同任务上预训练之后策略倾向于输出一个平均的结果。甚至在某些情形下，策略对于实际的领域知识学习的很少，取而代之的是学到了输出空间的范围。

11.3.2.2 Model-Agnostic Meta-Learning（MAML）

MAML的核心思想是找到在任务学习中对于变化敏感的策略参数（参数的微小改变会带来任意属于任务分布$\rho(\mathcal{T})$的新任务上的巨大loss改善）。MAML是通过内外双循环来实现上述目标的。

内循环中，通过优化在某个从$\rho(\mathcal{T})$中采样的任务$\mathcal{T}_i$来优化策略参数θ：

$$\theta_i' = \theta + \alpha_\theta \nabla_\theta \mathbb{E}_{S\sim T_i, \pi_\theta}\{V_i^{\pi_\theta}(s)\}.$$

外层循环则是通过遍历每个属于$\rho(\mathcal{T})$的训练任务$\mathcal{T}_i$来更新策略参数θ（注意，外层循环包含内层循环）：

$$\theta^* = \arg\max_{\theta} \mathbb{E}_{T_i \sim \rho(T)} \mathbb{E}_{s \sim T_i, \pi_{\theta_i'}} \left\{ V_i^{\pi_{\theta_i'}}(s) \right\}.$$

MAML内外双循环的设计与其核心思想相呼应。内循环模拟新任务快速适配（参数微小改变），外循环则是跨任务适应。MAML允许智能体对于新任务通过一次梯度下降来进行few-shot学习。

11.3.2.3 Model-Agnostic Exploration with Structured Noise（MAESN）

MAESN也采用内外双循环的方式来学习一个策略，并在此基础上学习结构化的探索噪声（而不是像传统强化学习简单地向动作中添加随机噪声）。MAESN的噪声z服从分布$q_\omega(z)$:

$$z \sim q_\omega(z).$$

比如，若这个分布是高斯分布，那么ω就是均值和标准差。注意，**噪声z一个episode只sample一次**，这样能保证时间上连续的随机性。并且这种结构化的噪声也能够将之前的知识编码到噪声参数ω中。

与MAML类似，MAESN内层循环如下：

$$\theta_i' = \theta + \alpha_\theta \nabla_\theta \mathbb{E}_{z_i \sim q_{\omega_i}, S \sim T_i, \pi_\theta(s, z_i)} \{ V_i^{\pi_\theta(s, z_i)}(s) \},$$

这里的$\pi_\theta(s, z_i)$是加入噪声z_i的策略。与MAML不同的是，MAESN内层循环还要更新噪声参数ω：

$$\omega_i' = \omega_i + \alpha_\omega \nabla_{\omega_i} \mathbb{E}_{z_i \sim q_{\omega_i}, s \sim T_i, \pi_\theta(s, z_i)} \{ V_i^{\pi_\theta(s, z_i)}(s) \}.$$

外层循环则是：

$$\{\theta^*, \omega_i^*\} = \arg\max_{\{\theta, \omega_i\}} \mathbb{E}_{\mathcal{T}_i \sim \rho(\mathcal{T})} \left\{ \mathbb{E}_{z_i' \sim q_{\omega_i'}, s \sim \mathcal{T}_i, \pi_{\theta_i'}(s, z_i')} \left\{ V_i^{\pi_{\theta_i'}(s, z_i')}(s) \right\} - D_{\mathrm{KL}}(q_{\omega_i'} \| p(z)) \right\},$$

上述公式基本与MAML类似。不同的是这里额外加的$D_{\mathrm{KL}}(q_{\omega_i'} \| p(z))$是相对于先验分布$p(z)$的KL散度。这个KL散度项是为了使噪声分布贴近于先验分布。

11.3.3 基于可迁移损失函数的元强化学习

基于可迁移策略的元强化学习通过将可迁移的知识来显示地编码到策略参数中来实现元学习。而基于可迁移损失函数的元强化学习则是通过将可迁移的知识隐式地编码到损失函数中来实现相同的目标。这样智能体就可以使用这种可迁移的损失函数来快速适应新任务。其代表性算法为Evolved Policy Gradient（EPG）。

EPG算法与传统的Policy Gradient算法最大的不同就是目标函数是通过学习而不是直接指定的。其目标函数形式如下：

$$L_\varphi(\pi_\theta, \tau),$$

这里的$\tau = \{s_0, a_0, r_0, s_1, a_1, r_1, \ldots\}$是一个采样得到的，由状态、动作和奖励组成的序列。EPG算法同样采用内外双循环的方式。内层循环是通过外层循环提供的损失函数（损失函数在内层看成是固定的）来更新策略参数：

$$\theta_i' = \theta - \alpha_\theta \nabla_\theta \mathbb{E}_{\tau \sim \mathcal{T}_i, \pi_\theta} \{ L_\varphi(\pi_\theta, \tau) \}.$$

外层循环则是通过优化损失函数的参数φ来最大化累计奖励的期望：

$$\varphi^* = \arg\max_{\varphi} \mathbb{E}_{\mathcal{T}_i \sim \rho(T)} \mathbb{E}_{S \sim \mathcal{T}_i, \boldsymbol{\pi}_{\theta_i'}} \left\{ V_i^{\boldsymbol{\pi}_{\theta_i'}}(s) \right\}.$$

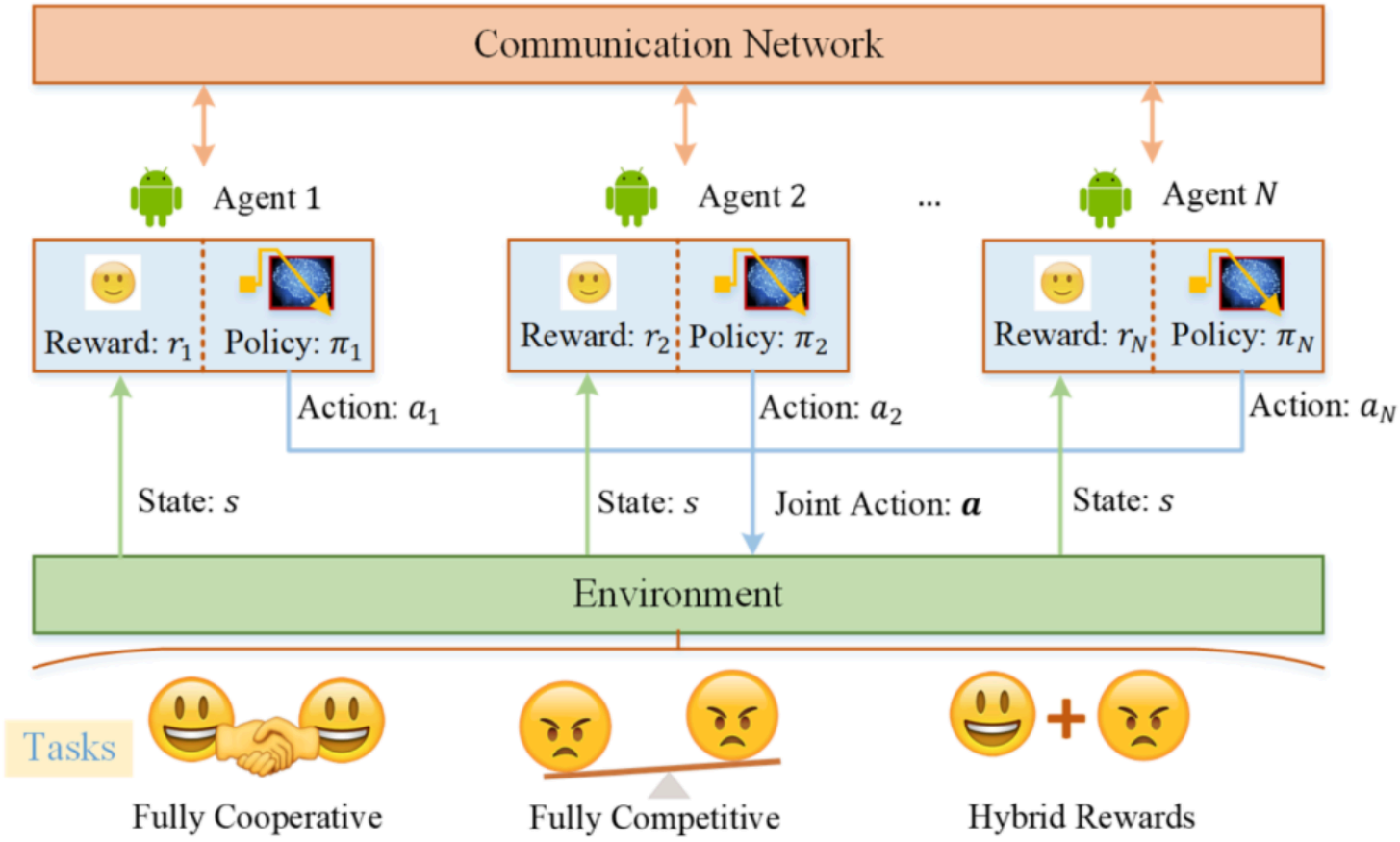
需要注意的是上式中的$V_i^{\boldsymbol{\pi}_{\theta_i'}}(s)$无法显示地表示为损失函数$L_\varphi(\pi_\theta, \tau)$的函数，因此在实际中需要使用进化策略（Evolution Strategy）来优化。

