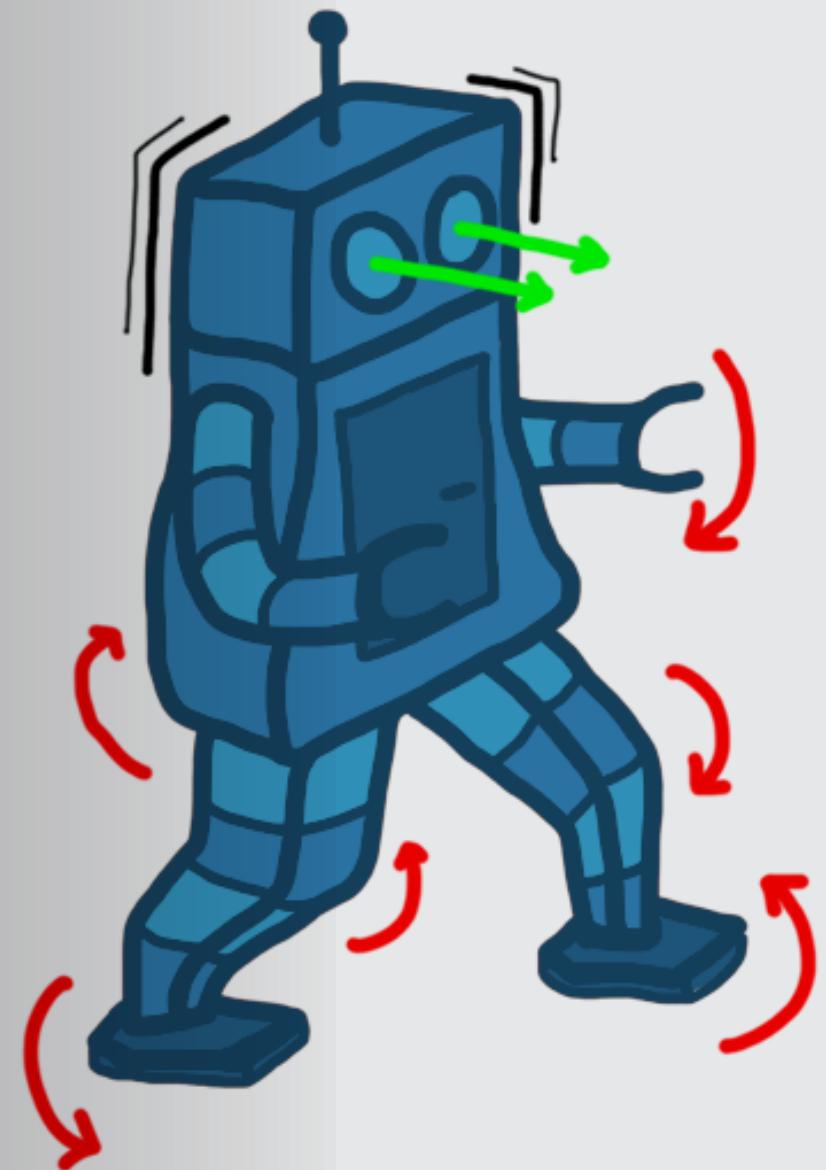


agent

使用 MATLAB 进行强化学习

environment



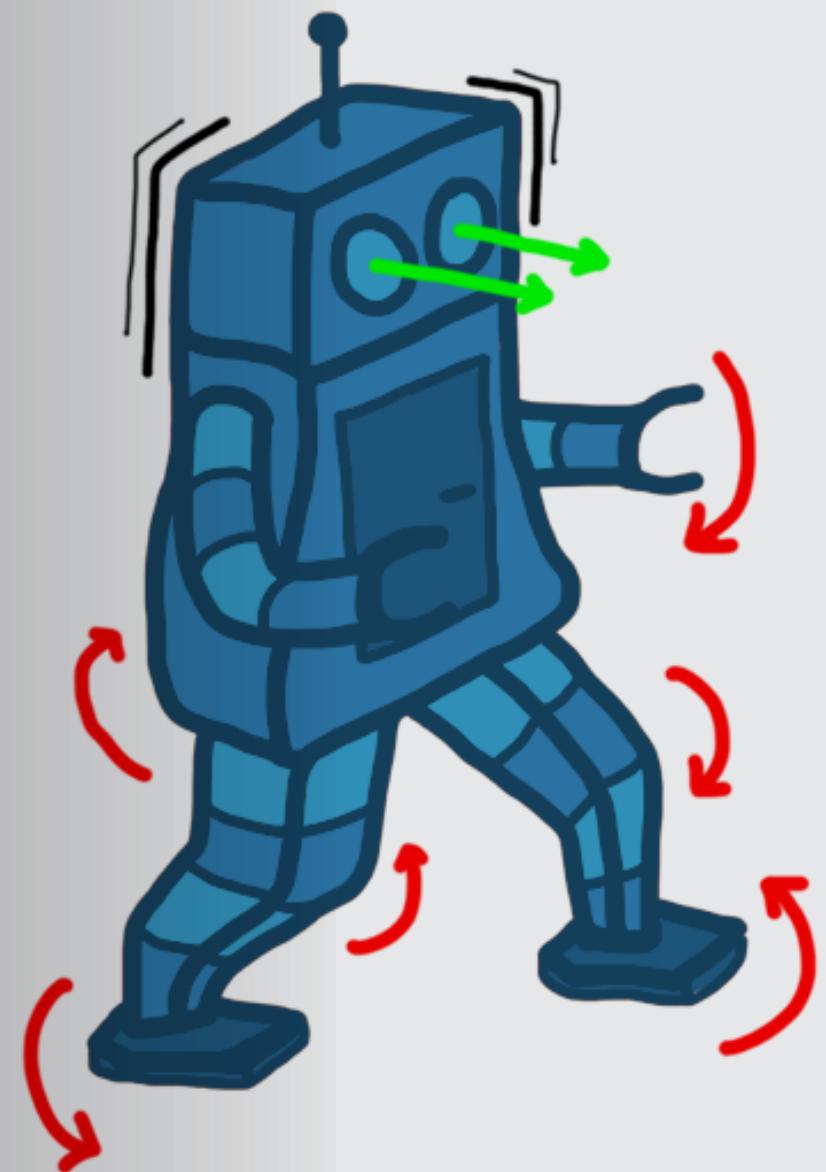
目录

1. 了解基础知识并设置环境
2. 了解奖励和策略结构
3. 了解训练和部署

agent

第 1 部分: 了解基础知识并设置环境

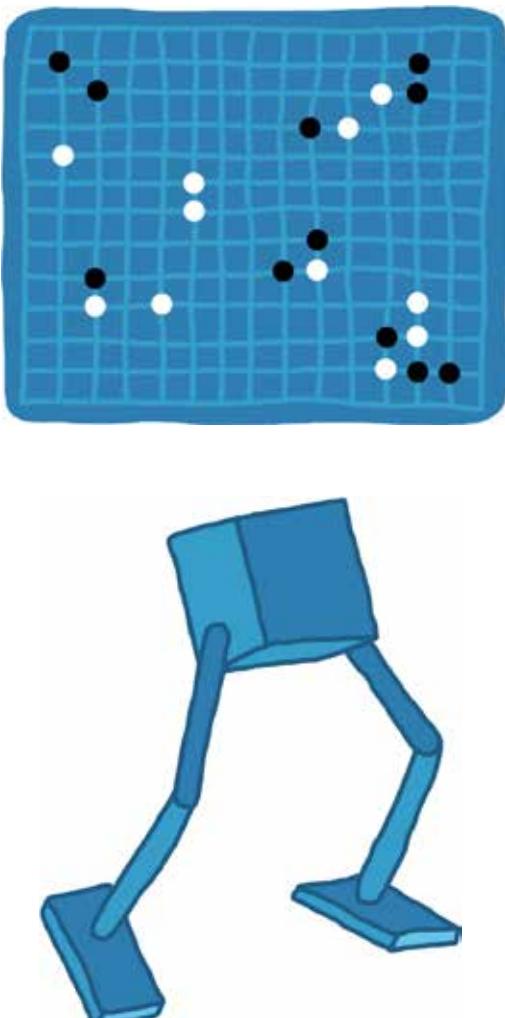
environment



什么是强化学习？

“强化学习旨在学习如何做，即如何根据情况采取动作，从而实现数值奖励信号最大化。学习者不会接到动作指令，而是必须自行尝试去发现回报最高的动作方案。”

—Sutton and Barto, [强化学习:简介](#)



强化学习 (RL) 已成功地训练计算机程序在游戏中击败全球最厉害的人类玩家。

在状态和动作空间较大、环境信息不完善并且短期动作的长期回报不确定的游戏中，这些程序可以找出最佳动作。

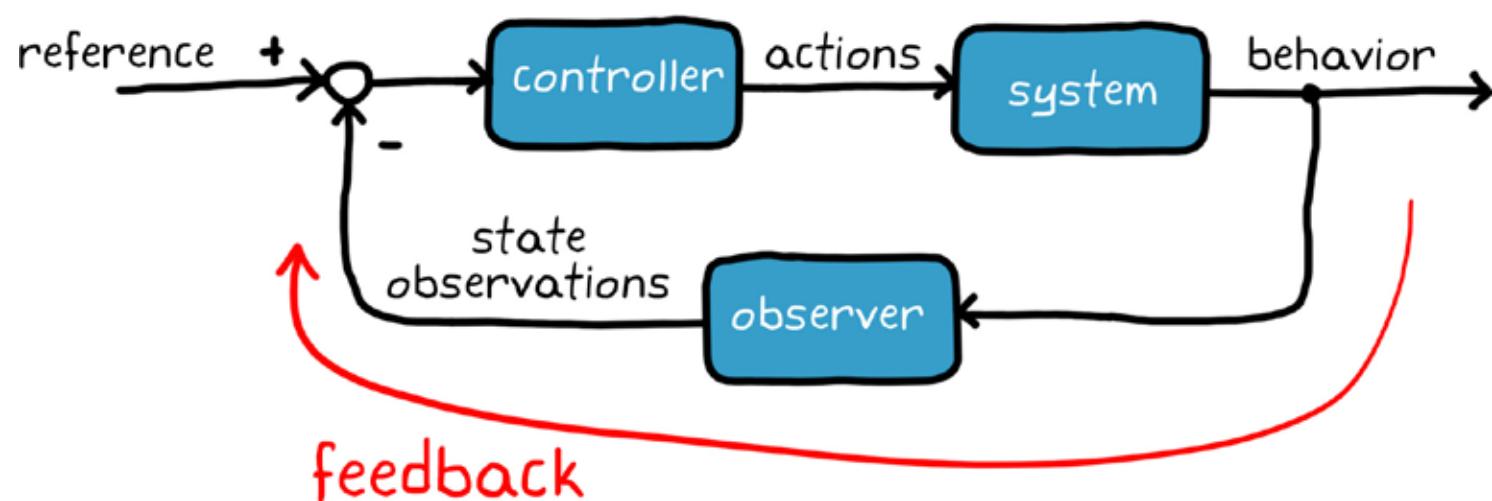
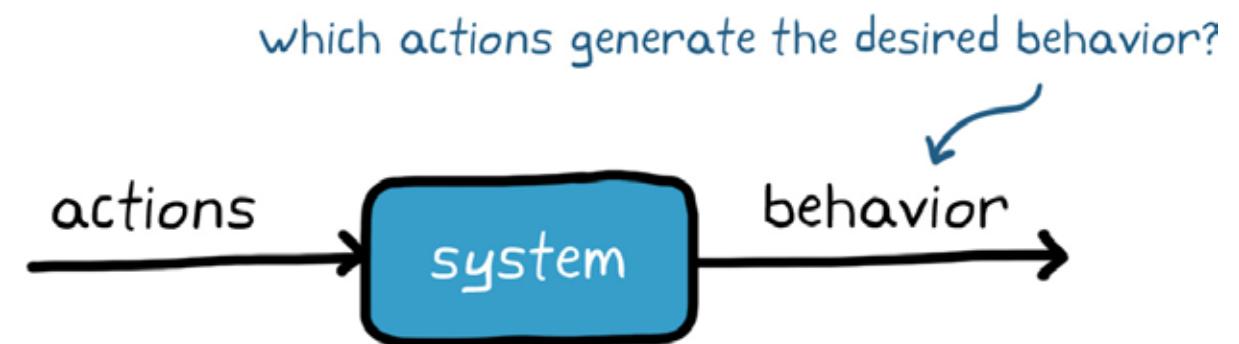
在为真实系统设计控制器的过程中，工程师面临同样的挑战。另外，强化学习能否帮助解决复杂的控制问题，例如训练机器人走路或驾驶自动驾驶汽车？

本电子书通过在传统控制问题的语境下解读什么是强化学习，帮助您了解如何设置和解决 RL 问题。



控制目标

从广义上而言,控制系统的目地是确定生成期望的系统行为的正确系统输入(动作)。



在反馈控制系统中,控制器使用状态观测提高性能并修正随机干扰。工程师运用反馈信号,以及描述被控对象和环境的模型,设计控制器,从而满足系统需求。

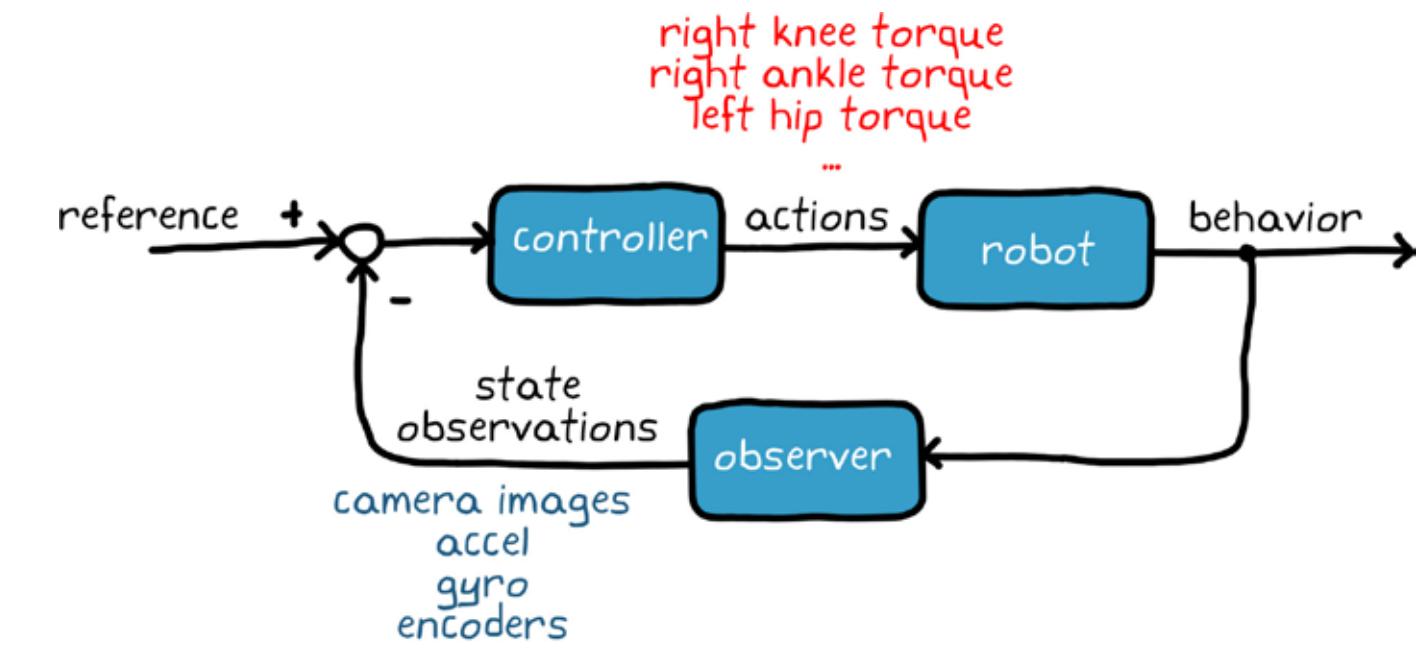
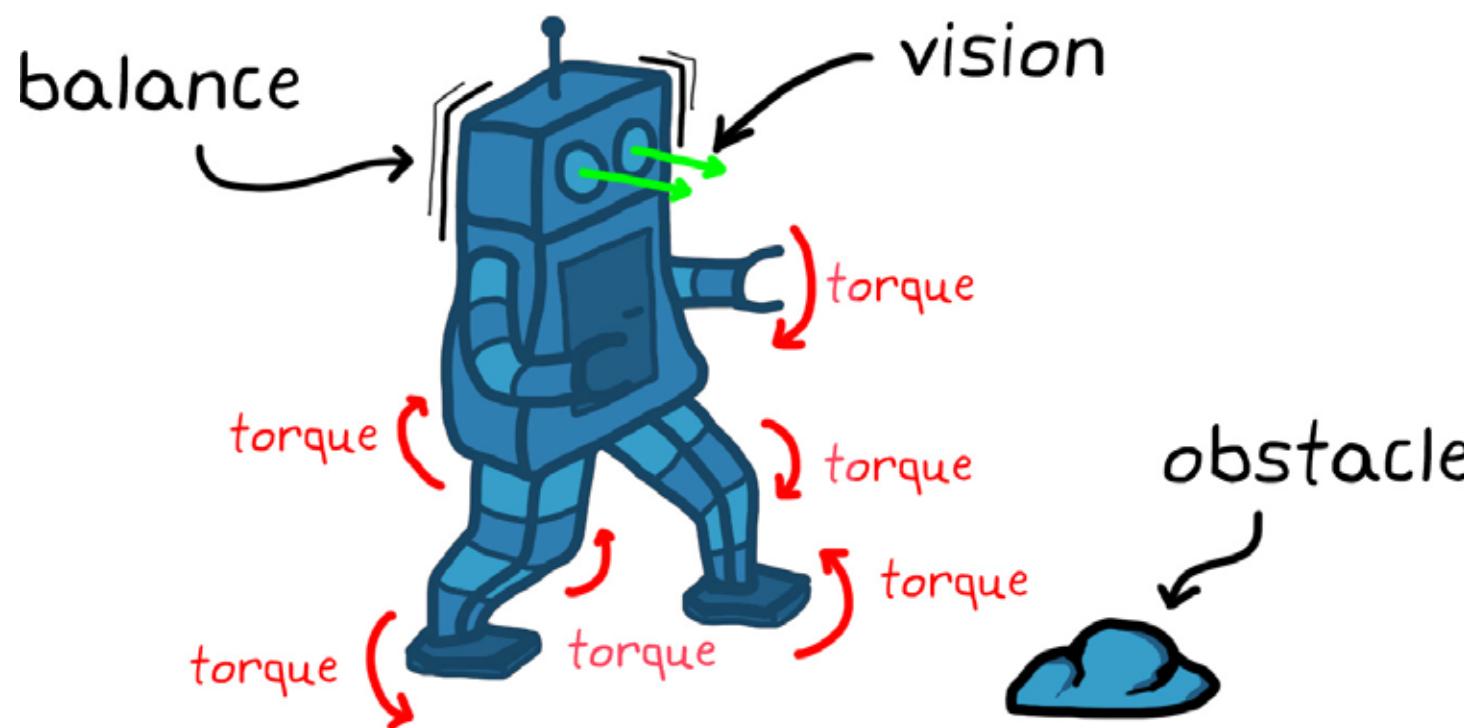
以上概念表述十分简单;然而,倘若系统难以建模、高度非线性或者状态和动作空间较大,则很难实现控制目标。

控制问题

为了理解此类难题对控制设计问题造成进一步后果，不妨设想一下开发步行机器人控制系统的场景。

要控制机器人（即系统），可能需要指挥数十台电机操控四肢的各个关节。

每一项命令是一个可执行的动作。系统状态观测量有多种来源，包括摄像机视觉传感器、加速度计、陀螺仪及各电机的编码器。



控制器必须满足多项要求：

- 确定适当的电机扭矩组合，确保机器人正常步行并保持躯体平衡。
- 在需要避开多种随机障碍物的环境下操作。
- 抗干扰，如阵风。

控制系统设计不仅要满足上述要求，还需满足其他附加条件，比如在陡峭的山坡或冰块上行走时保持平衡。

控制方案

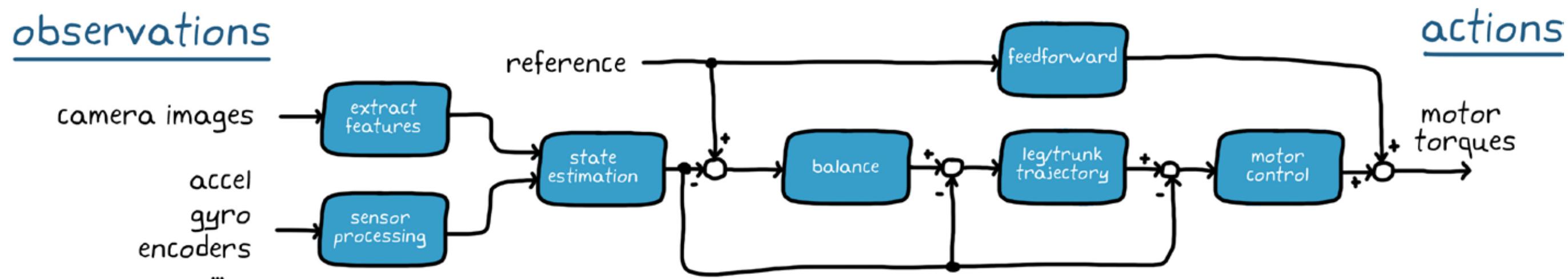
通常,解决此类问题的最佳方法是将问题分解成为若干部分,逐个击破。

例如,您可以构建一个提取摄像机图像特征的流程。比方说,障碍物的位置和类型,或者机器人在全局参照系中所处的位置。综合运用这些状态与其他传感器传回的处理后的观测值,完成全状态估测。

