

使用 MATLAB 进行强化学习

了解训练和部署

value-based

actor
critic

policy-based

强化学习工作流程概述

本系列电子书涉及强化学习的五个方面。首先讲解概念，然后介绍在 MATLAB® 和 Simulink® 中的具体操作方法。

第一本电子书重点介绍[建立环境](#)。第二本电子书探讨[奖励和策略结构](#)。本电子书介绍[训练和部署](#)。

1

您需要一个环境，供您的智能体开展学习。您需要选择环境里应该有什么，是仿真还是物理设置。

environment



2

您需要考虑最终想要智能体完成什么任务，并设计奖励函数，激励智能体实现目标。

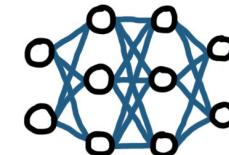
reward



3

您需要选择一种表示策略的方法。思考您想如何构造参数和逻辑，由此构成智能体的决策部分。

policy



4

您需要选择一种算法来训练智能体，争取找到最优的策略参数。

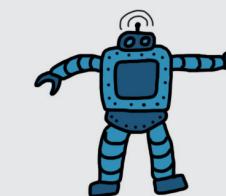
training



5

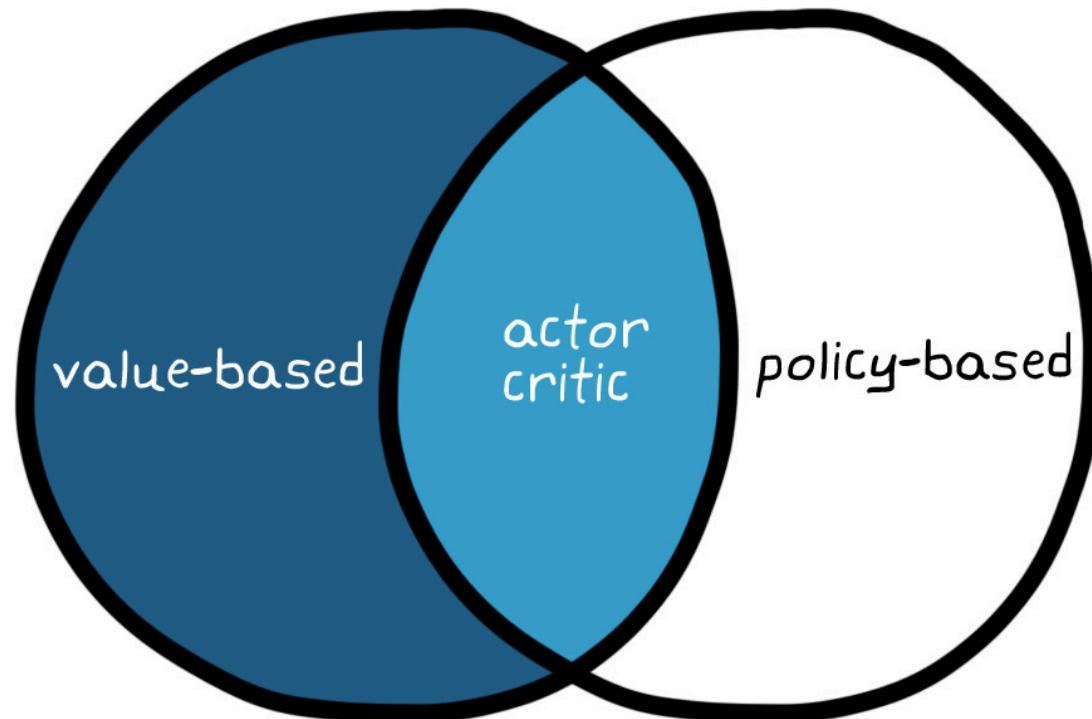
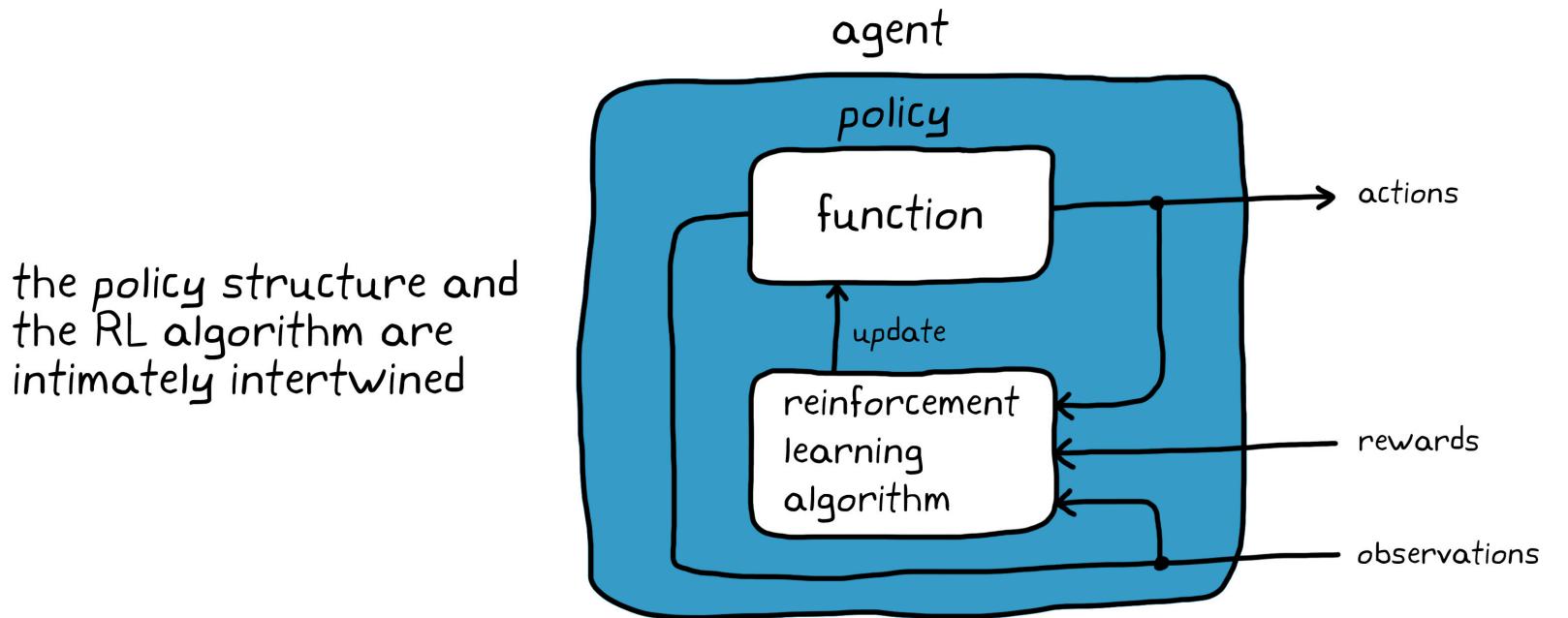
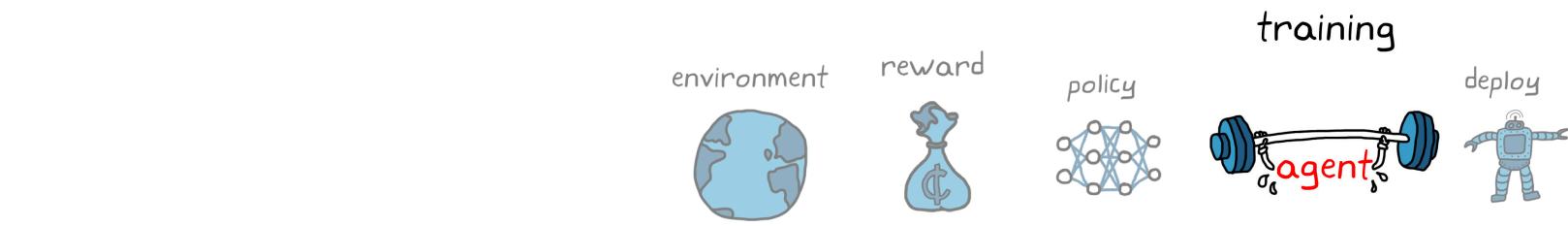
最后，您需要在实地部署该策略并验证结果，从而利用该策略。

deploy



如何构建策略

在强化学习 (RL) 算法中, 神经网络表示智能体策略。策略结构与强化学习算法密切相关;若未选择 RL 算法, 则无法构建策略。

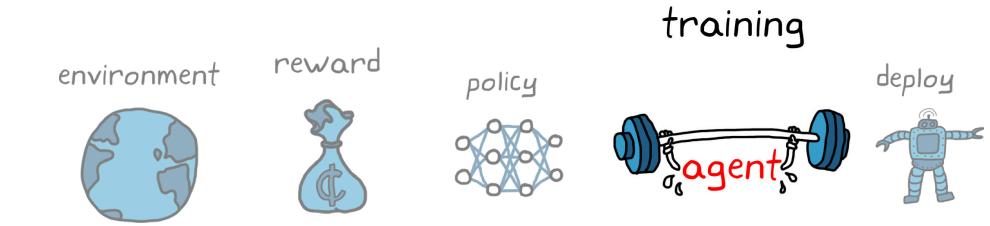


the policy structure and the RL algorithm are intimately intertwined

接下来的几页将介绍基于策略函数、基于价值函数及执行器-评价器强化学习方法, 以重点说明策略结构区别。当然, 这里只是简要概括说明;但是, 您如果想要对策略构建方法有个基本了解, 这些内容应该可以带您入门。

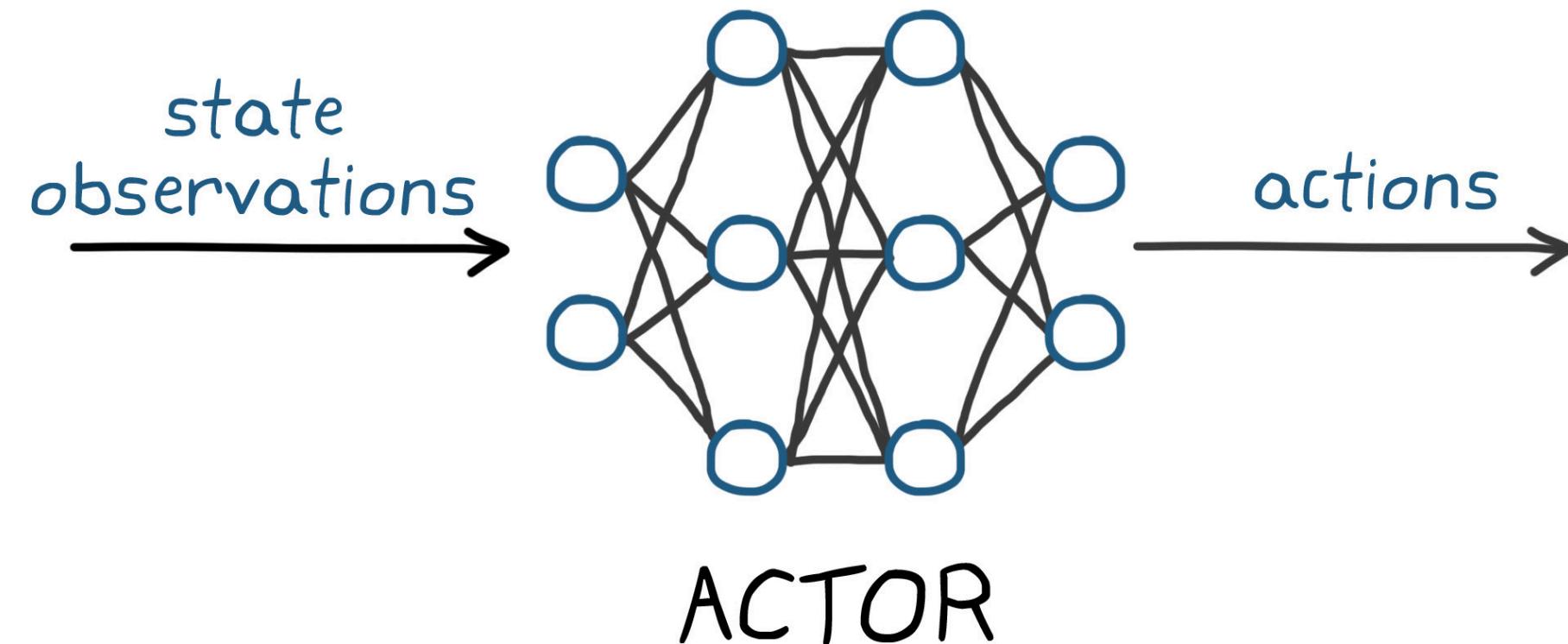
基于策略函数的学习

基于策略函数的学习算法,以状态观测量为输入,以动作为输出,来训练神经网络。这个神经网络就是完整的策略,因此称为基于策略函数的算法。神经网络称为执行器,因为它直接指挥智能体采取动作。

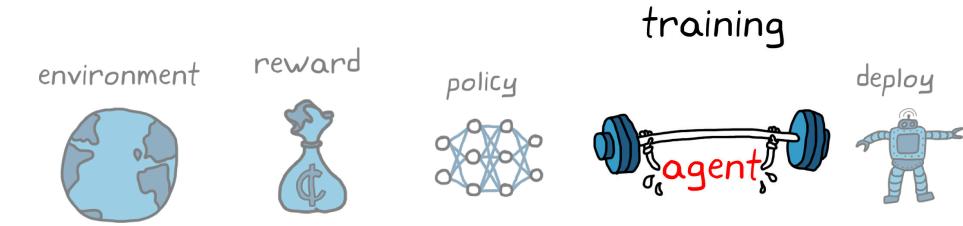


现在的问题在于,如何训练这个神经网络?为了大致地了解这一点,我们来看一款雅达利游戏:打砖块 (Breakout)。

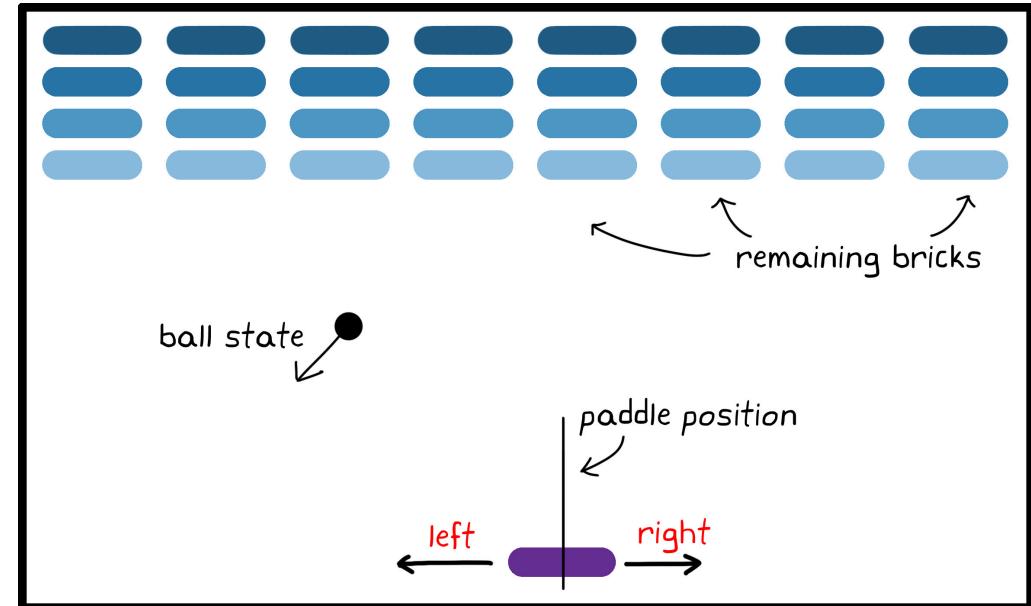
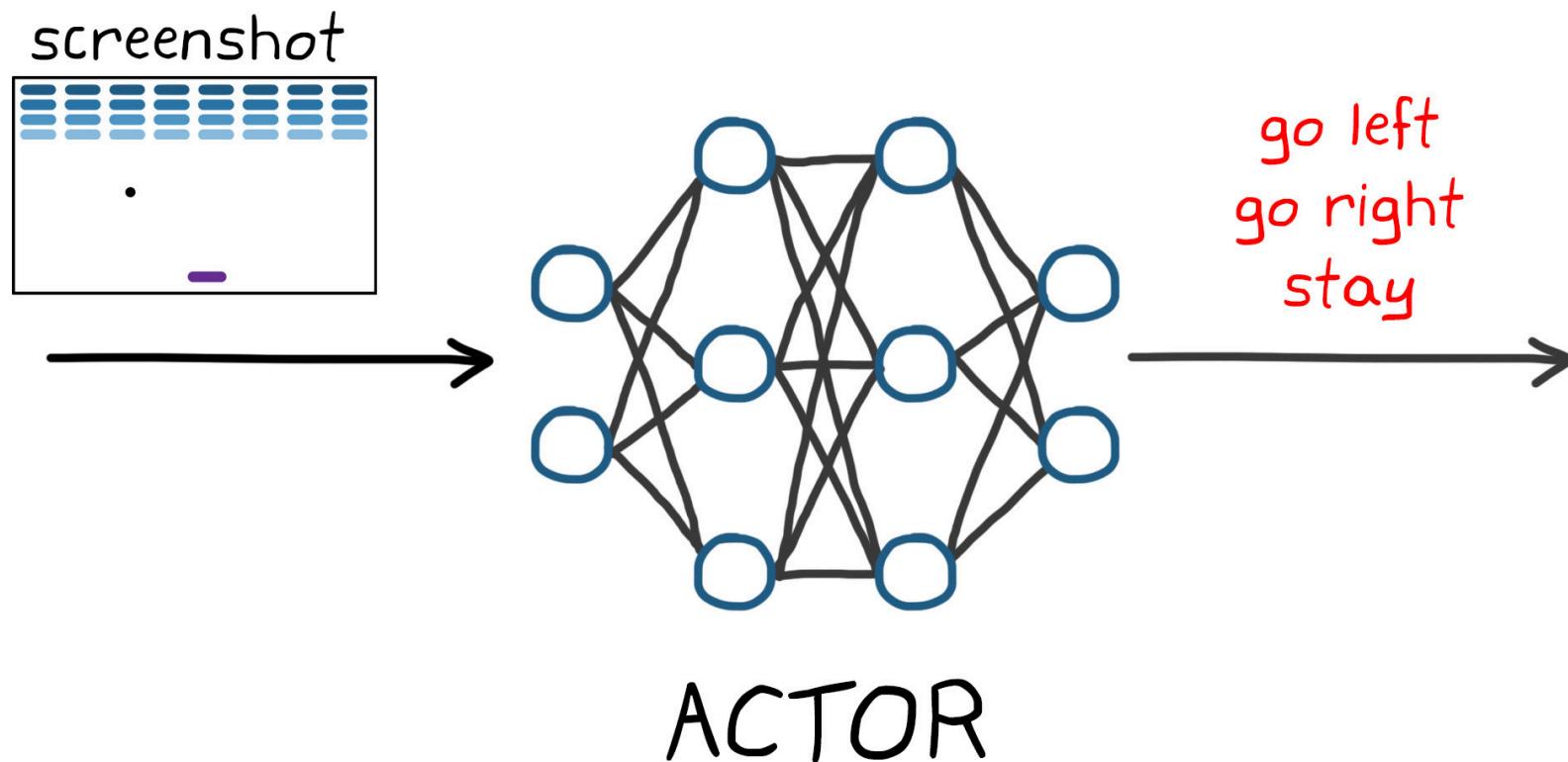
policy function-based learning