11.4 多智能体强化学习（Multi-Agent Reinforcement Learning，MARL）

多智能体强化学习（MARL）也被称为分布式强化学习（Distributed Reinforcement Learning），是指多个智能体在同一个环境中进行学习和决策的过程。每个智能体都需要考虑自己与环境及其它智能体的交互。

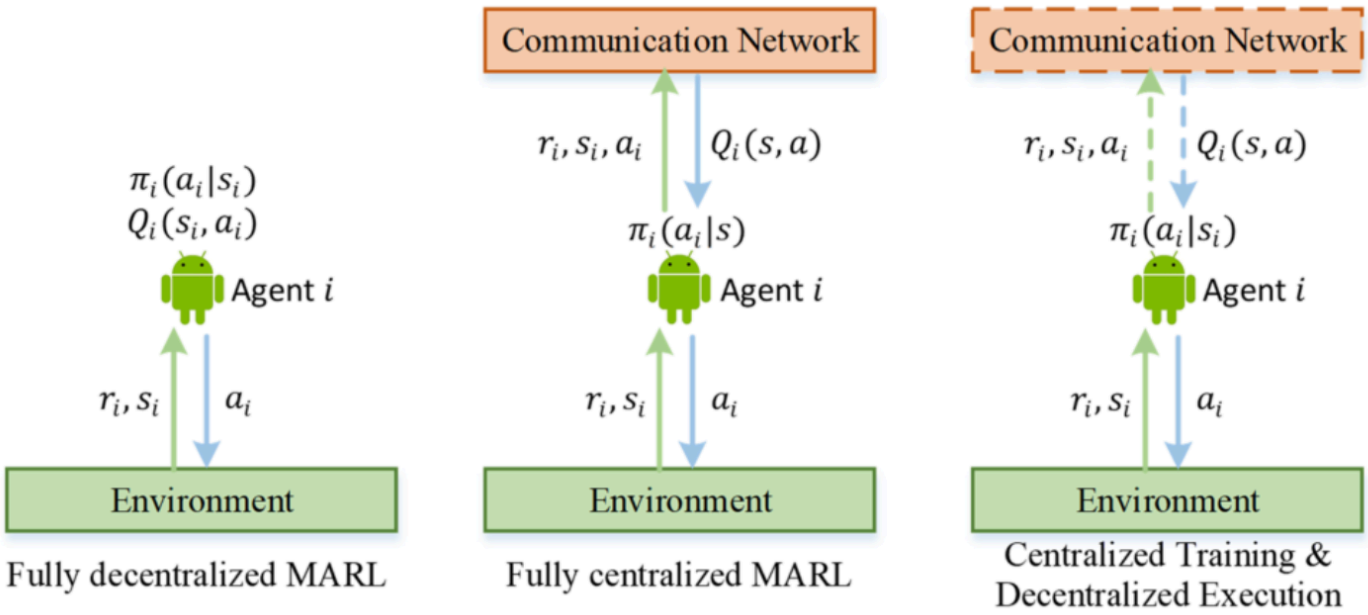
一个最简单的想法是对于每个智能体沿用单智能体强化学习算法，而把其它智能体当做环境的一部分。但是这种做法**会导致环境非平稳，因此破坏了马尔可夫性质的假设**。当其它智能体也根据他们之前的交互来更新策略时，这种做法就失效了。

MARL有许多分类方式，下面将介绍两种常见的分类方式。



第一种分类方式是根据智能体之间的关系来分类。如上图所示，MARL可以分为：

- **完全合作（Fully Cooperative）**：代表算法有distributed Q-learning、frequency maximum Q-Value等。
- **完全竞争（Fully Competitive）**：这类问题通常被建模为两玩家的零和博弈（Two-Player Zero-Sum Game），代表算法有minimax Q-learning。
- **混合奖励（Hybrid Rewards）**：这类算法每个agent的reward可能与其它agent的reward冲突，也可能一致。代表算法有Nash Q-learning、friend-or-foe、mean field regime等。



第二种分类方式是根据环境的部分可观性来分类。根据每个智能体是否能观测到完整的环境状态s还是只能观测到部分状态s_i，MARL可以分为：

- **Fully Decentralized**：智能体只能根据局部信息s_i来决策和更新值函数。然而，这种设置忽视了MARL系统的本质。
- **Fully Centralized**：智能体可以根据完整的环境状态s来决策和更新值函数，可以获取到其它智能体的动作和观测。然而，这种设置在实际中难以实现。
- **Centralized Training with Decentralized Execution（CTDE）**：平衡上述两种方式的做法。智能体使用完整的环境状态s来训练和更新值函数，但是在策略函数决策时只能使用局部信息s_i。

Method	Policy	Value function	Typical algorithm
Fully decentralized	$\pi_i(a_i\|s_i)$	$Q_i(s_i,a_i)$	Single - agent RL algorithm
Fully centralized	$\pi_i(a_i\|s)$	$Q_i(s,a)$	Nash Q - learning
CTDE	$\pi_i(a_i\|s_i)$	$Q_i(s,a)$	MADDPG, VDN

11.4.1 多智能体强化学习建模

多智能体强化学习的标准建模为随机多智能体博弈（Stochastic Multi-Agent Game），是MDP的推广。它被记作:

$$\{\mathcal{S},\mathcal{A},\mathcal{P},\mathcal{R}\},$$

其中，$\mathcal{S}$是各个智能体共享的状态空间，$\mathcal{A}=\mathcal{A}_1\times\mathcal{A}_2\times\ldots\times\mathcal{A}_N$是联合动作空间，$\mathcal{P}$是状态转移函数，$\mathcal{R}$是联合奖励函数。对于智能体$i$，我们使用下面的简记类表示除它之外其它所有智能体：

$$-i=\{1,2,\cdots,N\}\backslash\{i\}.$$

那么，联合动作a可以表示为：

$$a=(a_i,a_{-i})\in\mathcal{A},$$

联合策略π可以表示为：

$$\pi(a|s)=\prod_i\pi_i(a_i|s)=\pi_1(a_1|s)\pi_2(a_2|s)\cdots\pi_N(a_N|s),$$

那么，联合（状态）值函数V可以表示为：

$$V_i^{\pi}(s)=\sum_{a\in\mathcal{A}}\pi\sum_{s'\in\mathcal{S}}\mathcal{P}\big(r_i+\gamma V_i^{\pi}(s')\big),i=1,2,\cdots,N,$$

这里的$\pi=\pi(a|s)$是联合策略，$\mathcal{P}=\mathcal{P}(s'|s,a)=\mathcal{P}(s'|s,a_i,a_{-i})$是状态转移函数，$r_i=r_i(s,a,s')=r_i(s,a_i,a_{-i},s')$是联合奖励函数。接下来，就可以进一步得出对于智能体i的最优策略：

$$\begin{aligned}\pi_i^*(a|s,\pi_{-i})&=\arg\max_{\pi_i}V_i^{(\pi_i,\pi_{-i})}(s)\\&=\arg\max_{\pi_i}\sum_{a\in\mathcal{A}}\pi_i(a_i|s)\pi_{-i}(a_{-i}|s)\sum_{s'\in\delta}\mathcal{P}\left(r_i+\gamma V_i^{(\pi_i,\pi_{-i})}(s')\right).\end{aligned}$$

这里的$V_i^{(\pi_i,\pi_{-i})}(s)$与刚才的$V_i^{\pi}(s)$是一样的，只不过使用了简记记号。

另外注意，其它智能体的联合策略$\pi_{-i}(a_{-i}|s)$是非平稳的，也就是说会随着时间的推移而变化，因此MARL的收敛性很难保证。往往需要很强的假设。比如，对于Nash Q-learning来说，需要鞍点或是纳什均衡点。对于随机博弈来说，一个常见的收敛性质是纳什均衡（Nash Equilibrium）：

$$V_i^{(\pi_i^*,\pi_{-i}^*)}(s)\geq V_i^{(\pi_i,\pi_{-i}^*)}(s),\forall\pi_i.$$

纳什均衡确定的平衡点π^*保证了没有智能体有偏离该平衡点的动机。在该平衡点下，每个智能体的最优策略π_i^*都是对其它智能体的最优策略π_{-i}^*的最佳响应。对于折扣或平衡奖励的随机博弈，纳什均衡点总是存在的，尽管可能不是唯一的。