估算的状态值和参考值将馈送至控制器,其中很可能包含多个嵌套控制回路。外部环路负责管理高级机器人行为(如保持平衡),内部环路用于管理低级行为和各个作动器。

所有问题都解决了吗?那可未必。

各环路之间相互交互,使得设计和调优变得异常困难。同时,确定最佳的环路构造和问题分解也并不轻松。

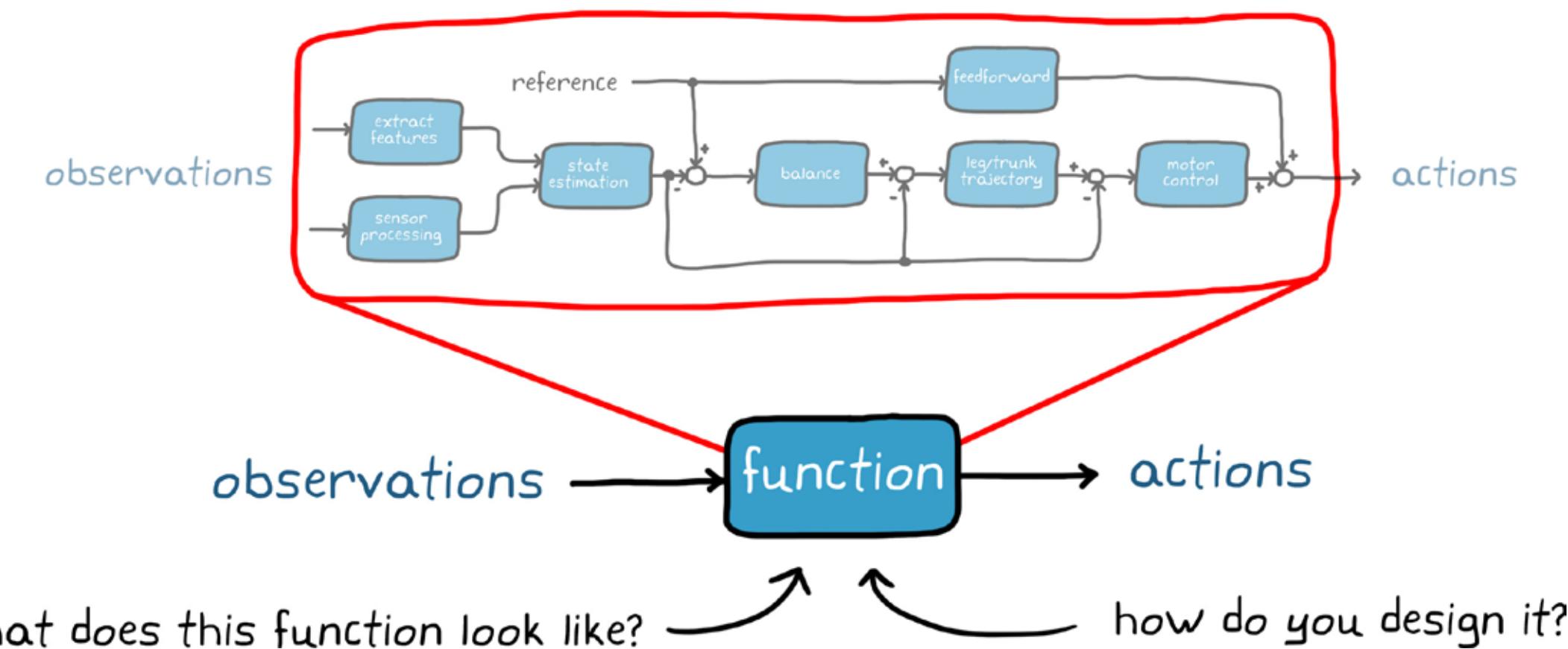


强化学习的魅力

不是尝试单独设计每一个组件，而是设想一下将其全部塞进一个函数里，由该函数负责接收所有观察结果并直接输出低级动作。

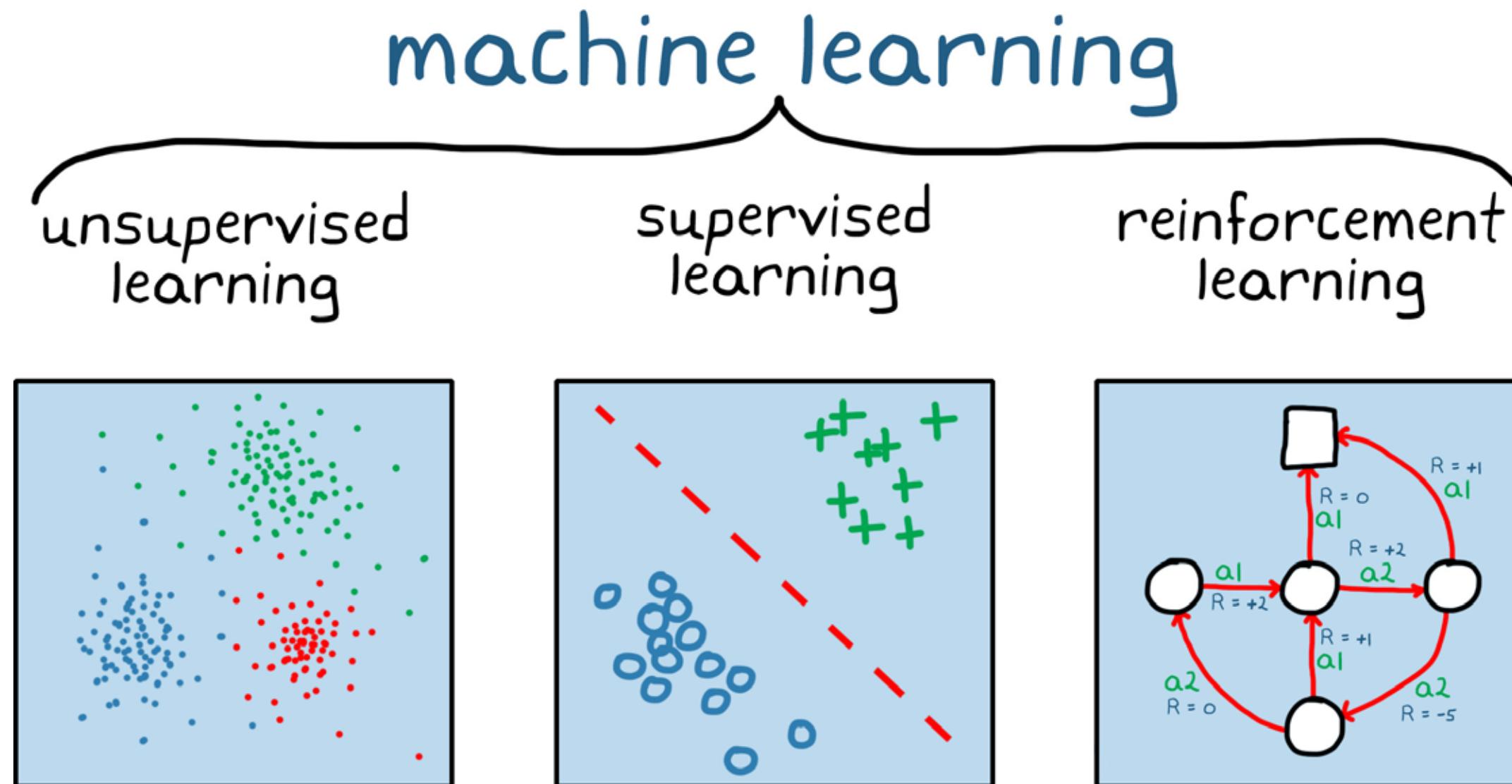
毋庸置疑，这可以简化系统方块图，但这个函数会是怎样的结构？你该如何设计这个函数呢？

创建一个单一的大函数比构建由分段子组件构成的控制系统，看起来难度要大；不过，强化学习可以助您达成目标。



强化学习：机器学习的子集

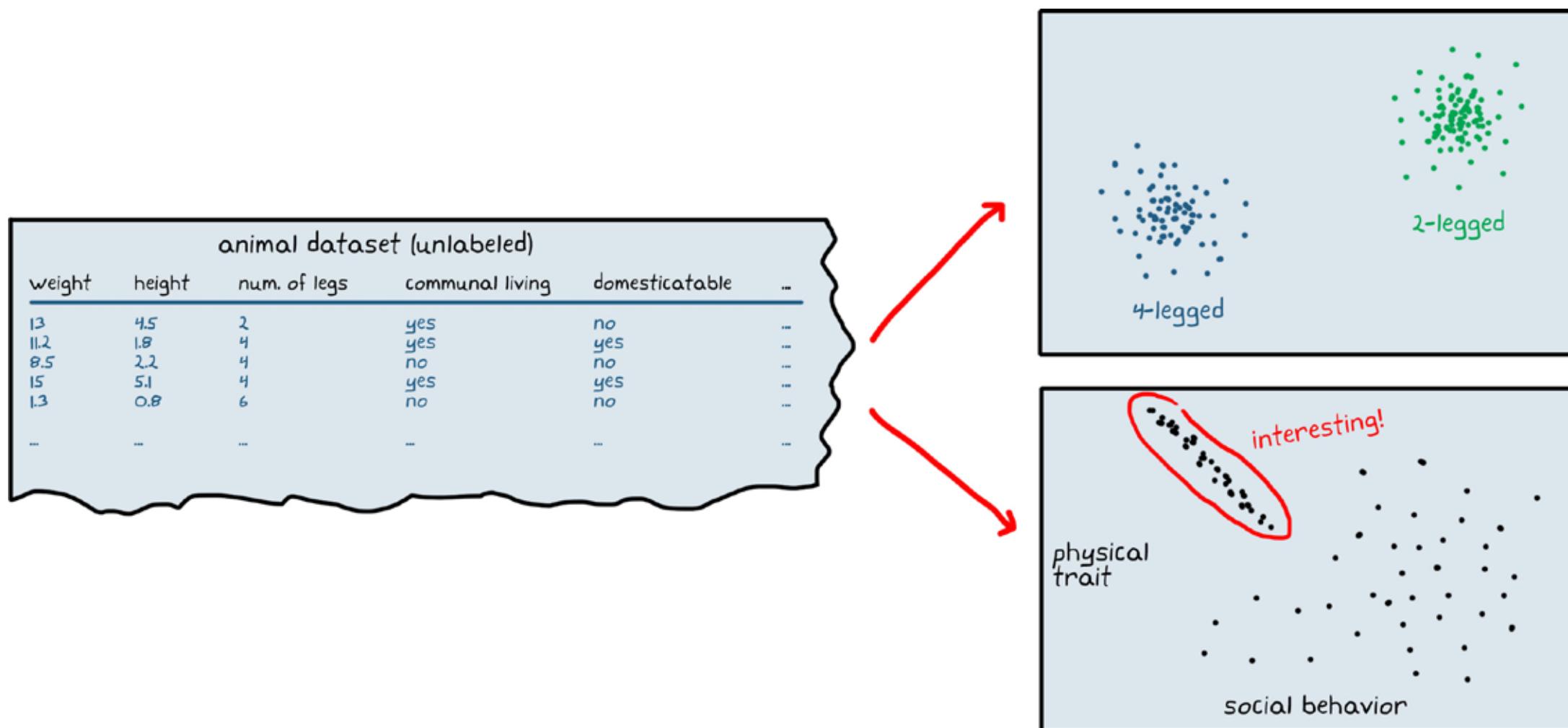
强化学习是机器学习的三个大类之一。无监督学习和监督式学习并不是本电子书关注的重点。但是理解强化学习与这二者的区别是值得的。



机器学习:无监督学习

无监督学习用于确定尚未被分类或标注的数据集的模式或隐藏结构。

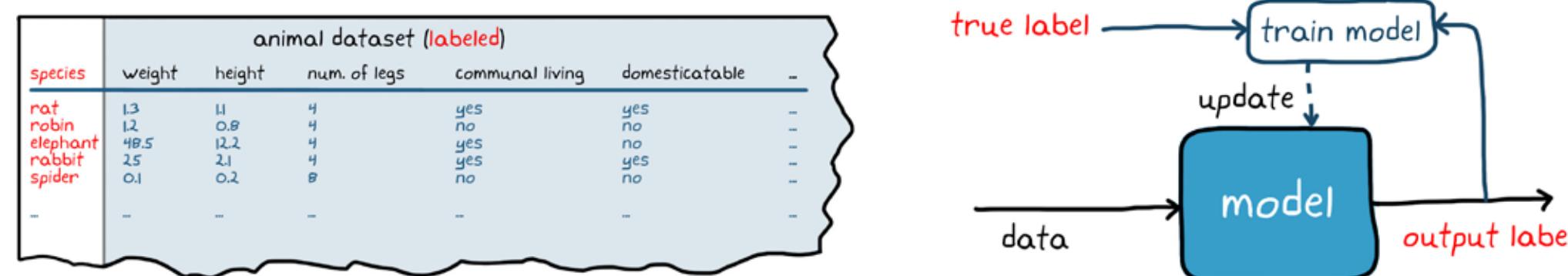
例如,假设您收集了 100,000 种动物的生理特征和社会倾向性信息。您可以使用无监督学习进行动物分组或总结相似特征。可以根据腿数进行分组,也可以根据不太显著的模式进行分组,例如,之前并不知道的生理特性和社会行为之间的关联性。



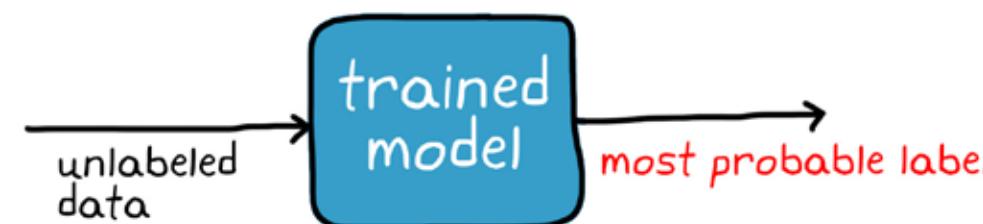
机器学习：监督式学习

您可以使用监督式学习训练计算机为给定输入加上标签。例如，如果动物特征数据集的其中一列是物种，则可以将物种作为标签，其余数据作为数学模型的输入。

您可以使用监督式学习训练模型，使其能够根据每一组动物的特征正确标记数据集。先由模型推断物种，再由机器学习算法系统性地调整模型。



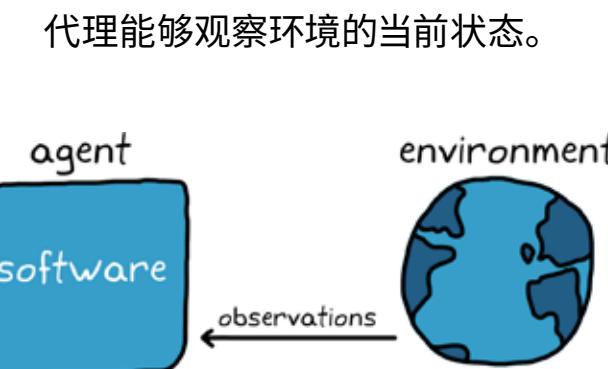
运用足够的训练数据获得可靠的模型后，再输入未标注的新动物的特征，经过训练的模型即能给出对应最有可能的物种标签。



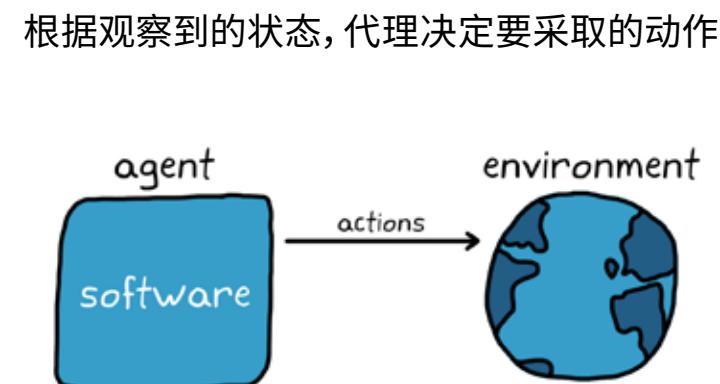
机器学习：强化学习

强化学习是一种截然不同的方法。不同于另外两种采用静态数据集的学习框架，RL 采用动态环境数据。其目标并不是对数据进行分类或标注，而是确定生成最优结果的最佳动作序列。为了解决这个问题，强化学习通过一个软件（即所谓的代理）来探索环境、与环境交互并从环境中学习。

1

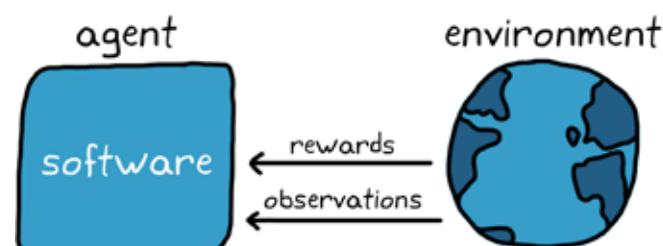


2



3

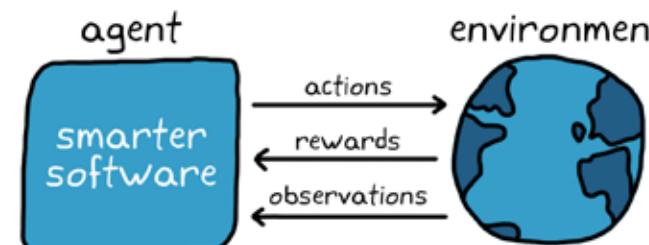
环境会改变状态并针对该动作生成奖励。
代理接收状态变化和动作奖励。



4

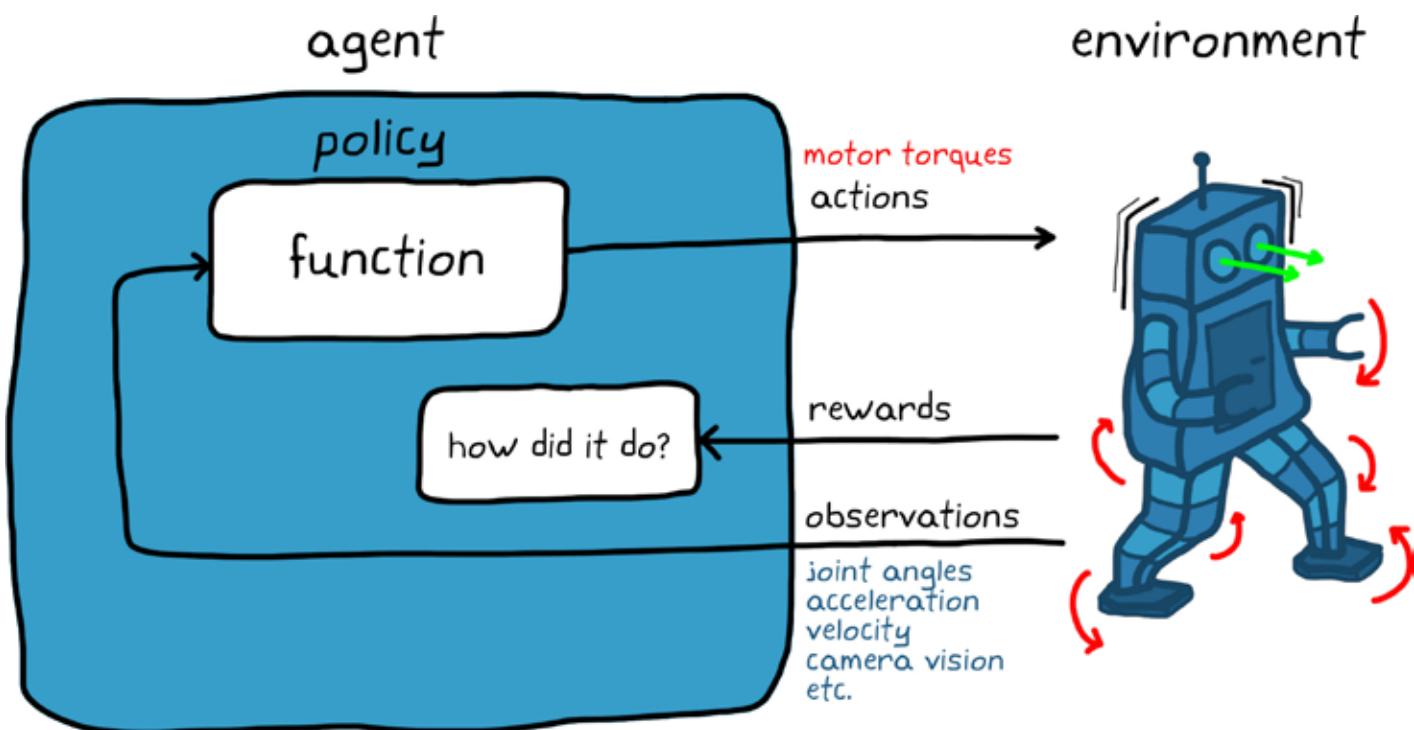
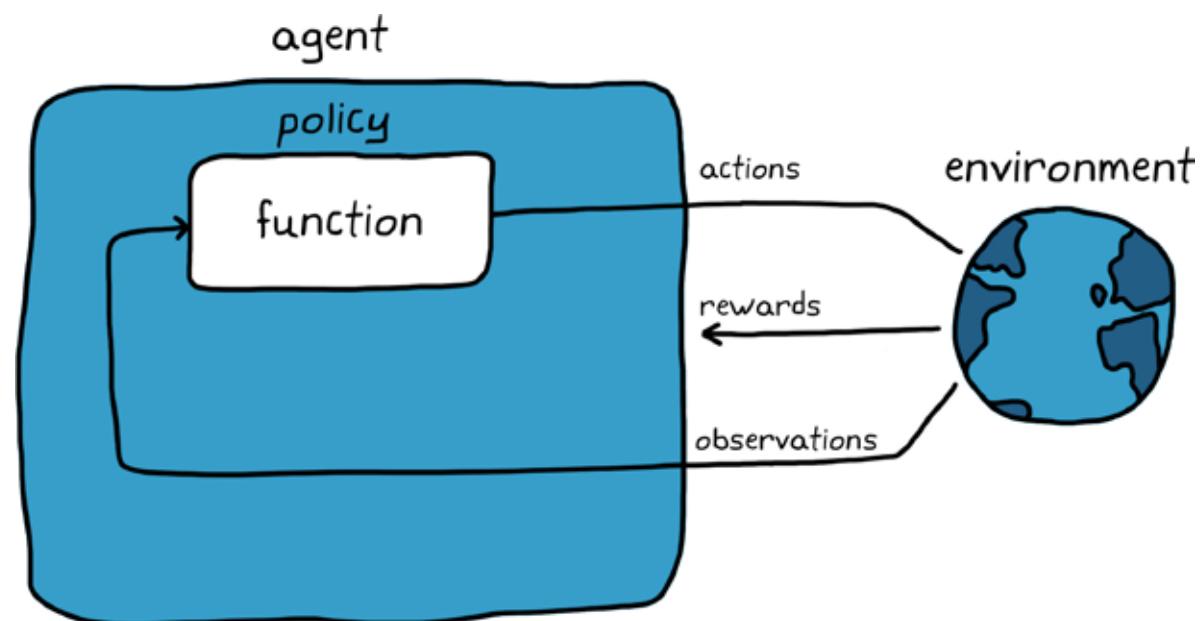
通过这些新信息，代理可以确定该动作是有用且应该重复，还是无益而应该避免。

观察-动作-奖励这个循环会一直持续，直至完成学习。



剖析强化学习

代理中有一个函数可接收状态观测量(输入),并将其映射到动作集(输出)。也就是前面讨论过的单一函数,它将取代控制系统的所有独立子组件。在RL命名法中,此函数称之为策略。策略根据一组给定的观测量决定要采取的动作。



以步行机器人为例,观察结果是指每个关节的角度、机器人躯干的加速度和角速度,以及视觉传感器采集的成千上万个像素点。策略将根据所有这些观测量,输出电机指令,使机器人移动四肢。

接着,环境将生成奖励,向代理反映特定作动器指令组合的效果。如果机器人能够保持直立并继续行走,则对应的奖励将高于机器人摔倒时的奖励。

学习最优策略

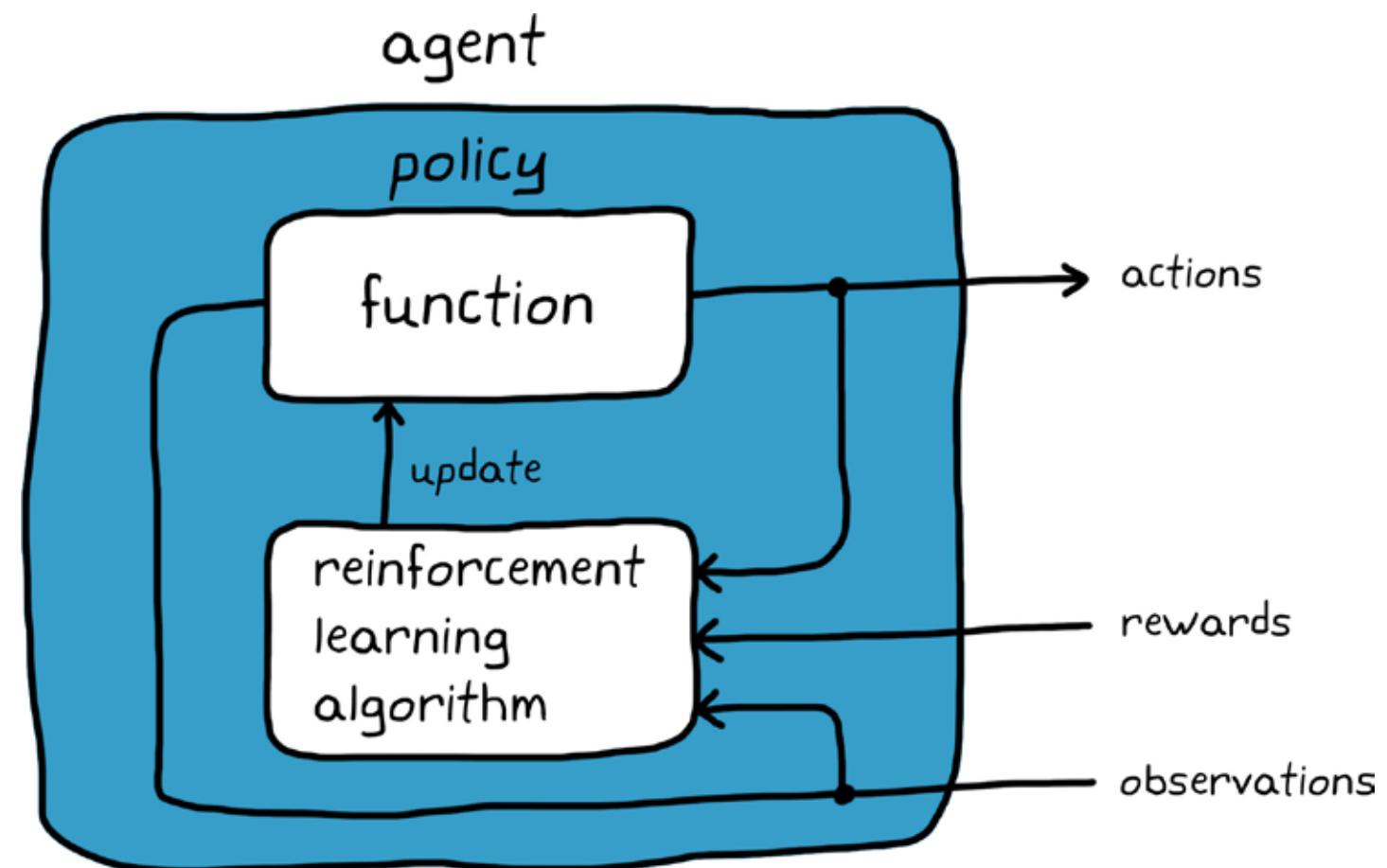
如果可以设计出一项完美的策略,针对观察到的每一种状态向适当的作动器发出适当的指令,那么目标就达成了。

当然,大多数情况下并非如此。即便你真的找到了完美的策略,环境也可能不断变化,因而静态映射不再是最优方案。

正因为如此,强化学习算法应运而生。

它可以根据已采取的动作、环境状态观测量以及获得的奖励值来改变策略。

代理的目标是使用强化学习算法学习最佳环境交互策略;这样一来,无论在任何状态下,代理都能始终采取最优动作—即长期奖励最丰厚的动作。



什么是学习?