学习打砖块的策略方法



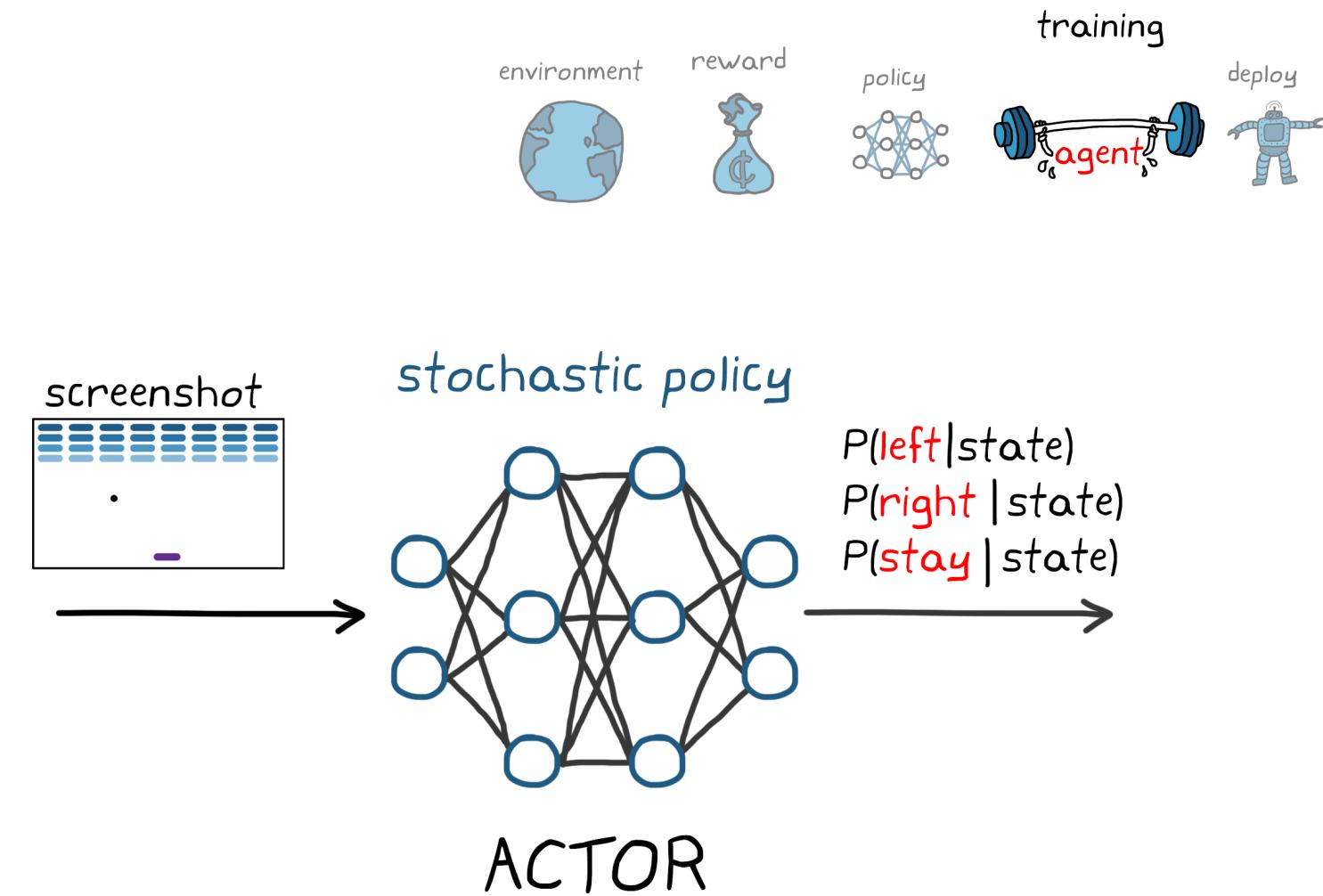
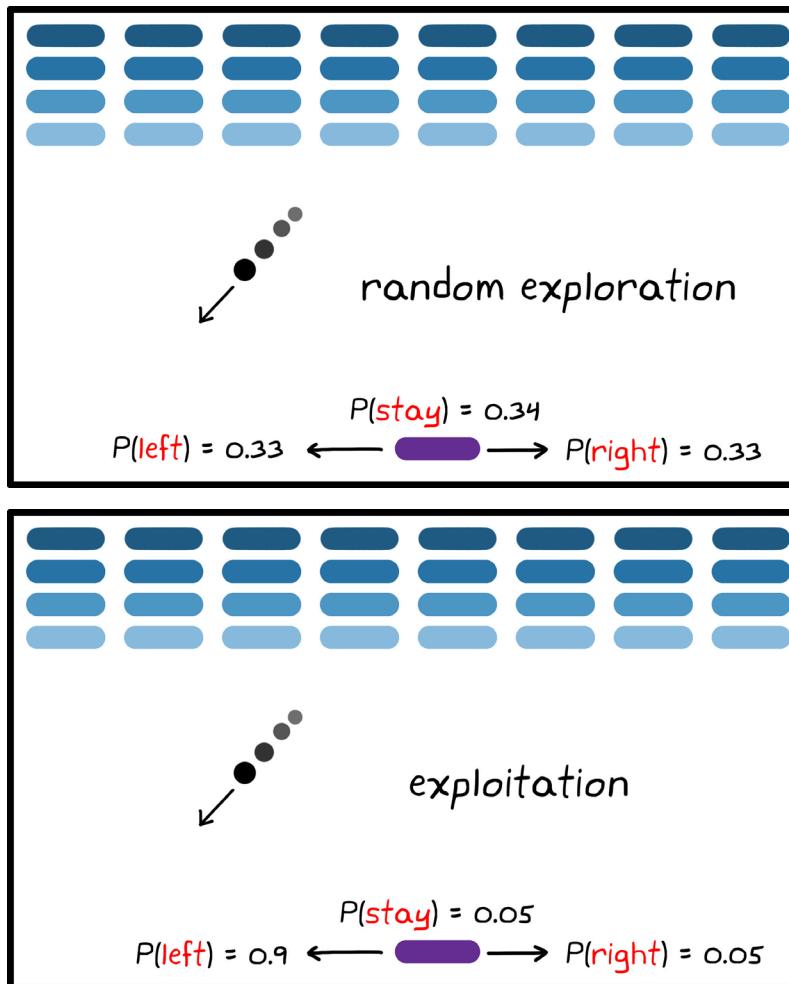
在打砖块这个游戏中，您需要使用球拍击打弹力球尽量消除砖块。这款游戏共包含三个动作，左移球拍、右移或保持不动，同时还有一个近乎连续的状态空间，其中包括球拍位置、弹力球位置和速度及剩余砖块位置。



在此示例中，执行器网络的输入是球拍、球和砖块状态。输出是指代表动作的节点：左移、右移和保持不动。您可以输入游戏屏幕快照，让网络学习图像中的哪些特征是决定输出的最关键因素，而不是手动计算状态，再将状态馈送至网络。执行器将成千上万个像素点强度映射到三个输出。

随机策略

设置好网络后，就可以着手研究训练方法了。策略梯度法是一个大类，本身包含大量变体。策略梯度法可以与随机策略一起使用，因此该策略不会生成确定性的“左移”指令，而是输出左移概率。概率与三个输出节点的值直接相关。

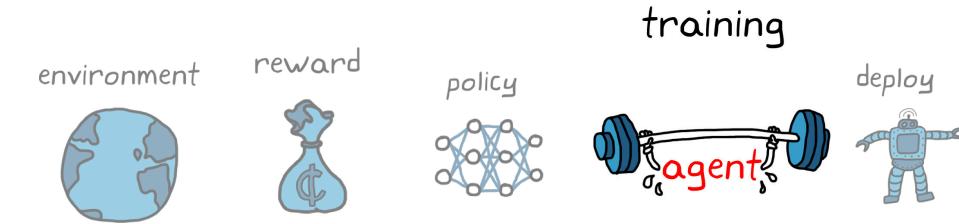


智能体是否应该利用环境，选择已知可以得到最多奖励的那些动作？还是应该选择探索环境中仍然未知部分的动作？

随机策略将通过探索概率来权衡动作利弊。现在，智能体在学习期间只需更新概率。左移比右移更好吗？如果确实如此，则提高此状态下的左移概率。

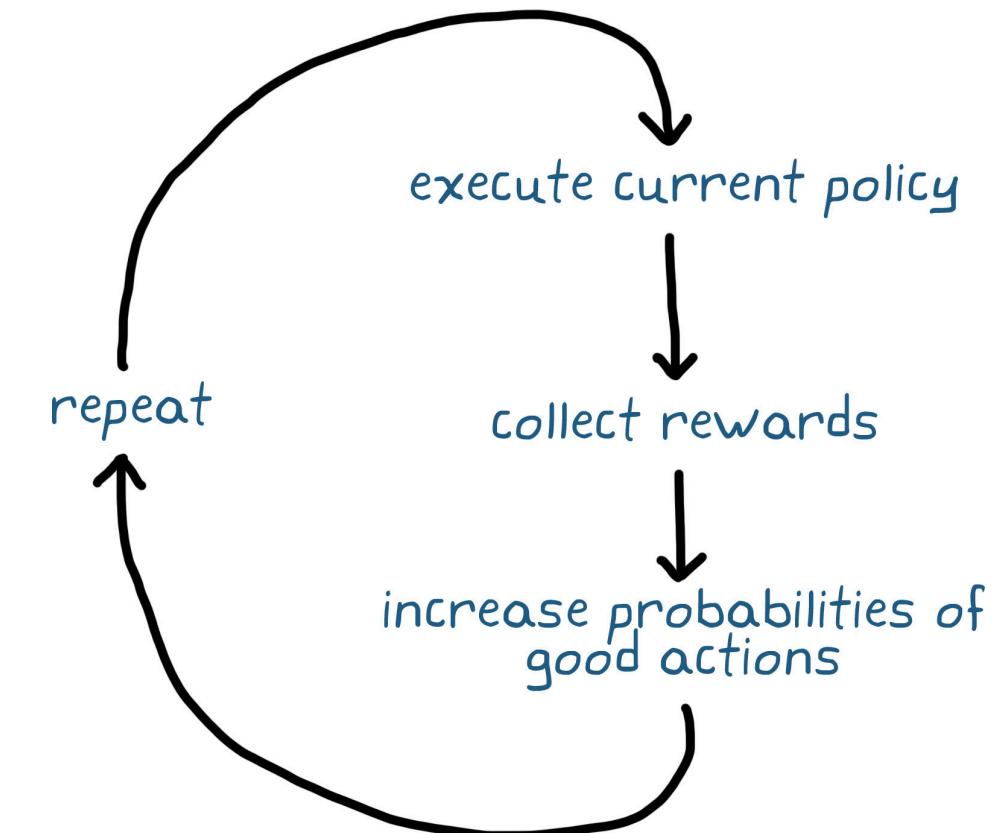
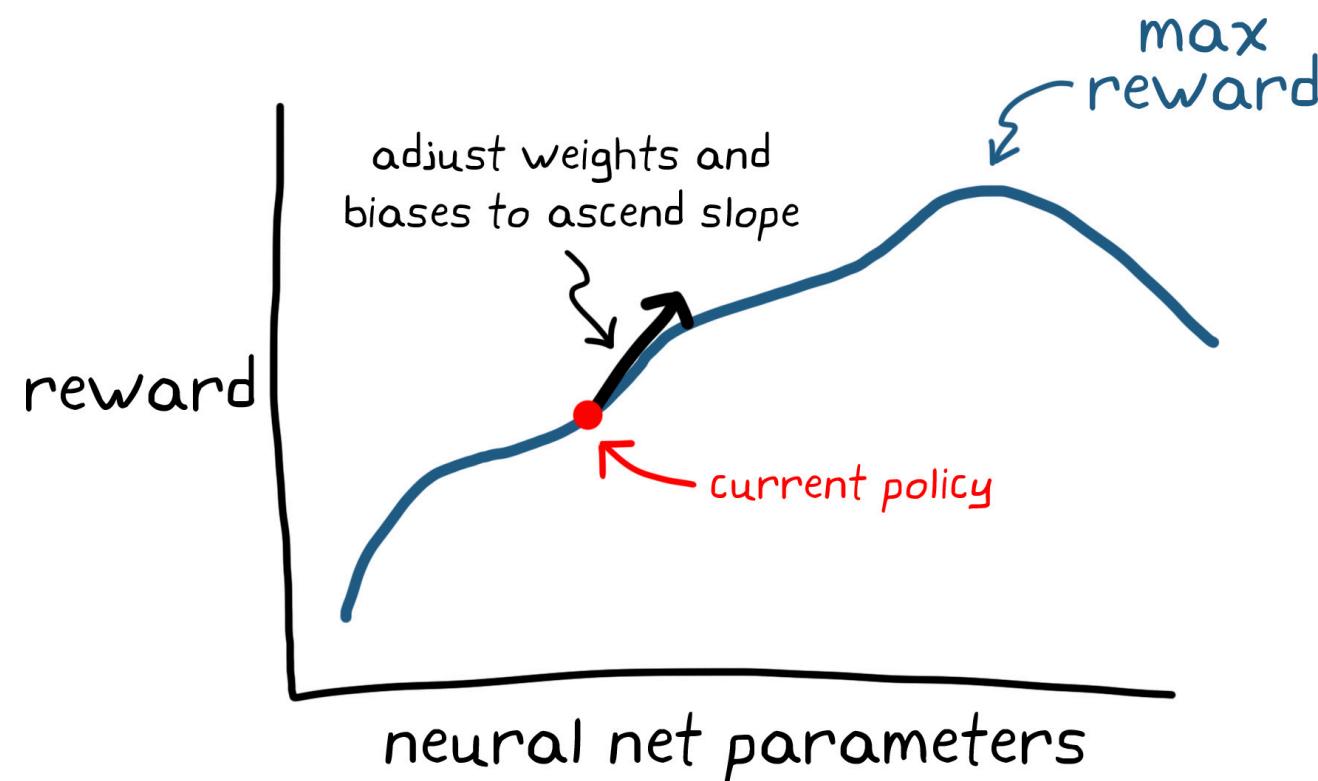
久而久之，智能体将沿回报最高的方向微调三个动作的概率。最终，每个状态对应的最有利动作的概率将最高，从而智能体将始终采取该动作。

策略梯度法



智能体如何确定动作是好是坏?这里的构想是:执行当前策略,沿途收取奖励,然后更新网络,以提高以高奖励值为导向的动作的概率。

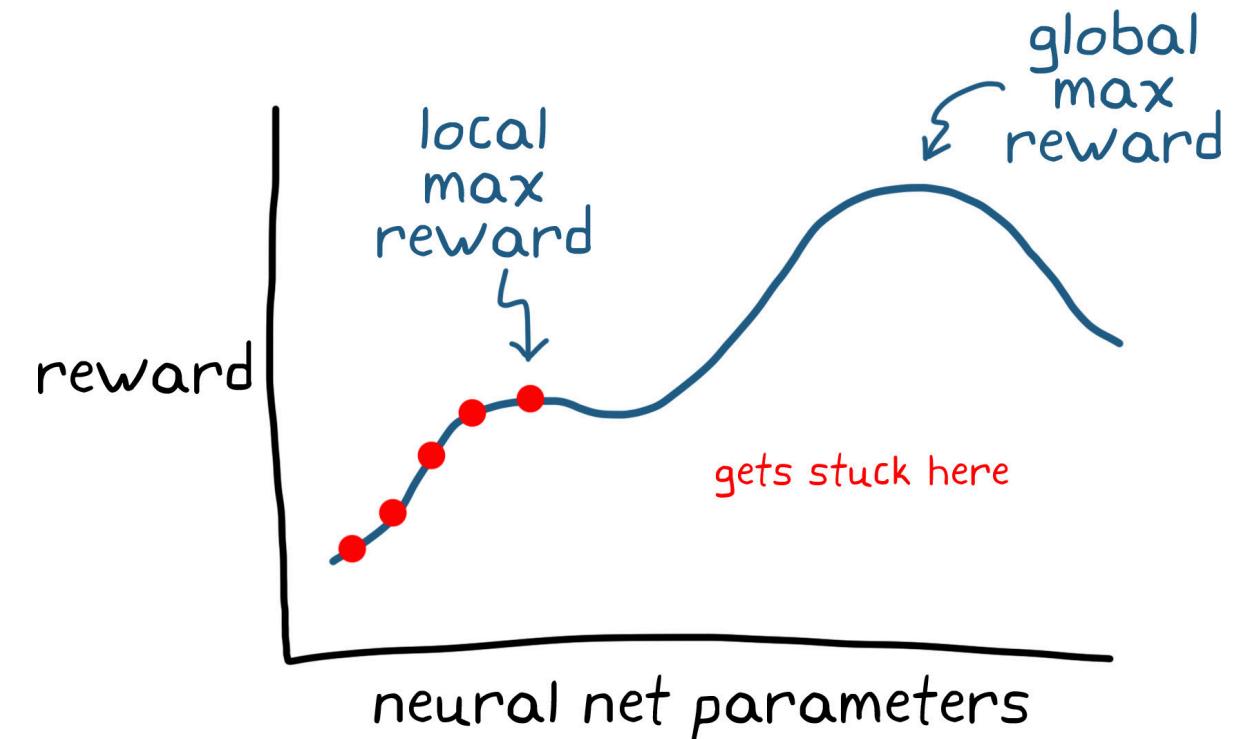
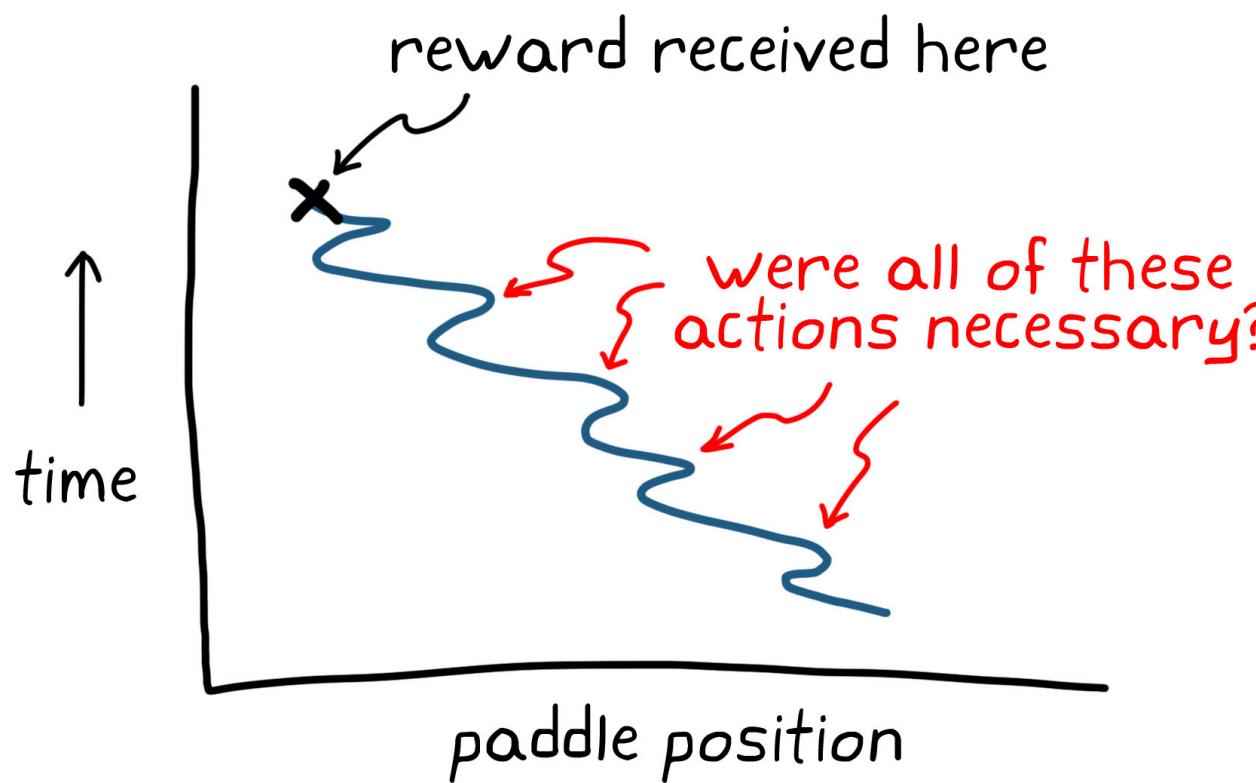
如果左移球拍未接住球并生成负奖励,则更改神经网络,提高智能体下次处于该状态时右移球拍的概率。



您可以计算网络权重和偏置关于奖励值的导数,然后调整权重和偏置,使奖励值正向提高。这样,学习算法将移动权重和偏置,沿着奖励函数斜坡攀升。这就是在名称中使用梯度这个词的由来。

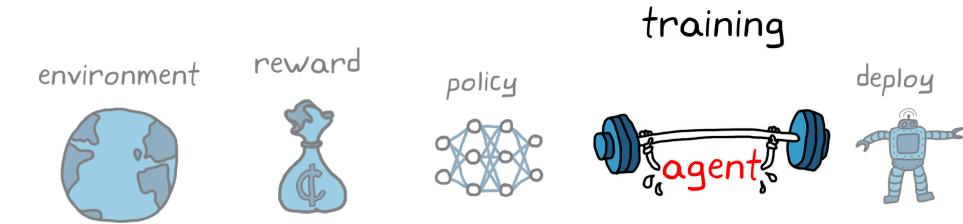
策略梯度法的缺点

策略梯度法的一个缺点在于，仅仅跟随最快上升方向这种初级方法可收敛到局部极大值，而不是全局极大值。另外，鉴于策略梯度法对测量噪声十分敏感，可能收敛速度较慢。例如，如果采取大量连续动作以得到奖励，并且累积奖励的各片段间方差较高时，就会发生这种情况。



例如，在打砖块游戏中，智能体可能会多次快速左右移动球拍，最终球拍穿越设定区域击球并得到奖励。智能体并不确定是否每一个动作都是得到奖励所必需的，因此策略梯度算法不得不将每一个动作均视为必要动作并相应调整概率。

基于价值函数的学习



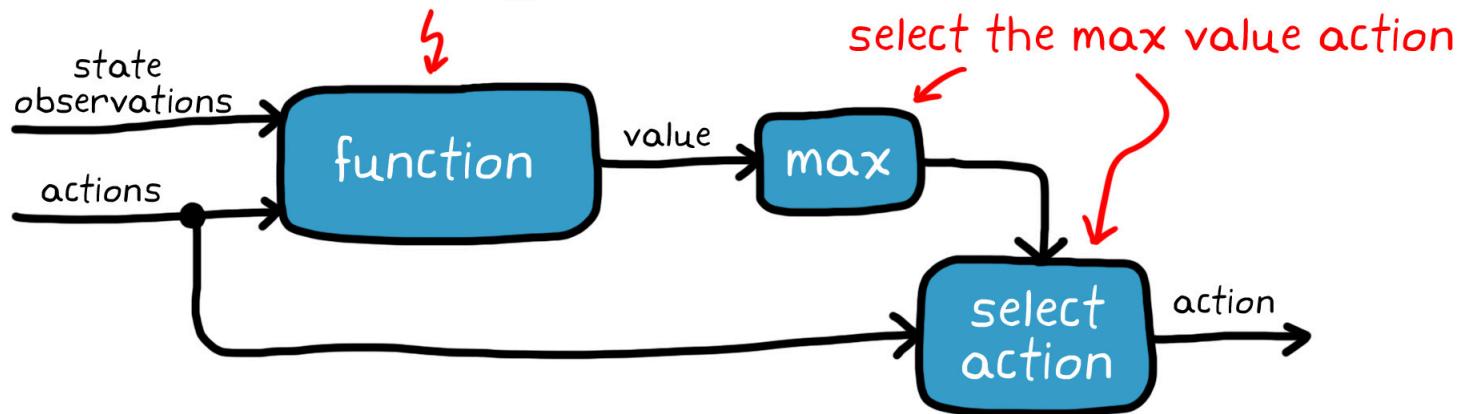
对于基于价值函数的智能体，函数的输入为状态和该状态下每一个可能动作，并输出采取该动作的价值。

what is the current state?