11.4.2 完全合作（Fully Cooperative）MARL

完全合作的MARL问题占据了大多数的MARL问题，按照奖励函数的不同可分为两类：

- **相同奖励**：所有智能体的奖励函数相同，即：

  $$r_1 = r_2 = \cdots = r_N = r,$$

  因此，**对于所有智能体来说值函数是相同的**。如果所有智能体被协调为一个决策者，那么就可以使用单智能体强化学习算法。
- **平均奖励**：在该设定下，每个智能体的奖励函数是不同的，优化目标是所有智能体长期的的平均奖励：

  $$\bar{r} = \frac{1}{N}\sum r_i.$$

  这种情况下，智能体之间需要更多的协作，因为无法在不知道其它智能体奖励函数的情况下在局部（locally）估计全局的值函数。相同奖励的MARL问题可以看成是平均奖励的特例。

这里介绍两种常见的完全合作MARL算法：

- **分布式Q学习（Distributed Q-learning）**：该算法中各个智能体通常具有相同的奖励函数。Distributed Q-learning就是把Q-learning算法推广到多智能体的情形。每个智能体i都有自己的Q值函数$Q_i(s,a_i)$和策略π_i。每个智能体的Q函数更新如下：

  $$Q_i(s,a_i) \leftarrow \max\left\{Q_i(s,a_i), r(s,a_i) + \gamma \max_{a_i'} Q_i(s',a_i')\right\}.$$

  按照上述公式更新得到的Q函数理论上应该满足下式：

  $$Q_i(s,a_i) = \max_{a_{-i}} \widehat{Q}_i(s,a),$$

  这里的$\widehat{Q}_i(s,a)$是智能体i的联合Q函数（即输入的动作为联合动作$a=(a_i,a_{-i})$）。这个式子的意思是智能体i的局部Q函数$Q_i(s,a_i)$应该在其他智能体所有可能动作a_{-i}里，选让$\hat{Q}_i(s,a)$最大的情况。这么做是想让智能体i的局部决策，尽可能适配全局联合动作的最优价值。但是实际上，该算法的收敛性并不一定。如果存在多个平衡点，那么每个智能体各自平衡点的组合并不一定是全局平衡点。各个智能体的策略可能朝着不同的方向更新，因此可能导致次优的解。
- **频率最大Q值（Frequency Maximum Q-Value，FMQ）**：该算法特别之处在于**其状态空间是空的（没有状态空间）**。智能体选择动作的依据是这个动作在过去是否有好的结果。而一个动作好不好是通过学到的其它智能体的模型或是访问次数等统计数据来评估的。每个智能体采用玻尔兹曼探索策略。修改后的Q函数如下：

  $$\tilde{Q}_i(a_i) = Q_i(a_i) + \lambda \frac{C^i_{\max}(a_i)}{C^i(a_i)} r_{\max}(a_i),$$

  这里的$r_{\max}(a_i)$是选择动作a_i后观察到的最大奖励，$C^i(a_i)$是动作a_i的访问次数，$C^i_{\max}(a_i)$是采取动作a_i后观察到最大奖励的次数。在该框架下，过去产生好的结果会驱动Q函数的增加。这实际上可以看成是一种统计学习方法，与蒙特卡洛树搜索（MCTS）类似。

11.4.3 完全竞争（Fully Competitive）MARL

完全竞争的MARL问题通常被建模为零和博弈问题，即：

$$\sum r_i(s,a,s') = 0.$$

而且大多数情况下玩家数被限定为2，因为即使是最简单的三人矩阵博弈也是PPAD-完全（Polynomial Parity Argument on Directed Graphs，有向图多项式奇偶性论证 ）问题。零和博弈通常使用纳什均衡来求解。下面介绍一种常见的完全竞争MARL算法：**Minimax Q-learning**。

这里我们考虑两个智能体，主智能体记作A，对手智能体记作B。值函数满足：

$$V_A^{(\pi_A,\pi_B)}(s) = -V_B^{(\pi_B,\pi_A)}(s).$$

纳什均衡策略满足：

$$V_A^{(\boldsymbol{\pi}_A,\boldsymbol{\pi}_B^*)}(s) \leq V_A^{(\boldsymbol{\pi}_A^*,\boldsymbol{\pi}_B^*)}(s),$$
$$V_B^{(\boldsymbol{\pi}_A^*,\boldsymbol{\pi}_B)}(s) \leq V_B^{(\boldsymbol{\pi}_A^*,\boldsymbol{\pi}_B^*)}(s).$$

因此根据minimax原理，最优值函数被定义为：

$$V_A^*(s) = \max_{\pi_A(\cdot|s)} \min_{a_B} \sum_{a_A} Q_A^*(s, a_A, a_B)\pi_A(a_A|s),$$
$$V_B^*(s) = \max_{\pi_B(\cdot|s)} \min_{a_A} \sum_{a_B} Q_B^*(s, a_A, a_B)\pi_B(a_B|s),$$

这里的$Q_A^*(s, a_A, a_B)$和$Q_B^*(s, a_A, a_B)$表示智能体遵循纳什策略选择动作各自得到的最优累计回报。那么，$V_i^*(s)$就可以通过求解一个线性规划问题来得到。而$Q_i^*(s, a_i, a_{-i})$本身的更新则类似于TD Learning的方式：

$$Q_A(s, a_A, a_B) \leftarrow (1-\alpha)Q_A(s, a_A, a_B) + \alpha\big(r_A(s, a_A, a_B) + \gamma V_A(s')\big),$$
$$Q_B(s, a_A, a_B) \leftarrow (1-\alpha)Q_B(s, a_A, a_B) + \alpha\big(r_B(s, a_A, a_B) + \gamma V_B(s')\big).$$

值得一提的是，Minimax Q-learning是“对手独立”的（Opponent Independent），即不管对手行为是什么或minimax优化有多个解，Minimax Q-learning中的每个智能体都至少达到minimax回报。

11.4.4 混合奖励（Hybrid Rewards）MARL

混合奖励的MARL问题对于各个智能体之间的关系没有限制，是最一般的MARL问题。这其实是一个general-sum博弈（General-Sum Game），因此难度也增加不少。即使是最简单的两玩家general-sum规范形式博弈也是PPAD-完全问题。只有在一些很强的假设下，才能保证收敛到纳什均衡。下面介绍三种常见的混合奖励MARL算法：

- **Nash Q-learning**：该算法具有智能体独立（Agent Independent）的性质，与Q-learning结构类似，但是其策略和状态值函数需要使用专门的博弈论求解器来求解。其核心更新公式如下：

  $$Q_i(s, a) \leftarrow (1-\alpha)Q_i(s, a) + \alpha(r_i(s, a) + \gamma \mathrm{Nash}[Q_i(s')]),$$

  这里的$\mathrm{Nash}[Q_i(s')]$是通过纳什策略得到的下一个状态的期望回报。Nash Q-learning在更新时需要知道所有智能体的Q表，每个智能体的Q表都应该可以被其它的智能体访问，因此这就要求所有智能体的动作和奖励可以被测量。当所有状态满足下面两个条件之一时可以保证收敛到纳什均衡：
  - 学习过程中遇到的每个博弈的阶段都必须有一个纳什均衡点（在该点所有智能体的的累计回报均最大）。
  - 每个博弈阶段都必须有一个纳什均衡点作为鞍点，此时任何一个智能体偏离该点都会使获益。
- **Friend-or-Foe**：对于每个智能体i，算法会将全部智能体分为两类：友方智能体（与i合作使得i的累计回报最大化，包含i自己）和敌方智能体（与i竞争使得i的累计回报最小化）。智能体i的值函数更新方式如下：

  $$V_i(s) = \max_{\pi_A} \min_{a_B} \sum_{a_A} Q_i(s, a_A, a_B)\pi_A\,,$$
  $$Q_i(s, a_A, a_B) \leftarrow (1-\alpha)Q_i(s, a_A, a_B) + \alpha\big(r_i + \gamma V_i(s')\big),$$