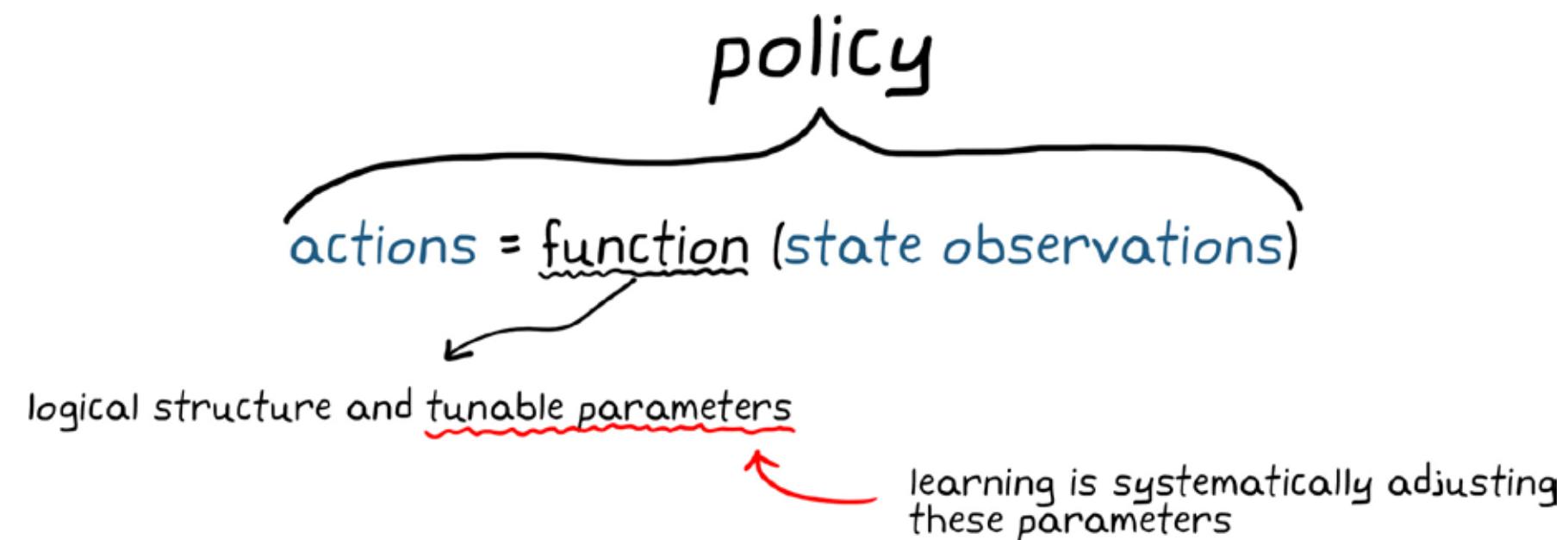
为了理解机器如何学习,请思考一下策略的含义:一个由逻辑和可调参数构成的函数。

倘若已有一套完善的策略结构(逻辑结构),对应有一组参数可生成最优策略,即可产生最丰厚的长期奖励的状态-动作的映射。

学习是指系统性调整这些参数以收敛到最优策略的过程。

这样,您将可以专注于设置适当的策略结构,而无需手动调整函数来获取确切的参数。

您可以让计算机通过稍后将要介绍的流程自行学习参数,但在现阶段,您可以将该流程视为一种复杂的试错过程。



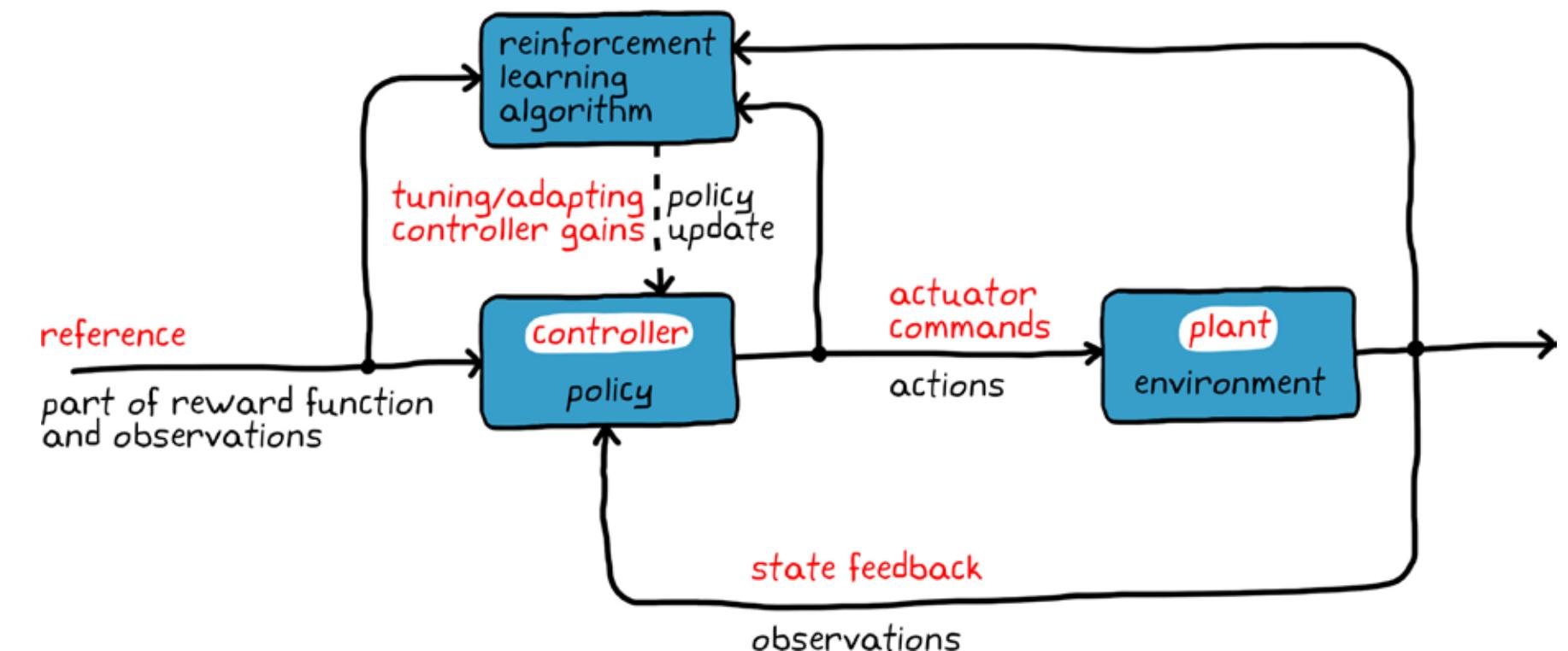
强化学习与传统控制有着怎样的相似之处？

强化学习的目标与控制问题相似；只不过方法不同，使用不同的术语表示相同的概念。

通过这两种方法，您希望确定正确的系统输入，以让系统产生期望的行为。

您的目的在于判断如何设计策略（或控制器），从而将环境（或被控对象）的状态观测量映射到最佳动作（作动器指令）。

状态反馈信号是指环境观察结果，参考信号则内置到奖励函数和环境观测量中。



强化学习工作流程概述

一般来说，强化学习涉及到五个方面。本电子书重点介绍第一个部分：建立环境。本系列的其他电子书将更深入地探索奖励、策略、训练和部署问题。

1

您需要一个环境，供您的代理开展学习。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代理做什么工作，并设计奖励函数，激励代理实现目标。

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，由此构成代理的决策部分。

policy



4

您需要选择一种算法来训练代理，争取找到最优的策略参数。

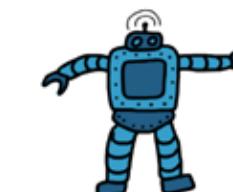
training



5

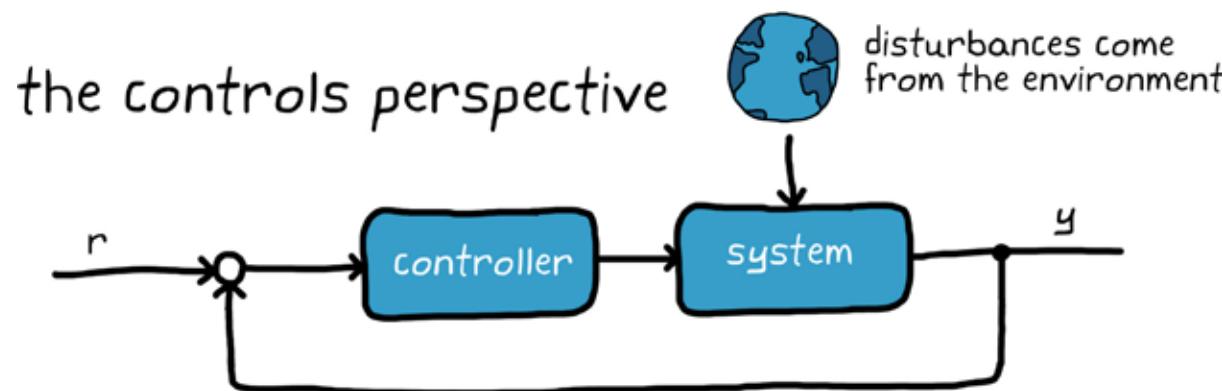
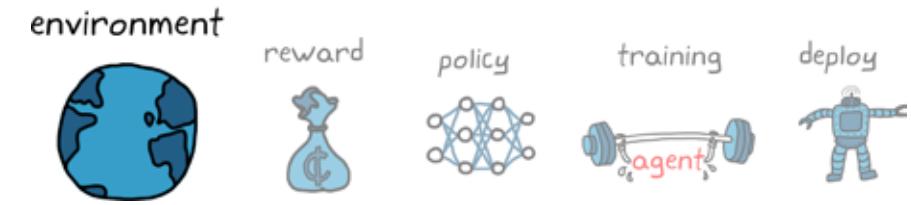
最后，您需要在实地部署该策略并验证结果，从而利用该策略。

deploy



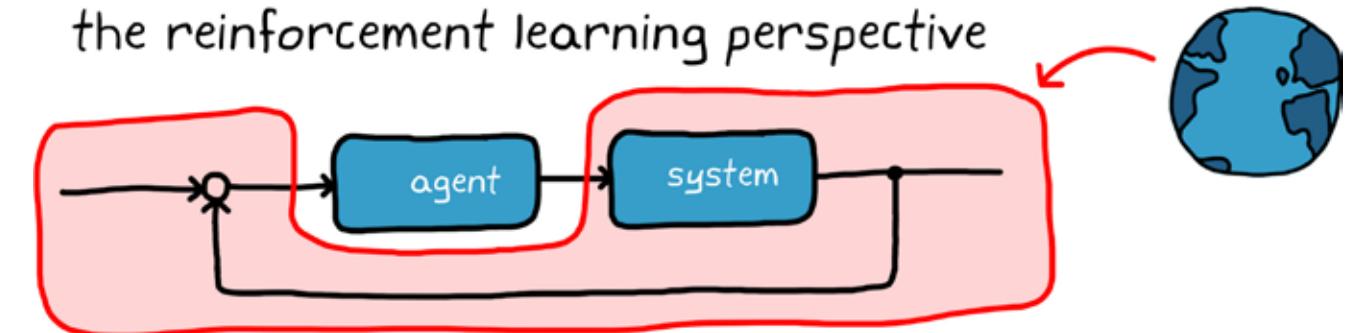
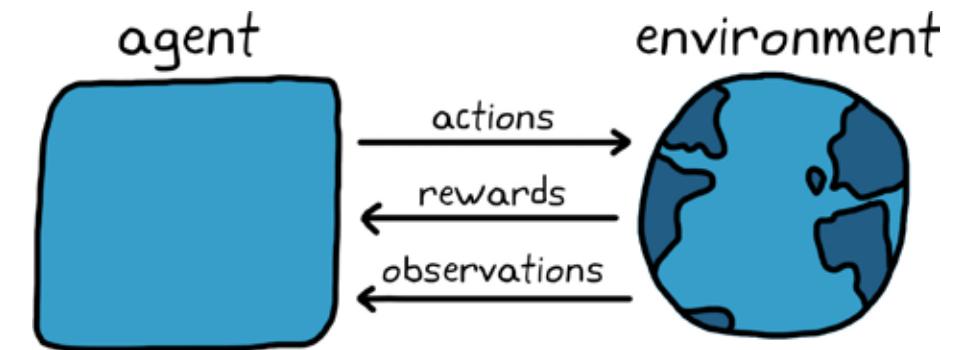
环境

环境是指存在于代理之外的一切元素。它既是代理动作产生作用的地方，又能生成奖励和观测量。



然而，在强化学习的术语中，环境是指除代理以外的一切元素，包括系统动态特性。因此，控制系统的一大部分实际上都属于环境。代理只不过是通过学习生成动作及更新策略的一个软件而已。

从控制的角度而言，这个定义可能令人费解，因为人们普遍将环境视为影响控制系统的干扰。

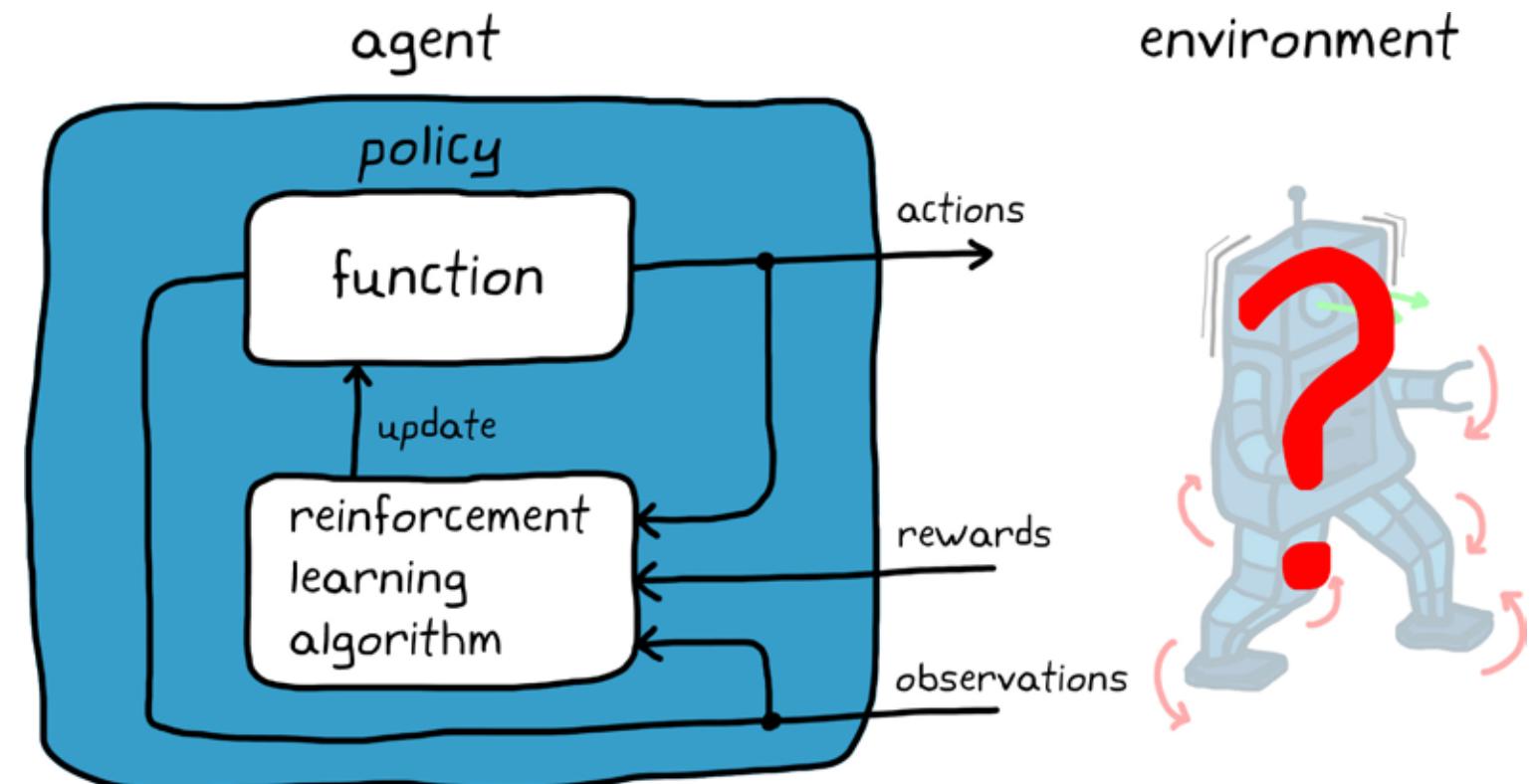
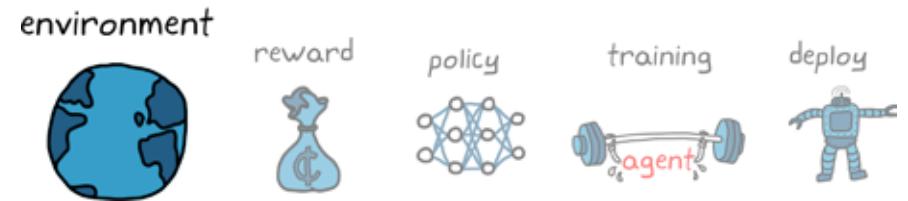


无模型强化学习

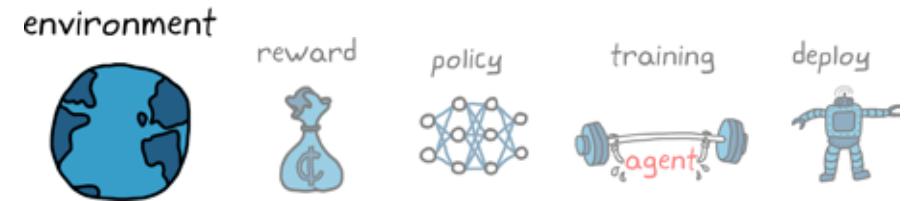
强化学习之所以功能强大,原因之一在于代理不需要对该环境有任何了解,但仍可学习如何与该环境交互。例如,代理不需要了解步行机器人的动力学或运动学原理,不必了解关节移动或附肢长度,却仍能确定如何获得最多的奖励。

这就是所谓的**无模型强化学习**。

在无模型 RL 中,您可以将采用 RL 的代理内置到任何系统,代理将能够学习最优策略。(假设您已给策略访问观测量、奖励、动作及足够的内部状态的权限。)

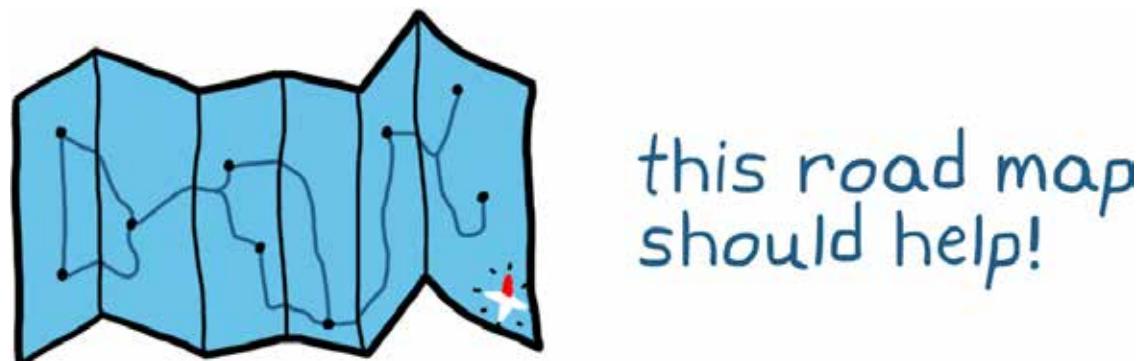


基于模型的强化学习



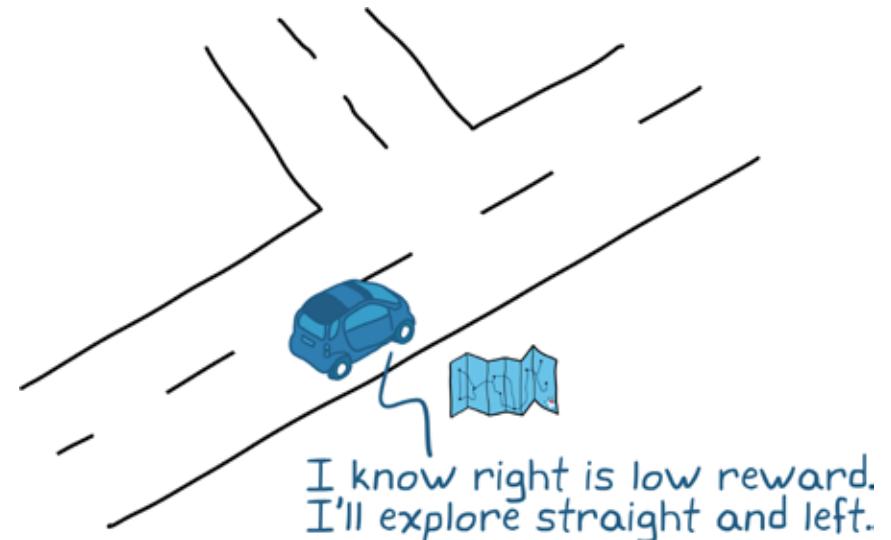
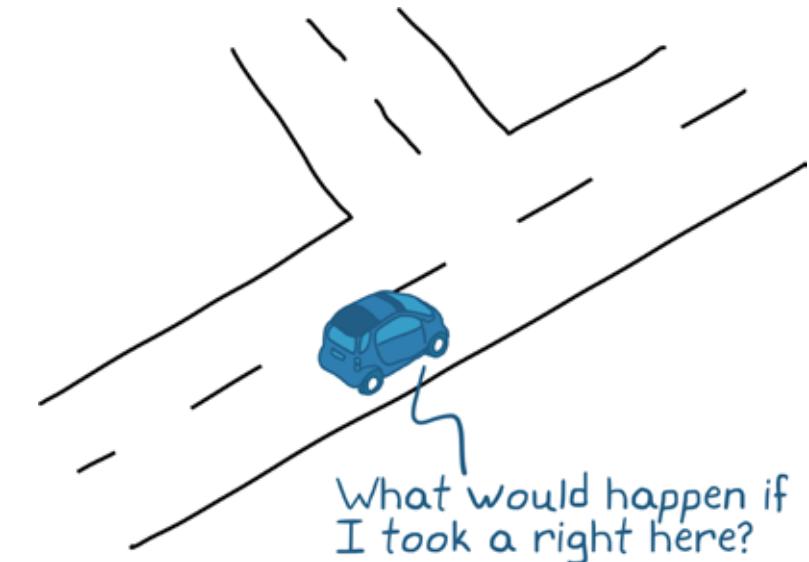
无模型 RL 面临一个问题：如果代理不了解环境，那么必须探索状态空间的所有区域，以确定如何获得最多的奖励。

这意味着，代理需要在学习过程中投入一些时间探索低奖励区域。



但是，您可能已经知道，状态空间的某些区域不值得探索。通过提供整个环境或部分环境的模型，将已知的信息提供给代理。

代理可以使用模型探索环境的某些部分，而无需采取实际的动作。模型可以补足学习过程，使其避开已知的无益区域，而集中探索剩余部分。



无模型与基于模型

基于模型的强化学习可以缩短学习最优策略所需的时间,因为您能够使用模型指导代理远离已知的低奖励状态空间区域。

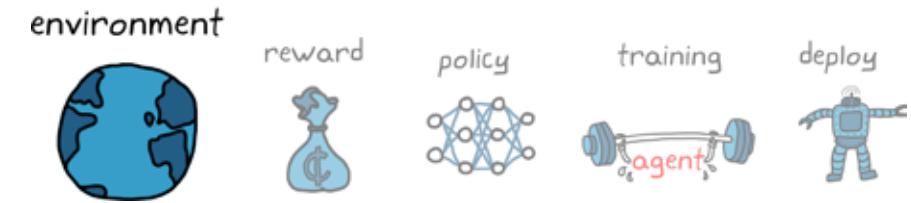
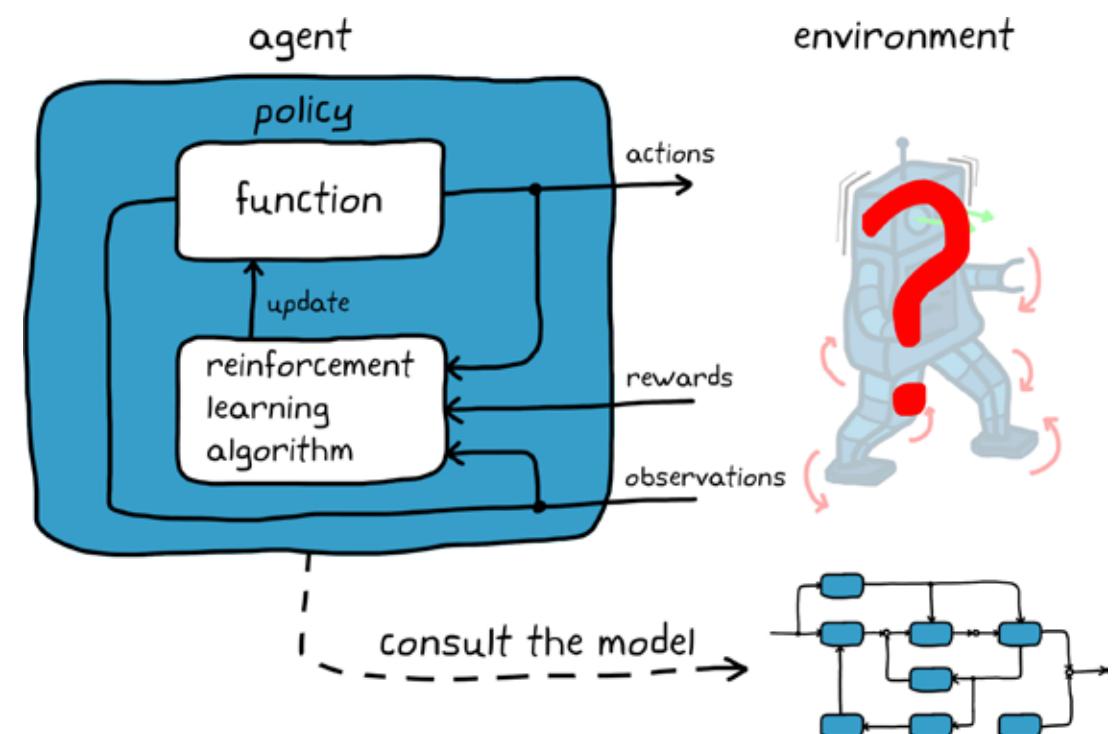
首先,您不希望代理进入这些低奖励状态,所以无需浪费时间学习低奖励状态下的最佳动作。