$value = function(state, observations, action)$

how good is the action from this state?

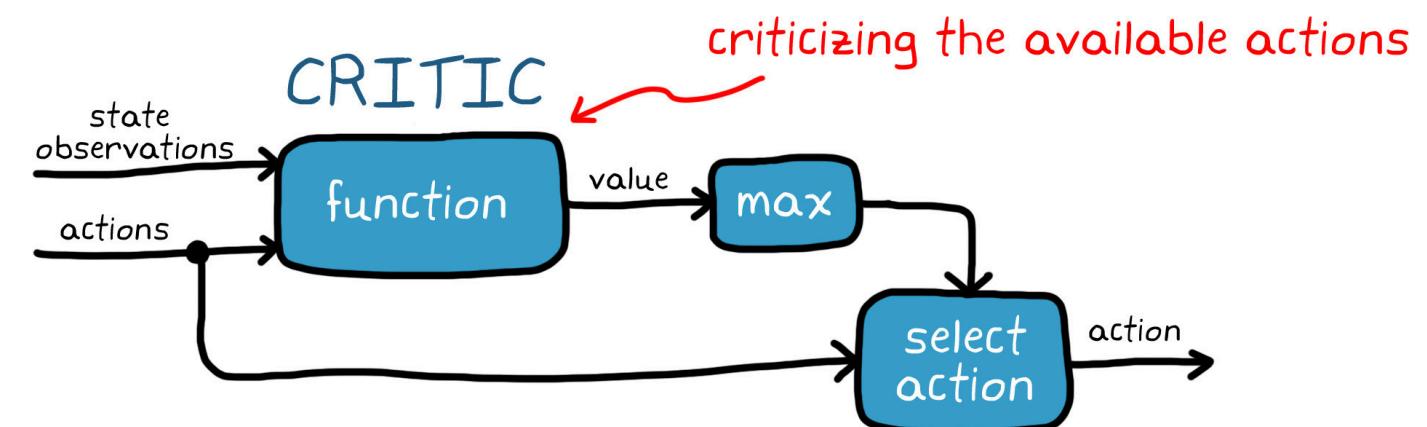
check the value of every action



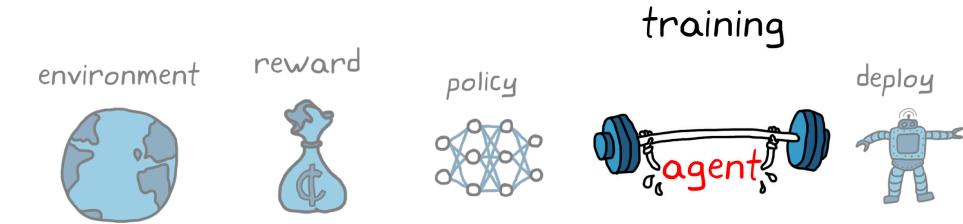
您可以将此函数视为评价器，因为它根据可能的动作并评价智能体的选择。

单纯凭借这个函数不足以表示策略，因为它输出的是价值，而策略需要输出动作。因此，策略是使用此函数检查给定状态下的每个可能动作的价值，然后选择具有最高价值的动作。

criticizing the available actions



价值函数与网格世界



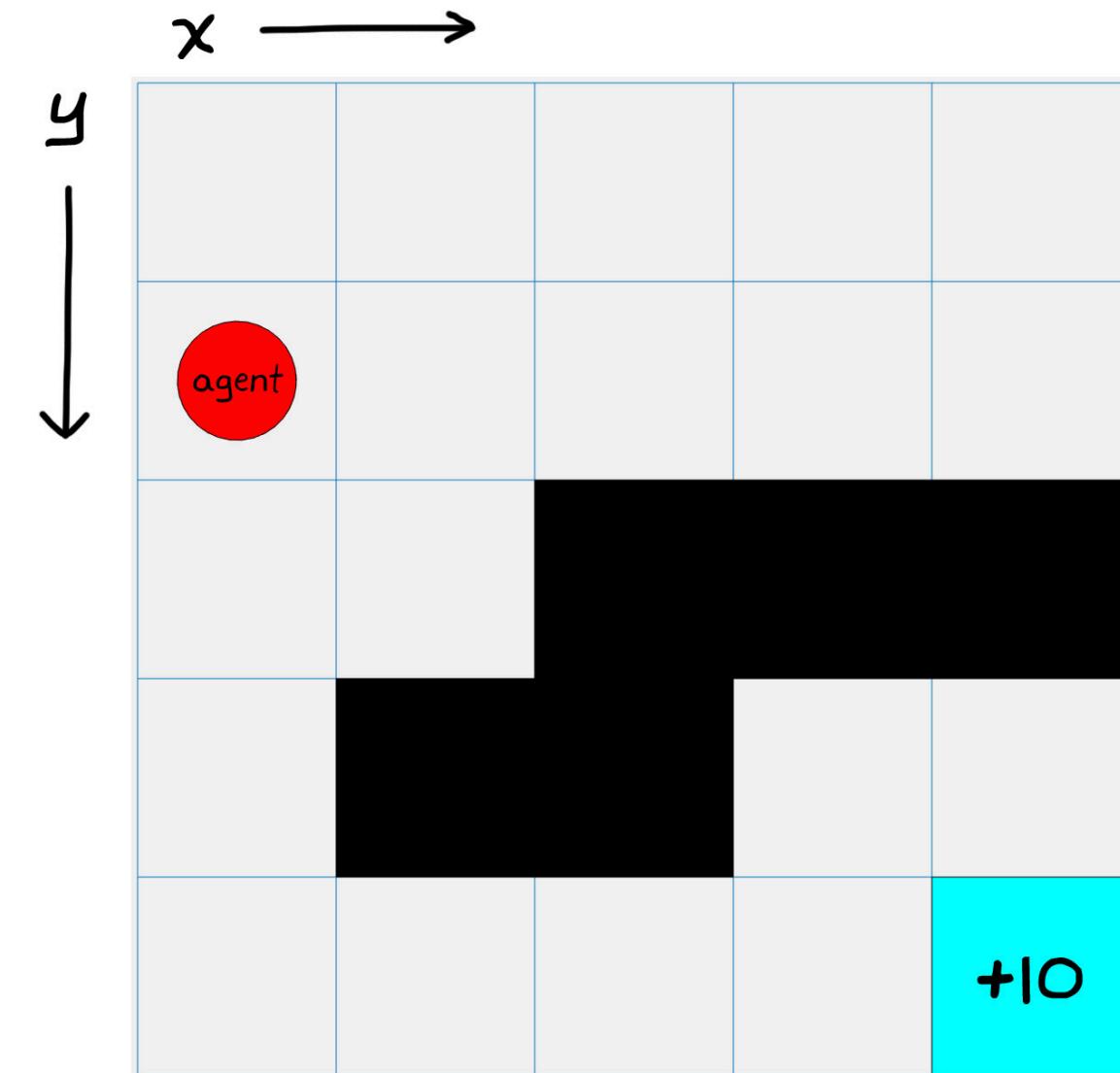
为了了解该函数是如何实际工作的,我们来看一个使用网格世界环境的示例。

在此环境中,共有两个离散状态变量:X 网格位置和 Y 网格位置。智能体一次只能向上、向下、向左或向右移动一个方格,而且每次采取一个动作都会产生 -1 奖励。

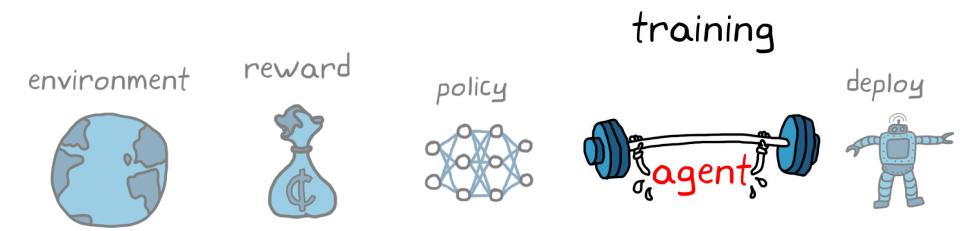
如果智能体试图脱离网格或进入黑色障碍区,则不会进入新状态,但仍会产生 -1 奖励。这样,智能体会因碰壁而受到惩罚,也不会取得任何实质性进展。

有一种状态可以产生 +10 奖励;构想是:为了收取最高奖励,智能体需要学会一种策略,使其以尽可能少的动作移动到奖励为 +10 的状态。

» [了解如何在 MATLAB 中求解网格世界环境](#)

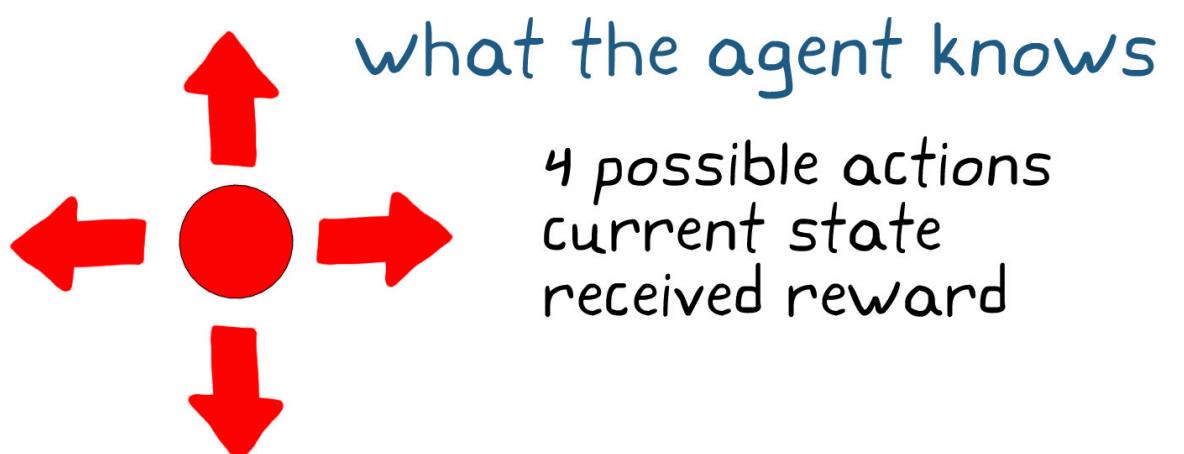
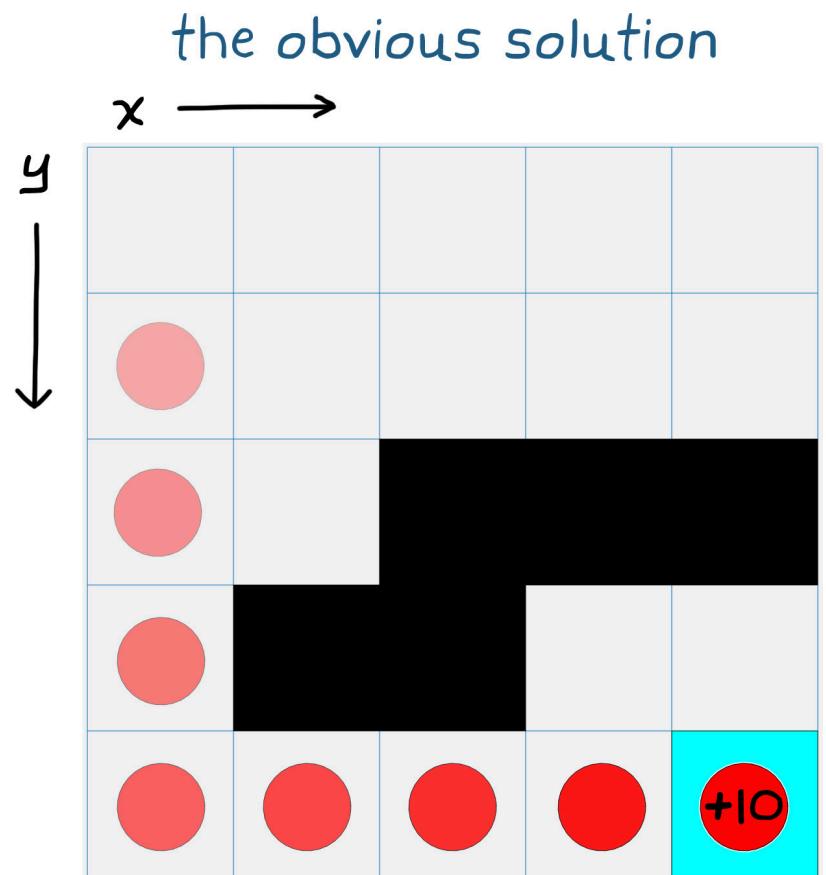


价值函数与网格世界 (续)

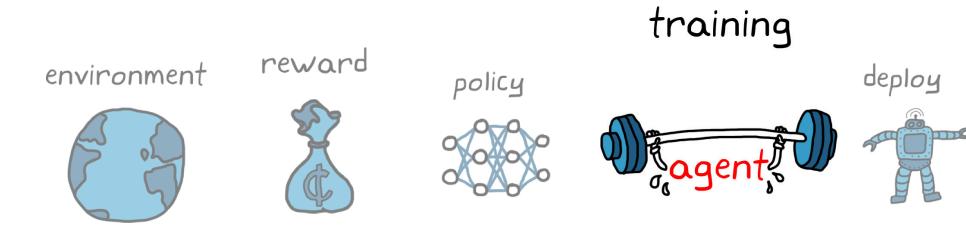


确定走哪一条路线才能得到奖励看起来可能很容易。

但必须谨记，在无模型强化学习中，智能体对所处的环境一无所知。它不知道目标是要到达 +10 奖励位置。只知道可以采取四个动作之一，而且在采取动作后会获得位置并从环境中得到奖励。

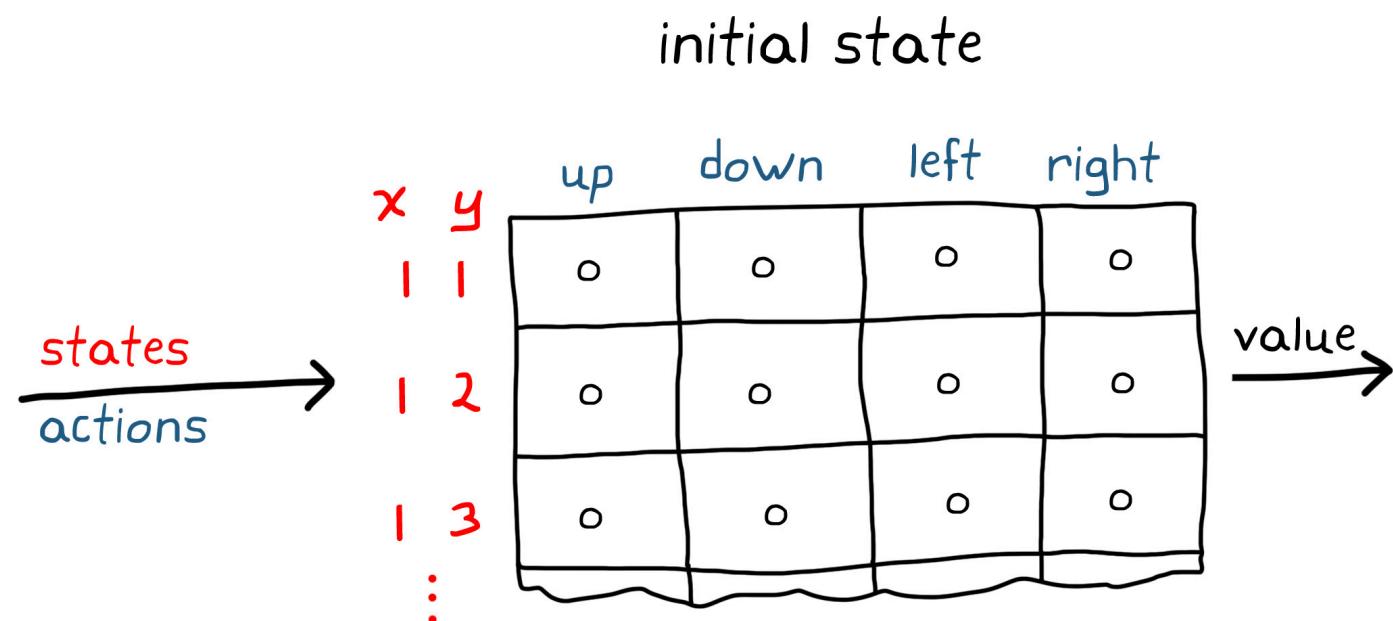
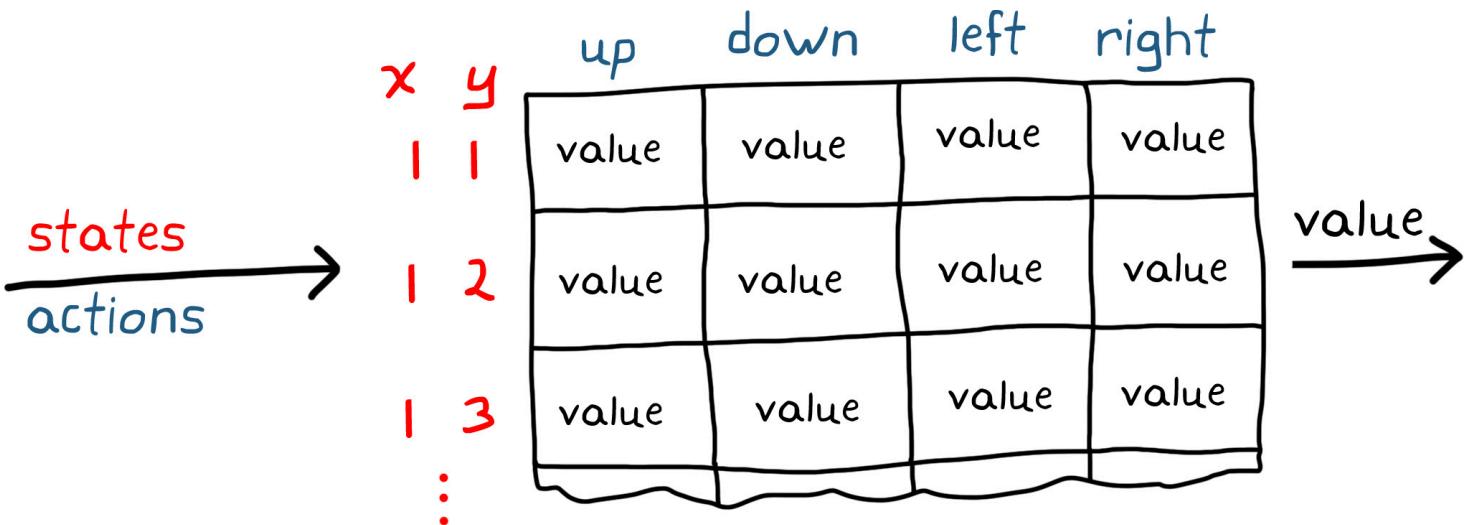