  这里的集合A是智能体i的友方智能体集合，集合B是敌方智能体集合。$a_A = \{a_1 \quad \cdots \quad a_{A_N}\}, a_B = \{a_1 \quad \cdots \quad a_{B_N}\}$。$A_N$和$B_N$分别是友方智能体（包含自己）和敌方智能体的数量。而$\pi_A = \prod_{i=1}^{A_N} \pi_i(a_i|s)$是友方智能体的联合策略。与Nash Q-learning相比，Friend-or-Foe算法不需要估计对方的Q值，并且能提供更强的收敛性保证。
- **Mean Field Regime**：该算法是为了解决智能体数量的scalability问题而提出的。在该算法下，每个智能体都有local的状态s_i和动作a_i以及一个邻居集合。那么，智能体之间的交互就可以使用聚合效果（Mean Field）μ来建模。以智能体i为例，其μ_i如下：

  $$\mu_i = \frac{1}{N_i}\sum_j a_j\,, a_j \sim \pi_j\big(\,\cdot\,|s_i, \mu_j\big),$$
  $$\pi_i(a_i|s_i, \mu_i) = \frac{\exp(-\beta Q_i(s_i, a_i, \mu_i))}{\sum_{\tilde{a}_i \in \mathcal{A}_i}\exp(-\beta Q_i(s_i, \tilde{a}_i, \mu_i))}\,,$$

  这里的N_i是智能体i的邻居数量，a_j是邻居智能体j采取的动作。这里的策略π_i的输入不仅包含状态s_i，还包含聚合效果μ_i。因此，在这种mean field游戏中，通过聚合效果μ_i来对于交互建模，每个智能体实际面临的是一个时变的MDP问题。它的最优解是一个mean field平衡点（mean field equilibrium），为最优策略和聚合效果的组合(π^*, μ^*)。在该平衡点下，π^*是时变MDP的最优策略，而μ^*是当所有智能体均遵循π^*时的聚合效果。

11.5 逆强化学习（Inverse Reinforcement Learning，IRL）

逆强化学习（IRL）是指从专家的行为中推断出奖励函数的过程，也被称为学徒学习（Apprenticeship Learning）、模仿学习（Imitation Learning）或通过示教学习（Learning by Demonstration）。

传统的强化学习通常假设奖励函数已知，但实际上奖励函数在实际应用中有时很难手工显示地指定，常常需要手工指定后观察效果再反复修正。IRL的目标是从专家的行为中推断出奖励函数。这不是简单地模仿专家的动作，而是试图理解为什么这样决策。尽管不一定能完美还原出专家的奖励函数，但是IRL算法往往能得到与专家性能相当的策略。IRL主要可分为两类：最大间隔逆强化学习（Maximum Margin Inverse Reinforcement Learning）和最大熵逆强化学习（Maximum Entropy Inverse Reinforcement Learning）。

11.5.1 逆强化学习问题建模

IRL假设奖励函数可以被表示为已知特征的线性组合：

$$r_\psi(s,a) = \psi^{\mathrm{T}} f(s,a),$$

这里的ψ是特征权重，而$f = [f_1, f_2, \ldots, f_n]^{\mathrm{T}}$是特征向量，其中的每个$f_i$都是一个特征函数，将状态和动作映射到实数。IRL将轨迹上的状态-动作对序列记为τ：

$$\tau = \{(s_0,a_0),(s_1,a_1),\ldots,(s_{T-1},a_{T-1})\}.$$

定义轨迹上特征的累计值为：

$$\mu(\tau) = \sum_{i=0}^{T-1} \gamma^i f(s_i,a_i).$$

那么，IRL学到的奖励应该使得专家轨迹和模仿出来的轨迹的特征累计值的期望相等：

$$\mathbb{E}_{\pi\psi}\{\mu(\tau)\} = \mathbb{E}\{\mu(\tau^*)\},$$

这里的π_ψ是IRL学到的对于参数化奖励函数$r_\psi(s,a)$的策略，τ^*是专家轨迹。关键问题是**如何计算左右两边的期望**。右边的期望好说，只通过专家轨迹就可以计算出来。但是左边的期望要求首先求解RL问题，之后使用学到的策略再与环境交互得到samples。

另外，给定一系列专家轨迹，也可能存在多种可能的奖励函数使得上述等式成立。因此，IRL算法通常在某种标准下选择最优的奖励函数。常见的标准有最大间隔（Maximum Margin）和最大熵（Maximum Entropy）。

另外注意的是，**逆强化学习学习奖励函数只是第一步，之后还需要在已学得的奖励函数的基础上使用强化学习算法来学习策略**。IRL通常是两阶段的：第一阶段是学习奖励函数，第二阶段是使用强化学习算法来学习策略。

11.5.2 最大间隔逆强化学习

最大间隔逆强化学习（Maximum Margin Inverse Reinforcement Learning，MMIRL）的核心思想在于最大化专家轨迹与其它策略的轨迹之间的间隔，从而使得第一阶段学到的奖励函数可以最大限度地分离专家轨迹与其它轨迹，从而使得第二阶段学到的策略能够更贴近专家轨迹而不是学到其它次优策略。

MMIRL的目标是最大化专家轨迹与其它轨迹之间的间隔：

$$\begin{aligned} &\max_{\psi} m, \\ &\text{s.t.} \\ &\psi^{\mathrm{T}}\mathbb{E}\{\mu(\tau^*)\} \geq \max_{\psi} \psi^{\mathrm{T}}\mathbb{E}_{\pi_\psi}\{\mu(\tau)\} + m, \end{aligned}$$

这里的m是间隔，τ^*是专家轨迹，τ是其它轨迹。注意到上述问题构建与支持向量机（SVM）类似，本质是找到最大间隔的超平面来区分两组点，因此可以使用SVM的求解方法来求解。可转化为下述优化问题：

$$\begin{aligned} &\max_{\psi} \frac{1}{2}\|\psi\|^2, \\ &\text{s.t.} \\ &\psi^{\mathrm{T}}\mathbb{E}\{\mu(\tau^*)\} \geq \max_{\psi} \psi^{\mathrm{T}}\mathbb{E}_{\pi\psi}\{\mu(\tau)\} + 1. \end{aligned}$$

之后就可通过二次规划的求解器来求解奖励函数。MMIRL的优缺点如下：

- **优点**：
  - 几次迭代快速收敛
  - 保证最终策略与专家策略性能相似
- **缺点**：
  - 容易过拟合，因为最大化间隔导致了无法考虑到其它次优的情况

11.5.3 最大熵逆强化学习

最大熵逆强化学习（Maximum Entropy Inverse Reinforcement Learning，MEIRL）的核心思想在于最大化观测到同样的专家轨迹的概率。最大熵IRL原始优化问题如下：

$$\begin{aligned}&\max_{\psi}\mathcal{H}(\pi_\psi),\\&\text{s.t.}\\&\mathbb{E}_{\pi\psi}\{\mu(\tau)\}=\mathbb{E}\{\mu(\tau^*)\},\end{aligned}$$

这里的$\mathcal{H}(\pi_\psi)$是策略π_ψ的熵。已经证明上述优化问题的最优解形式如下：

$$p(\tau|\psi)=\frac{p(\tau)e^{\psi^{\mathrm{T}}\mu(\tau)}}{\int p(\tau)e^{\psi^{\mathrm{T}}\mu(\tau)}\mathrm{d}\tau},$$

这里的$p(\tau)$是使用从均匀分布中采样得到的动作产生的轨迹的概率密度。因此，上述优化问题可以转化为：

$$\max_{\psi}L(\psi).$$

这里的$L(\psi)$是对数似然函数，形式如下：

$$\begin{aligned}L(\psi)&=\sum_{i=1}^{N}\log p(\tau_i^*|\psi)\\&=\sum_{i=1}^{N}\left[\log p(\tau_i^*)+\psi^{\mathrm{T}}\mu(\tau_i^*)-\log\int p(\tau)e^{\psi^{\mathrm{T}}\mu(\tau)}\mathrm{d}\tau\right],\end{aligned}$$

这里的τ_i^*是第i个专家轨迹。可进一步证明上述目标函数的梯度为：

$$\nabla L(\psi)=\mathbb{E}\{\mu(\tau^*)\}-\mathbb{E}_{\tau\sim p(\tau|\psi)}\{\mu(\tau)\}.$$

关于上述梯度的一个直观的理解是当某条轨迹的期望值大于对应的专家轨迹的期望值时，需要减少对应的特征的权重ψ，这样又降低了所有具有高$\mu(\tau)$的轨迹的概率，从而使得期望值$\mathbb{E}_{\tau\sim p(\tau|\psi)}\{\mu(\tau)\}$减小。