在基于模型的强化学习中,不需要了解整个环境模型;只需为代理提供您自己了解的那部分环境。

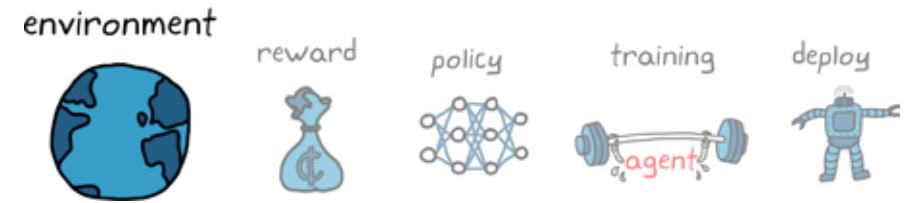
无模型强化学习应用更广泛,本电子书接下来将重点介绍这种方法。

如果您对无模型强化学习的基础知识有所了解,那么继续研究基于模型的 RL 也会更为直观。

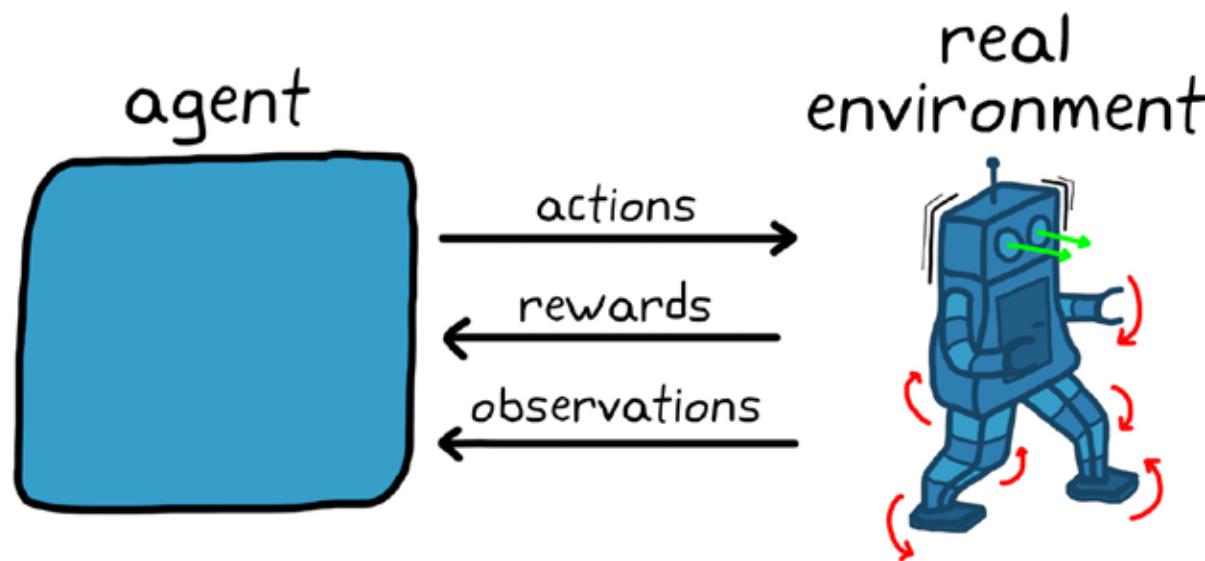
现阶段,无模型 RL 更受欢迎,因为人们希望通过它来解决一些难以开发模型(甚至是简单模型)的问题。例如,通过像素观测来控制汽车或机器人。在大多数情况下,像素强度与汽车或机器人动作之间的关系并不明显。



真实环境与仿真环境



鉴于代理通过与环境交互来开展学习，您需要设法使代理与环境进行实际交互。可以采用真实物理环境，也可以通过仿真，需根据具体情况加以选择。

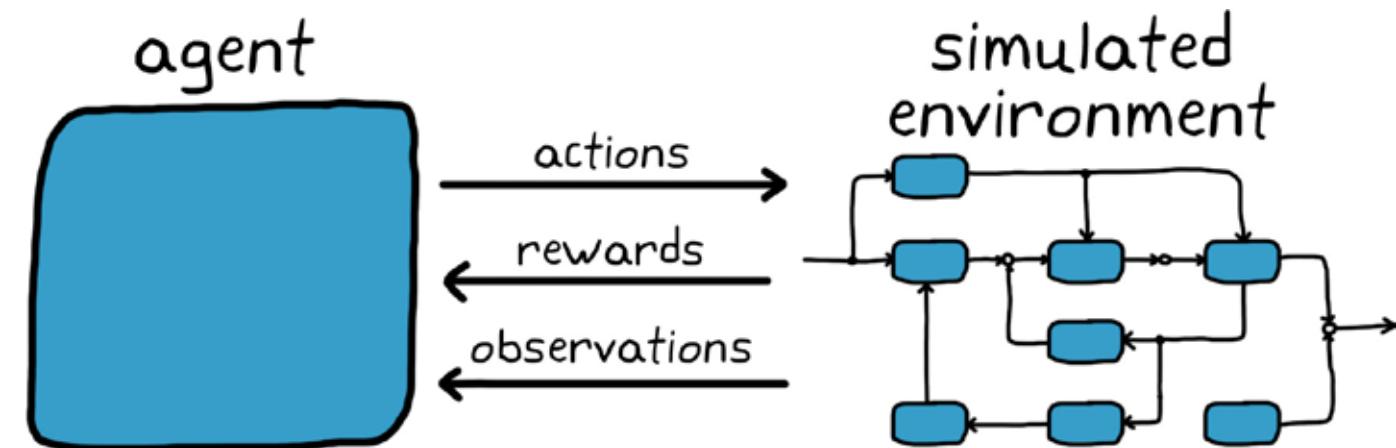


真实

准确性：没有什么能比真实环境更全面地反映环境状态了。

简便性：无需花时间创建和验证模型。

必要性：如果真实环境不断变化或难以准确建模，可能需要根据该真实环境进行训练。



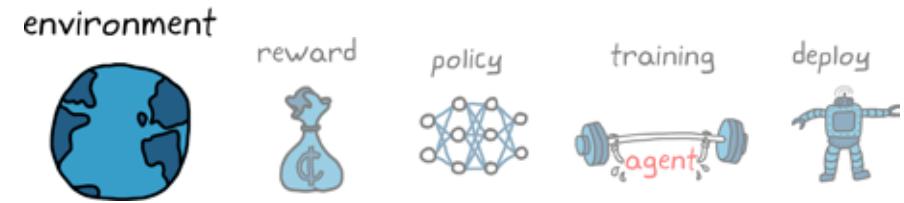
仿真

高速性：仿真的运行速度可能高于实时环境或可并行化，从而可以加快缓慢的学习过程。

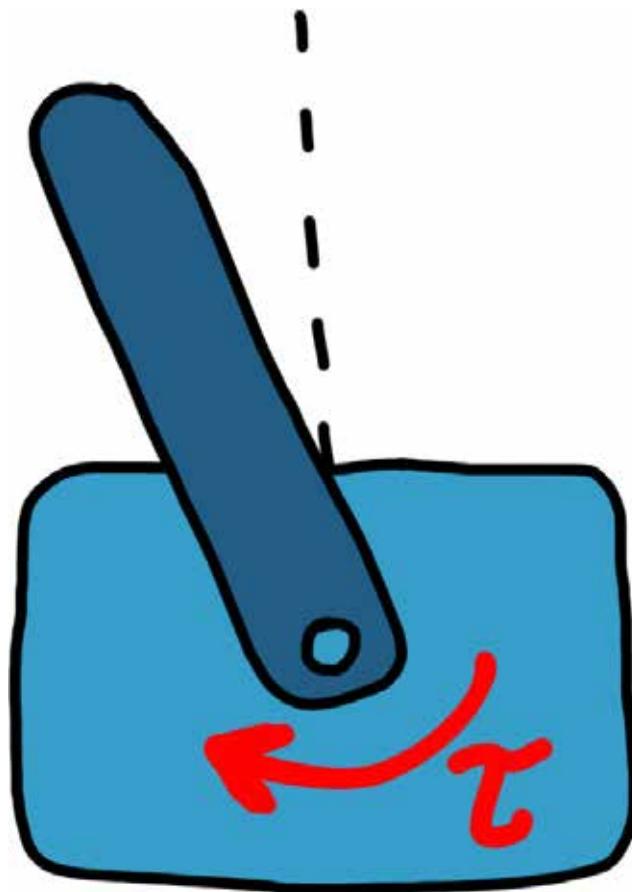
仿真条件：对于难以测试的情况，建模更容易。

安全性：不存在损坏硬件的风险。

真实环境与仿真环境



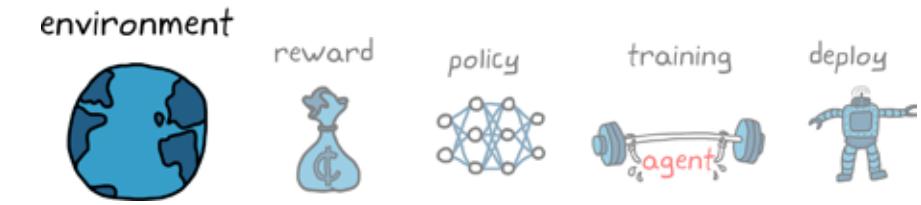
例如，您可以通过运行真实的物理系统，让代理学习如何平衡倒立摆。这应该是个不错的解决方案，因为硬件损坏的情况不大可能发生。鉴于状态和动作空间相对较小，训练大概不会花费太长时间。



但若是训练步行机器人，这种方法可能并不那么奏效。如果开始训练时策略不够理想，机器人会不停地摔倒或踉跄，以至于无法学会移动双腿，更不用说学习如何走路了。这不仅会损坏硬件，而且每次还得扶起机器人，相当耗时。效果不理想。



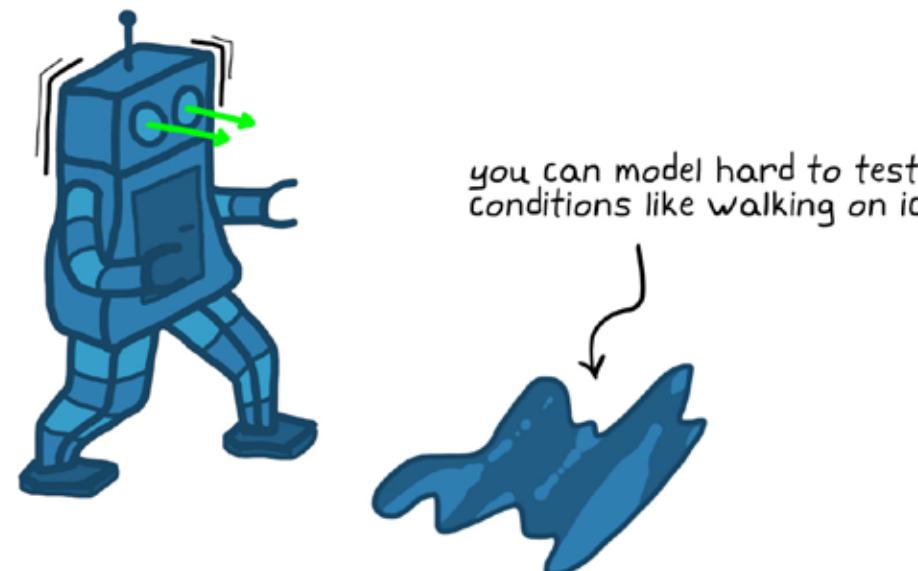
仿真环境的优势



仿真环境是最常用的代理训练方法。这对于控制问题的一大优势在于，通常已经具备良好的系统和环境模型，因为您往往需要系统和环境模型来进行传统控制设计。如果已在 MATLAB® 或 Simulink® 中搭建好模型，您可以将现有控制器替换为强化学习代理，向环境添加奖励函数，然后启动学习过程。

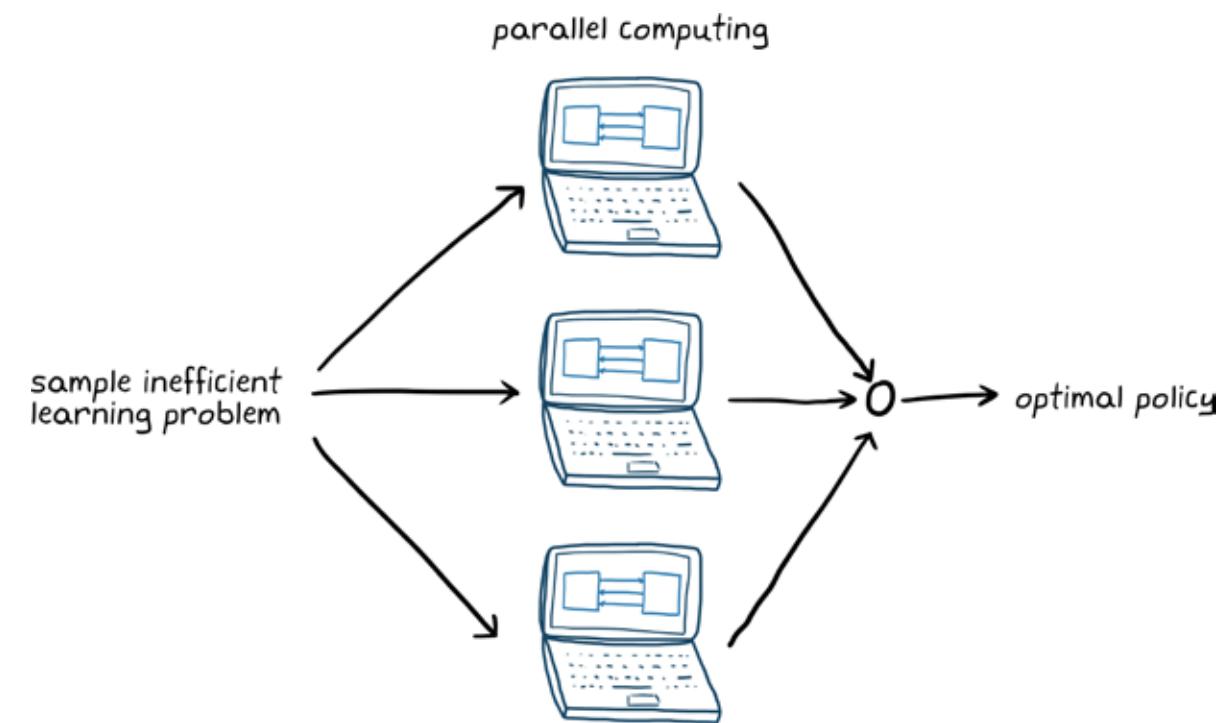
学习过程需要大量样本：试验、误差和修正。从这个意义上而言，学习过程效率极低，因为这期间可能需要经历数千乃至数百万个片段，才能收敛到最优解决方案。

环境模型的运行速度可能比实时环境快，您可以启动大量仿真让其并行运行。这两种方法都可以加快学习过程。



相较于在真实世界中让代理暴露在环境中，您对仿真状况的控制力要大得多。

例如，您的机器人或许必须能够在任意数量的不同表面上行走。通过仿真技术模拟在低摩擦表面（如冰面）上行走比实际冰面测试容易得多。此外，在低摩擦环境中训练代理其实还有助于机器人在各种平面上保持直立姿势。通过仿真可以营造更适宜的训练环境。



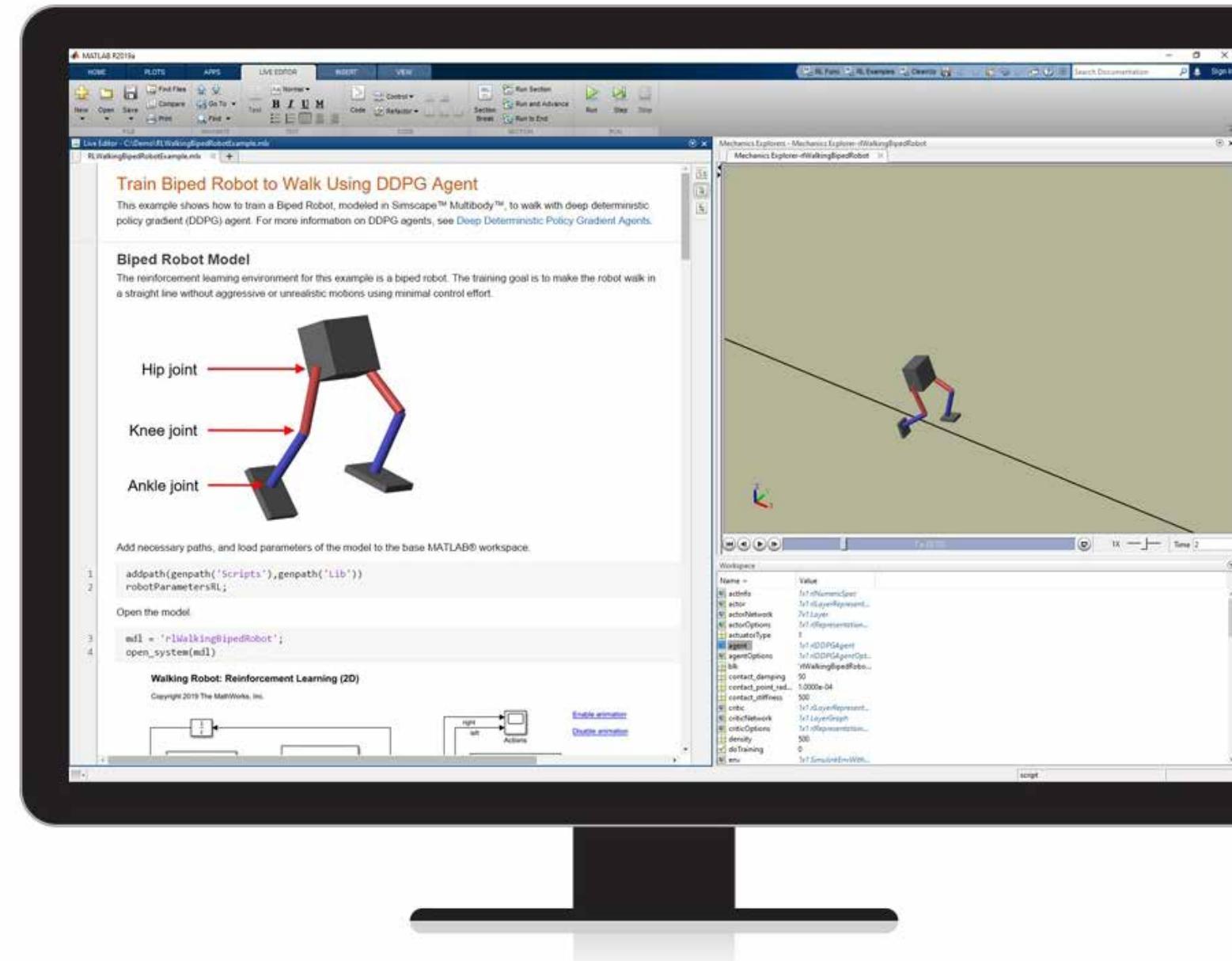
使用 MATLAB 和 Simulink 进行强化学习

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。使用该工具箱,您可以通过让代理与 MATLAB 或 Simulink 模型表征的环境进行交互,来训练策略。

例如,如需在 MATLAB 中定义强化学习环境,您可以使用现成的模板脚本和类,根据应用场合适当修改环境动态特性、奖励、观测量和动作。

在 Simulink 中您可以模拟大量不同的环境,以用于解决控制或强化学习问题。例如,您可以进行车辆动力学和飞行动力学建模;使用 Simscape™ 进行多种物理系统建模;使用 System Identification Toolbox™ 进行基于测量数据的近似动力学建模;对雷达、激光雷达和惯导模块等传感器建模等等。

mathworks.com/products/reinforcement-learning



了解更多 agent

观看

什么是强化学习? (14:05)

了解环境和奖励 (13:27)

对步行机器人进行建模与仿真 (21:19)

深入了解

Reinforcement Learning Toolbox 入门

在 MATLAB 中创建环境

在 Simulink 中创建环境

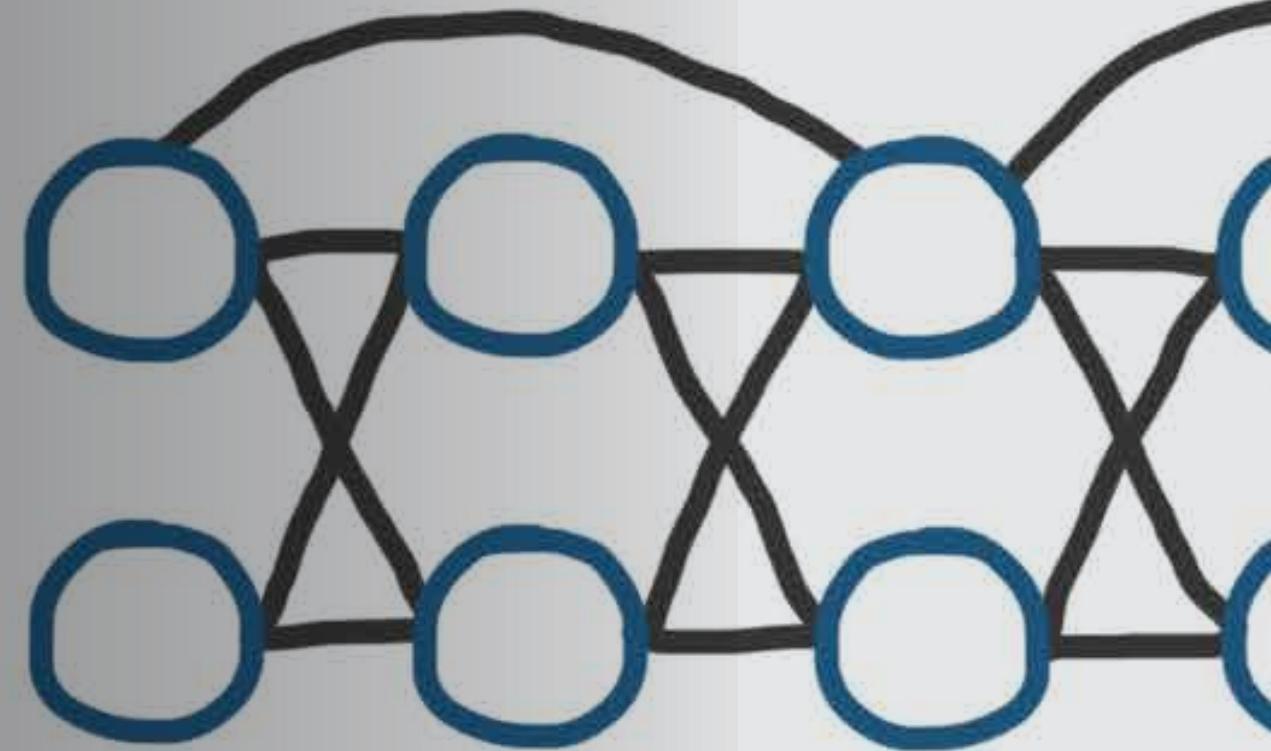
在 Simulink 中进行飞行动力学建模

在 Simulink 中进行整车动力学仿真

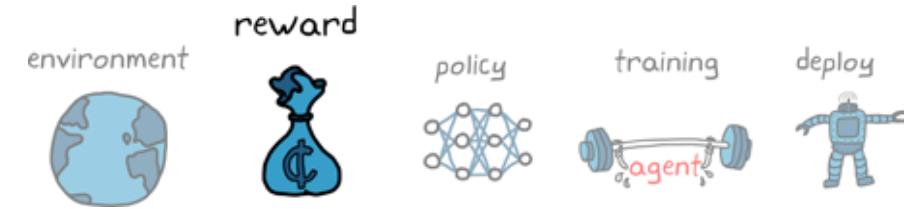
environment



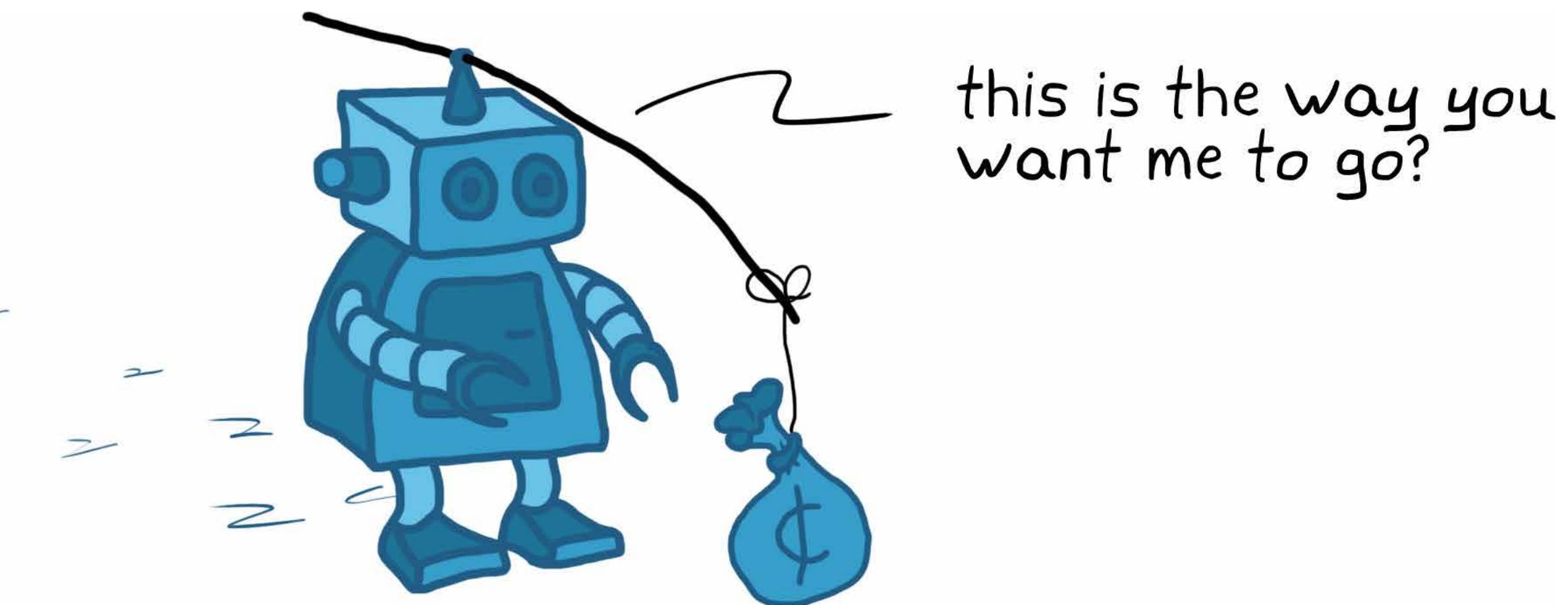
第 2 部分: 了解奖励和策略结构



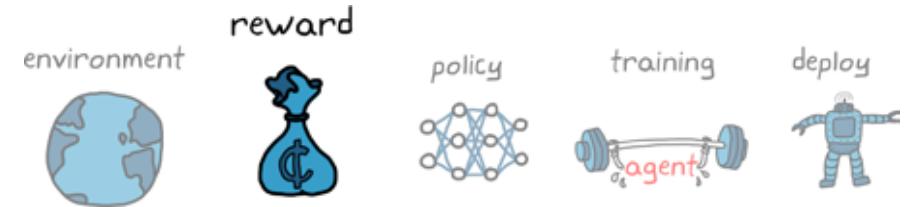
奖励



设置好环境后，下一步是思考您想要代理做什么，以及在它完成后，您会如何进行奖励。
这需要设计一个奖励函数，让学习算法“明白”什么情况下策略变得更好，最终趋向您所寻求的结果。



什么是奖励?