使用 Q-table 求解网格世界



智能体了解环境的方式是：采取动作，然后根据获得的奖励认识到状态/动作对的价值。由于网格世界的状态和动作数量有限，您可以使用 Q-table 将其映射到价值。

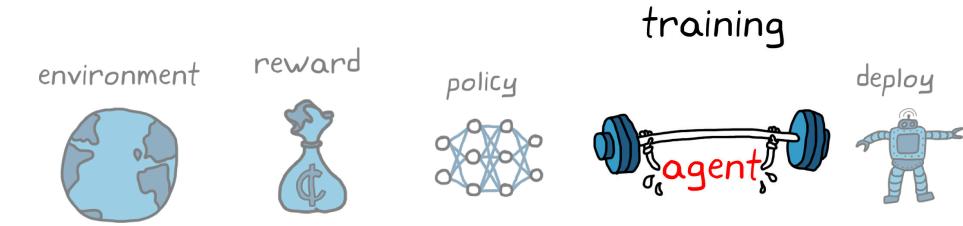
那么，智能体如何认识到这些价值呢？通过所谓的 Q-learning 过程。



借助 Q-learning，您可以首先将表格初始化为零，这样从智能体的角度而言，所有动作看上去并无不同。智能体采取随机动作后，将进入新状态并从环境中收取奖励。

智能体将奖励作为新信息，根据著名的贝尔曼方程，更新前一状态和刚采取的动作对应的价值。

贝尔曼方程



$$\text{new } Q(s, a) = Q(s, a) + \alpha [R(s, a) + \gamma \cdot \max Q'(s', a') - Q(s, a)]$$

利用贝尔曼方程，智能体可以将整个问题分解为多个简单步骤，来逐步求解 Q-table。智能体不会求解单个步骤的状态/动作对的真实价值，而是通过动态规划在每次访问状态/动作对被访问时，更新价值。贝尔曼方程对于 Q-learning 及另外一些学习算法（如 DQN）很重要。下面详细介绍方程中每一项的具体含义。

智能体根据状态 s 采取动作 a 后，将得到奖励。

价值不仅仅是动作的即时奖励；而是将来所能得到的最大预期回报。因此，状态/动作对的价值是指智能体刚刚得到的奖励加上智能体将来预计能得到的奖励。

您可以运用 gamma 折算未来奖励，避免智能体过于依赖将来的奖励。Gamma 取值介于 0（不根据未来奖励评估价值）和 1（考虑到无穷远的未来奖励）之间。

$$\text{new } Q(s, a) = Q(s, a) + \alpha [R(s, a) + \gamma \cdot \max Q'(s', a') - Q(s, a)]$$

↑
reward for taking action, a , from state, s

$$\text{new } Q(s, a) = Q(s, a) + \alpha [R(s, a) + \gamma \cdot \underbrace{\max Q'(s', a')}_{\uparrow} - Q(s, a)]$$

maximum expected value from state, s'

$$\text{new } Q(s, a) = Q(s, a) + \alpha [R(s, a) + \underbrace{\gamma \cdot \max Q'(s', a')}_{\uparrow} - Q(s, a)]$$

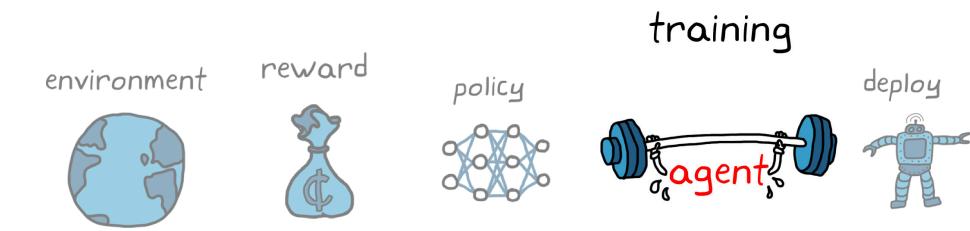
discount future rewards

贝尔曼方程 (续)

现在, 得出的总和是状态和动作对 (s, a) 的新价值, 将新价值与之前的估计价值进行比较以得出误差。

用误差乘以学习率, 从而控制将价值的旧估计值替换为新值 ($\alpha = 1$), 或者沿新值 ($\alpha < 1$) 的方向微调旧价值。

最后, 得到的增量值会添加到旧估计值, 从而更新 Q-table。



$$\text{new } Q(s, a) = Q(s, a) + \alpha \left[\frac{R(s, a) + \gamma \cdot \max Q'(s', a') - Q(s, a)}{\text{new best estimate of value}} \right]$$

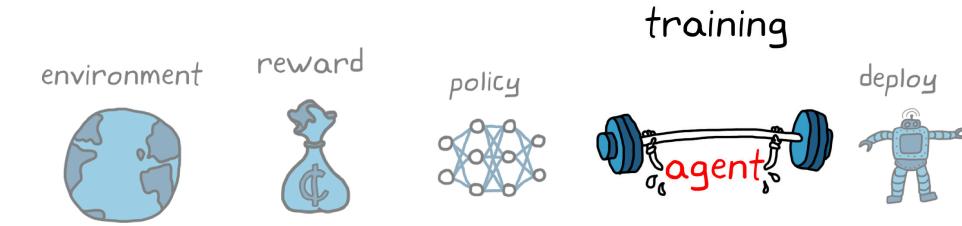
↑
previous estimate of value

$$\text{new } Q(s, a) = Q(s, a) + \frac{\alpha}{\text{error is multiplied by learning rate}} \left[R(s, a) + \gamma \cdot \max Q'(s', a') - Q(s, a) \right]$$

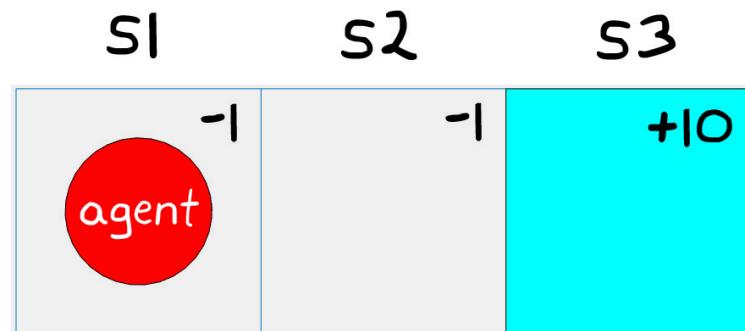
$$\text{new } Q(s, a) = \underbrace{Q(s, a)}_{\text{delta value is added to old estimate}} + \alpha \left[R(s, a) + \gamma \cdot \max Q'(s', a') - Q(s, a) \right]$$

贝尔曼方程是强化学习与传统控制理论之间的另一个关联点。如果您熟悉最优控制理论, 可能会发现该方程是哈密顿-雅可比-贝尔曼方程的离散形式; 当面向整个状态空间求解时, 该方程是实现最佳化的充分必要条件。

贝尔曼方程 (续)



仔细观察简单网格世界示例的前几个步骤，了解贝尔曼方程的意义，可能会有所帮助。在此示例中，alpha 设置为 1, gamma 设置为 0.9。如果两个动作的价值相同，智能体将采取随机动作；否则，智能体将选择价值最高的动作。

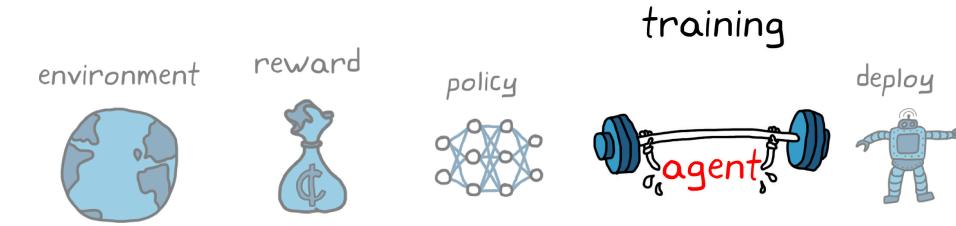


片段	步骤	状态	当前 Q(s, a)	动作	R(s, a)	新 Q(s, a)																								
1	1	S1	<table border="1"><tr><td></td><td>S1</td><td>S2</td><td>S3</td></tr><tr><td>左</td><td>0</td><td>0</td><td>0</td></tr><tr><td>右</td><td>0</td><td>0</td><td>0</td></tr></table>		S1	S2	S3	左	0	0	0	右	0	0	0	右 (随机)	-1	<table border="1"><tr><td></td><td>S1</td><td>S2</td><td>S3</td></tr><tr><td>左</td><td>0</td><td>0</td><td>0</td></tr><tr><td>右</td><td>-1</td><td>0</td><td>0</td></tr></table>		S1	S2	S3	左	0	0	0	右	-1	0	0
	S1	S2	S3																											
左	0	0	0																											
右	0	0	0																											
	S1	S2	S3																											
左	0	0	0																											
右	-1	0	0																											
					Bellman equation: $0 + 1 \cdot [-1 + 0.9 \cdot 0 - 0] = -1$																									
1	2	S2	<table border="1"><tr><td></td><td>S1</td><td>S2</td><td>S3</td></tr><tr><td>左</td><td>0</td><td>0</td><td>0</td></tr><tr><td>右</td><td>0</td><td>0</td><td>0</td></tr></table>		S1	S2	S3	左	0	0	0	右	0	0	0	右 (随机)	+10	<table border="1"><tr><td></td><td>S1</td><td>S2</td><td>S3</td></tr><tr><td>左</td><td>0</td><td>0</td><td>0</td></tr><tr><td>右</td><td>-1</td><td>10</td><td>0</td></tr></table>		S1	S2	S3	左	0	0	0	右	-1	10	0
	S1	S2	S3																											
左	0	0	0																											
右	0	0	0																											
	S1	S2	S3																											
左	0	0	0																											
右	-1	10	0																											
					Bellman equation: $0 + 1 \cdot [10 + 0.9 \cdot 0 - 0] = +10$																									

片段结束

当智能体进入终止状态 **S3** 时，片段结束，智能体以初始状态 **S1** 重新初始化。保存 Q-table 中的价值，学习进入下一片段，下一页将继续介绍。

贝尔曼方程 (续)

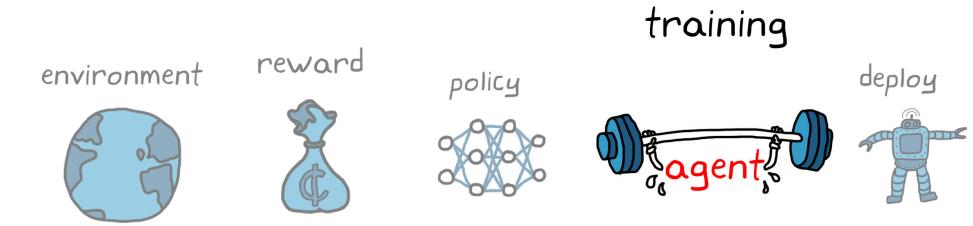


片段	步骤	状态	当前 Q(s, a)	动作	R(s, a)	新 Q(s, a)																								
2	1	S1	<table border="1"> <thead> <tr> <th></th><th>S1</th><th>S2</th><th>S3</th></tr> </thead> <tbody> <tr> <td>左</td><td>0</td><td>0</td><td>0</td></tr> <tr> <td>右</td><td>-1</td><td>10</td><td>0</td></tr> </tbody> </table>		S1	S2	S3	左	0	0	0	右	-1	10	0	左 (贪婪)	-1	<table border="1"> <thead> <tr> <th></th><th>S1</th><th>S2</th><th>S3</th></tr> </thead> <tbody> <tr> <td>左</td><td>-1</td><td>0</td><td>0</td></tr> <tr> <td>右</td><td>-1</td><td>10</td><td>0</td></tr> </tbody> </table>		S1	S2	S3	左	-1	0	0	右	-1	10	0
	S1	S2	S3																											
左	0	0	0																											
右	-1	10	0																											
	S1	S2	S3																											
左	-1	0	0																											
右	-1	10	0																											
					Bellman equation: $o + 1 \cdot [-1 + 0.9 \cdot 0 - o] = -1$																									
2	2	S1	<table border="1"> <thead> <tr> <th></th><th>S1</th><th>S2</th><th>S3</th></tr> </thead> <tbody> <tr> <td>左</td><td>-1</td><td>0</td><td>0</td></tr> <tr> <td>右</td><td>-1</td><td>10</td><td>0</td></tr> </tbody> </table>		S1	S2	S3	左	-1	0	0	右	-1	10	0	右 (随机)	-1	<table border="1"> <thead> <tr> <th></th><th>S1</th><th>S2</th><th>S3</th></tr> </thead> <tbody> <tr> <td>左</td><td>-1</td><td>0</td><td>0</td></tr> <tr> <td>右</td><td>8</td><td>10</td><td>0</td></tr> </tbody> </table>		S1	S2	S3	左	-1	0	0	右	8	10	0
	S1	S2	S3																											
左	-1	0	0																											
右	-1	10	0																											
	S1	S2	S3																											
左	-1	0	0																											
右	8	10	0																											
					Bellman equation: $-1 + 1 \cdot [-1 + 0.9 \cdot 10 - (-1)] = +8$																									

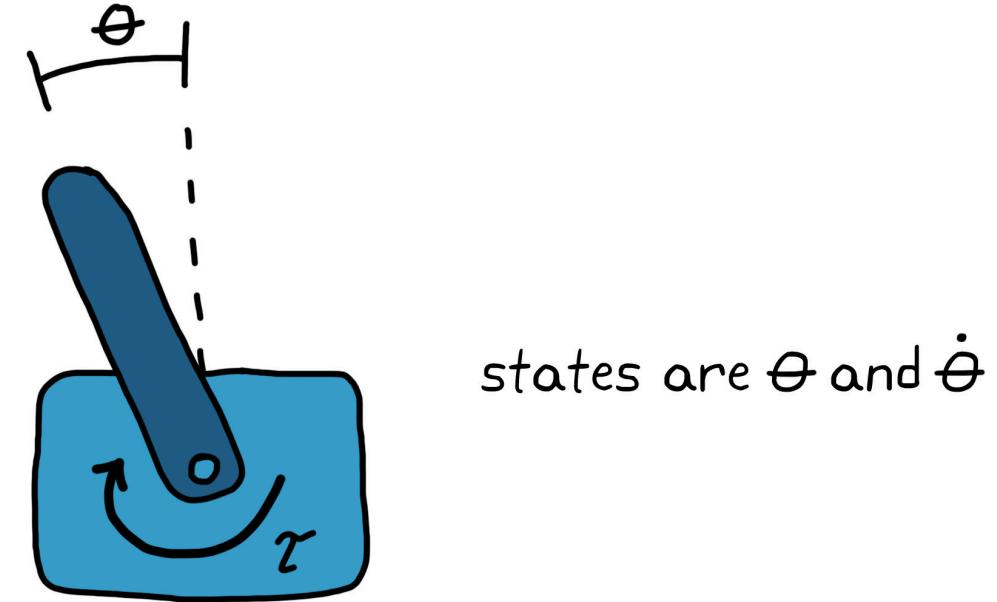
片段结束

只需四个动作，智能体已经确定产生最优策略的 Q-table；在状态 S1 下，由于价值 8 大于 -1，因此将右移；在状态 S2 下，由于值 10 大于 0，因此将再次右移。关于这个结果，有趣的是，Q-table 并未逐一确定每个状态/动作对的真实价值。如果继续学习，价值将继续朝实际价值的方向移动。但是，您不必确定真实价值即可产生最优策略；只需将最优动作价值设置为最高数值。

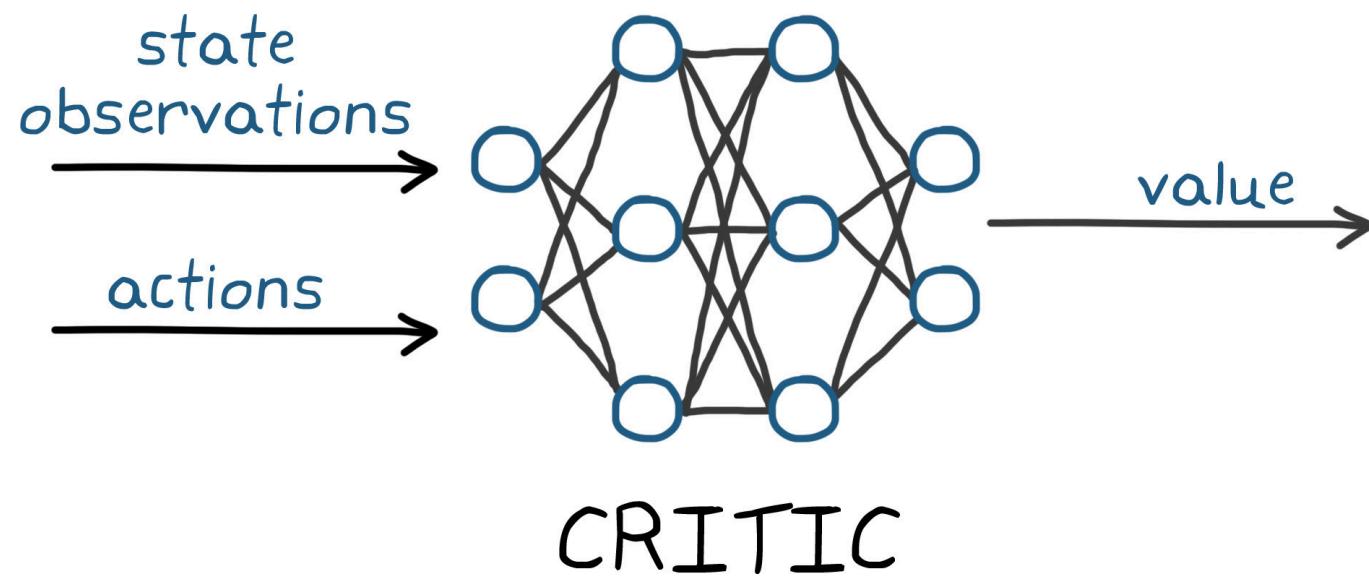
基于神经网络的评价器



将这一构想延伸应用于倒立摆。与网格世界一样，包含角度和角速率两个状态量，只不过现在的状态量是连续的。



value function-based learning



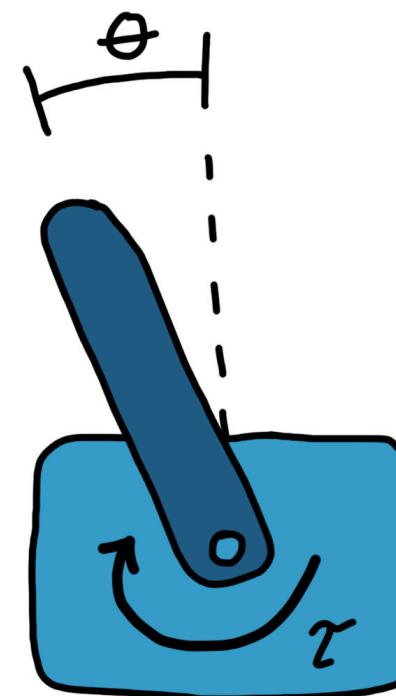
用神经网络表示价值函数（评价器）。该构想与使用表格相同：输入状态观测量和动作，神经网络返回状态/动作对的价值，策略将选择具有最高价值的动作。

久而久之，网络将缓慢收敛到一个函数，针对连续状态空间任意位置的每个动作输出真实价值。

基于价值的策略的缺点

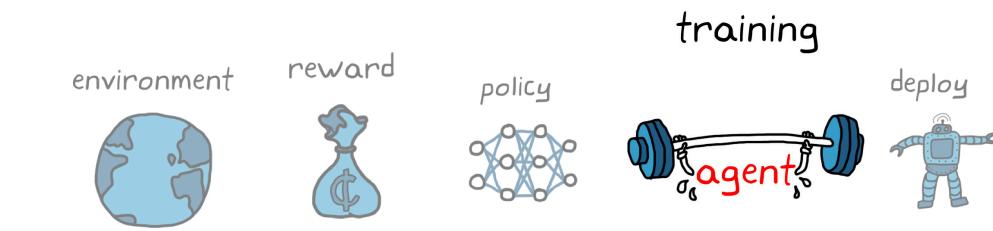
您可以使用神经网络为连续状态空间定义价值函数。如果倒立摆具有离散动作空间，则可逐一将离散动作馈送至评价器网络。

基于价值函数的策略不适用于连续动作空间。这是因为无法针对每个取值有无限种可能的动作逐一计算价值来确定最大价值。即使是大(但非无限)的动作空间, 耗费的计算资源也会很大。这一点颇为遗憾, 因为在控制问题中, 经常会出现连续动作空间, 如施加一系列连续取值范围内的扭矩, 以求解倒立摆问题。



continuous states: θ and $\dot{\theta}$

discrete action space: $\mathcal{A} = [-2, -1, 0, 1, 2] \text{ Nm}$

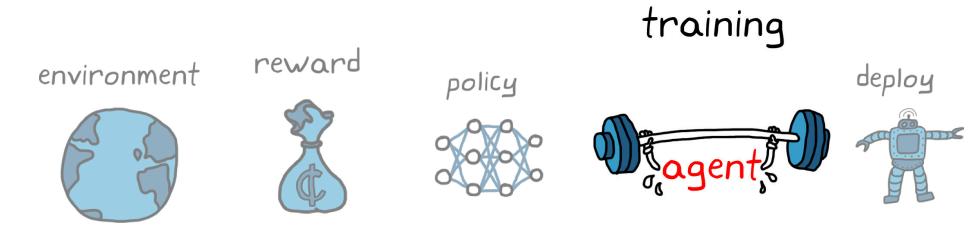


那么, 您可以怎么做?