该算法通过引入概率视角优雅地解决了过拟合问题，并提供了一个计算高效的凸优化问题供求解，并能保证学到的策略的性能。

11.6 离线强化学习（Offline Reinforcement Learning）

在真实世界中使用不完美的策略探索环境收集数据可能是昂贵的，甚至可能是危险的。而许多问题又过于复杂，难以进行高保真建模。使用低精度的仿真环境进行训练又会导致sim2real gap。另一方面，很多现实系统又收集了大量数据（如电动汽车就包含了大量传感器数据），这些数据中蕴含了丰富的信息。离线强化学习（Offline Reinforcement Learning，也被称为批量强化学习，Batch Reinforcement Learning）就是从这些离线的、静态的数据中学习策略，而不是通过与环境交互来学习。

离线强化学习首先使用behavior policy（行为策略）$\pi_\beta(a|s)$从环境中收集数据，得到数据集$\mathcal{D}=\{(s_t,a_t,r_t,s_{t+1})\}$，之后通过这些数据来学到策略$\pi_\theta(a|s)$。

然而，离线强化学习也面临诸多挑战，比较典型的挑战有：

- **只能从静态数据集中学习**：离线强化学习不能通过进一步地与环境交互来收集更多的样本。然而，静态数据集$\mathcal{D}$可能只能覆盖了状态-动作空间的一部分，因此贝尔曼算子可能没有被应用到真实的MDP中，而成为一个有偏（biased）的算子。采样误差和函数近似误差在离线的设定下尤为严重。

- **分布偏移（Distribution Shift）**：函数近似器（策略函数和值函数）在一个分布（静态数据集$\mathcal{D}$）上训练，然而可能在另一个分布（真实分布）上验证。造成分布偏移的原因包括与学到的策略相关的边缘分布的变化以及优化过程的max算子。当与策略相关的数据分布与真实分布偏离时，可能会出现OOD（Out-of-Distribution）问题。而最大化过于乐观估计的OOD动作会导致严重的过估计

因此，离线强化学习通常还需要使用一个增长的在线sample批量来提升表现。

最近的离线强化学习算法通常使用如下方法来解决分布偏移问题：

- **Policy Constraint Offline RL**：限制学到的策略尽量贴近行为策略$\pi_\beta(a|s)$。
- **惩罚OOD动作**：一些基于模型的离线强化学习算法会在reward函数中惩罚OOD动作。
- **值函数正则化**：修改Q函数训练目标来实现较为保守的行为。
- **基于不确定性的离线强化学习**：这类方法利用Q函数的认知不确定性对分布外（OOD）的动作进行惩罚；
- **In-Sampling离线强化学习**：在已有的数据样本内学习值函数来避免OOD问题。
- **Goal-conditioned模仿学习**：将强化学习问题转化conditional、filtered或weighted的模仿学习问题，从而去除易受分布偶安逸影响的PEV过程。

离线强化学习正迅速发展，以下方向是当前的研究热点：

- **数据高效离线强化学习算法**：设计数据高效的离线强化学习算法，能够在覆盖状态-动作空间的有限的小数据集上学习到高性能的策略。
- **寻找更好的不确定性和泛化性评估指标**：为在未知数据样本上的模型或值函数找到更好的评估指标。
- **探索新的离线强化学习方案（scheme）**：应对分布偏移，同时避免过度保守的策略学习
- **提升策略的鲁棒性**：提升策略在有限数据或输入数据中存在不准确性时的鲁棒性。

方法	核心想法	优势	劣势
策略约束（BCQ、BEAR、TD3BC）	利用行为策略约束学习到的策略	当行为数据易于建模时效果良好； 建模框架简单直接	许多算法需要估计行为策略； 往往过于保守
基于模型（MOREl、MOPO）	利用不确定性对奖励进行惩罚	在高覆盖/混合数据集上效果好； 泛化性更佳	受学习到的动态模型影响大； 需要进行不确定性估计
价值正则化（CQL）	修改 Q 函数训练目标	通常表现良好	在连续动作空间上， 策略评估通常成本更高； 有时可能过于保守
基于不确定性（BEAR、MOPO、EDAC）	利用不确定性对 Q 函数进行惩罚	在高覆盖数据集上效果好； 通常表现良好	需要进行不确定性估计
样本内学习（IQL）	修改策略评估方案	表现良好；策略学习稳定	性能取决于数据质量
目标条件化模仿学习（RvS）	学习条件化策略	绕过策略学习过程中的分布偏移问题	最优策略仅在一些限制性假设下得到保证

11.6.1 Policy Constraint离线强化学习

Policy Constraint离线强化学习的想法很简单，既然OOD动作会在贝尔曼更新Q函数时造成误差，那么将提供给Q函数的动作限制在样本集$\mathcal{D}$里如何？那为了保证这一点，就需要限制学到的策略$\pi_\theta(a|s)$尽量贴近行为策略$\pi_\beta(a|s)$。因此，Policy Constraint离线强化学习的优化目标如下：

$$
\begin{gathered}
Q(s,a) \leftarrow r(s,a)+\gamma \mathbb{E}_{a^{\prime} \sim \pi_\theta\left(a^{\prime} \mid s^{\prime}\right)}\left\{Q\left(s^{\prime}, a^{\prime}\right)\right\}, \\
\pi=\arg \max _\pi \mathbb{E}_{s \sim \mathcal{D}, a \sim \pi(a \mid s)}\{Q(s,a)\} \text{ or } \arg \max _\pi \mathbb{E}_{s \sim \mathcal{D}, a \sim \pi(a \mid s)}\{A(s,a)\}, \\
\text{s.t.} \\
D(\pi(a|s), b(a|s)) \leq \epsilon,
\end{gathered}
$$

这里的$D(\cdot,\cdot)$是某种距离度量，而$b(a|s)$是行为策略。那么显然，Policy Constraint离线强化学习的核心为以下两点：

- **从数据集$\mathcal{D}$中学习得到行为策略$b(a|s)$**：可通过行为克隆（Behavior Cloning，BC）或隐式建模来实现
- **选择合适的距离度量$D(\cdot,\cdot)$**

Policy Constraint Offline RL可以根据约束的显式或隐式分为两类：

- **显式约束**
  显式约束的Policy Constraint Offline RL使用某些距离度量来衡量学到的策略和行为策略之间的距离。例如，BEAR使用maximum mean discrepancy（MMD）来衡量上述距离。在比如，Fujimoto等人发现即时使用最简单的行为克隆（Behavior Cloning，BC）作为距离度量就能得到很好的结果。由此他们开发了TD3BC（Twin Delayed Deep Deterministic Policy Gradient with Behavior Cloning）算法，其策略更新公式如下：

  $$\pi = \arg\max_{\pi} \mathbb{E}_{s,a\sim\mathcal{D}}\{\alpha Q(s,\pi(s)) - (\pi(s)-a)^2\},$$

  这里大括号里面的第一项是标准的强化学习项，第二项是行为克隆项。α是一个超参数，用来平衡两者的权重。
- **隐式约束**
  隐式约束不使用显式的距离度量，而是使用某种方法来迫使学到的策略尽量贴近行为策略。比如，BCQ算法通过向行为策略$\pi_\beta(a|s)$添加扰动来实现隐式约束：

  $$\pi_\theta(\cdot|s) = \arg\max_{a_i\sim b(\cdot|s)} Q\left(s, a_i + \xi_\theta(s,a_i)\right) \forall i,$$

  这里的$\xi_\theta(s,a_i)$是一个扰动模型，具有确定的上下界$\xi_\theta(s,a_i)\in[-\xi_{bnd},\xi_{bnd}]$。这里的思路可以这样理解，在添加了随机扰动的情况下若还要优化上述损失函数，那么只能尽量往中心值（即原始未添加扰动的行为策略）靠拢。

Model-free的策略梯度算法通常表现超过行为克隆。但是总体来说，Policy Constraint离线强化学习算法只有在单一模态（即行为模式）的数据集下采表现较好。对于多模态数据集，这类算法往往因为过于保守而无法学到好的策略。但是多模态数据集又是现实世界中常见的情况。另外，这种方法只能贴近而无法超越行为策略。

11.6.2 基于模型的离线强化学习

基于模型的离线强化学习（Model-based Offline Reinforcement Learning）并不是与model-based的在线强化学习那样使用模型，这里的模型只不过是为了度量**不确定性**。这里的思路其实也是让学到的策略尽量贴近行为策略，但是不像Policy Constraint离线强化学习那么保守。至于怎么让两个策略贴近？靠不确定性度量来实现。对于具有较大不确定性的的状态-动作对施加惩罚（逻辑是这样的：远离数据集的分布$\rightarrow$更不准确$\rightarrow$更大的不确定性$\rightarrow$更大的惩罚）。因此，这类方法主要是通过对奖励函数进行惩罚来实现的：

$$\tilde{r}(s,a) = r(s,a) - \varphi(s,a),$$

这里的$\varphi(s,a)$是对状态-动作对(s,a)的不确定性度量。不同的度量方法会导致不同的算法。下面介绍两种常见的基于模型的离线强化学习算法：MORel和MOPO。