奖励是一个函数，会生成一个标量，代表处于某个特定状态并采取特定动作的代理的“优度”。

reward = function (state, action)
scalar representing "goodness"

这个概念类似于 LQR 中用于惩罚系统性能差和作动器工作量增加的代价函数。当然，区别在于，代价函数试图让值最小化，而奖励函数试图让值最大化。但这也是在解决同一个问题，因为奖励可以看作代价的相反数。

LQR cost function, $J = \int_0^{\infty} (x^T Q x + u^T R u) dt$

performance effort
quadratic

主要区别在于，LQR 中的成本函数为二次方程，而在强化学习 (RL) 中，对于创建奖励函数没有任何限制。您可以采用稀疏奖励，或在每个时间步长后奖励，或者仅在较长一段时间后一个片段完全结束时给予奖励。奖励可以使用非线性函数计算，也可以通过几千个参数来计算。这完全取决于采取什么方式才能有效地训练代理。

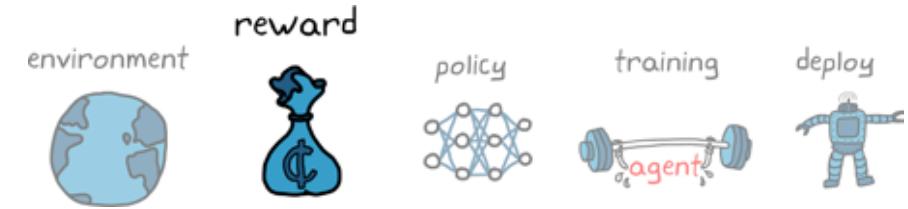
稀疏奖励

既然对于如何创建奖励函数没有限制，您可能会遇到奖励稀疏的情况。这意味着您想要给予激励的目标要在一长串动作后才能实现。步行机器人可能就属于这种情况，您可以进行如下设置：只有当机器人成功行进 10 米后，代理才会得到奖励。因为这是您训练机器人想要达到的最终目的，所以这样设置奖励函数完全合理。

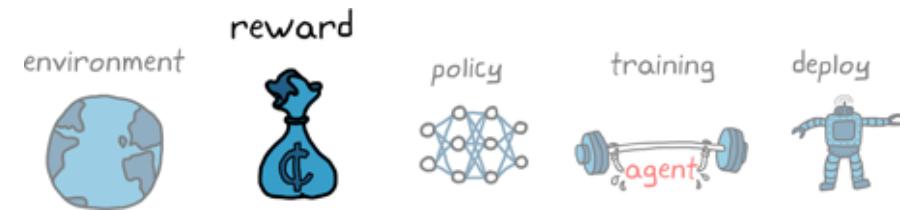


$$\text{reward} = \begin{cases} 1 & \text{for } \text{state} = 10 \text{ meters} \\ 0 & \text{for } \text{state} \approx 10 \text{ meters} \end{cases}$$

稀疏奖励的问题是，代理可能在很长的时期内蹒跚而行，尝试不同的动作，经历许多不同的状态，沿途却没有得到任何奖励，因此，在该过程中没有学到任何东西。代理能够随机生成确切的动作序列以获得稀疏奖励的概率非常小。想像一下生成所有正确的马达指令，让机器人直立行走 10 米，而不是摔倒在地面上，这得需要多好的运气。



奖励重塑



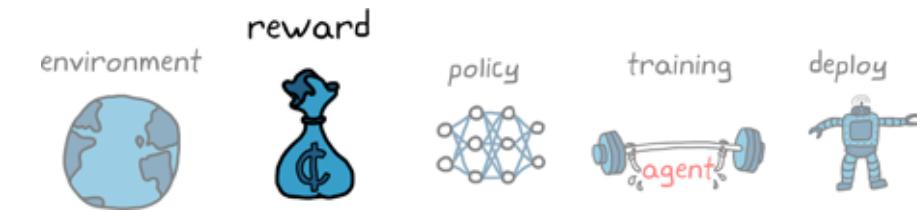
可通过奖励重塑改进稀疏奖励，即提供较小的中间奖励，引导代理沿正确的道路前进。



不过，奖励重塑也有它自己的问题。如果您给优化算法提供一条捷径，算法就会采取这条捷径。而捷径隐藏在奖励函数内，当您开始重塑奖励时，更有可能出现。奖励函数设计得不好可能造成代理收敛到一个不理想的解，即使该解会为代理获得最多的奖励。看上去我们的中间奖励可能引导机器人成功地走向 10 米的目标，但最优解可能不是向那第一个奖励走去。反而可能朝它笨拙地跌倒，收取该奖励，从而强化该行为。除此以外，机器人可能趋向于缓慢地沿地面蠕动，以收取其余的奖励。对代理来说，这是合情合理的高奖励的解，但是对设计者来说，这显然不是首选结果。



特定领域知识

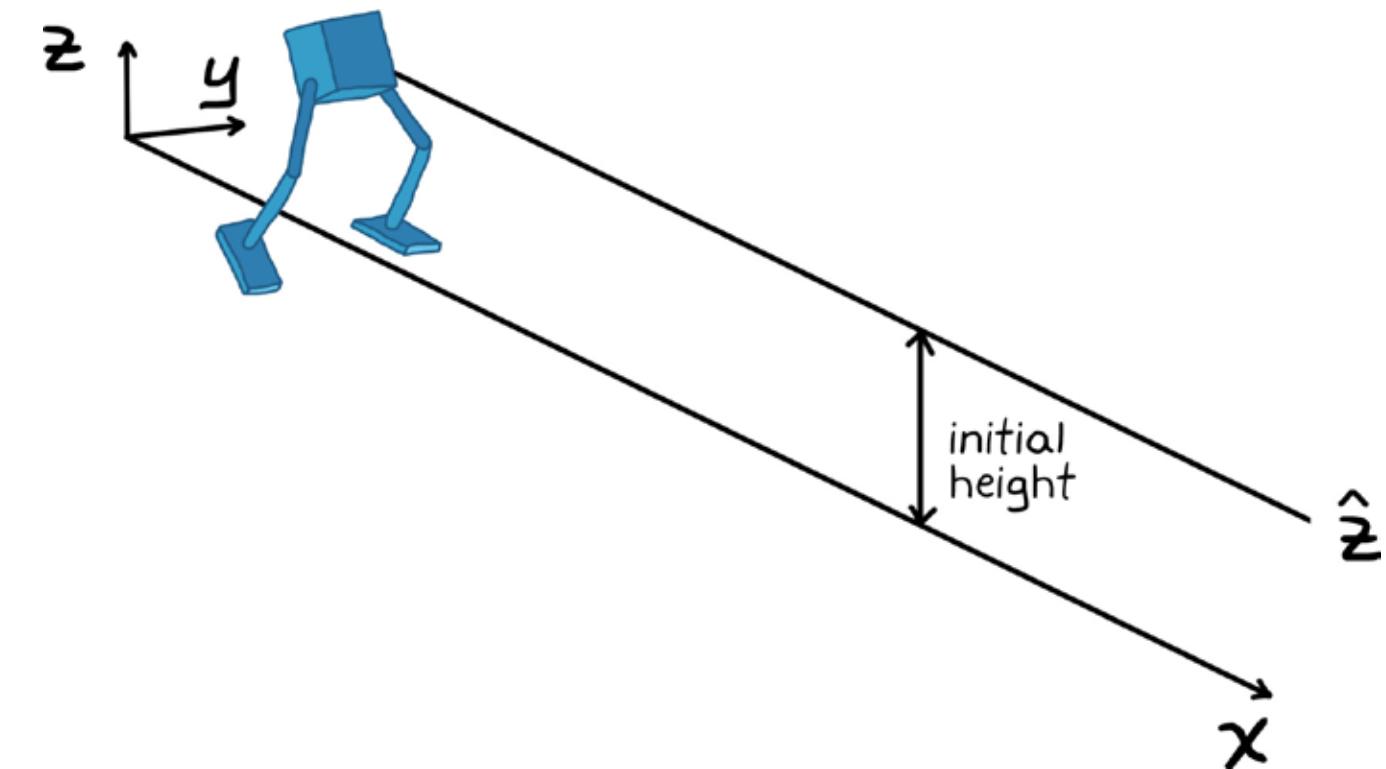


奖励重塑并不总是用于代替稀疏奖励。它还是工程师将特定领域的知识注入代理的一种方法。例如，如果您确定想让机器人行走，而不是沿地面爬行，您可以奖励代理，让机器人的躯干保持在行走高度。您还可以奖励作动器工作量的降低，更长时间地站立，不偏离预定路线。

$$r_t = v_x - 3y^2 - 50\hat{z}^2 + 25 \frac{T_s}{T_f} - 0.02 \sum_i u_{t-1}^i$$

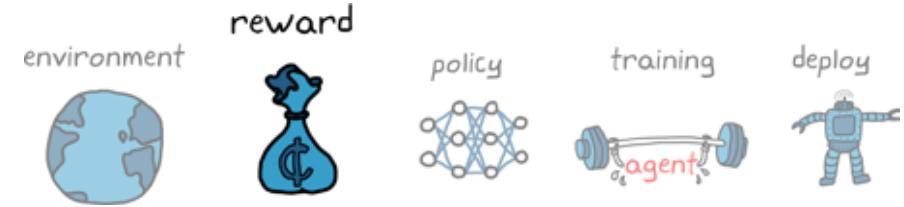
Annotations for the reward function components:

- v_x : forward velocity
- $3y^2$: don't stray from path
- $50\hat{z}^2$: keep trunk high
- $25 \frac{T_s}{T_f}$: walk as long as possible
- $0.02 \sum_i u_{t-1}^i$: minimize actuator effort

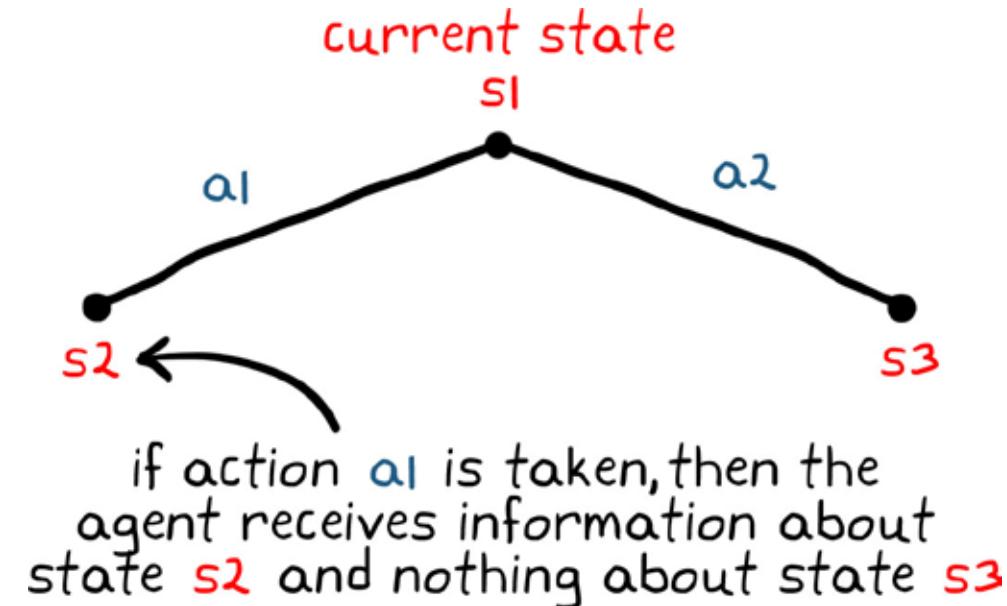


这不是说设计奖励函数就很容易，使之正确运作可能是强化学习中比较难的任务之一。例如，直到您花了很长时间训练代理，而它无法产生您所寻求的结果之后，您才可能知道奖励函数是不是设计得不好。但是，有了这个总体的认知，您至少能够更好地理解哪些事情需要注意并且可能使设计奖励函数容易一点。

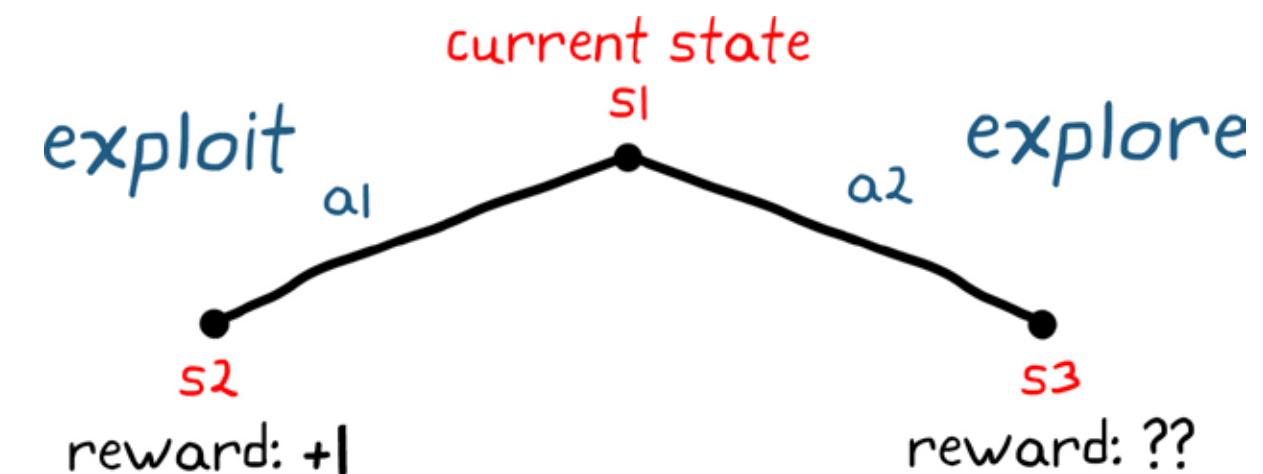
探索 (Exploration) 与利用 (exploitation)



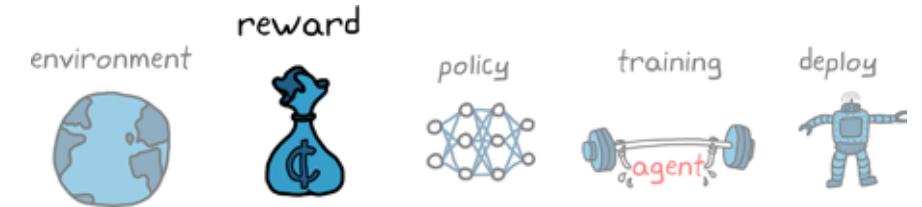
强化学习的一个重要方面是在代理与环境交互时权衡探索与利用之间的利弊。强化学习时需要做这个决定的原因在于学习是在线实现的。不是利用静态数据集，而是由代理的动作决定从环境中返回哪些数据。代理做出的选择决定了它会接收到的信息，以及因此它可从中学习的信息。



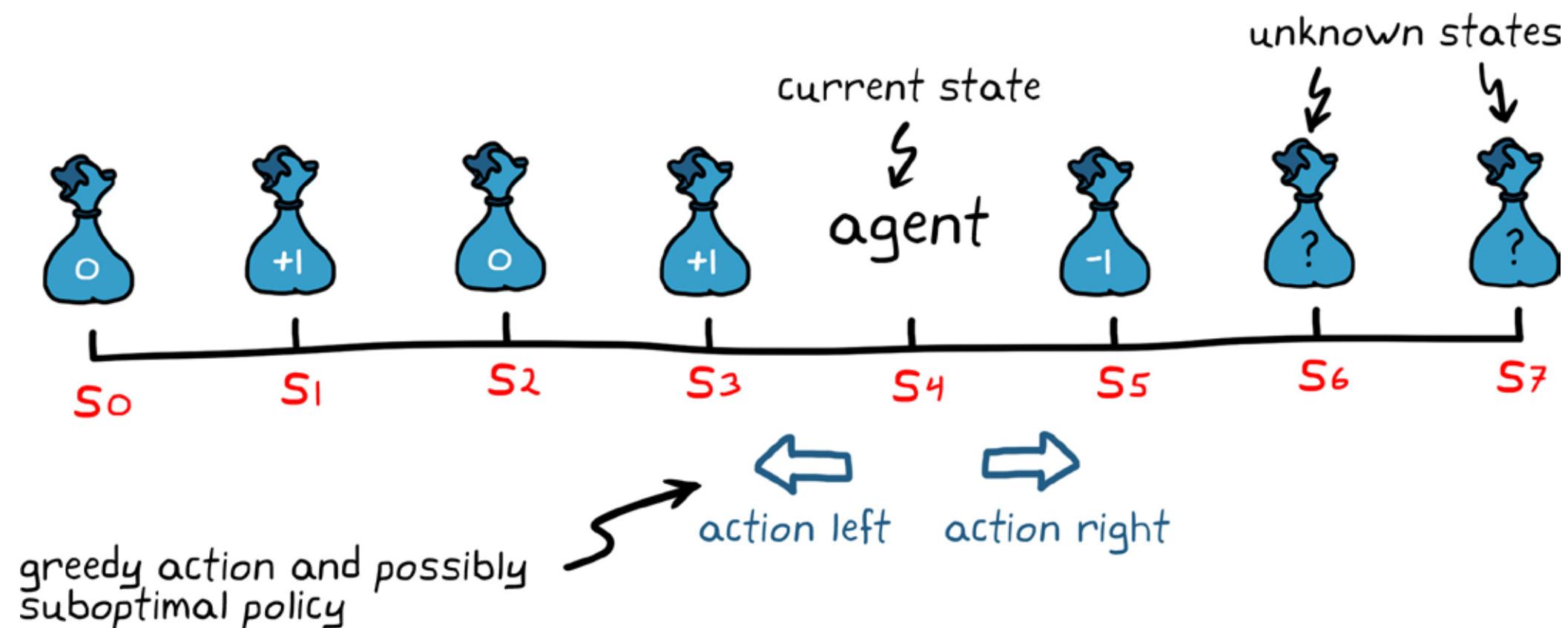
这里的构想是：代理是否应该利用环境，选择收取它已经知道的最多奖励的那些动作？还是应该选择探索环境中仍然未知部分的动作？



纯利用面临的问题



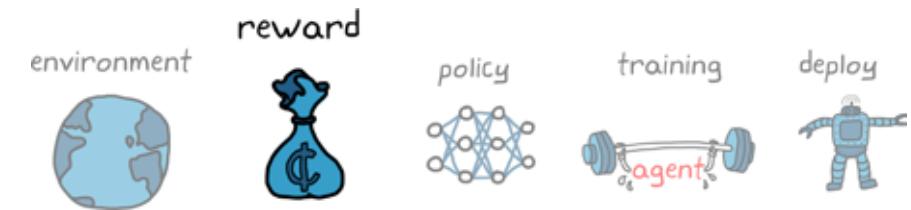
比如说，代理处于特定的状态，它可以采取两个动作之一：向左或向右。它知道，向左会产生 +1 奖励，向右会产生 -1 奖励。代理对于初始低奖励状态右侧的环境一无所知。如果代理采取贪婪的方法，总是利用环境，那么它会选择向左并收取它知道的最高奖励，而完全忽略其他状态。



所以，您可以看到，如果代理总是利用它认为在当前时刻的最佳动作，则可能永远得不到在低奖励动作之外存在的状态的信息。这种纯利用可能会延长找到最优策略的时间，也可能造成学习算法收敛到次优的策略，因为可能永远探索不到状态空间的全部区域。

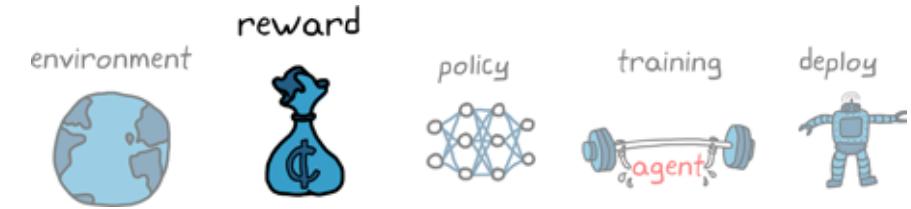
纯探索面临的问题

反之，如果您偶尔让代理探索，即使冒着收取较少奖励的风险，也可能扩大其对于新状态的策略。这打开了找到它不知道的更高奖励的可能性，增加了收敛到全局解的概率。但您不想让代理过度探索，因为这种方法也有缺点。举例来说，在物理硬件上训练时，纯探索不是一个好方法，因为代理会面临因探索某个动作而造成硬件损坏的风险。思考一下在高速公路上探索随机方向盘输入的自动驾驶汽车可能造成的破坏力。



但是，即使对于不存在硬件损坏风险的仿真环境，纯探索也不是有效的学习方法，因为代理可能花时间探查较大部分的状态空间。虽然这对找到全局解有好处，但过度的探索可能会减缓学习速度，以至于在合理的学习时间内找不到足够好的解。因此，最好的学习算法在探索与利用环境之间达到一种平衡。