您可以使用普通策略梯度法, 如基于策略函数的算法部分所述。此类算法能够处理连续动作空间;但是, 如果奖励方差大且梯度噪声大, 将难以收敛。您也可以将两种学习方法合并为一类算法, 即所谓的执行器-评价器算法。

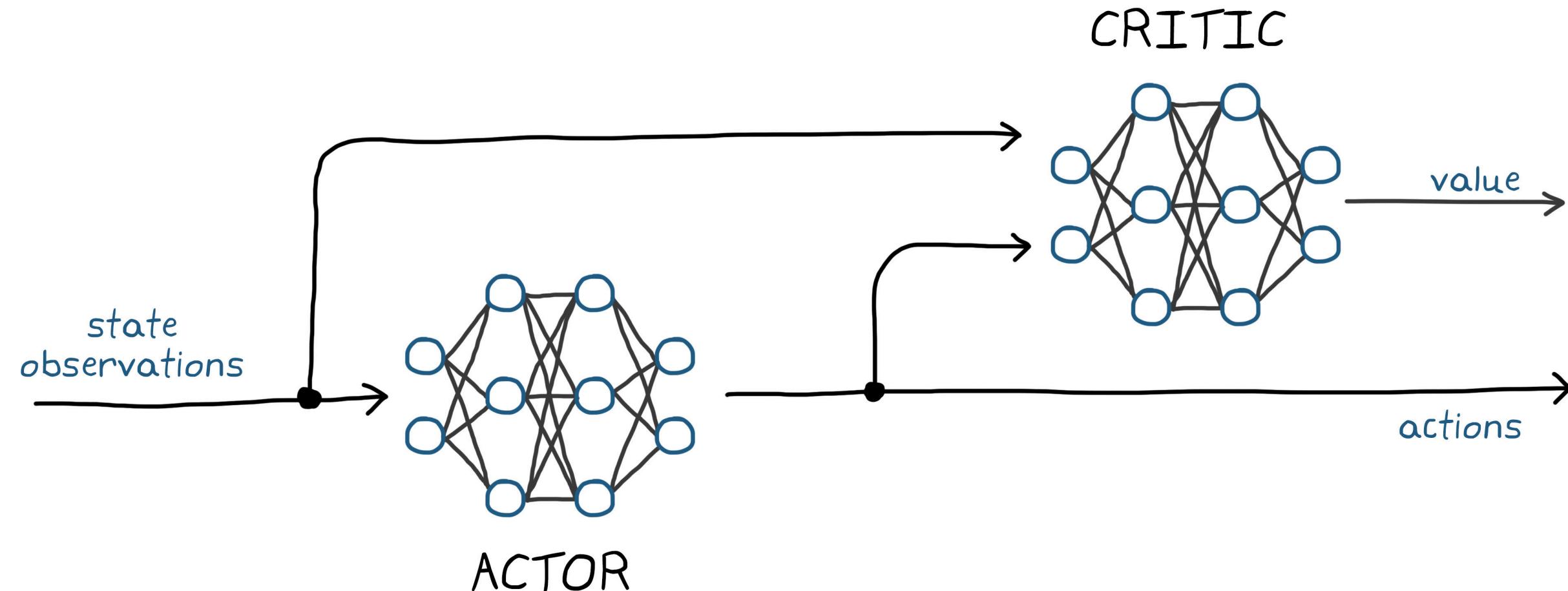
» [了解如何在 MATLAB 中训练执行器-评价器智能体保持倒立摆平衡](#)

执行器-评价器方法

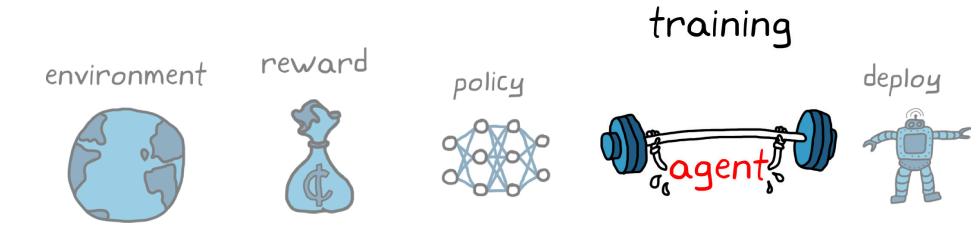


执行器是一个网络，用于在当前状态给定的情况下尝试采取自认为最佳的动作，如策略函数方法所示。评价器是另一个网络，用于根据状态和执行器采取的动作估计价值，如价值函数方法所示。此方法适用于连续动作空间，因为评价器只需查看执行器采取的单个动作，无需评估所有动作来尝试确定最佳动作。

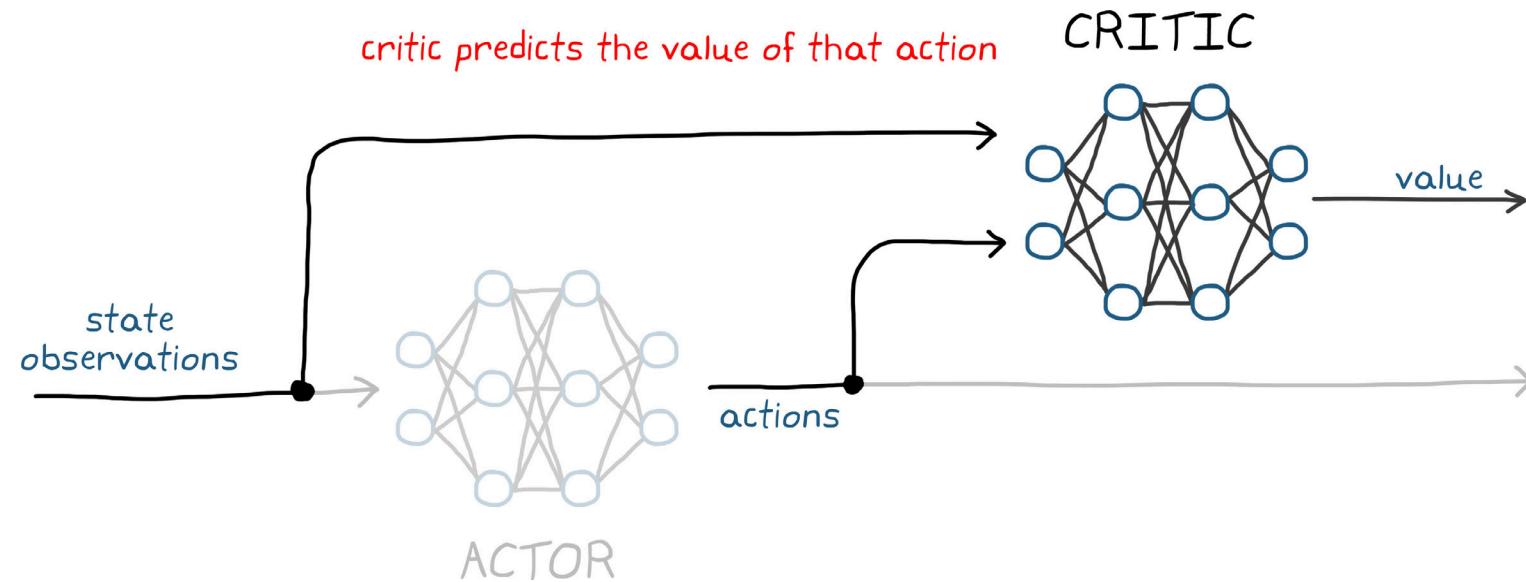
actor-critic learning algorithms



执行器-评价器学习周期

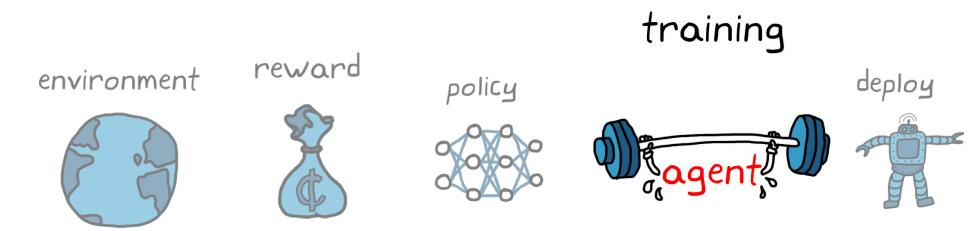


执行器采用与策略函数算法相同的方式选择动作，并将其应用于环境。

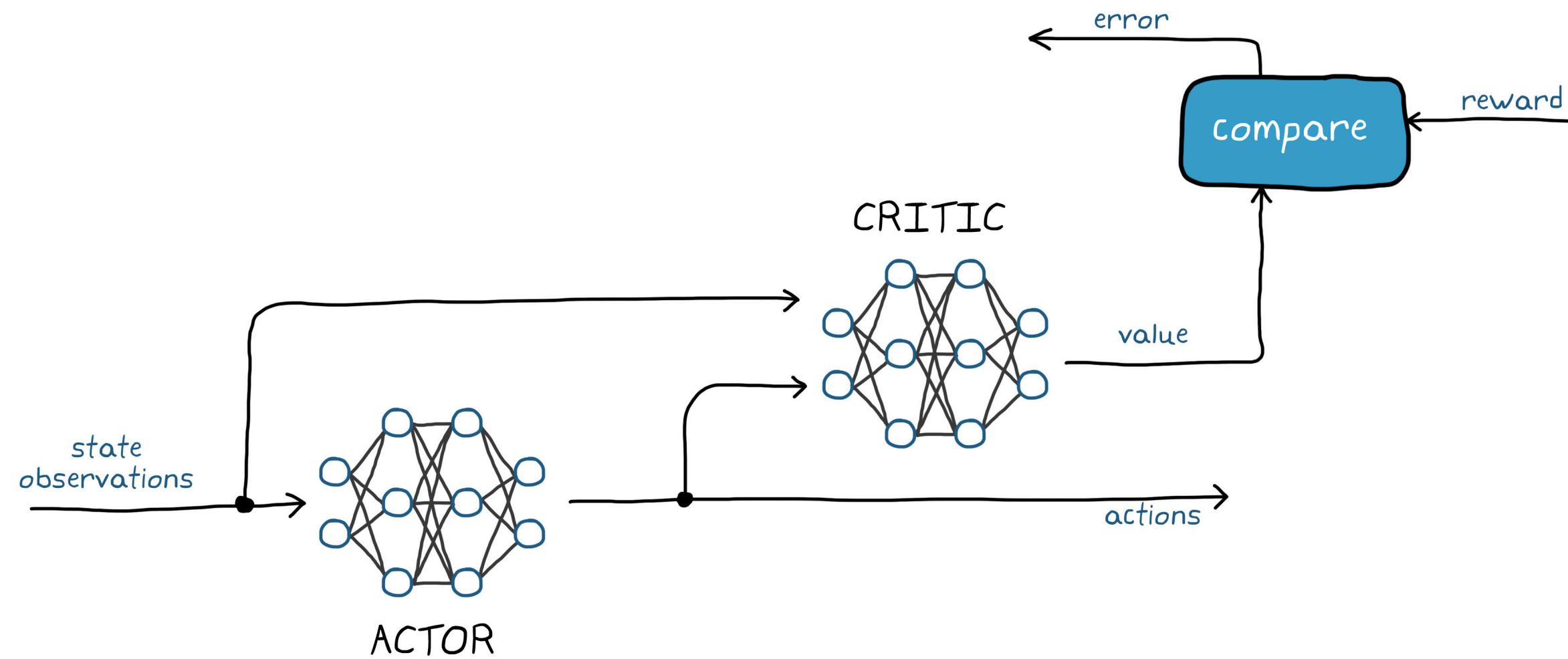


评价器根据当前状态和动作对预测相应动作的价值。

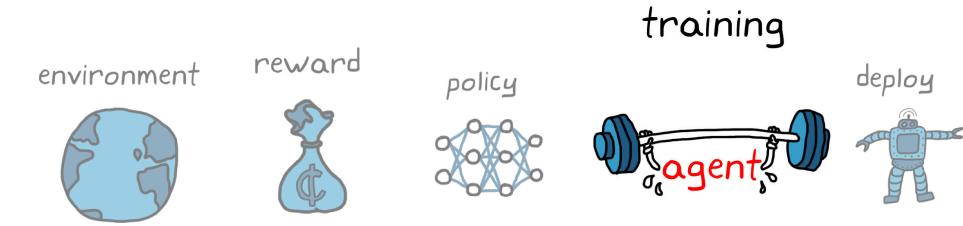
执行器-评价器学习周期 (续)



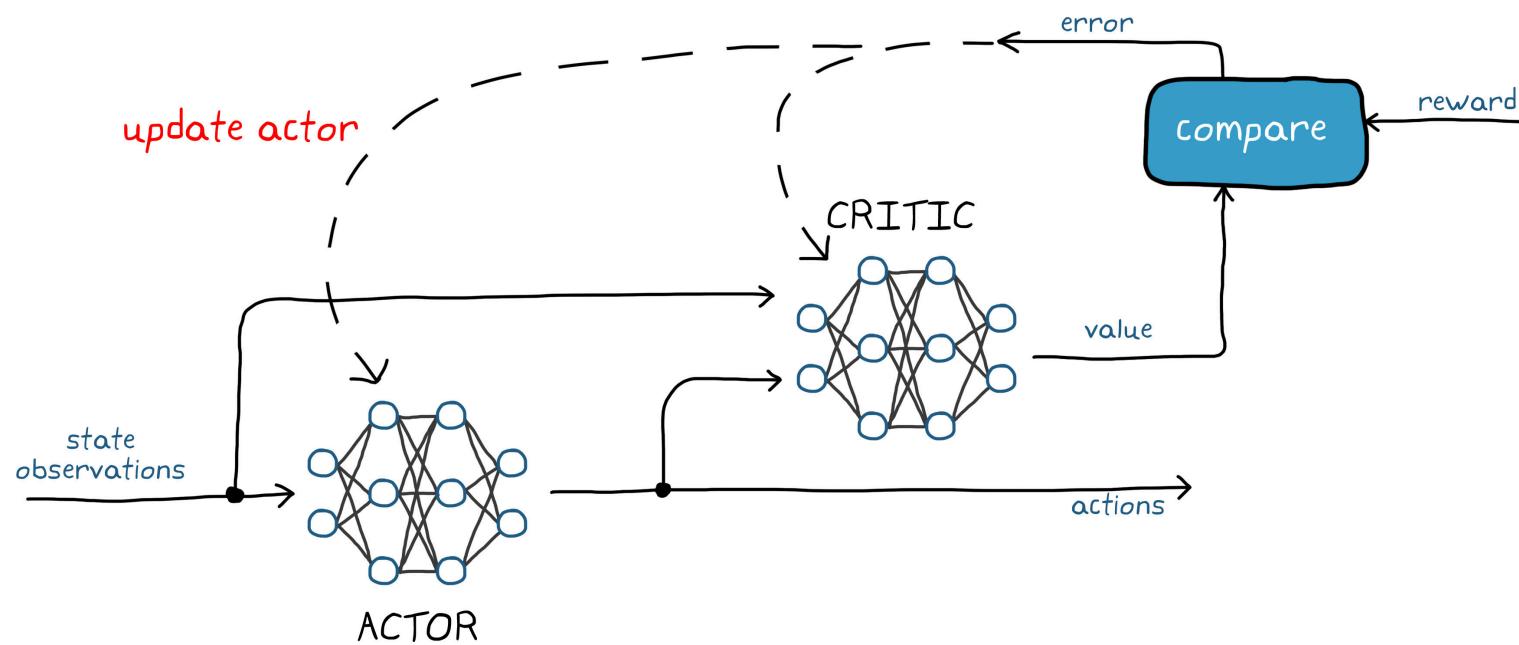
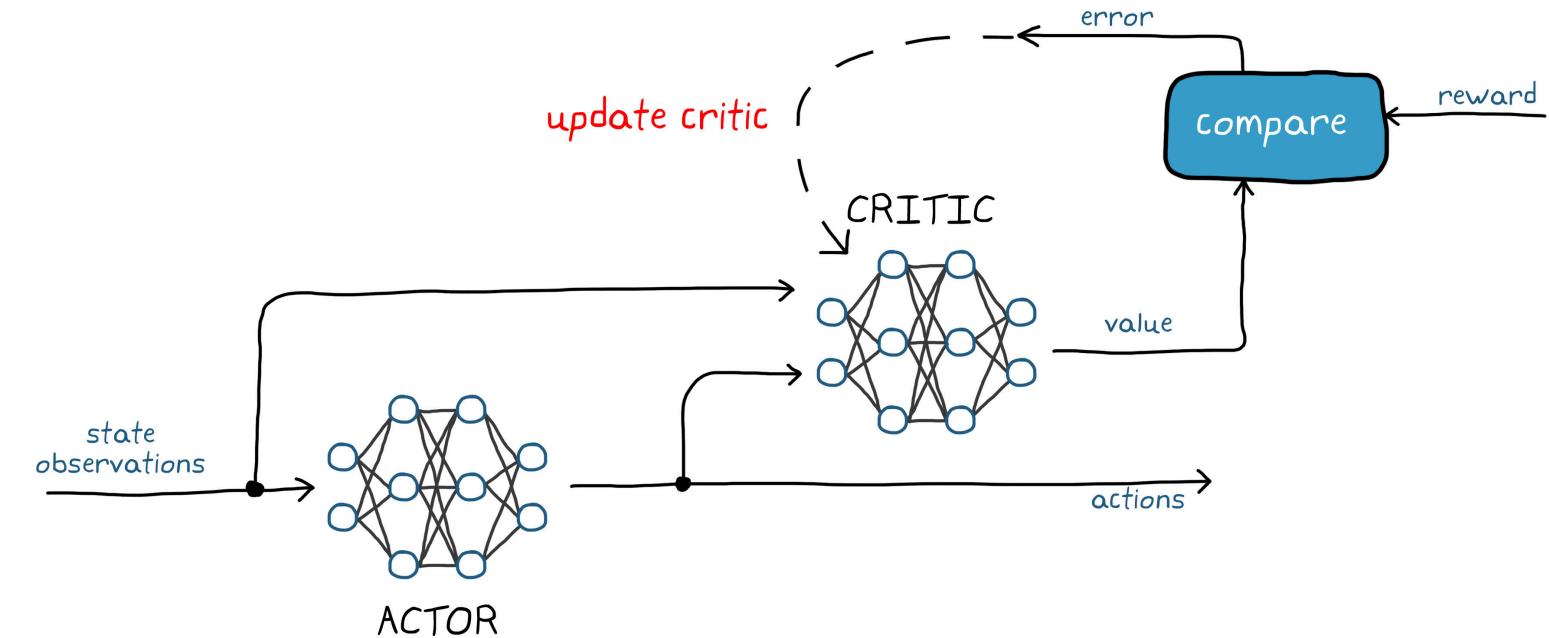
而后,评价器使用环境奖励确定其价值预测的准确性。误差是对前一状态的价值的新估计值和评价器网络给出的旧估计值之间的差异。价值的新估计值基于得到的奖励和当前状态的折扣价值得出。评价器可通过误差判断结果好于预期还是逊于预期。



执行器-评价器学习周期 (续)



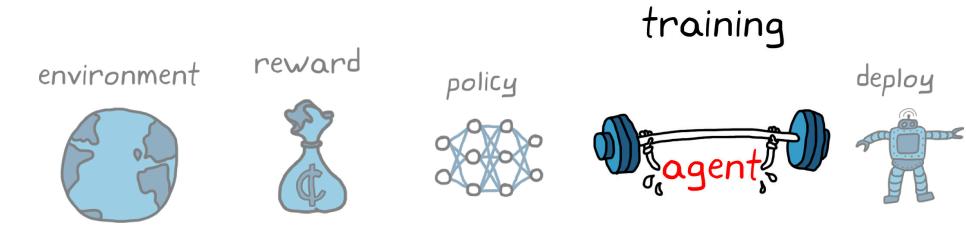
评价器使用此误差自我更新,采用的方式与价值函数完全相同,以便下次进入此状态时更好地进行预测。