MORel首先从数据集中学习一系列动力学模型$f(s,a;\phi_i), i=1,2,...,N$，之后根据这些模型来估计不确定性。MORel使用动力学模型来产生轨迹，当模型间的不一致性超过一定阈值时停止该条轨迹。其具体的奖励函数如下：

$$\tilde{r}(s,a) = \begin{cases} -R_{\max} & \text{if } \max_{i,j}(s,a;\phi_i) - f(s,a;\phi_j)\|_2 > \epsilon \\ r(s,a) & \text{otherwise} \end{cases}$$

MOPO算法也是学习一系列动力学模型，但这些动力学模型的输出是高斯分布的均值和标准差：

$$f(s'|s,a;\phi_i) = \mathcal{N}(\mu_i(s,a), \sigma_i^2(s,a)), i=1,2,...N$$

那么其奖励函数如下：

$$\tilde{r}(s,a) = r(s,a) - \max_i\{\|\sigma_i(s,a)\|_F\}.$$

这里是用标准差来度量相对于学到的动力学模型的不确定性。学到的动力学模型和新的奖励函数结合在一起来产生新的数据。之后，通过model-free的off-policy RL算法来学习策略。

Model-based Offline RL在高覆盖/混合数据集上上动力学模型能学得很好，表现得会比Policy Constraint Offline RL更不保守，效果也更好。另外因为使用了动力学模型，泛化性也更好。但其严重依赖于学到的动力学模型的质量，如何开发更好的度量不确定性，能准确检测到OOD状态-动作对是当前的研究中仍未有定论的难题。

11.6.3 价值正则化离线强化学习

这里提供了另一种解决OOD样本引起的过估计问题的方法：对于值函数进行正则化。**具体来说，既然会出现过估计，不如在学习值函数时就学一个低估（underestimated）的、较为保守的值函数**。这类方法里面最典型的方法就是CQL（Conservative Q-Learning）。其修改后的Q函数学习目标如下：

$$Q_{\mathrm{CQL}}^{\pi}=\arg\min_{Q}\mathbb{E}_{s,a,s'\sim\mathcal{D}}\left\{\left(Q(s,a)-Q^{\mathrm{target}}(s,a)\right)^2\right\}+\\ \alpha\cdot\left(\mathbb{E}_{s\sim\mathcal{D},a\sim v(a|s)}\{Q(s,a)\}-\mathbb{E}_{s,a\sim\mathcal{D}}\{Q(s,a)\}\right).$$

这里第一项就是标准的TD Error，关键在于第二项。第二项试图最小化在一个指定分布$v(a|s)$下的Q值与在数据集$\mathcal{D}$下的Q值之间的差异。通过上述学习目标就可以得到一个低估的Q函数。上述做法被证明等价于向值函数中添加了一个“Q-function aware”的惩罚项，或是在玻尔兹曼策略下隐式的KL散度正则化。

CQL表现超过了很多Policy Constraint Offline RL算法，而且相较而言没有那么保守（因为没有显式地限制学到的策略向行为策略靠拢）。

11.6.4 基于不确定性的离线强化学习

这种方法的核心思想在于，**OOD样本的Q函数值会显著大于那些在数据集分布内的样本的Q函数值，因此可用作惩罚项**。因此，这类方法会首先学习一个用于评估不确定性的集合$\mathcal{P}_{\mathcal{D}}(Q^{\pi})$，那么策略优化目标就可以写成：

$$\pi'=\arg\max\mathbb{E}_{s\sim\mathcal{D},a\sim\pi(a|s)}\{\mathbb{E}_{Q^{\pi}\sim\mathcal{P}_{\mathcal{D}}}\{Q^{\pi}(s,a)\}-\alpha\mathrm{Unc}(\mathcal{P}_{\mathcal{D}})\},$$

这里的$\mathrm{Unc}(\cdot)$是一个不确定性度量函数。最简单的不确定度量就是已经在model-based离线强化学习中提到的一系列Q函数的不一致性。

与Policy Constraint Offline RL相比，基于不确定性的离线强化学习算法拥有更严格的次优界限（tighter suboptimality bound），并且展示出了更好的表现。但是其核心问题仍在于对神经网络等非线性近似器缺乏容易且有理论保证的不确定性度量方法。上述基于一系列Q函数的不一致性来度量不确定性的方法在这个Q函数集合中各个Q函数差异很小时表现很差，且训练和维护这些Q函数集合的成本也很高。

11.6.5 样本内学习离线强化学习

前述的绝大多数离线强化学习算法都是在知行互动（Actor-Critic）框架下进行的，需要同时训练值函数和策略函数，它们的更新是一个相互耦合的过程（PEV和PIM）。然而，在使用策略函数构建值函数的Target的时候，如果取到了OOD的动作，就会造成分布漂移带来的过估计问题。样本内学习离线强化学习（In-Sample Reinforcement Learning）采用了完全不同的思路。这类算法不使用策略函数，而是直接从数据中学习值函数或者简单地使用行为策略的值函数$Q^{b}(s,a)$来替代策略函数。

比较有代表性的样本内学习离线强化学习算法是IQL（Implicit Q-Learning）。首先，IQL先学习一个状态值函数$V_{\psi}(\cdot)$：

$$L_V(\psi)=\mathbb{E}_{s,a\sim\mathcal{D}}\left\{L_2^{\tau}\left(Q_{\widehat{w}}(s,a)-V_{\psi}(s)\right)^2\right\}$$

这里的$L_2^{\tau}(\cdot)$是一种修改后的特殊的L2损失函数，具体形式如下：

$$L_2^{\tau}(x)=\begin{cases}|1-\tau|x^2 & \text{if } x<0\\ \tau x^2 & \text{if } x\geq 0\end{cases}$$

这里的τ是一个超参数，控制了对负值和正值的惩罚程度。之后，利用状态值函数$V_{\psi}(\cdot)$来学习Q函数：

$$L_Q(w)=\mathbb{E}_{s,a,s'\sim\mathcal{D}}\left\{\left(r(s,a)+\gamma V_{\psi}(s')-Q_w(s,a)\right)^2\right\}$$

这里利用下一个状态的状态值函数$V_{\psi}(s')$来构建TD目标，这样就避免了在计算TD目标时使用策略函数。最后，可通过下面的Advantage Weighted Regression（AWR）来学习策略函数：

$$L_{\pi}(\phi)=\mathbb{E}_{s,a\sim\mathcal{D}}\left\{\exp\left(\beta\left(Q_{\widehat{W}}(s,a)-V_{\psi}(s)\right)\right)\log\pi_{\phi}(a|s)\right\}$$

IQL算法易于实施，表现也很好，并能避免OOD问题，因此训练过程也更稳定。

11.6.6 Goal-conditioned模仿学习

最后一种方法被称为Goal-conditioned模仿学习（Goal-conditioned Imitation Learning，GCIL）。这类方法在RvS框架下进行，采用了条件下的模仿学习。GCIL学习一个以结果（可以使目标状态、奖励或目标回报等）为条件的策略：

$$L(\pi) = -\sum_{\tau\sim\mathcal{D}}\sum_{1\le t\le H}\log\pi(a_t|s_t, o(\tau)),$$