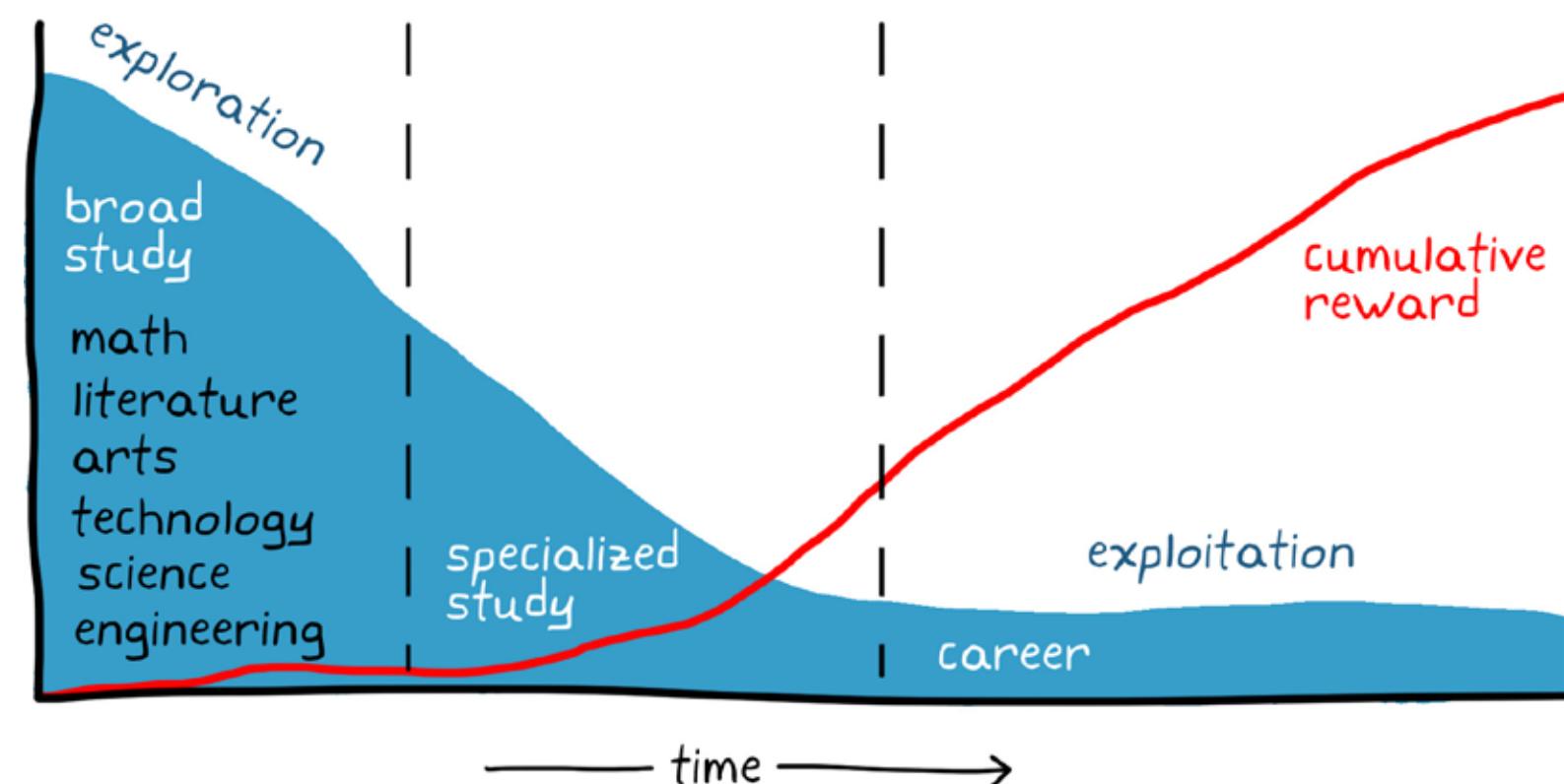
探索与利用的平衡



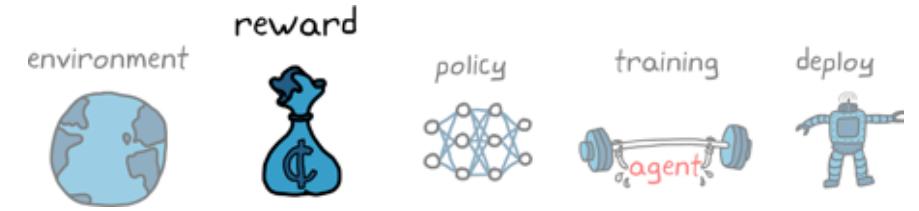
考虑一下学生在选择职业道路时可能采取的做法。学生们在低年级时会去探索不同的学科和课程，对新的体验一般持开放态度。在进行一番探索后，他们可能趋向于更多地学习某一专业学科，然后最终趋向于他们认为财务收益和工作满意度（奖励）最大化的职业。

探索每一种可能的职业选项，用一生的时间可能都不够。
因此，学生必须从目前他们已经探索的职业选项中选取最优的职业道路。如果他们推迟太长时间不去运用他们所学的知识，而是继续探索新的职业选项，那么留给他们收取回报的时间就不多了。

即使强化学习算法提供了一个简单的方法来平衡探索与利用，但是在整个学习过程中的什么位置设置平衡点可能不是那么显而易见的，从而让代理在所分配的学习时间内，收敛到一个足够好的策略。但是，一般来说，代理在开始学习时探索比较多，在结束前逐渐过渡到更多的利用，就像学生一样。



价值的值

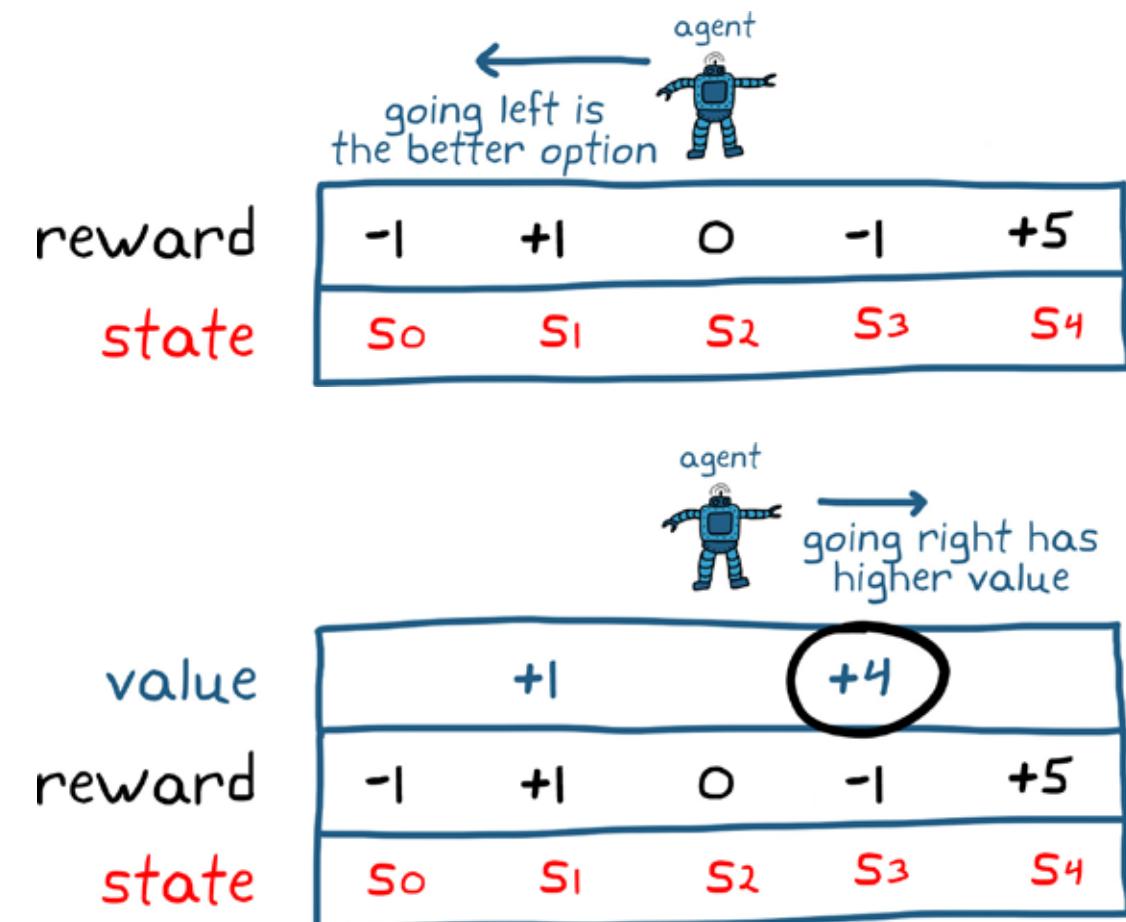


强化学习的第二个重要方面是**价值**的概念。评估一个状态或动作的价值，而不是奖励，可以帮助代理选择将会在一段时间内收取最多奖励（而不是短期利益）的动作。

奖励: 处于某一状态或采取特定动作的即时收益

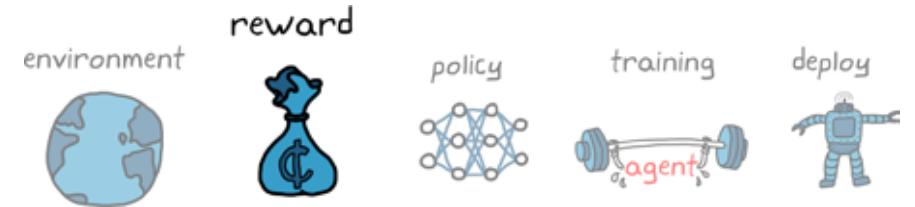
价值: 代理预期从某一状态和往后将会获得的总回报

例如，假设我们的代理正尝试收取两步内的最多奖励。如果代理只看每个动作的奖励，它会先向左一步，因为这样产生的奖励比右侧高。然后它会向右退，因为这同样是最奖励，最后，一共收取 +1。



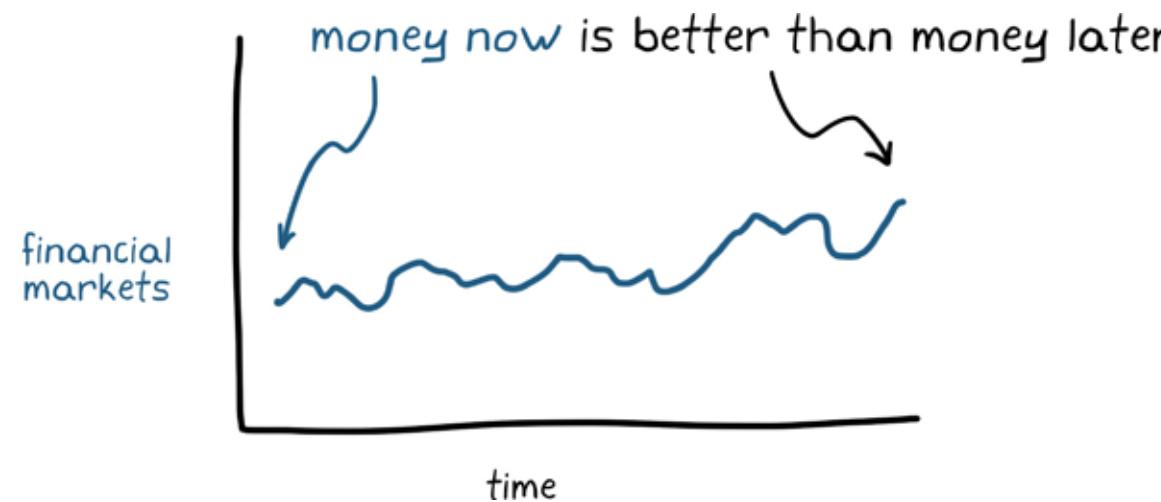
但是，如果代理能够评估状态的价值，它就会看到，向右走比向左走有更高的价值，即使奖励较低。使用价值作为其向导，最后代理就会得到 +4 的总回报。

短视的好处

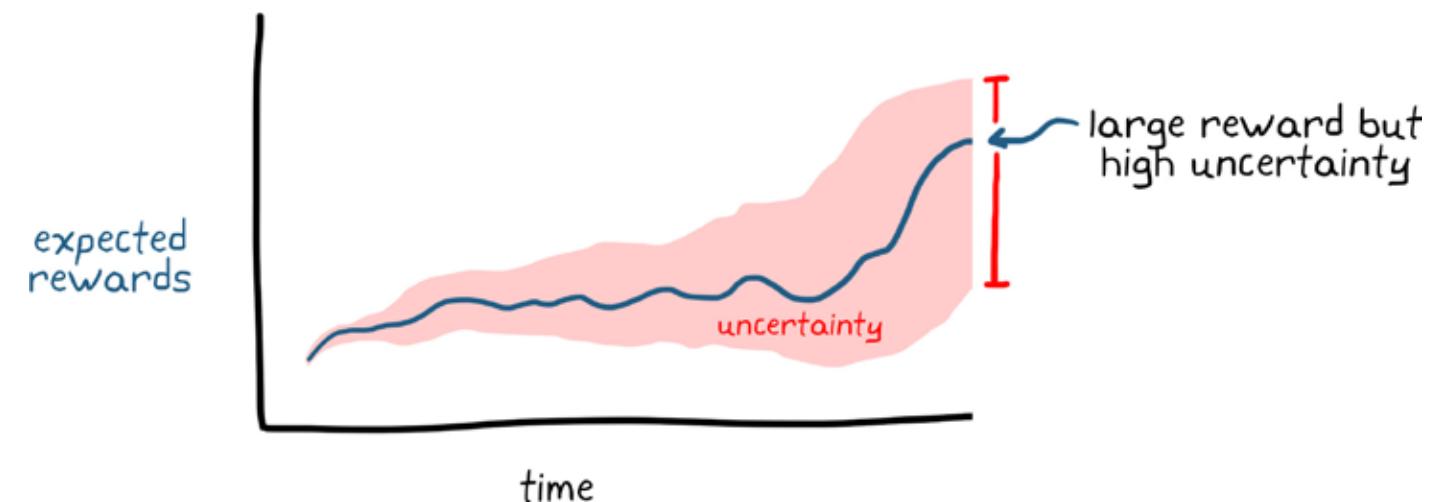


当然，承诺在多个连续动作后得到高奖励并不意味着第一个动作肯定最好；这里至少有两个好理由。

首先，像金融市场一样，您口袋里现在的钱可能比一年后口袋里多一点的钱更值钱。



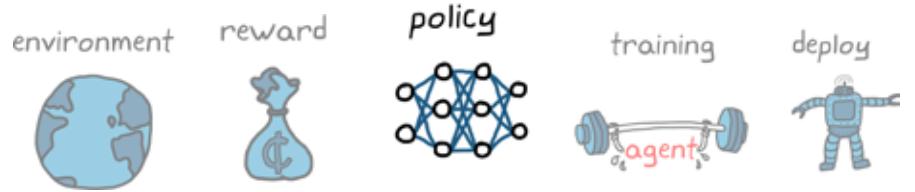
其次，对更远的将来所能获取的奖励的预测变得不大可靠；因此，等到那时，该奖励可能已不存在了。



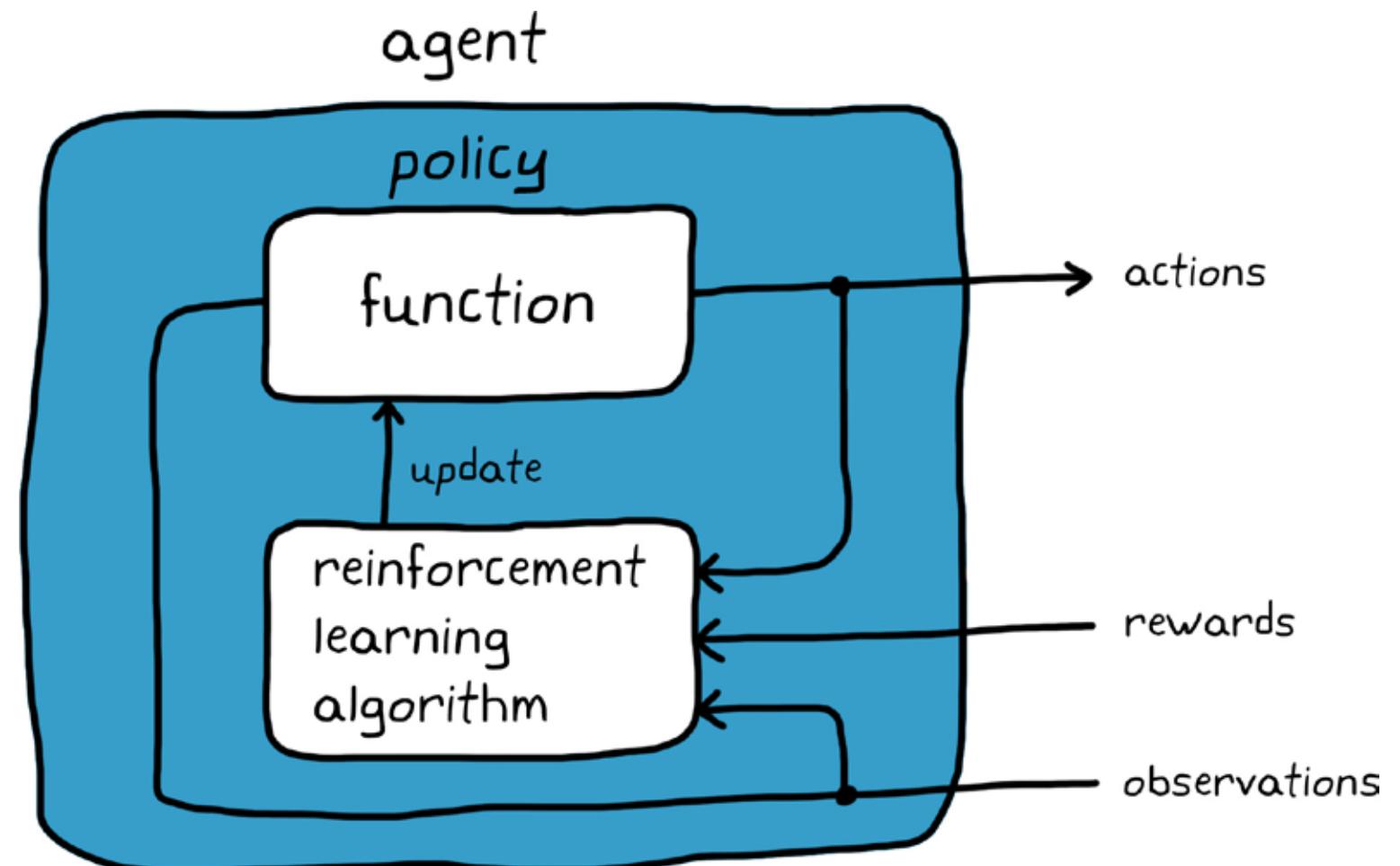
在这两种情况下，在评估价值时多一点短视比较有利。在强化学习中，通过对奖励打折，越远的未来折扣越大，您可以设置想让代理短视的程度。具体实现方式是设置折扣系数 gamma，介于 0 到 1 之间。

$$\text{total discounted reward} = r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 r_4 + \dots = \sum_{i=1}^T \gamma^{i-1} r_i$$

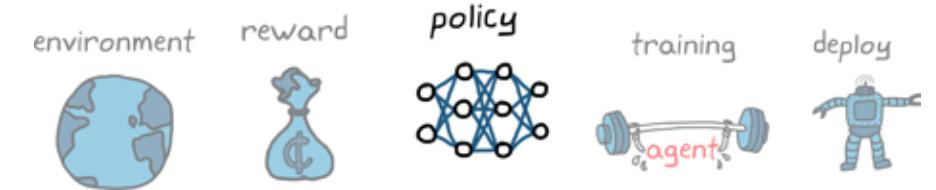
什么是策略?



既然您已了解环境以及它在提供状态和奖励方面的角色，现在可以开始探讨代理本身了。代理由策略和学习算法组成。策略是将观测量映射到动作的函数，学习算法是用来查找最优策略的优化方法。



如何表示策略



在最基本的级别，策略是以状态观测为输入、以动作为输出的函数。所以，如果您在寻找表示策略的方法，任何具有这种输入和输出关系的函数都可以。

policy \Rightarrow *actions = function (state observations)*

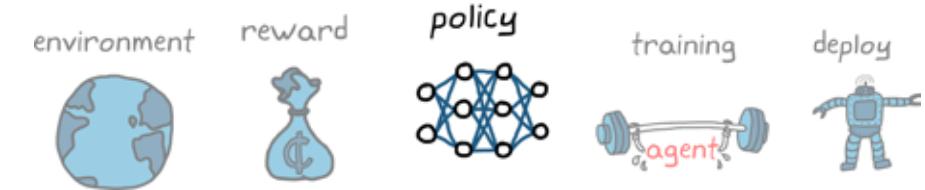
一般来说，有两种构造策略函数的方法：

- 直接：状态观测和动作之间存在特定映射。
- 间接：您着眼于其他指标（如价值）来推断最优映射。^{*}

接下来的几页将介绍如何使用基于价值的方法，重点说明可用来表示策略的不同类型的数学结构。但是请记住，这些结构对基于策略的函数也适用。

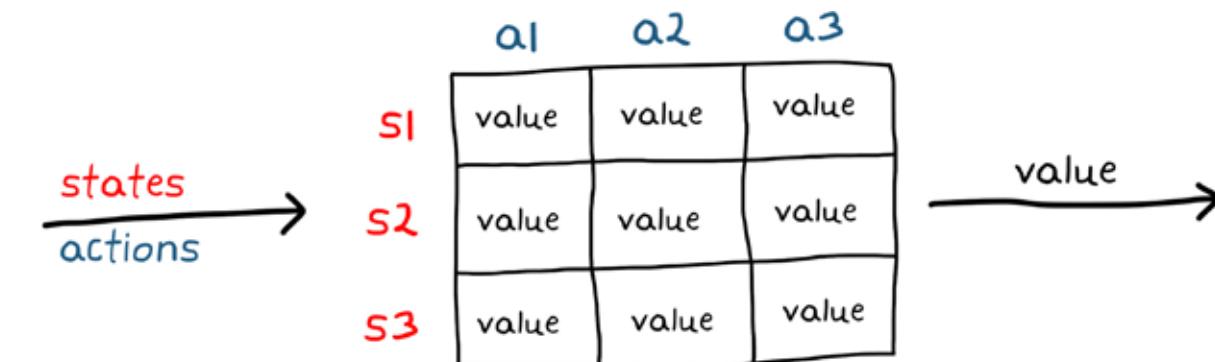
^{*} 事先提醒！您可以在名为 *Actor-Critic* 的第三种方法中结合直接策略映射和基于价值映射的好处，稍后介绍这种方法。

用表格表示策略

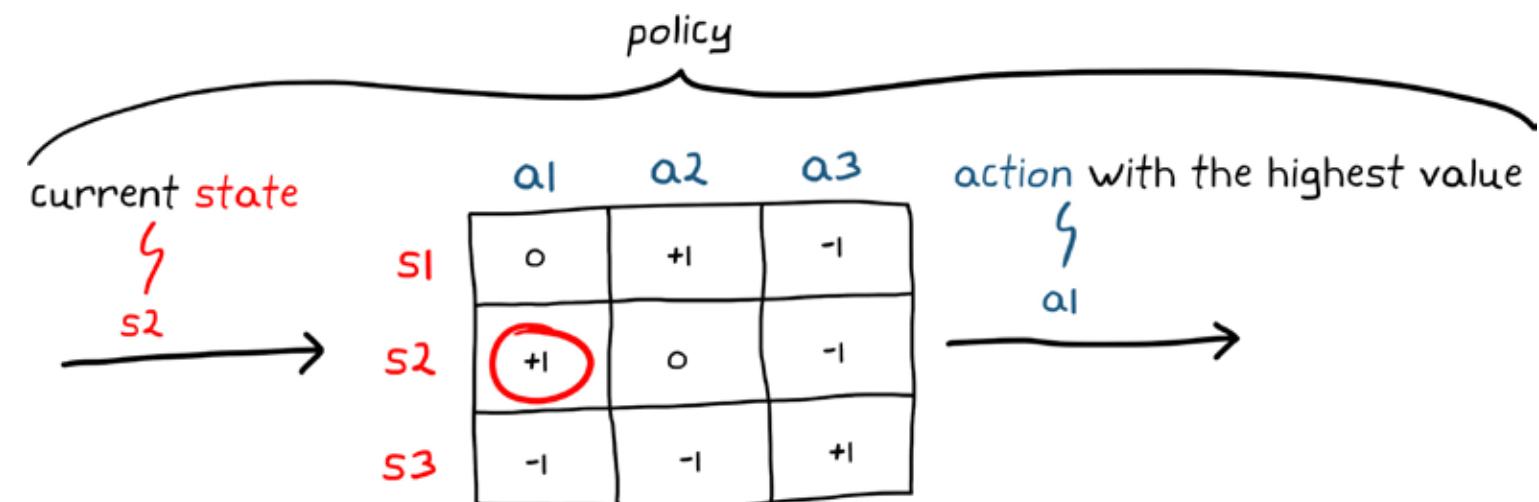


如果环境的状态和动作空间离散，且数量少，则可以使用简单表格来表示策略。

表格正是您期望的形式：一个数组，其中，输入作为查询地址，输出是表格中的相应数字。有一种基于表格的函数类型是 Q-table，它将状态和动作映射到价值。

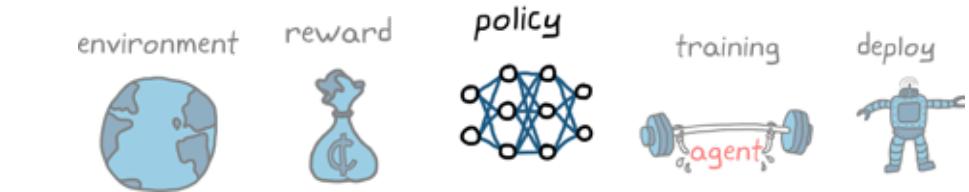
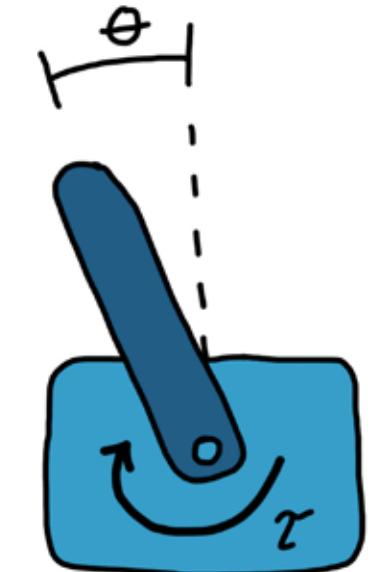