执行器也会根据评价器响应自我更新,以便调整将来再次采取该动作的概率。

因此,策略现在按照评价器建议的方向沿着奖励函数的斜坡攀升,而不是直接使用奖励。

两种互补网络

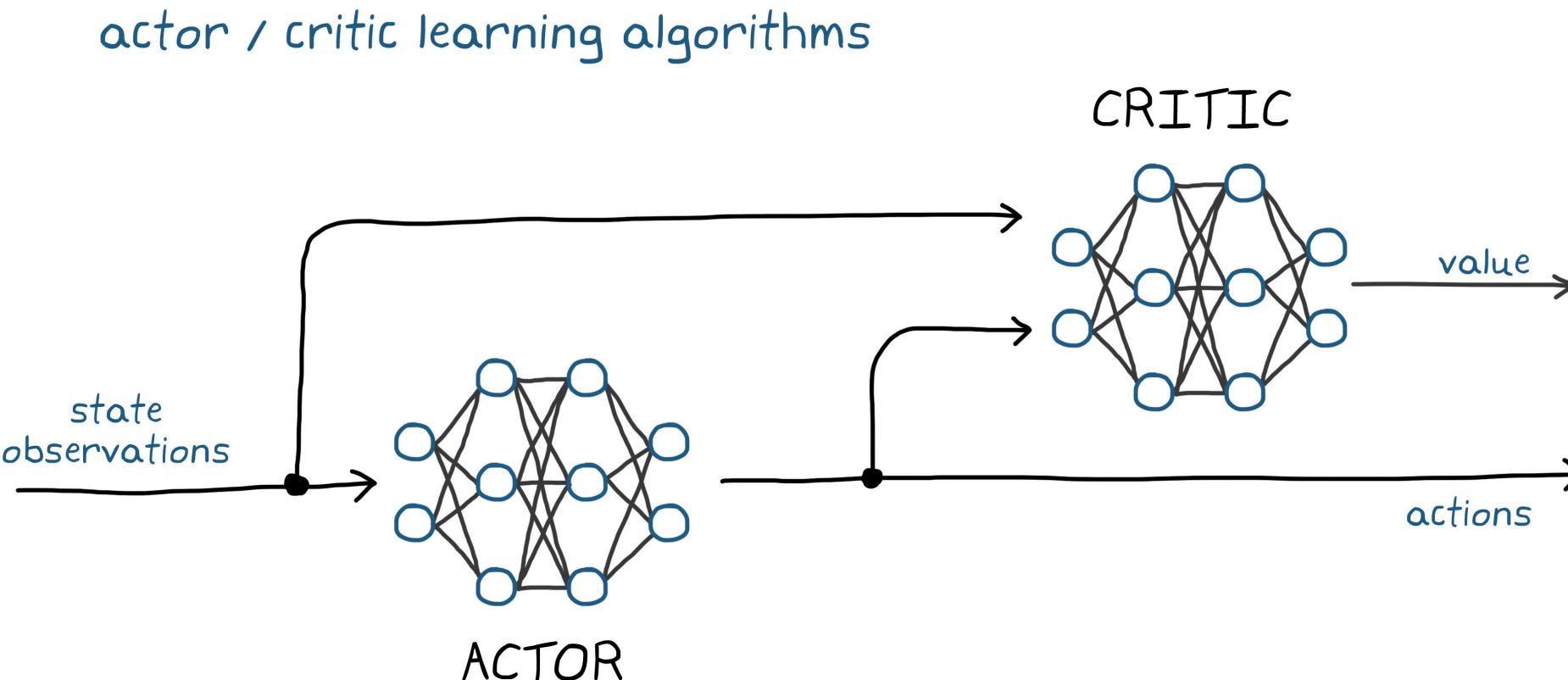


许多不同类型的学习算法使用执行器-评价器策略；本电子书对这些概念进行了归纳，不特定于某种算法。

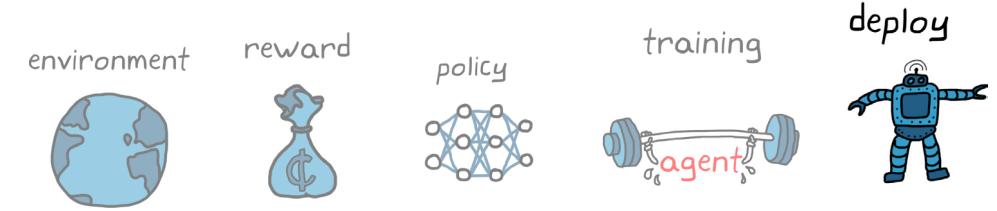
执行器和评价器均为尝试学习最优行为的神经网络。执行器使用评价器的反馈来确定动作好坏，从而学习需要采取的正确行为；评价器通过得到的奖励学习价值函数，以便正确评价执行器采取的动作。对于执行器-

评价器方法，智能体可以利用策略函数算法和价值函数算法的最佳部分。执行器-评价器可以处理连续状态空间和动作空间，并在返回的奖励方差较高时加快学习速度。

希望大家对于为什么在创建智能体时必须设置两个神经网络有了清楚的认识；每个神经网络都扮演着特定的角色。

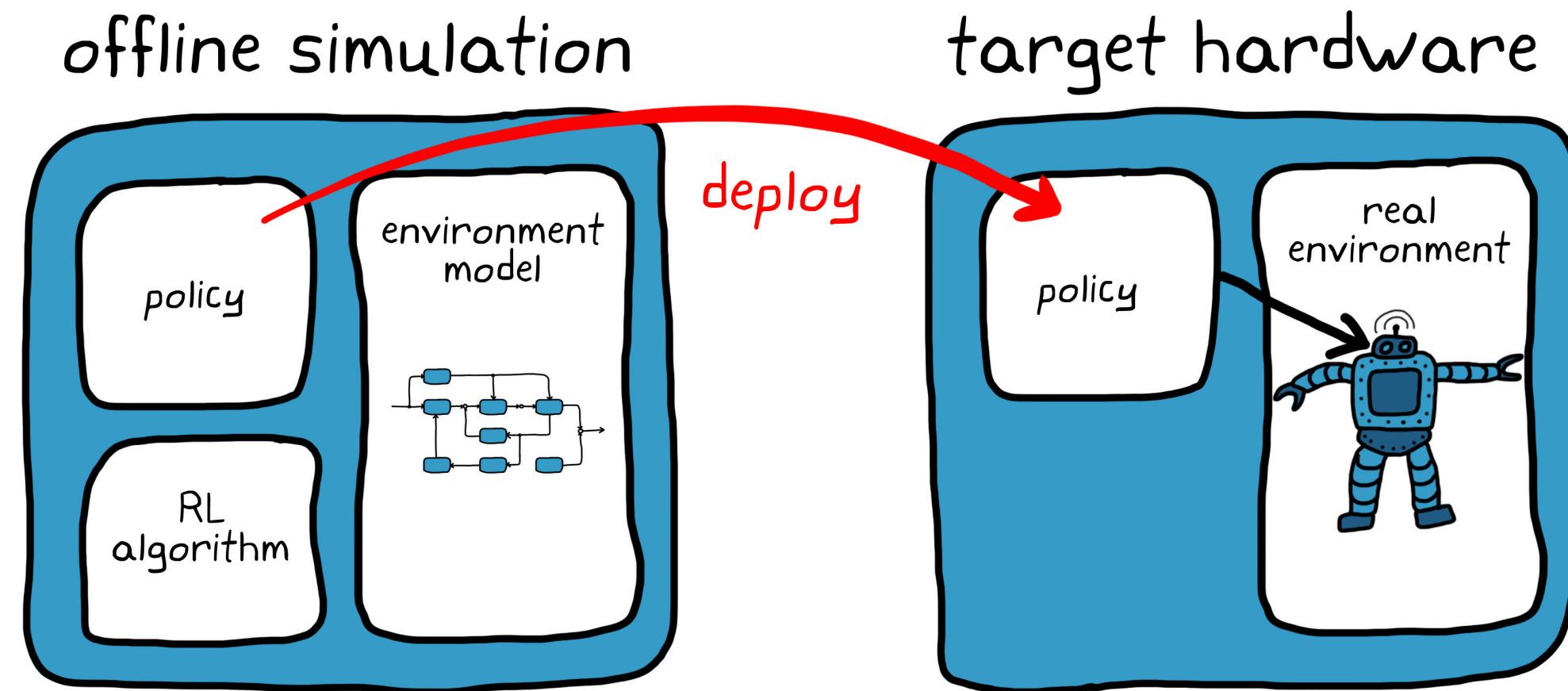


策略部署

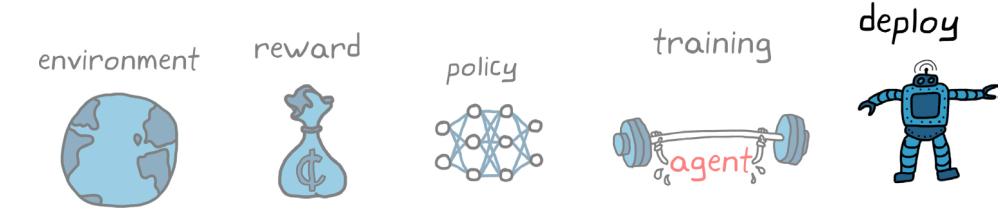


强化学习工作流程的最后一步是部署策略。

如果在真实环境中使用实物智能体开展学习，则学习的策略已存在于智能体上且可供利用。本电子书假设智能体已通过与仿真环境交互进行离线学习。一旦策略足够理想，则可停止学习过程，将静态策略部署到任意数量的目标，就像部署采用传统方式开发的控制律一样。



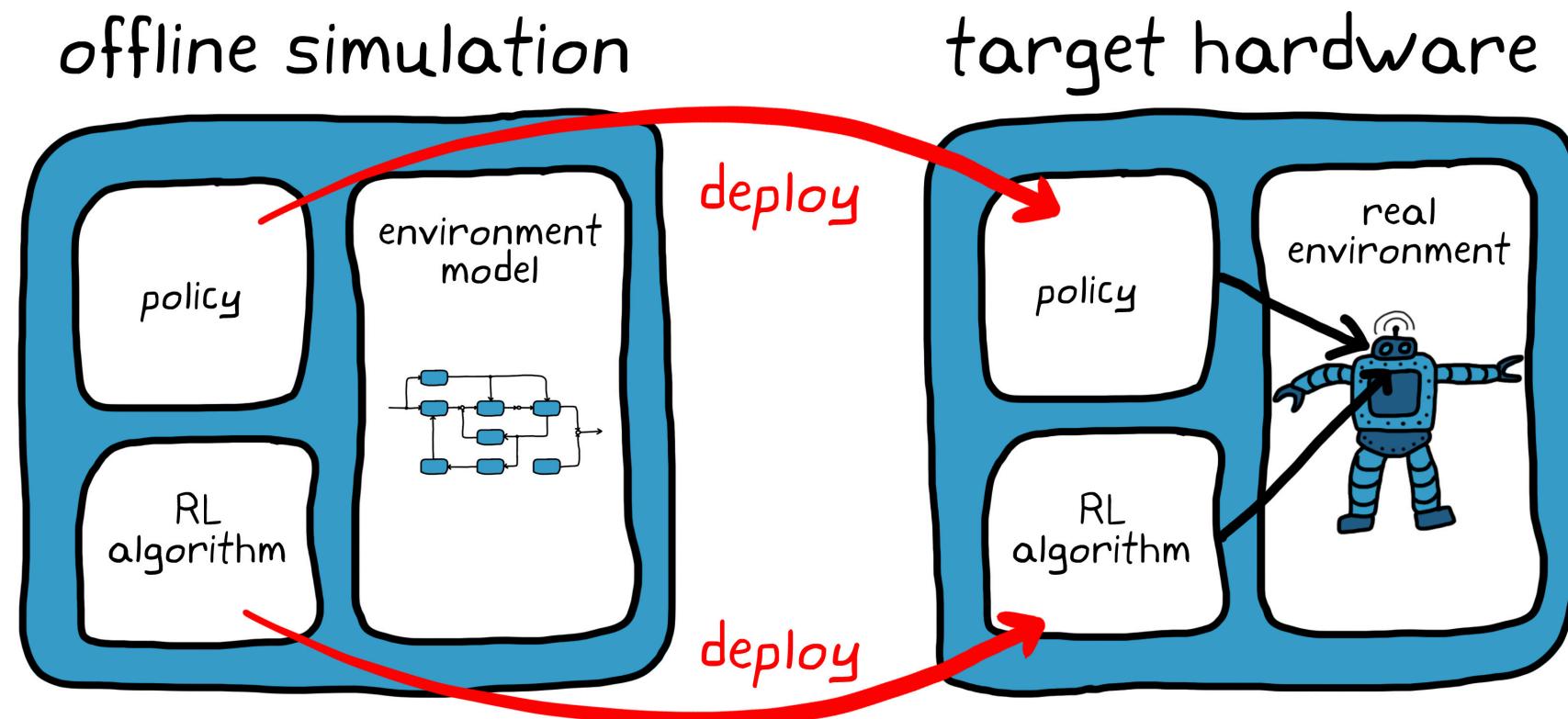
部署学习算法



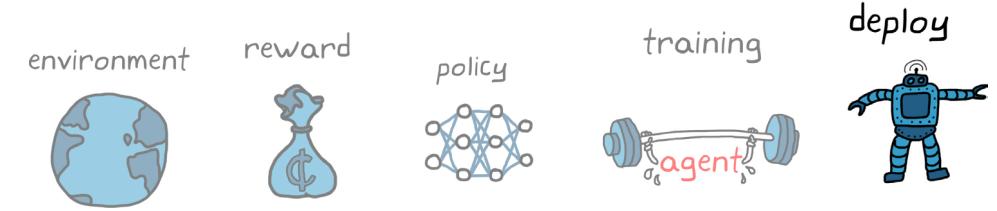
即便大部分学习在仿真环境下离线完成，部署后仍然可能需要使用真实物理硬件继续开展学习。

这是因为部分环境可能难以准确建模，因此最适合模型的策略可能不一定最适合真实环境。另一个原因可能是环境会随着时间慢慢发生变化，智能体必须不定期地继续学习，才能适应这些变化。

出于这些原因，您需要将静态策略和学习算法都部署到目标。对于此设置，您可以选择执行静态策略（停止学习）或继续更新策略（启动学习）。

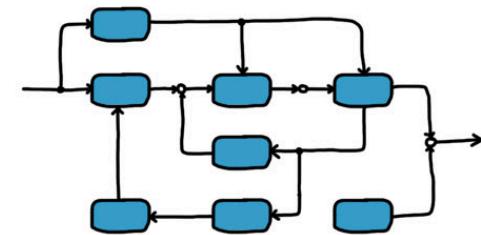


互补关系



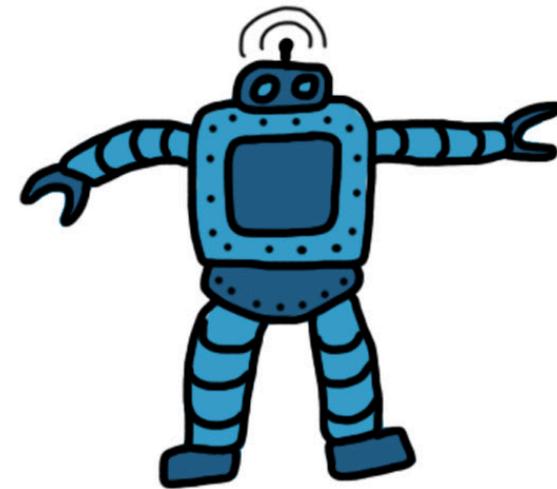
在仿真环境下学习与在真实环境下学习属于互补关系。您可以通过仿真相对快速、安全地学习足够理想的策略 — 该策略将保障硬件安全, 即使不完美也会接近期望行为。然后, 您可以通过物理硬件和在线学习来调整策略, 从而创建针对环境微调的策略。

start learning here



coarse-tuned
optimal policy

finish learning here



fine-tuned
optimal policy

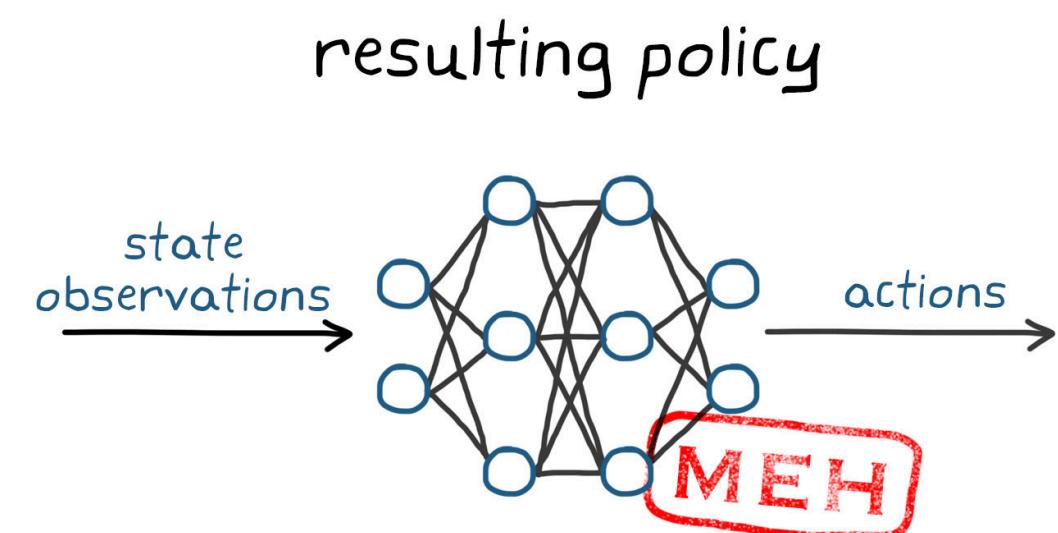
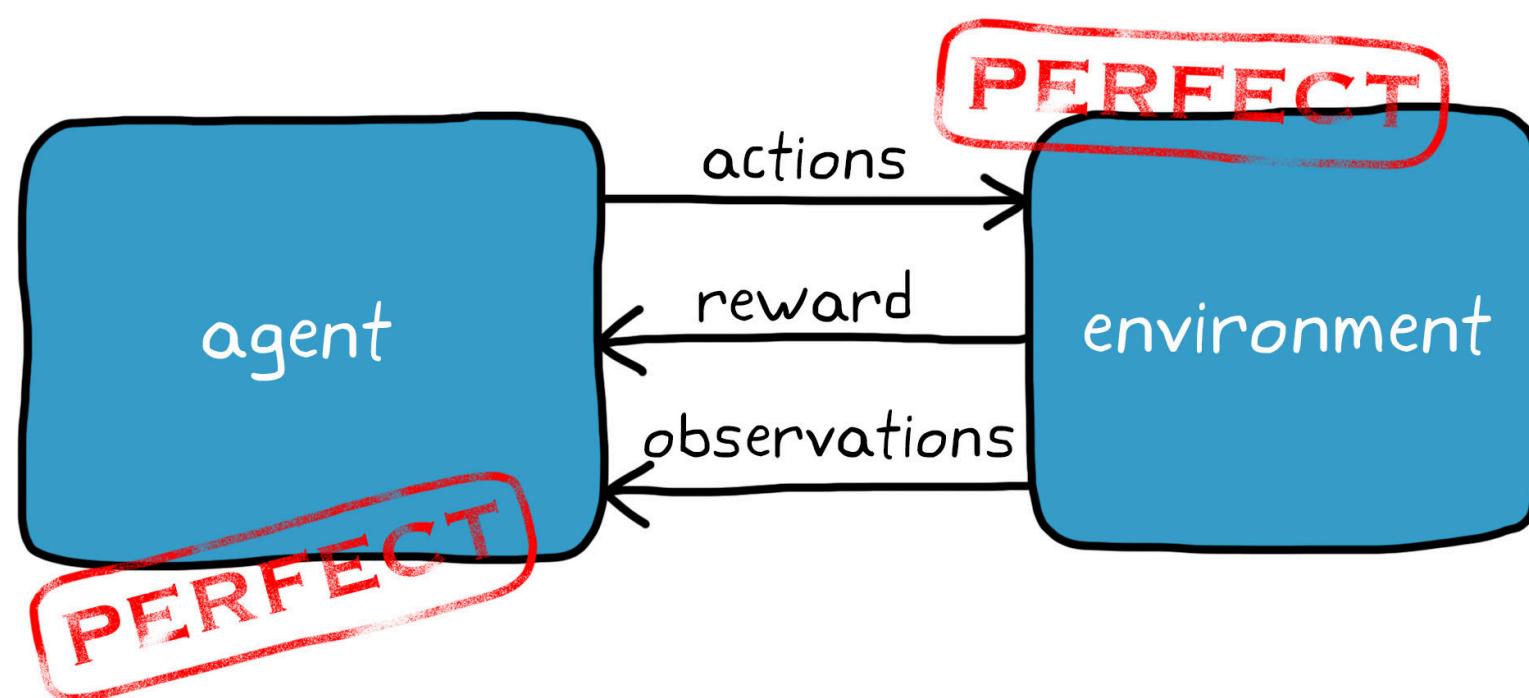


RL 的缺点

此时,您可能认为自己可以设置环境,在环境中部署强化学习智能体,然后让计算机去解决您的问题,而您可以走开喝杯咖啡了。遗憾的是,即使设置了完美的智能体和完美的环境,并且学习算法收敛到解决方案,该方法仍然存在缺陷。