这里的τ是轨迹,$o(\tau)$是轨迹τ的结果。通过上式学好策略之后，就可以在推理时设定期望的结果之后，和当前状态一起输入到策略函数中，得到当下的动作。

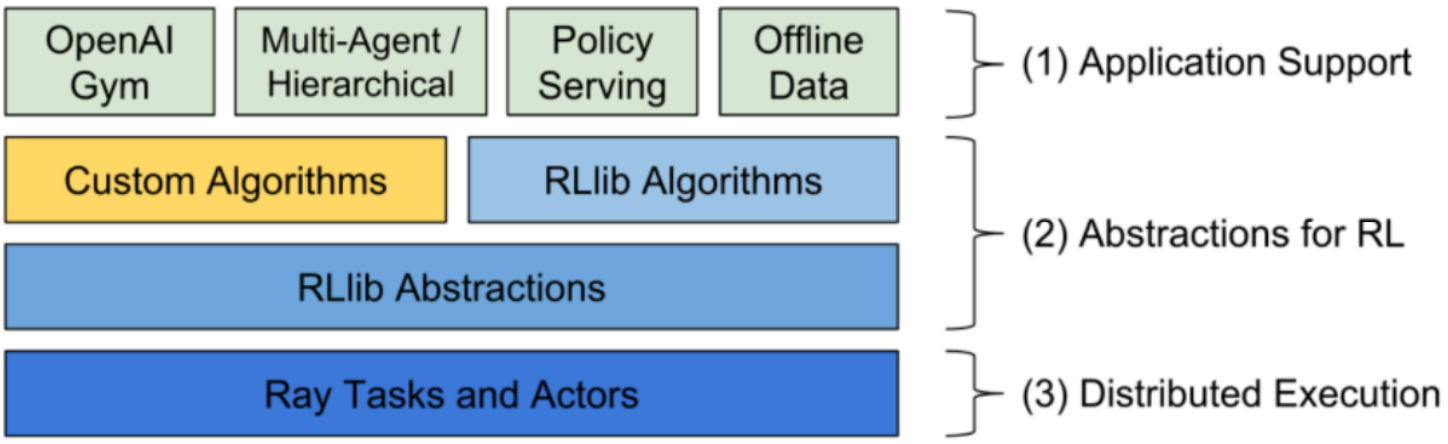
GCIL方法本质上是将强化学习问题转化为监督学习来处理，因此没有强化学习中的稳定性困扰。但是与强化学习基于的动态规划相比，GCIL为了达到最优策略需要一系列更严格的假设。而且GCIL也不是一种解决离线强化学习的通用方法，但是在具有高质量数据的确定性MDP中表现还是很好的，可以用一用。

11.7 主要的RL框架/库

目前，已经有很多RL框架/库可供使用。使用这些库可以抽象强化学习问题和算法中的共享组件，简化开发过程，帮助研究人员专注于算法创新，并提高代码的可重用性和可迁移性。下面列出了一些主要的RL框架/库。总结表格如下：

框架/库	开发者	支持的深度学习框架	特点
RLlib	UC Berkeley RISE Lab、Anyscale Inc.	TensorFlow、PyTorch	基于Ray分布式计算框架，支持分布式训练，简单统一的API，支持多种强化学习算法
OpenAI Baselines	OpenAI	TensorFlow	最早的强化学习库之一，缺乏文档，混合的代码风格，一些早期算法在训练时可能不稳定
GOPS	清华大学iDLab	PyTorch	高度模块化的设计，支持多种RL/ADP算法和训练方式，支持多种环境
Tianshou	清华大学	PyTorch	模块化设计，支持多种强化学习算法，代码量少，支持单元测试，但只支持平行采样

11.7.1 RLlib



- **开发者**：UC Berkeley RISE Lab、Anyscale Inc.
- **支持的深度学习框架**：TensorFlow、PyTorch
- **特点**：
  - **优点**：
    - 基于Ray分布式计算框架，支持分布式训练
    - 简单统一的API
    - 支持多种强化学习算法

        - 可定制的环境、预处理器、网络模型、采样过程及强化学习算法
        - 包含主流的强化学习算法实现
    - **缺点**：
        - 高度模块化和可复用的代码结构使学习曲线陡峭
        - 修改或定制时需要深入研究 RLlib 的源代码去寻找相应位置

11.7.2 OpenAI Baselines

- **开发者**：OpenAI
- **支持的深度学习框架**：TensorFlow
- **特点**：
    - **优点**：
        - 最早的强化学习库之一
    - **缺点**：
        - 缺乏文档
        - 混合的代码风格
        - 一些早期算法在训练时可能不稳定
- **继任者**：Stabe-Baselines（SB）及 Stable-Baselines3提升了训练稳定性、代码可读性和简易性。迁移至PyTorch后结构进行了简化。但是仍然不支持多GPU训练，因此与分布式框架相比训练速度较慢。

11.7.3 GOPS

- **开发者**：清华大学iDLab
- **支持的深度学习框架**：PyTorch
- **开发目的**：使用RL算法求解通用的最优控制问题
- **特点**：
    - **优点**：
        - 高度模块化的设计，如环境、算法、训练器、Buffer、采样器等
        - 不同模块可通过超参数灵活配置
        - 不仅支持OpenAI Gym环境，还支持多种第三方环境，如MATLAB/Simulink、CarSim、and SUMO
        - 支持多种RL/ADP算法，如infinite-horizon ADP、finite-horizon ADP、DQN、DDPG、TD3、SAC、TRPO、PPO、DSAC
        - 支持多种serial和parallel训练方式，包含 on-policy serial trainers、off-policy serial trainers、synchronous parallel trainers、asynchronous parallel trainers
        - 支持添加硬约束、对抗动作（adversarial actions）等

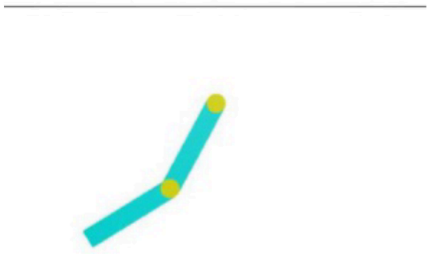
11.7.4 Tianshou

- **开发者**：清华大学
- **支持的深度学习框架**：PyTorch
- **特点**：
    - **优点**：
        - 模块化设计，可定制环境、算法、训练pipeline等
        - 代码量少
        - 支持多种强化学习算法
        - 支持单元测试（unit test）
    - **缺点**：
        - 只支持平行采样，但不支持平行训练
        - 大规模复杂问题的训练速度较慢

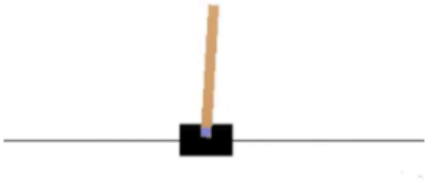
11.8 主要的RL仿真平台

强化学习仿真平台提供虚拟环境来代替现实世界进行实验和测试，这有助于降低实验成本和风险，也有助于快速验证和开发算法。下面列出了一些主要的RL仿真平台。

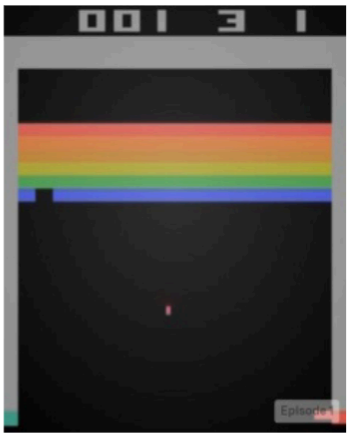
11.8.1 OpenAI Gym



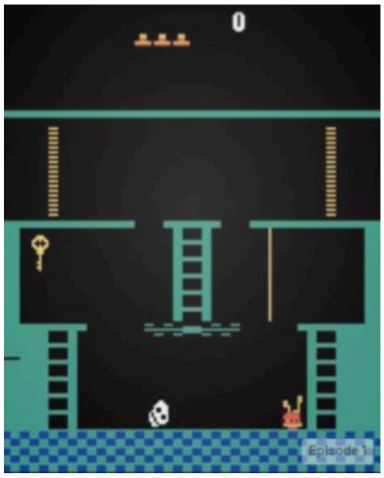
(a) Acrobot
(Classic control)



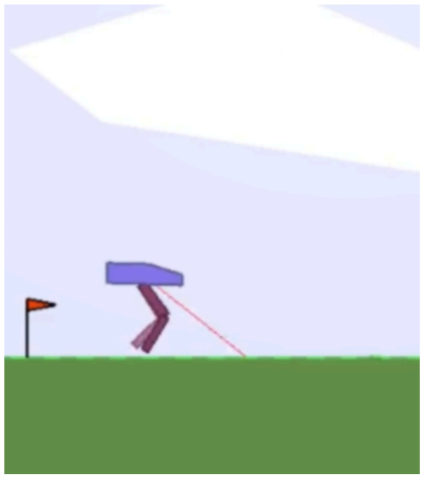
(b) CartPole
(Classic control)



(c) Breakout
(Atari)

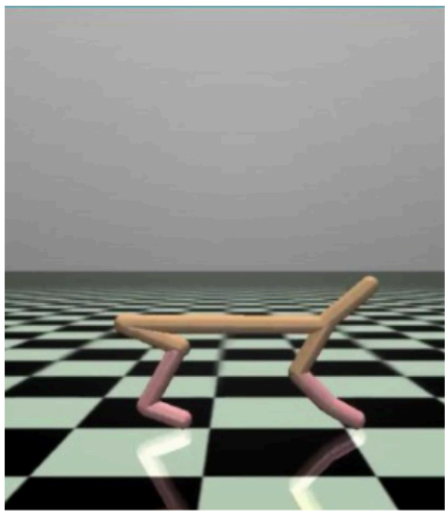


(d) Montezuma's Revenge
(Atari)