使用Q-table，策略会在当前状态给定的情况下检查每个可能动作的价值，然后选择具有最高价值的动作。使用 Q-table 训练代理将包括确定表格中每个状态/动作对的正确价值。在表格完全填充正确的值之后，选择将会产生最多长期奖励回报的动作就相当直接了。



连续的状态/动作空间

当状态/动作对的数量变大或变为无穷大时，在表格中表示策略参数就不可行了。这就是所谓的维数灾难。为了直观地理解这一点，让我们考虑一个用于控制倒立摆的策略。倒立摆的状态可能是从 $-\pi$ 到 π 的任何角度和任何角速率。另外，动作空间是从负极限到正极限的任何电机转矩。试图在表格中捕获每个状态和动作的每一种组合是不可能的。



angle rate	angle	torque			
-0.3	0.124	value	value	value	value
0.17	0.137	value	value	value	value
0.175	0.139	value	value	value	value
0.223	0.204	value	value	value	value
:	:				

可以用一个连续函数来表示倒立摆的连续特性，该函数以状态为输入，以动作为输出。但是，在您开始学习此函数中的正确参数之前，您需要定义逻辑结构。对于高自由度系统或非线性系统来说，这可能很难设计。

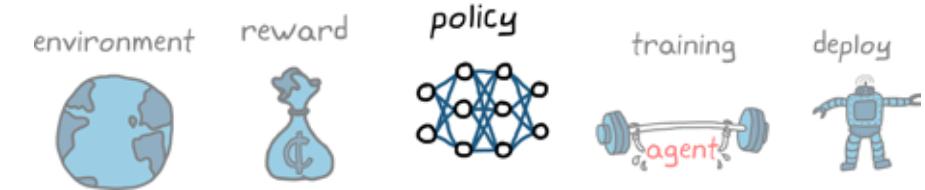
$$\text{value} = -(\dot{\theta}^2 + \theta^2) + \tau ?$$

we need to define the logical structure

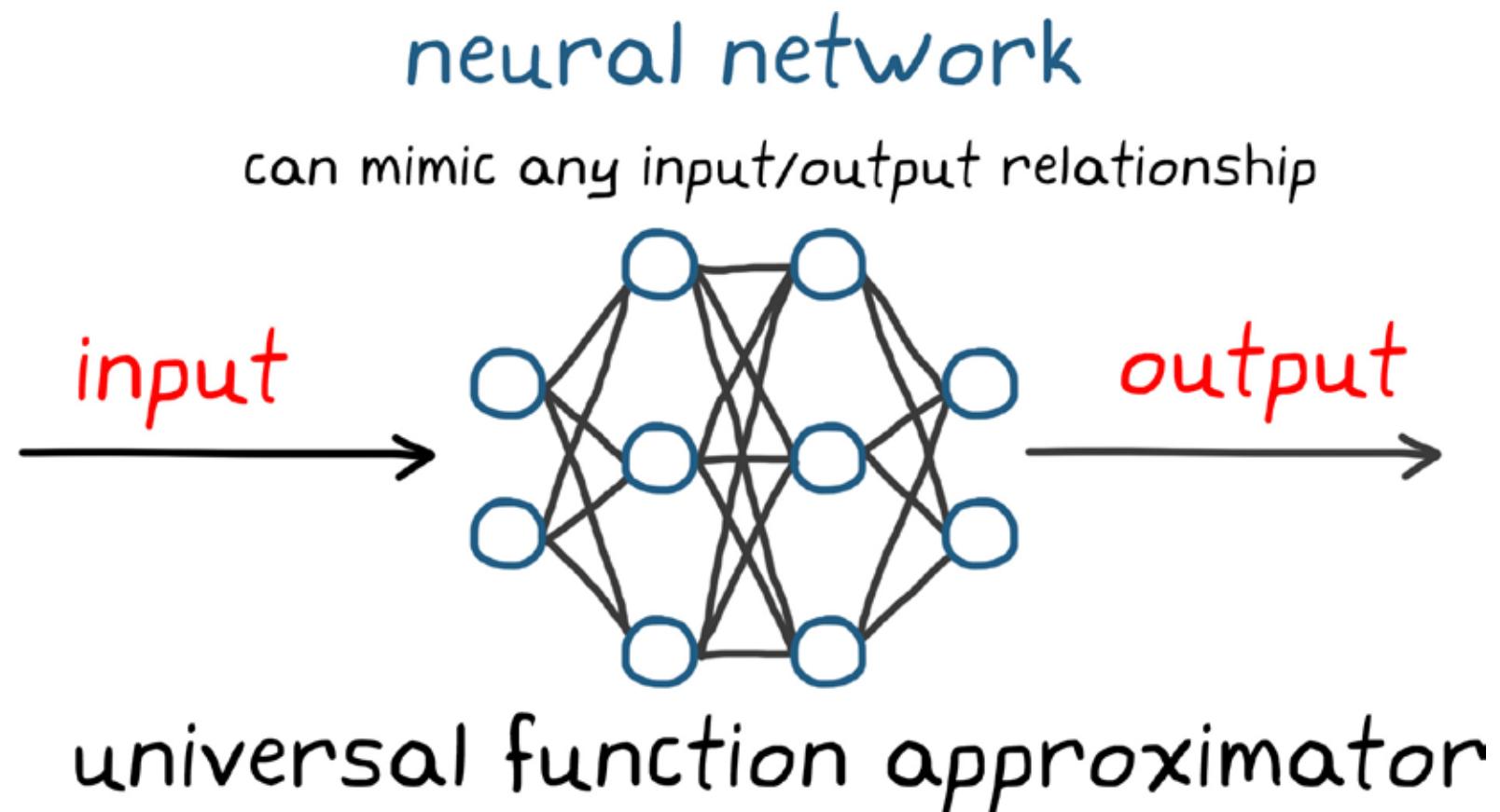
$$\text{value} = \dot{\theta} \sin(\theta) + \tau ?$$

所以，您需要一种方法来表示能处理连续状态和动作的函数，而不必为每种环境状况设置对应难以设计的逻辑结构。这就是神经网络发挥作用的地方。

通用函数逼近器

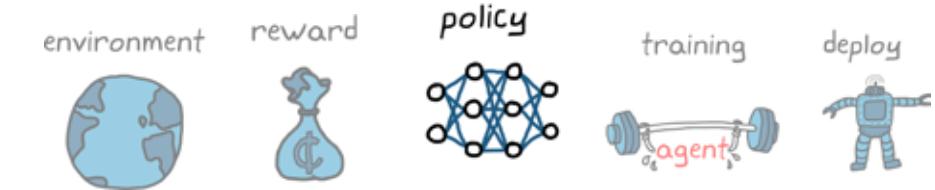


神经网络是一组节点或人工神经元，采用一种能够使其成为通用函数逼近器的方式连接。这意味着，给出节点和连接的正确组合，您可以设置该网络，模仿任何输入与输出关系。尽管函数可能极其复杂，神经网络的通用性质可以确保有某种神经网络可以实现目标。



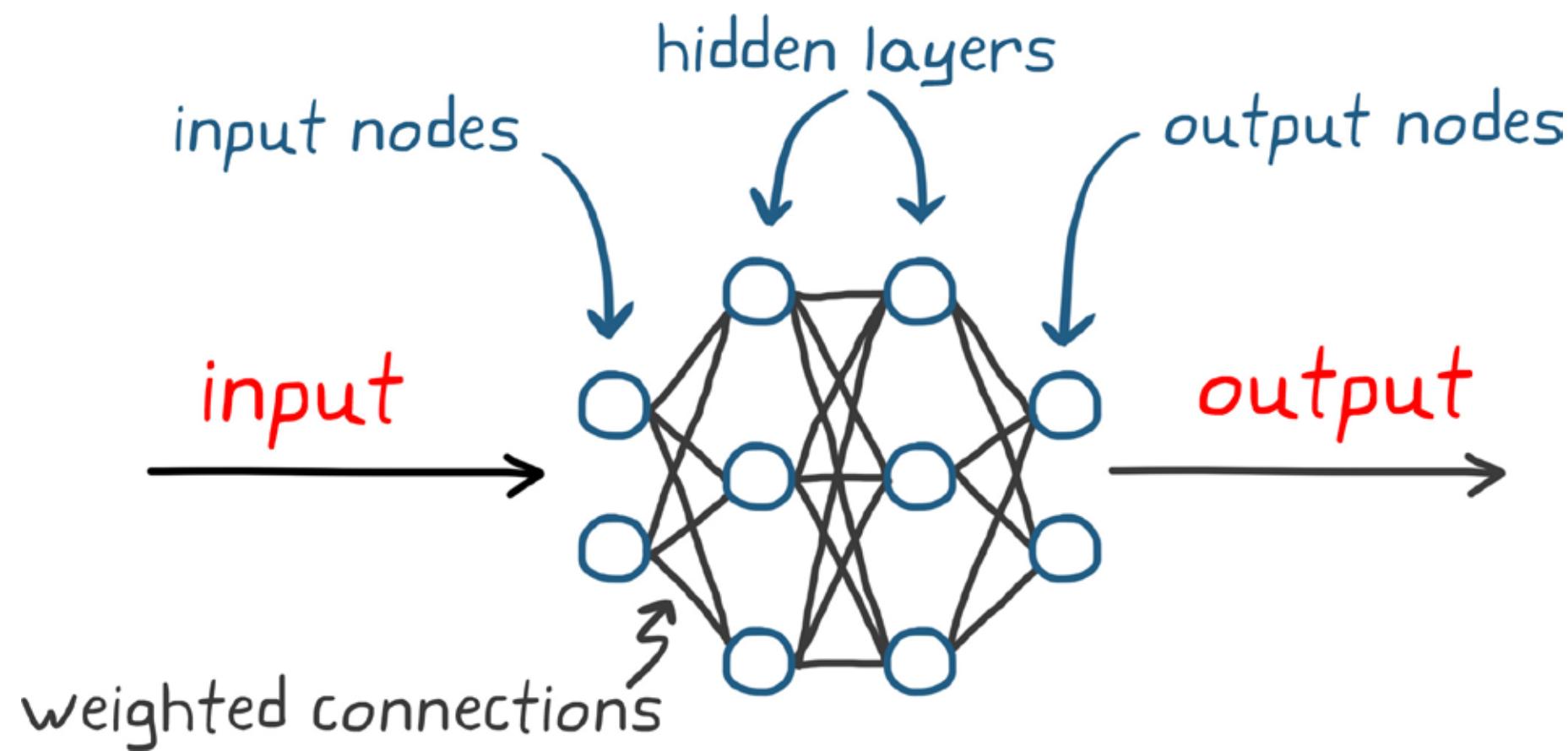
所以，与其尝试寻找适合特定环境的完美非线性函数结构，不如使用神经网络，这样就可以在许多不同环境中使用相同的节点和连接组合。唯一的区别在于参数自身。学习过程将包括系统地调节参数，找到最优输入/输出关系。

什么是神经网络？

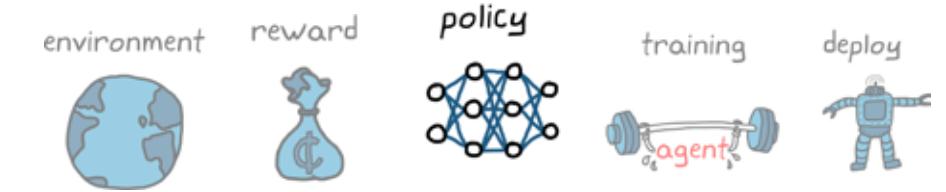


这里不会深入探讨神经网络的数学原理。但强调一些事情十分重要，方便解释稍后建立策略时的一些决策。

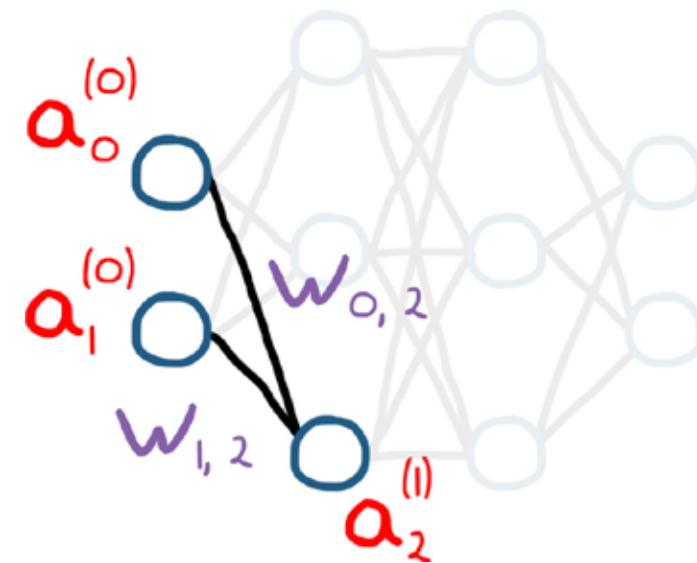
左边是输入节点，一个节点对应函数的一个输入，右边是输出节点。中间是称为隐藏层的节点列。此网络有 2 个输入、2 个输出和 2 个隐藏层，每层 3 个节点。对于全连接的网络，存在从每个输入节点到下一层中每个节点的加权连接，然后是从这些节点连接到后面一层，直到输出节点为止。



图形背后的数学



任何给定节点的值等于馈入该节点的每个节点乘以各自权重系数的总和再加上一个偏置。



value of $a_2^{(1)}$ = $w_{0,2} \cdot a_0^{(0)}$ + $w_{1,2} \cdot a_1^{(0)} + b_2^{(1)}$

\nwarrow layer
 \swarrow node position

您可以对某个层中的每个节点执行此计算，以紧凑的矩阵形式写出来，作为一个线性方程组。这组矩阵运算实质上是将一层节点的数值变换为下一层节点的值。

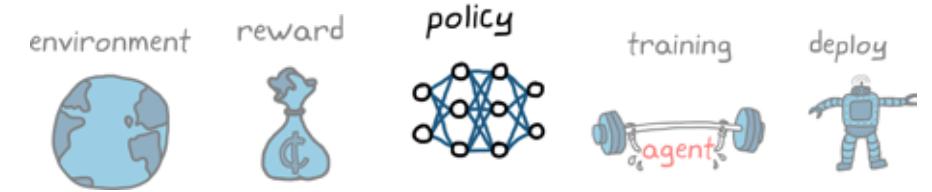
transform from layer 0 to layer 1 transform from layer 1 to layer 2 transform from layer 2 to layer 3

$$a^{(1)} = W_0 a^{(0)} + b^{(1)}$$

matrices

$$a^{(2)} = W_1 a^{(1)} + b^{(2)}$$
$$a^{(3)} = W_2 a^{(2)} + b^{(3)}$$

缺失的关键步骤



一连串级联的线性方程组如何充当通用函数逼近器? 具体来说, 它们如何表示非线性函数? 好, 有一个步骤可能是人工神经网络最重要的一个方面。在计算一个节点的值后, 会应用一个激活函数, 更改它前面节点(作为下一层输入)的值。

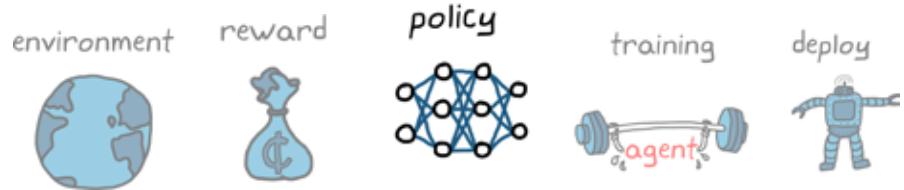
$$a^{(l)} = \text{act} [W_o a^{(o)} + b^{(l)}]$$

activation function is applied after linear operations

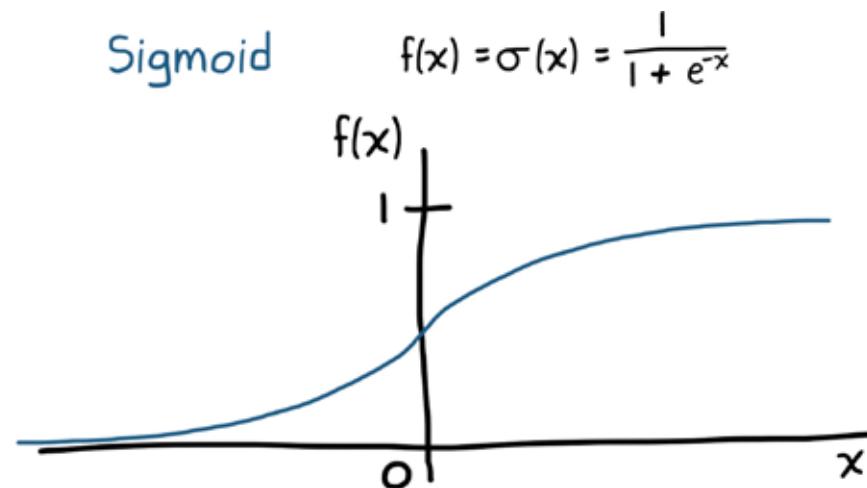
有若干不同的激活函数。它们共同的特点是非线性, 这对于构造可以逼近任何函数的网络至关重要。为什么出现这种情况? 因为许多非线性函数可以分解成加权的激活函数输出组合。

有关详细信息, 请阅读[通用逼近定理可视化](#)。

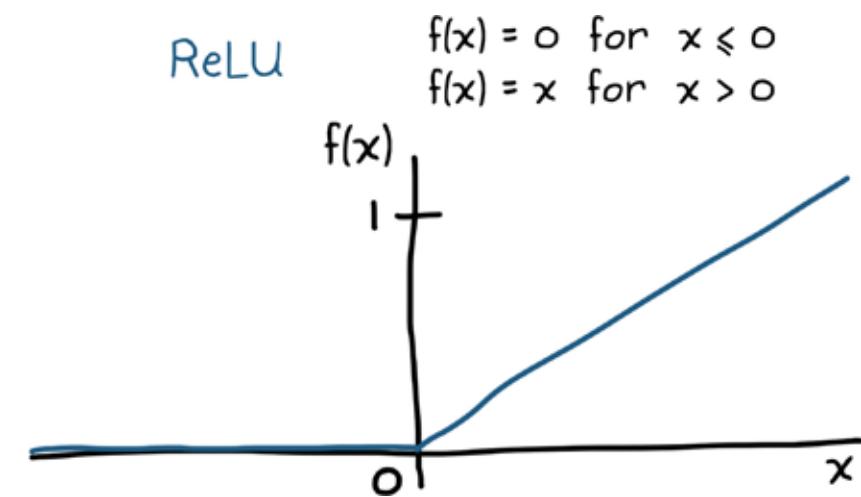
ReLU 和 Sigmoid 激活



sigmoid 激活函数会生成一条平滑的曲线，使介于负无穷大和正无穷大之间的任何输入都被压缩到 0 到 1 之间。



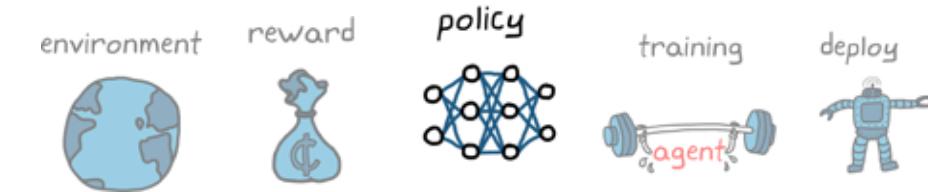
修正线性单元 (ReLU) 函数可以使任何负的节点值归零，正值则保持不变。



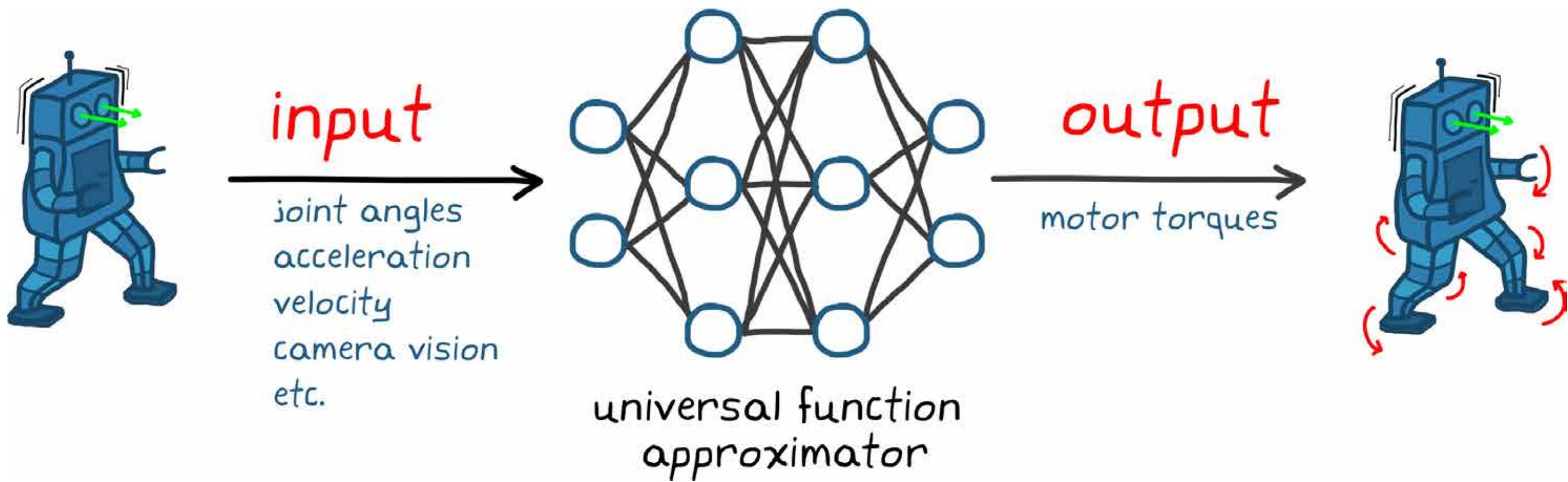
举个例子，预激活节点值 -2 使用 sigmoid 激活将变为 0.12，使用 ReLU 激活将变为 0。

preactivation node value	postactivation sigmoid	ReLU
-2	0.12	0
-1	0.27	0
1	0.73	1
2	0.88	2

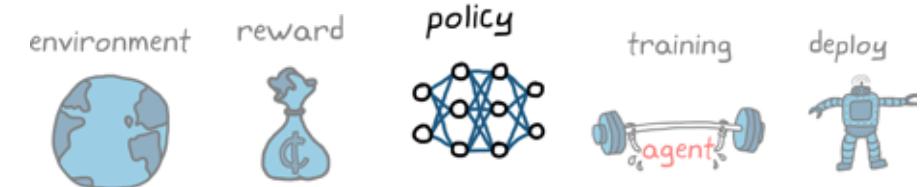
用神经网络表示策略



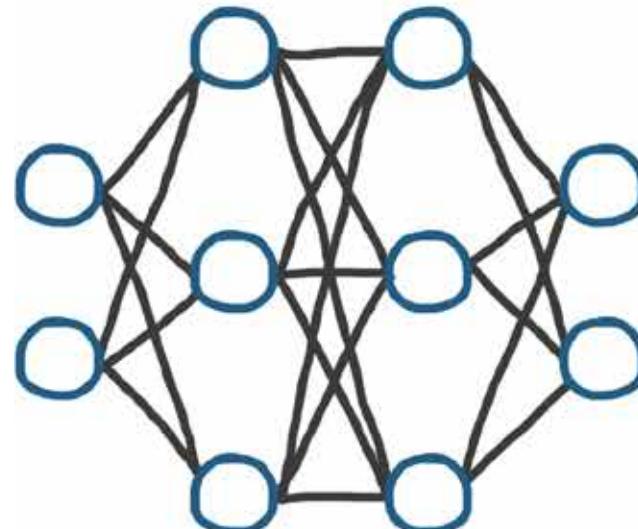
我们先归纳要点。您想找到一个函数，以大量的观测量为输入，并将其变换为能控制一些非线性环境的一组动作。由于此函数的结构通常过于复杂，难以直接求解，您想借助神经网络进行近似，随时间推移，学习该函数。人们倾向于认为，可以接入任何神经网络，然后放手让强化学习算法自行去寻找合适的权重和偏置的组合，任务就完成了。遗憾的是，事情往往不是这样。