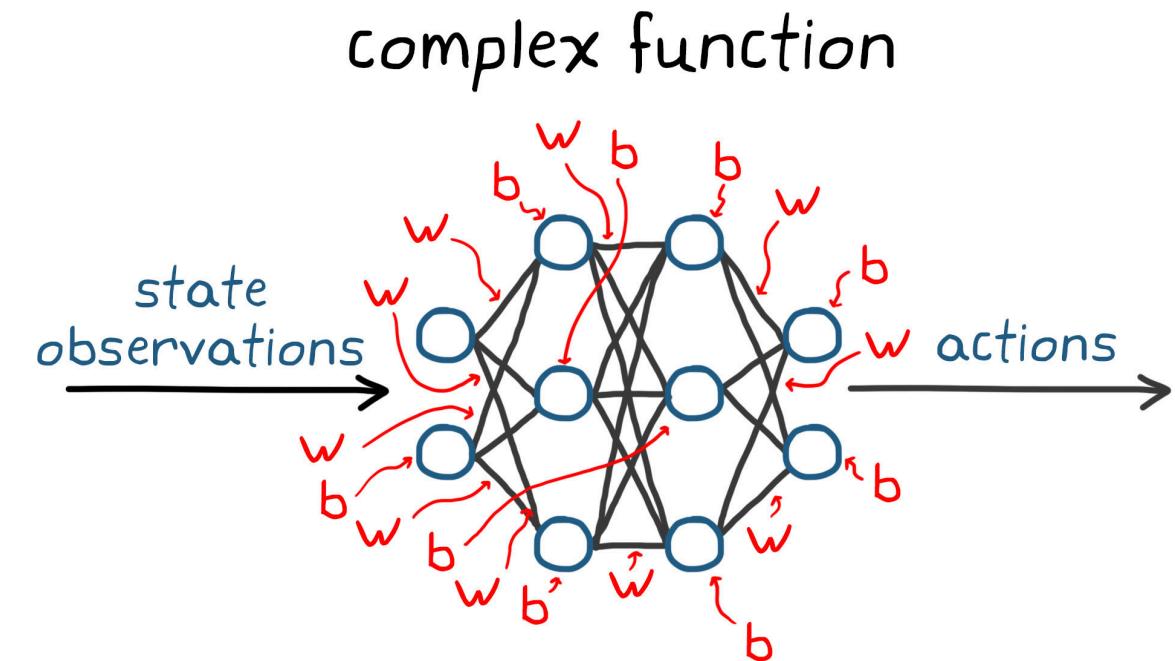
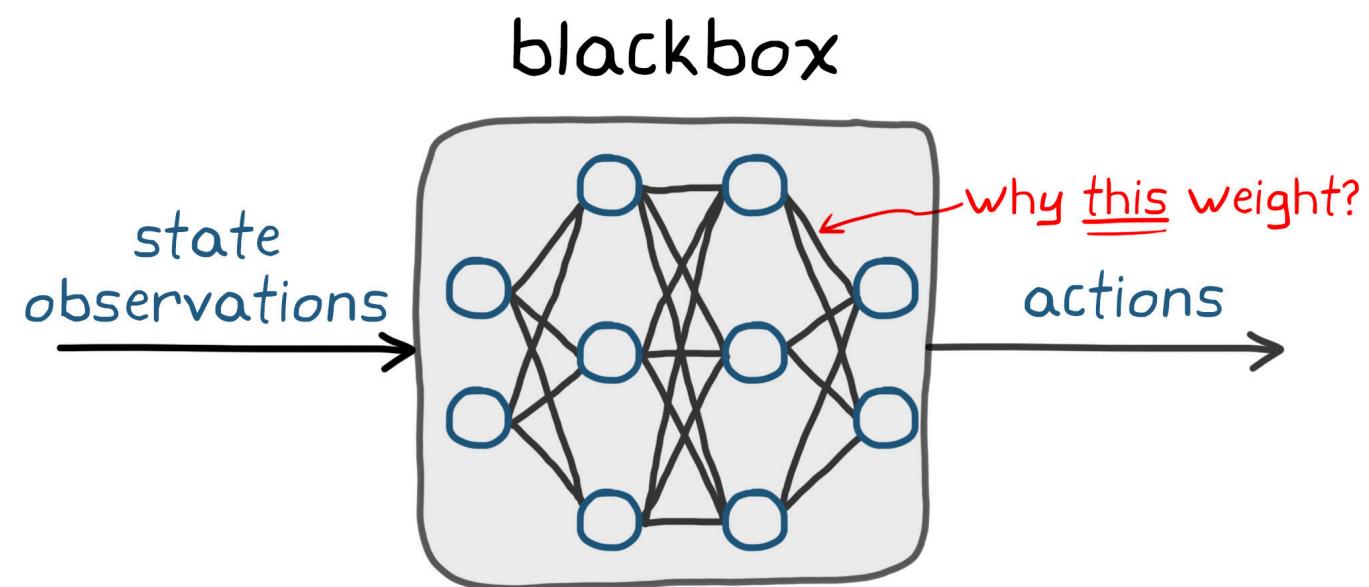
这些挑战可归结为两大问题:

- 如何知道解决方案会发挥作用?
- 如果不够完美,可以手动调整吗?



无法解释的神经网络

从数学角度而言,策略由神经网络构成,其中可能包含数十万个权重和偏置及非线性激活函数。这些值与网络结构组合形成了一个复杂函数,将高级观测值映射到低级动作。



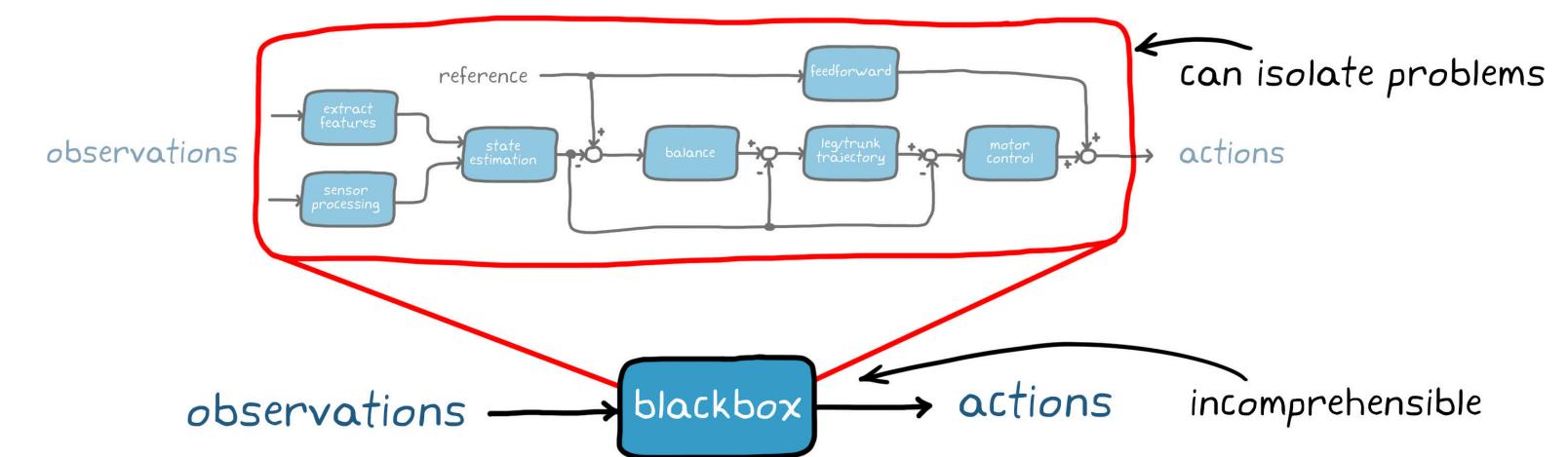
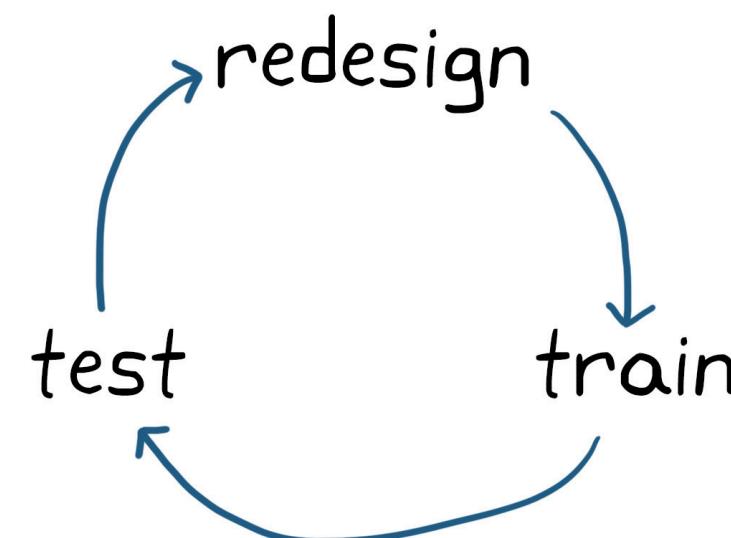
对于设计者来说,此函数是一个“黑盒”。您可能对此函数的运行原理及此网络已识别的隐藏特征具有直观的认识,但恐怕还不了解给定权重或偏置值背后的原因。因此,如果策略不符合规范或操作环境发生变化,势必不知如何调整策略才能解决问题。

目前正在开展的研究,尝试推广可解释的人工智能(explainable artificial intelligence)这一概念。根据这一构想,您可以设置网络,以便人们能够轻松理解和核查。目前,强化学习生成的大部分策略仍归类为黑盒,这个问题迫切需要解决。

精准定位问题

此时面临一个问题：将难度较大的逻辑精简为单一黑盒函数降低了控制问题的解决难度，但却导致最终解决方案无法解释。将其与采用传统方式设计的控制系统做个对比：传统控制系统通常采用层次结构，其中包含环路和级联控制器，每个环路和控制器用于控制系统的特定动态特性。思考一下如何通过物理属性（如附肢长度或电机常数）推导增益；即使物理系统发生变化，更改这些增益也很方便。

此外，如果系统未按预期方式运行，对于传统设计，通常可以将问题精确定位到特定控制器或环路，而后进行重点分析。您可以隔离、测试和修改控制器，确保控制器在指定条件下正常运行，然后将其带回更大的系统。

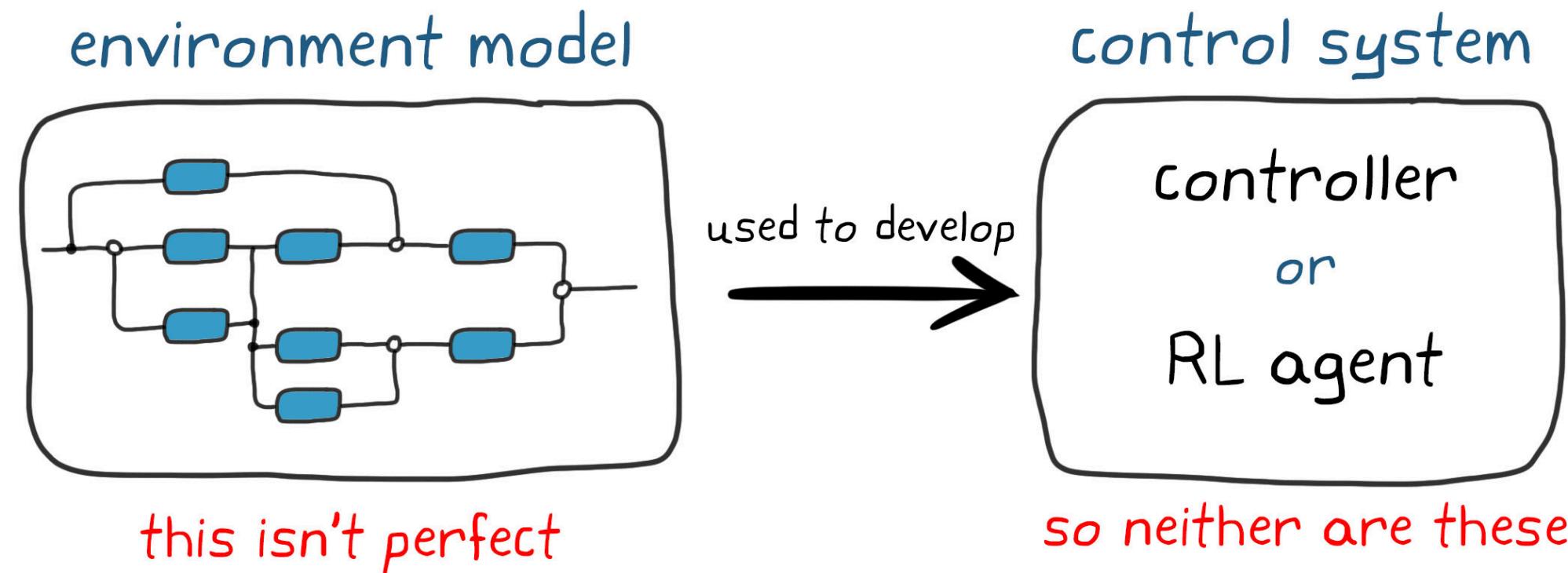


如果解决方案是一个庞大的神经元、权重和偏置的集合，则很难隔离问题。因此，如果最终得出的策略不太适当，而又无法单独修改出问题的策略部分，则必须重新设计智能体或环境模型，然后再次进行训练。重新进行一轮设计、训练和测试可能极为耗时。

更严峻的问题

届时还会可能面临一个更严峻的问题：不但训练智能体所需的时间长，还会影响环境模型的准确性。

想要开发出足够逼真的模型，充分考量所有重要的系统动态特性、干扰和噪声，是很困难的。有些时候，它无法完美反映实际情况，因此使用该模型开发的所有控制系统也不会完美。因此，仍然必须进行实物测试，而不仅仅是通过模型验证各个方面。



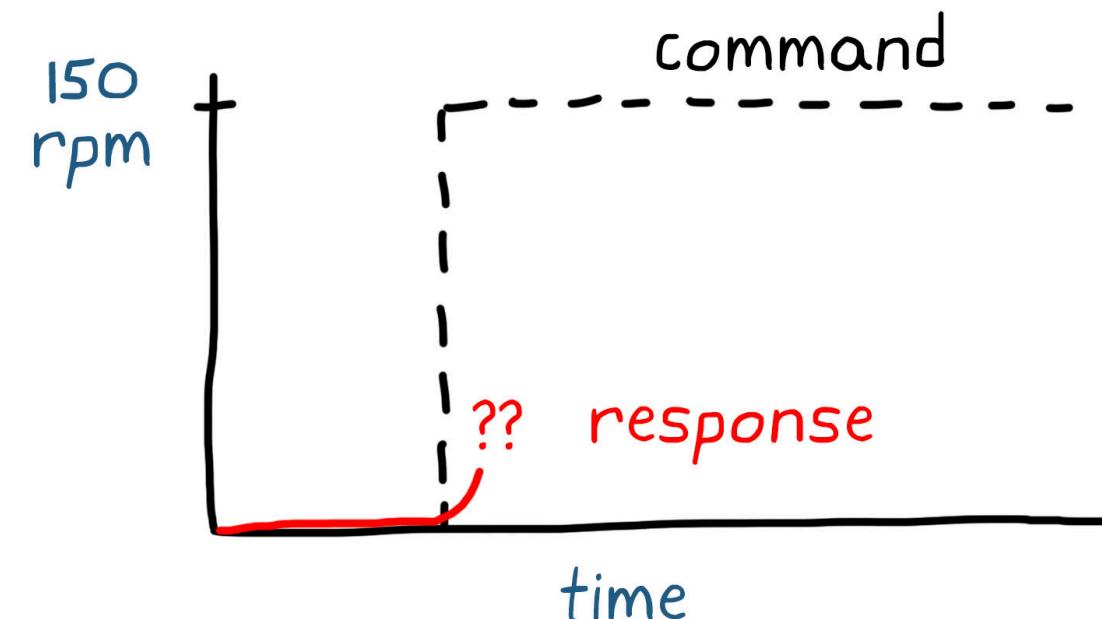
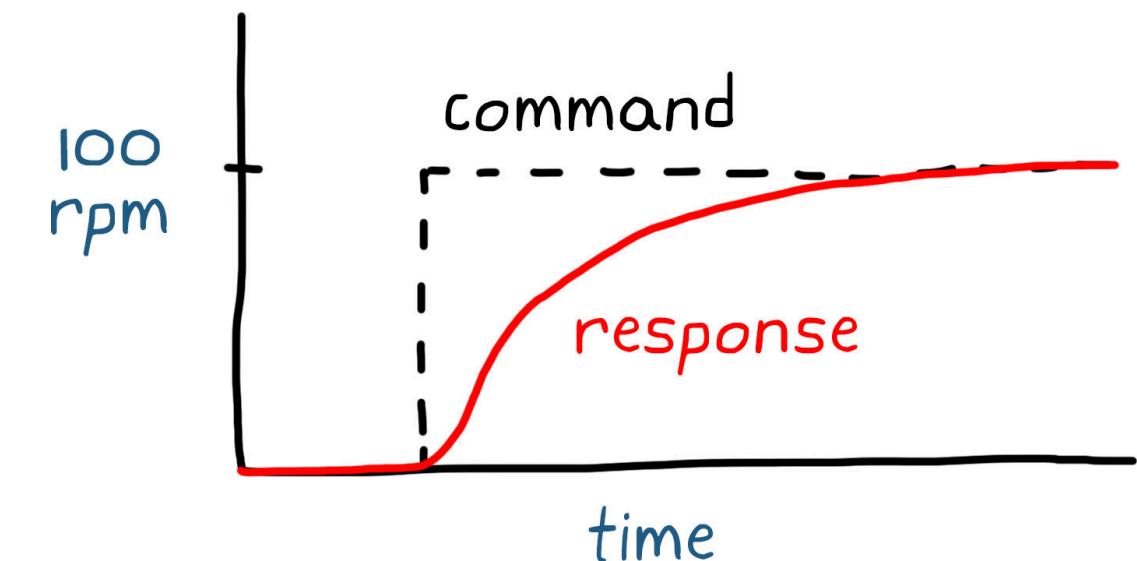
如果使用模型设计传统控制系统，这将不再是问题，因为您可以理解函数并调整控制器。但是，对于神经网络策略，则没有那么好办。鉴于永远无法构建绝对真实的模型，因此使用该模型训练的所有智能体都会存在轻微错误。为了解决这个问题，唯一方法是基于物理硬件训练智能体，这本身就颇具挑战性。

验证学习的策略

使用神经网络同样很难验证策略是否符合规范。出于某种原因，使用学习的策略很难根据系统在某一状态下的行为预测系统在另一状态下的行为。举例来说，如果您通过让智能体学习跟踪一个从 0 到 100 RPM 的阶跃输入，训练它控制电机速度，那么除非进行测试，否则您无法确定同一策略是否能够跟踪一个类似的从 0 到 150 RPM 的阶跃输入。即使电机呈线性运行也是如此。

一个微小的变化都有可能激活一组截然不同的神经元并产生意外结果。除非进行测试，否则无从得知。测试更多的条件确实可以降低风险，但除非能够测试每一个输入组合，否则无法保证策略完全正确。

不得不额外运行一些测试似乎没什么大不了，但是必须谨记，深度神经网络的一大优势在于，它们可以处理各种传感器的数据，比如输入空间极大的摄像机图像；设想一下成千上万个像素，每个像素点的值介于 0 到 255 之间。在这种情况下，根本不可能测试每一个组合。



形式化验证方法

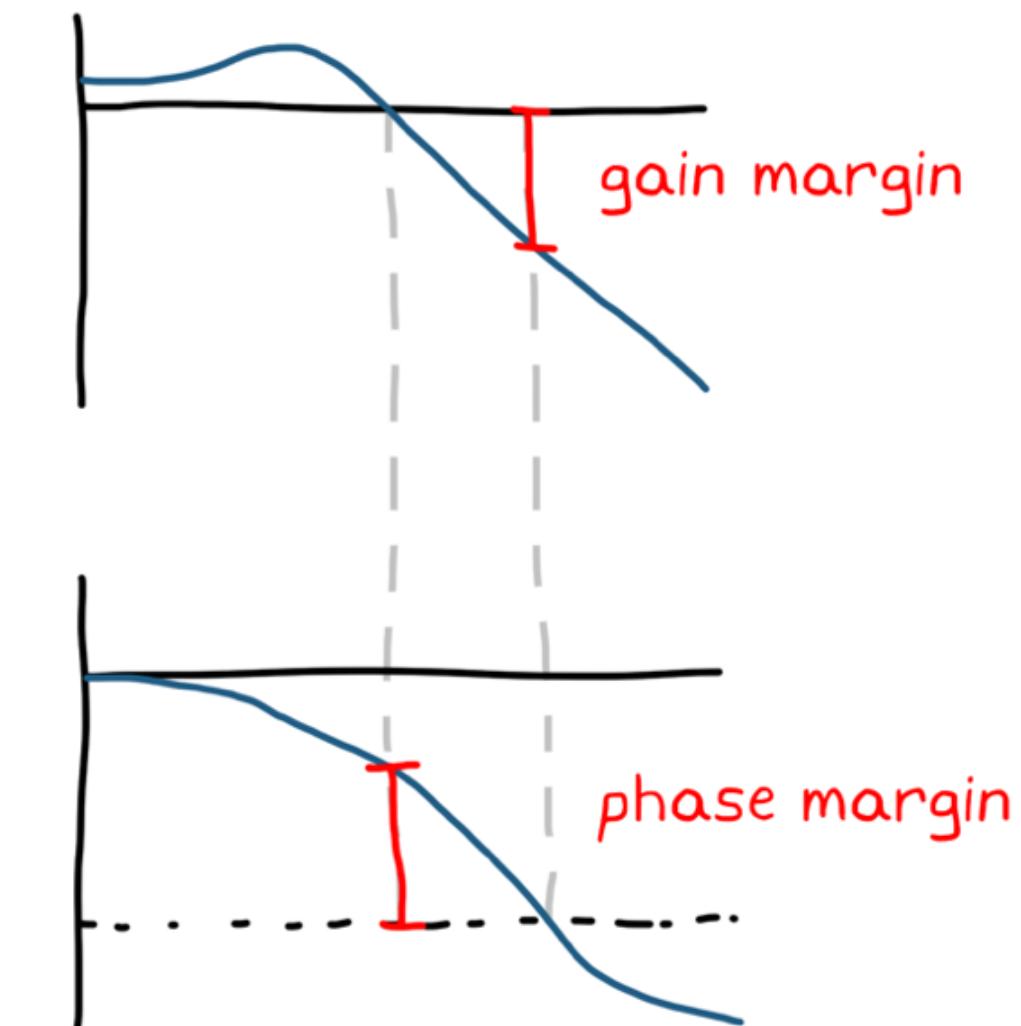
学习的神经网络同样会增加形式化验证的难度。此类方法包括通过提供形式化证明(而不是采用测试)保证满足某项条件。例如,如果在软件中对信号进行了绝对值运算,则无需通过测试确保该信号始终为非负信号。只需检查代码并证明始终符合条件即可完成验证。其他类型的形式化验证包括计算鲁棒性和稳定性系数,如幅值裕度和相位裕度。