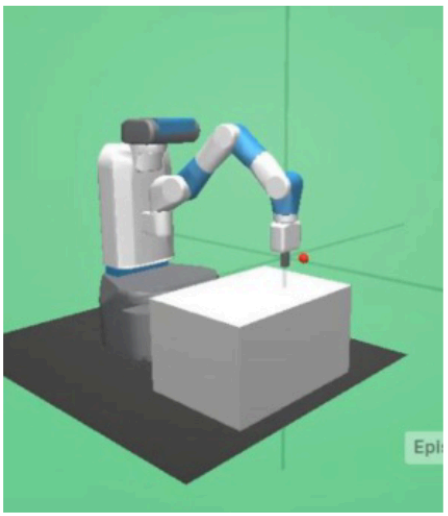
(e) BipedalWalker
(Box2D)

(f) CarRacing
(Box2D)



(g) HalfCheetah (MuJoCo)

(h) Humanoid (MuJoCo)



(i) FatchReach (MuJoCo)

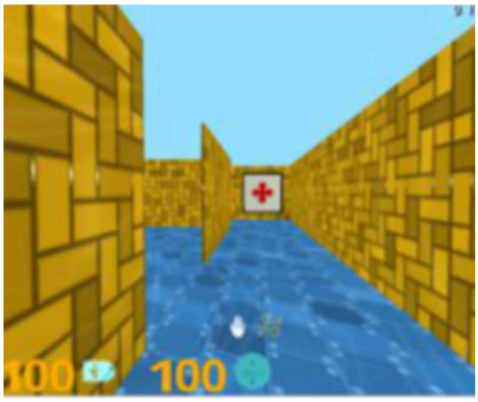
- **开发者**：OpenAI
- **简介**：OpenAI Gym是一个开源的强化学习平台，提供了多种标准化的环境和接口，方便研究人员和开发者进行实验和测试。它支持多种类型的环境，如经典控制（小规模问题）、棋类游戏、Atari游戏、Doom游戏、Box2D、MuJoCo等。智能体育环境交互以episode为单位，episode起始点从预设的分布中随机初始化得到，当环境到达终止状态或达到最大步数时episode结束。
- **兼容性**：
  - **支持的深度学习框架**：TensorFlow、PyTorch
- **特点**：
  - **优点**：
    - 统一的API交互接口
    - 环境对于每个版本进行编号，有助于结果复现和比较

11.8.2 TORCS

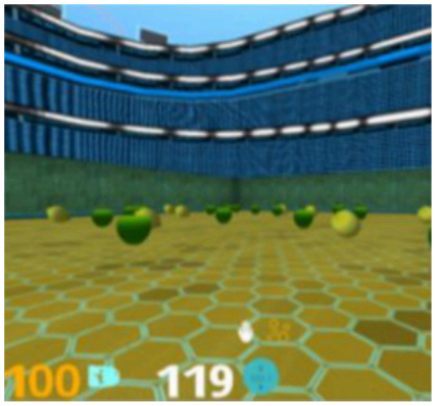
- **开发者**：开源项目，项目地址：TORCS
- **简介**：TORCS（The Open Racing Car Simulator）是一个开源的赛车模拟器，允许玩家与虚拟赛车进行对抗。也允许多个玩家在线对战或使用程序/AI控制赛车。基于OPenGL开发，效果逼真。
- **兼容性**：
  - **语言**：C++
  - **系统**：Linux,、FreeBSD、OpenSolaris、Windows

- **特点**：
  - **优点**：
    - 真实的赛车模拟环境、多种模式供选择（单人、多人、AI对战等）
    - 车辆动力学使用复杂模型建模，包含损伤模型、碰撞检测模型、轮胎和车轮模型、空气动力学模型等
    - 开源、模块化、可扩展

11.8.3 DeepMind Lab



(a) nav_maze

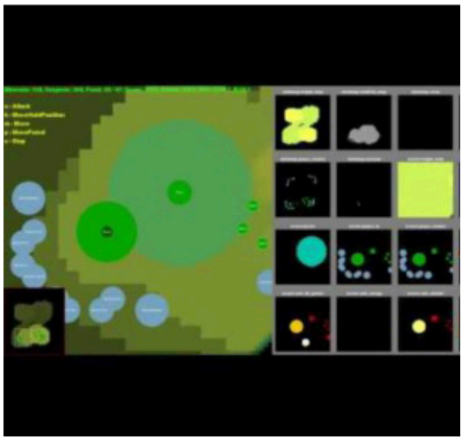


(b) seekavoid_arena

(c) stairway_to_melon

- **开发者**：谷歌DeepMind
- **简介**：谷歌的深度思维实验室（DeepMind Lab）于2016年发布，作为一个基于智能体的人工智能研究平台，它在第一人称3D游戏引擎雷神之锤III竞技场的基础上构建。根据用户提供的帧率可以进行lock-stepped的交互。当用户给出下一个动作前会暂停在当前观测。可采取的动作包括四处观察以及移动，并支持同时提供若干个动作。
- **兼容性**：
  - **语言**：C语言API、Python binding以及扩展级别的Lua API
- **特点**：
  - **优点**：
    - 每步提供奖励信号、pixel观测（第一人称视角）、速度信息
    - 可通过DeepMind Lab的API定制环境变体机基本的多智能体交互

11.8.4 StarCraft II



(a) Map visualization

(b) Typical operation

(c) Agent demonstration

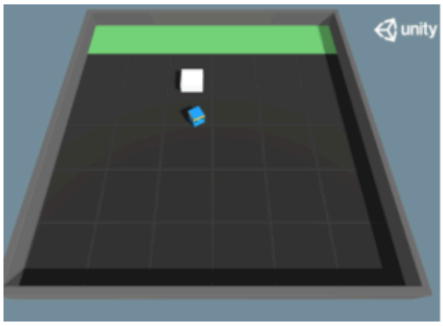
- **简介**：基于星际争霸II的平台。相比之前的平台，其对算法提出了更高的要求。具有多智能体、不完美信息、大规模状态空间和动作空间及状态需要从原始输入中提取的特点
- **兼容性**：
  - **语言**：Python接口

    - **平台**：Windows、Linux、macOS
- **特点**：
    - **优点**：
        - 除了主地图外还包含若干个小地图，小地图涵盖星际争霸 II的不同方面
        - 主地图包含人类专家的重放数据，并在此基础上训练了一个基线模型

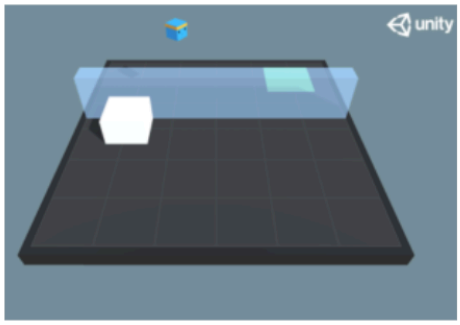
11.8.5 微软的Project Malmo

- **开发者**：微软
- **简介**：微软的Project Malmo是一个基于Minecraft的人工智能研究平台。它提供了一个结构化、动态的虚拟世界，具有高度灵活性和复杂性。支持导航、建造、生存等任务。
- **特点**：
    - **优点**：
        - 高度复杂灵活，支持多种任务和环境
        - 支持AI-AI和AI-人类交互
        - 提供不同层级抽象的观测和动作

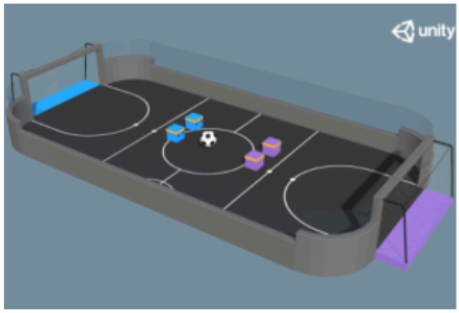
11.8.6 Unity ML-Agents



(a) Push block



(b) Wall jump



(c) Soccer twos

- **简介**：包含一个游戏引擎和叫做Unity Editor的交互界面。Unity Editor提供多种内置组件，如相机、mesh、刚体等。支持2D、3D、AR/VR。平台提供了一组预先构建的室内场景以及Python API，可以使用第一人称智能体与这些环境进行交互。
- **兼容性**：
    - **语言**：Python API、C#
    - **平台**：Windows、Linux、macOS
- **特点**：
    - **优点**：
        - 包含一系列基线算法
        - 可定制多种环境和任务，如简单的网格世界问题、复杂的策略游戏、基于物理的谜题和多智能体竞技游戏。支持具备传感器和物理复杂性的环境
        - 支持动态的多智能体交互