$x = \text{abs}(y)$

verify this
is positive

code inspection shows
this is true

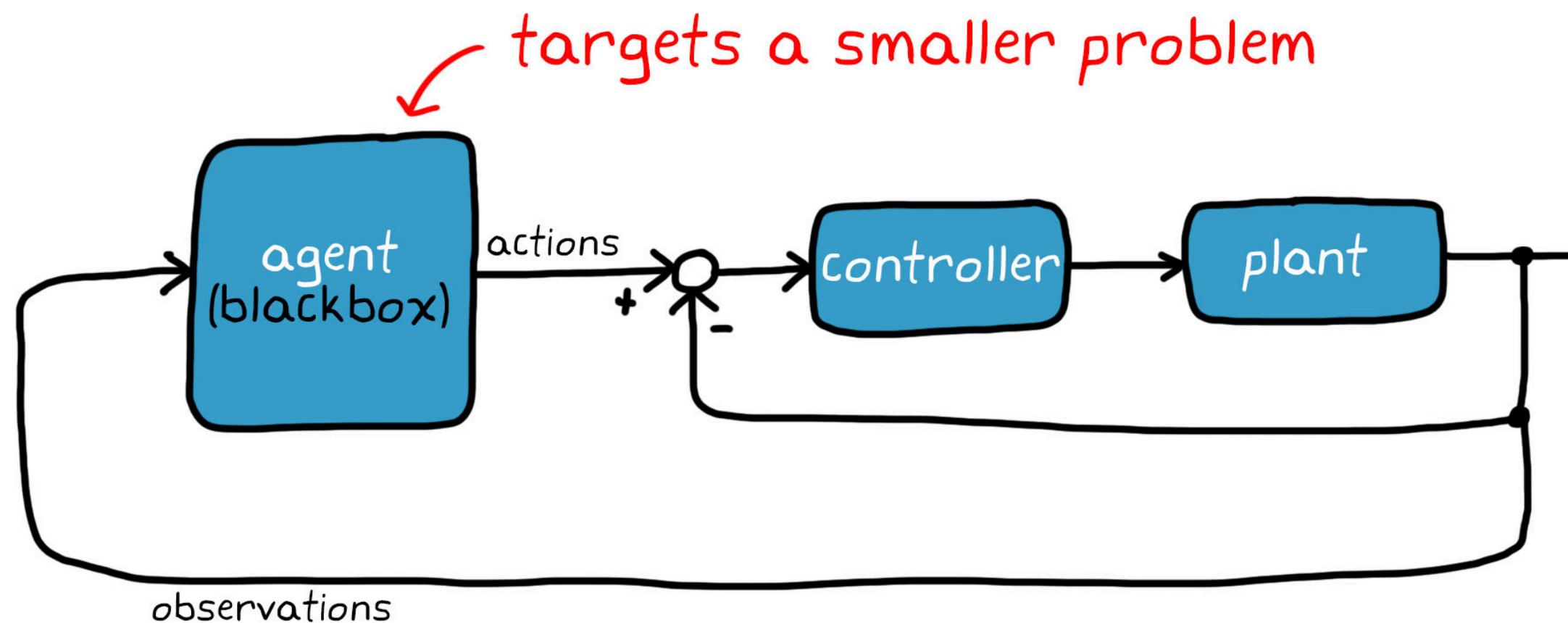
对于神经网络,这种形式化验证难度更大。我们说过,要检查代码并对其行为做出任何保证是很难的。也没有方法可以确定其鲁棒性或稳定性。一切都归结于一个事实:无法从内部解释函数。



缩小问题范围

为了缩小这些问题的范围，限制强化学习智能体的作用域是一个好主意。不再学习输入最高级观测值并输出最低级动作指令的策略，我们可以使传统控制器围绕在 RL 智能体四周，确保它只解决某一类专业问题。通过让一个强化学习智能体只解决一个小问题，我们将无法解释的黑盒缩小为难以通过传统方法求解的系统部分。

策略越小会越专注，因此更易于理解其操作，对整个系统的影响也有限，而且训练时间会相应缩短。但是，缩小策略并不能解决您面临的问题；只是降低了问题复杂度。您仍然不知道它对不确定性是否具有鲁棒性、是否可以保证稳定性，或者可否验证系统是否符合规范。



问题的替代方法

即使无法量化鲁棒性、稳定性和安全性，也可以在设计中使用替代方法来解决这些问题。

为提高鲁棒性和稳定性，每次运行仿真时均可在调整过重要环境参数的环境中训练智能体。

例如，如果在每个片段开始时为步行机器人选择不同的最大扭矩值，则策略最终将收敛到对制造公差具有鲁棒性的参数。按此方法调整所有重要参数将有助于最终得到整体可靠的设计。您可能无法要求特定幅值裕度和相位裕度，但信心将大大增强，确信结果在工作状态空间内的适用范围更大。

episode	torque	length	delay	reference	...
1	2 Nm	1 cm	10 ms	step	.
2	2.5 Nm	1.3 cm	8 ms	ramp	.
3	2.1 Nm	1.7 cm	14 ms	impulse	.
.
.
.

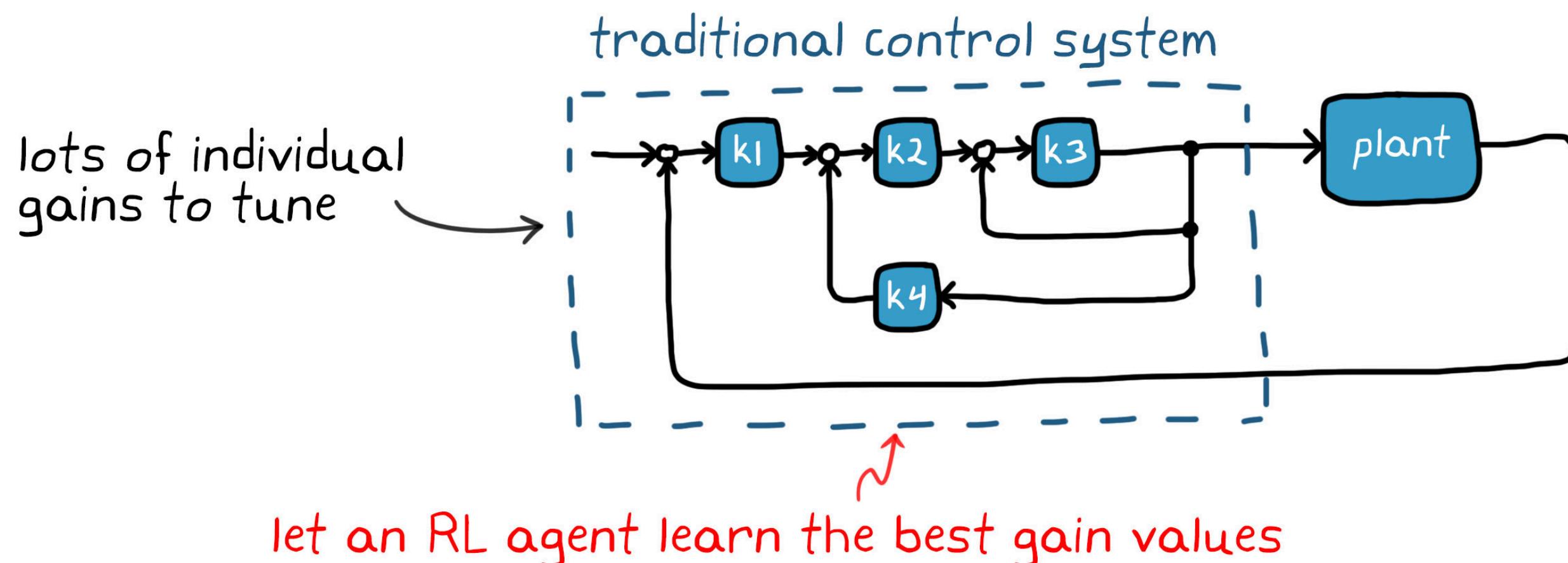
通过设定系统无论如何都要避免的情况，您可以提高安全性，并在策略之外构建软件，以监控此类情况。如果触发该监控器，则可在造成损坏之前限制系统或者接管系统，并将其设置为某种安全模式。

这不会防止您部署危险策略但将保护系统，让您可以借此了解故障过程、调整奖励及训练环境排除此类故障。

```
% software monitor  
  
if abs(body_angle) > 45; % monitor for falling  
    mode = "safe"; % set safe mode  
    extend_arms(); % prepare for impact  
end
```

解决不同问题

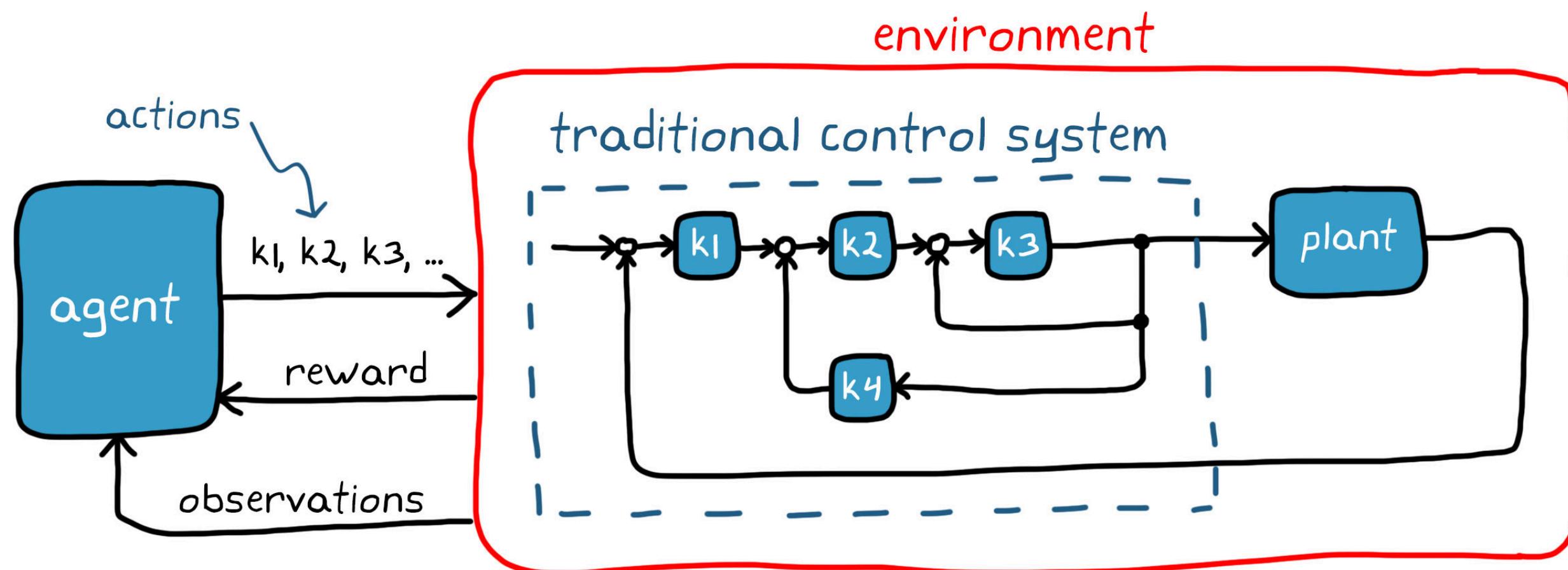
替代方法很好，但也可以干脆通过解决另一个问题来直接解决问题。您可以将强化学习作为工具，用于优化基于传统架构控制系统的控制器增益。想象一下设计包含数十个嵌套环路和控制器的架构，每个嵌套环路和控制器都有若干增益。最终可能会遇到一种情况：需要调整一百个乃至更多的单独增益值。您可以设置 RL 智能体，同时学习所有增益的最佳值，而不是尝试手动调整每一个增益。



RL 对传统方法进行补充

想象一下由控制系统和被控对象构成的环境。奖励是指系统运行效果以及实现该效果需要付出的努力，动作是指系统增益。每个片段结束后，学习算法会调整神经网络，确保增益朝提高奖励的方向移动（即改善表现并减少学习量）。

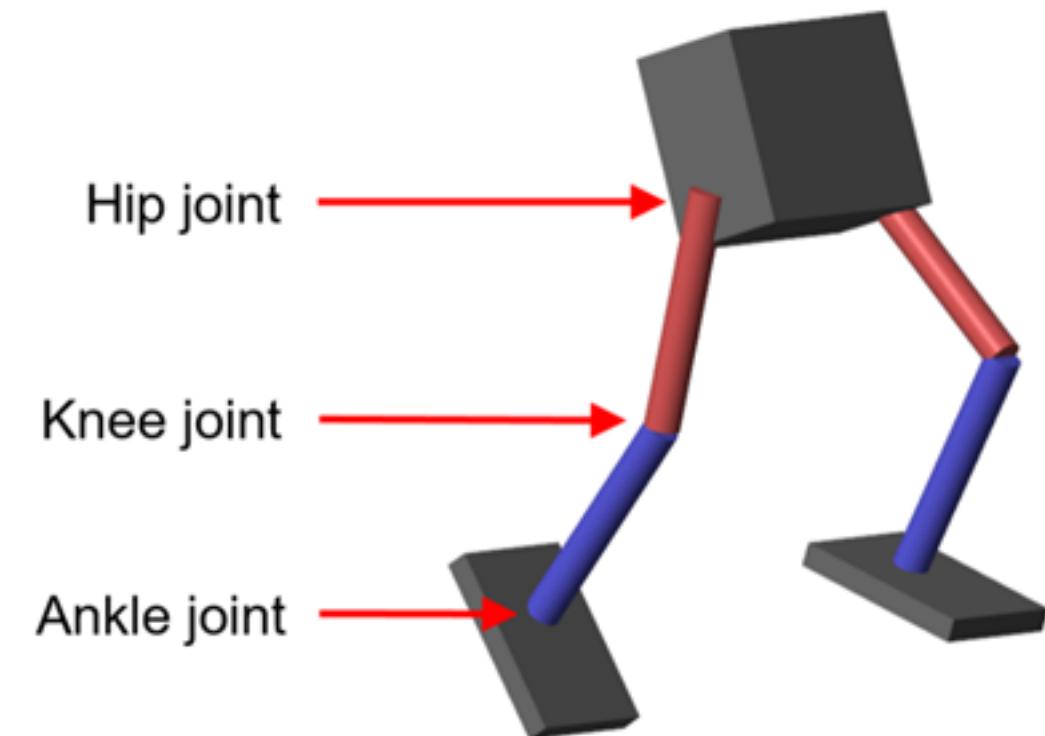
采用这种方法，则可以两全其美。无需部署和验证任何神经网络，也不必担心被迫进行更改；只需将最终静态增益值编码到控制器中。这样，您仍然采用传统架构系统，也就是可以在硬件上进行验证和手动调整，但其中填入了使用强化学习优选出的增益值。



强化学习的未来

强化学习是解决各种难题的强大工具。对于如何理解解决方案及验证解决方案是否可行,还存在一些挑战,但正如我们所说,现在可以通过几种方法应对这些挑战。尽管强化学习远远没有发挥全部潜力,但是可能不久就会成为所有复杂控制系统的设计方法。

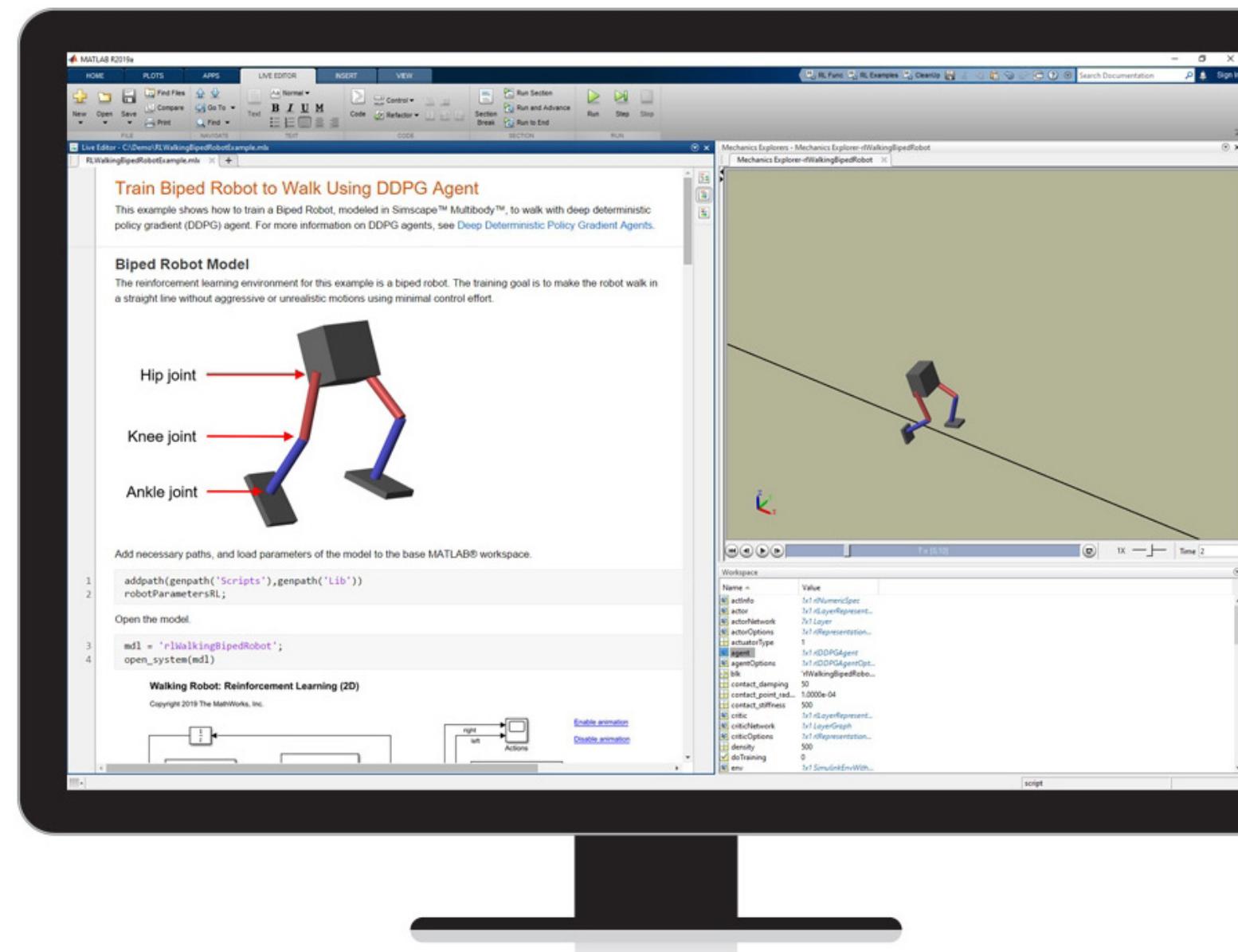
学习算法、强化学习设计工具(如 MATLAB 和 Reinforcement Learning ToolboxTM)及验证方法一直在不断进步。



使用 MATLAB 进行强化学习

Reinforcement Learning Toolbox 为使用强化学习算法进行策略训练提供了一些函数和模块。您可以使用这些策略为复杂系统(如机器人和自主系统)实现控制器和决策算法。

借助该工具箱,您可以使用深度神经网络、多项式或查找表来实现策略。然后,通过与 MATLAB 或 Simulink 模型所表示的环境进行交互,训练策略。



了解更多

[在基本网格世界中训练强化学习智能体 - 产品文档](#)
[训练执行器-评价器智能体平衡车摆系统 - 产品文档](#)
[使用 DDPG 智能体训练双足机器人走路 - 产品文档](#)
[强化学习入门 - 代码示例](#)
[强化学习技术讲座 - 视频系列](#)