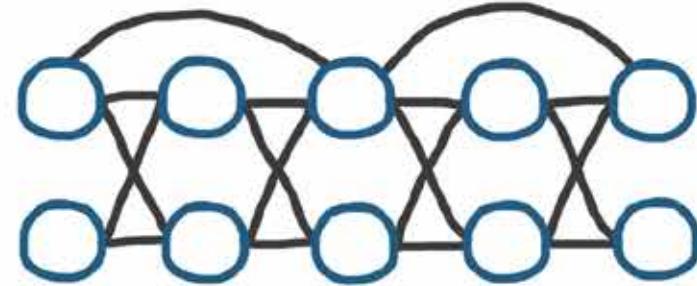
神经网络结构



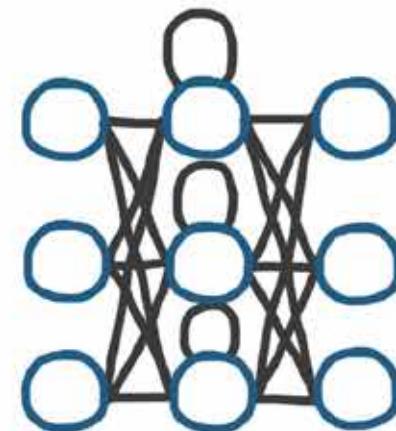
您必须提前对神经网络做出一些选择，确保它的复杂度足以近似您所寻求的函数，但也不要复杂得无法进行训练或慢得不可想像。例如，如您所见，您需要选择激活函数、隐藏层的数量以及每层的神经元数量。但除此之外，您还可以控制网络的内部结构。是否应该像您一开始使用的网络那样全连接？还是说应该像残差神经网络中那样，连接时跳过一些层？使用循环神经网络，利用自身环路形成内部记忆？各组神经元是否应该像卷积神经网络那样共同工作？



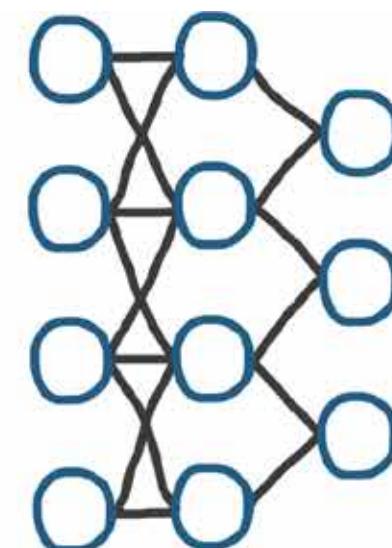
fully connected



residual



recurrent



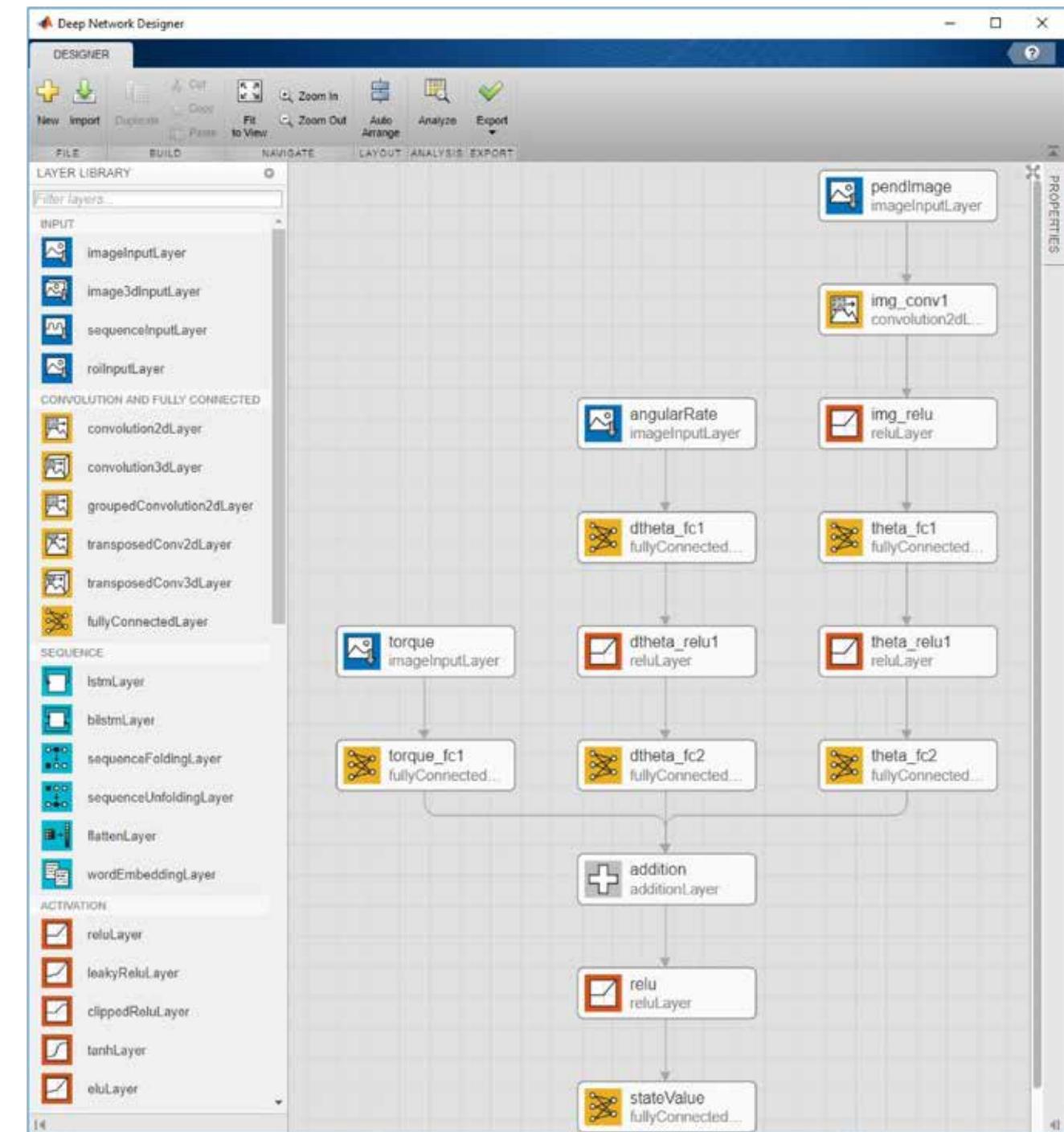
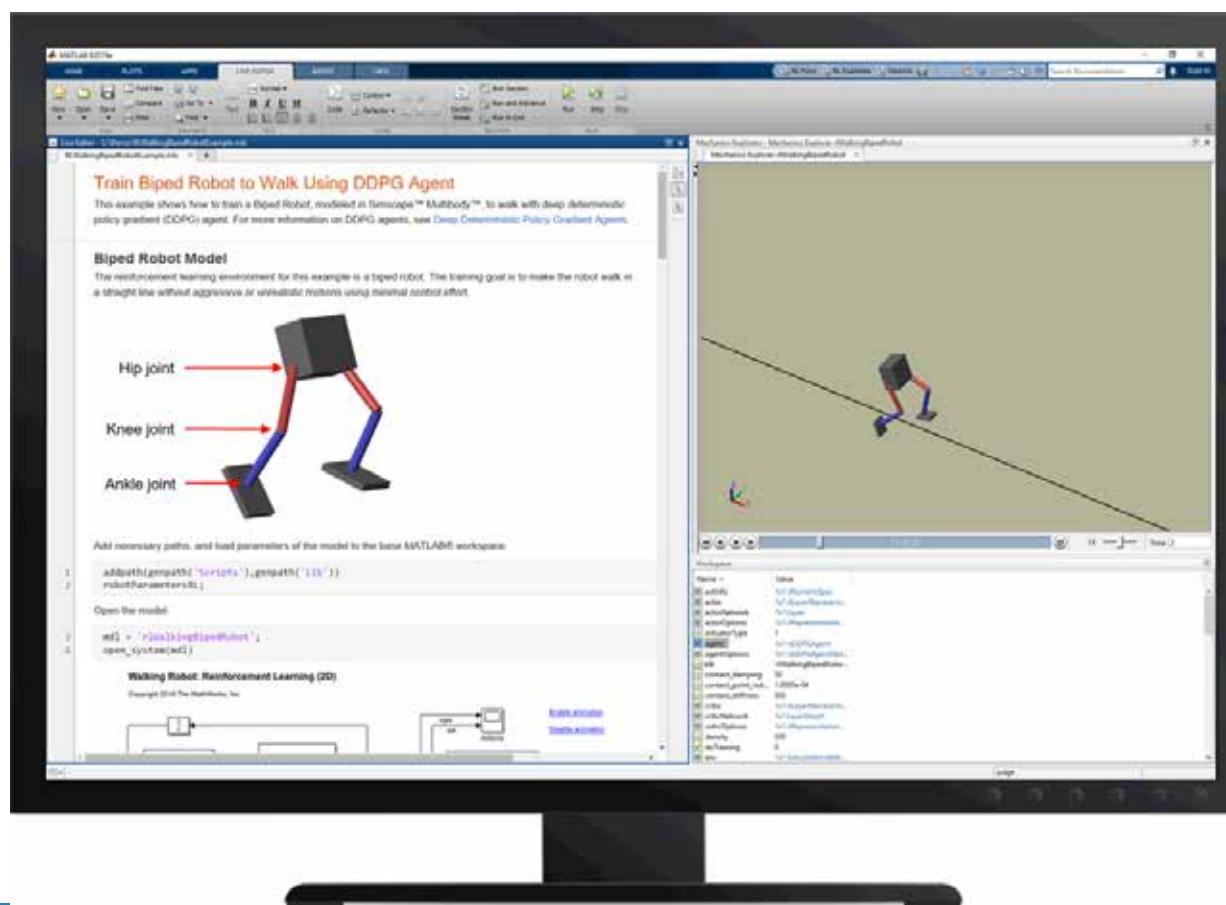
convolutional

与其他控制技术一样，没有一个适当的方法能确定神经网络结构。许多方法可归纳为，从已适用于您试图解决的问题类型的结构开始，然后加以微调。

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。



使用 Deep Network Designer 应用程序创建的 Deep Q-learning network (DQN) 代理。

了解更多

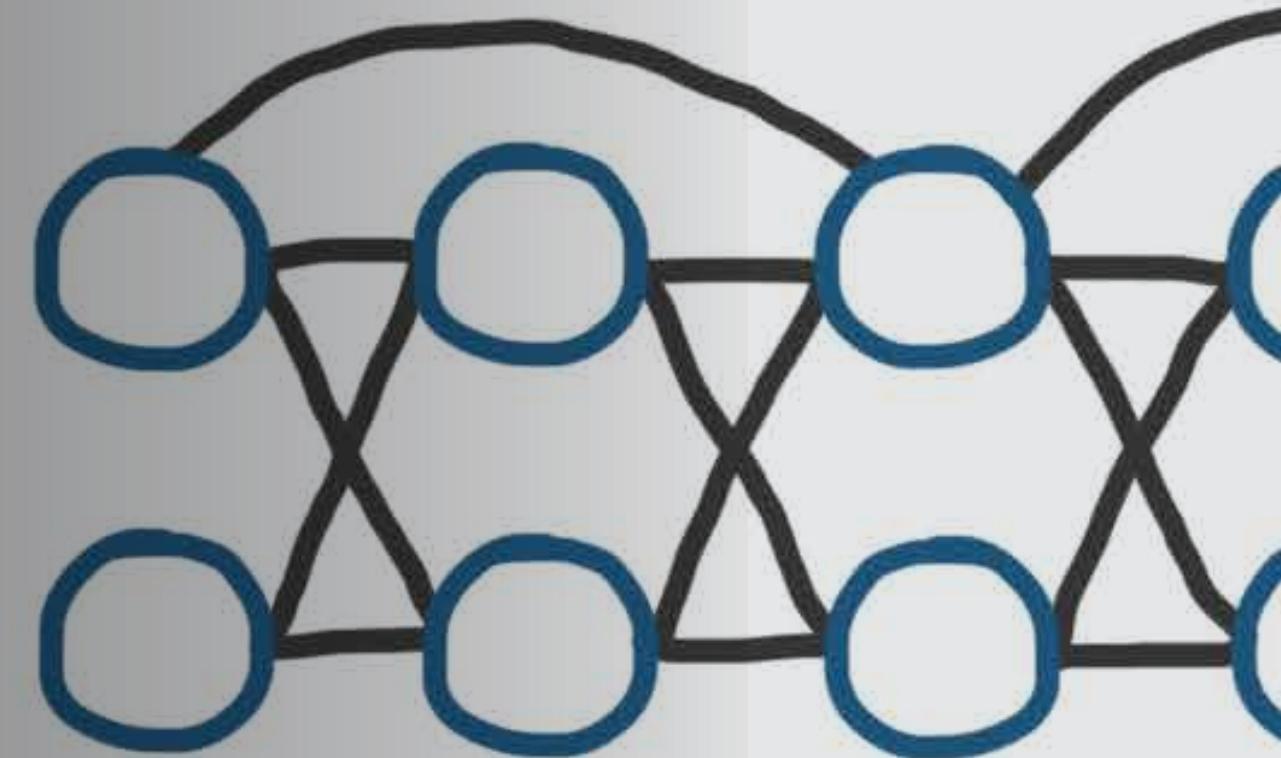
[Reinforcement Learning Toolbox - 概述](#)

[了解策略和学习算法 \(17:50\) - 视频](#)

[在 MATLAB 和 Simulink 中定义奖励信号 - 产品文档](#)

[策略和价值函数表示形式 - 产品文档](#)

[入门参考示例 - 示例](#)



第 3 部分: 了解训练和部署

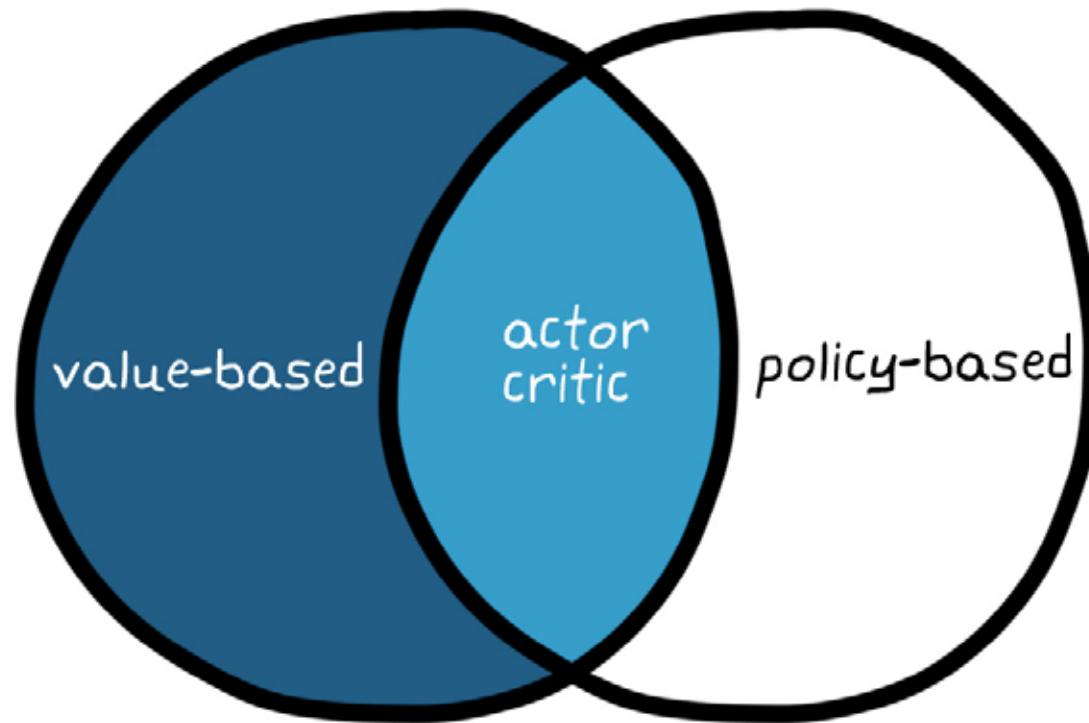
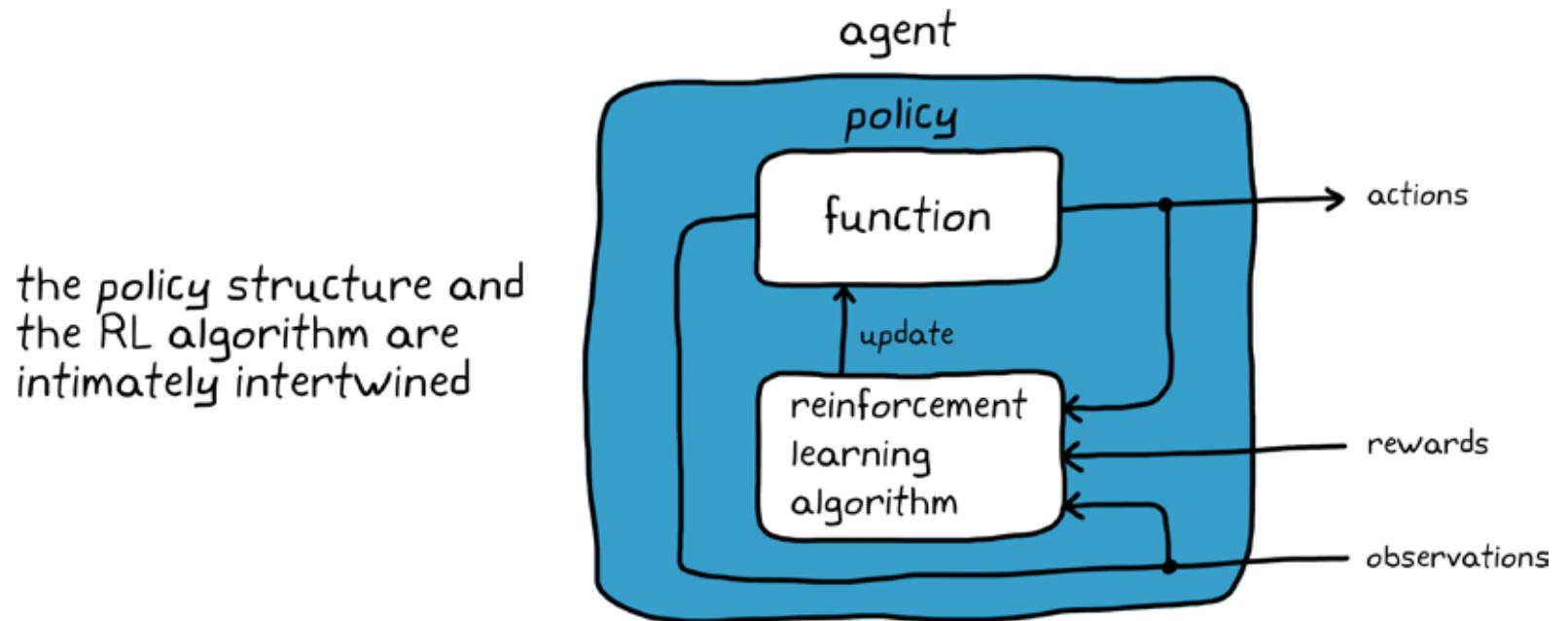
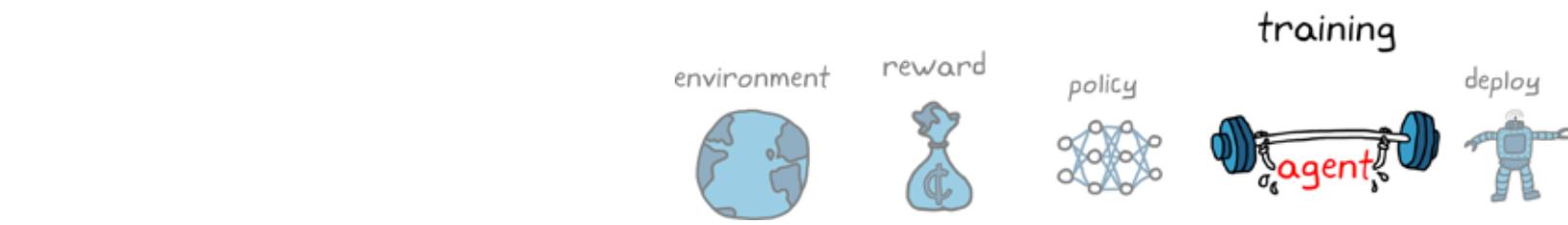
value-based

actor
critic

policy-based

如何构建策略

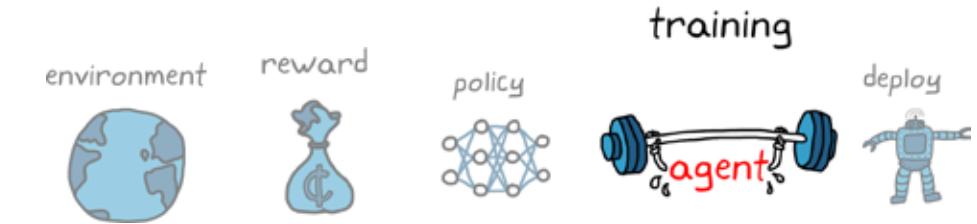
在强化学习 (RL) 算法中, 神经网络表示智能体策略。策略结构与强化学习算法密切相关;若未选择 RL 算法, 则无法构建策略。



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。

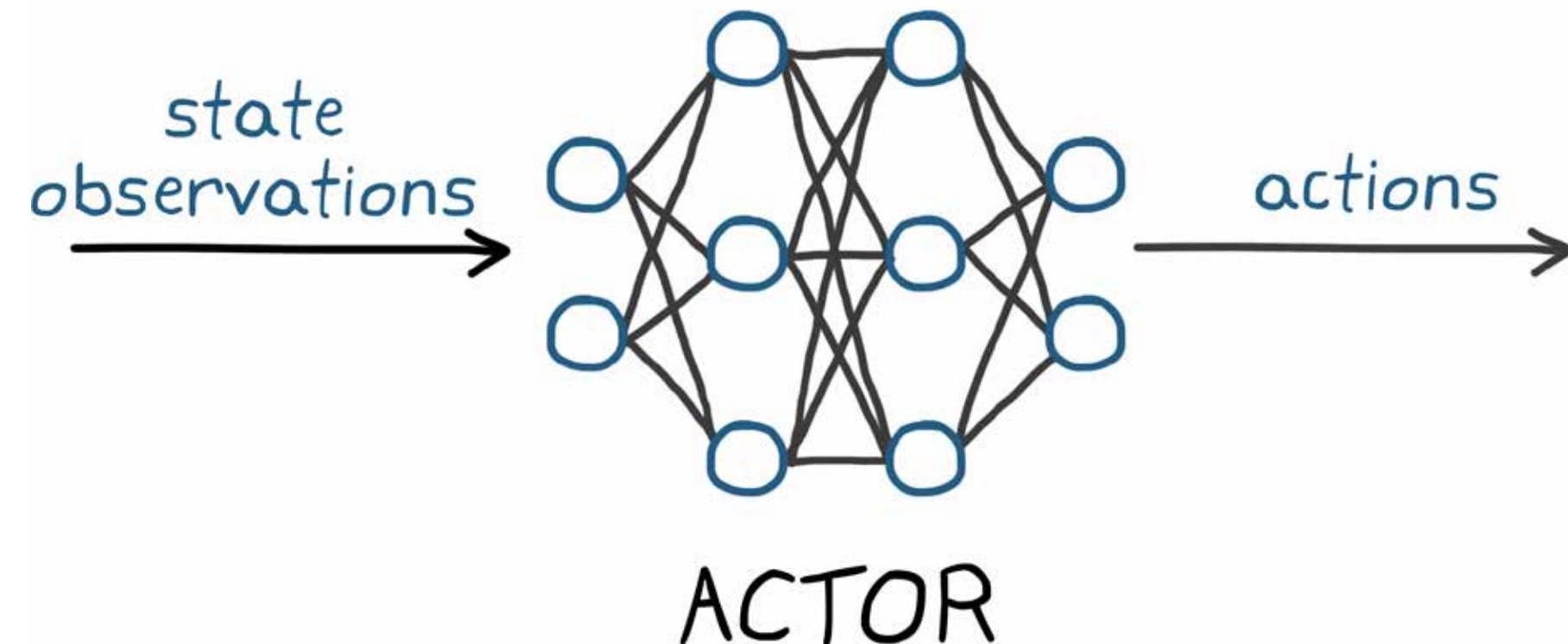
基于策略函数的学习

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。

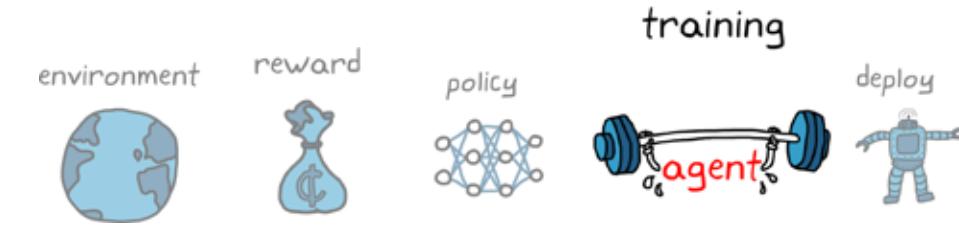


现在的问题在于,如何训练这个神经网络?为了大致地了解这一点,我们来看一款雅达利游戏:打砖块(Breakout)。

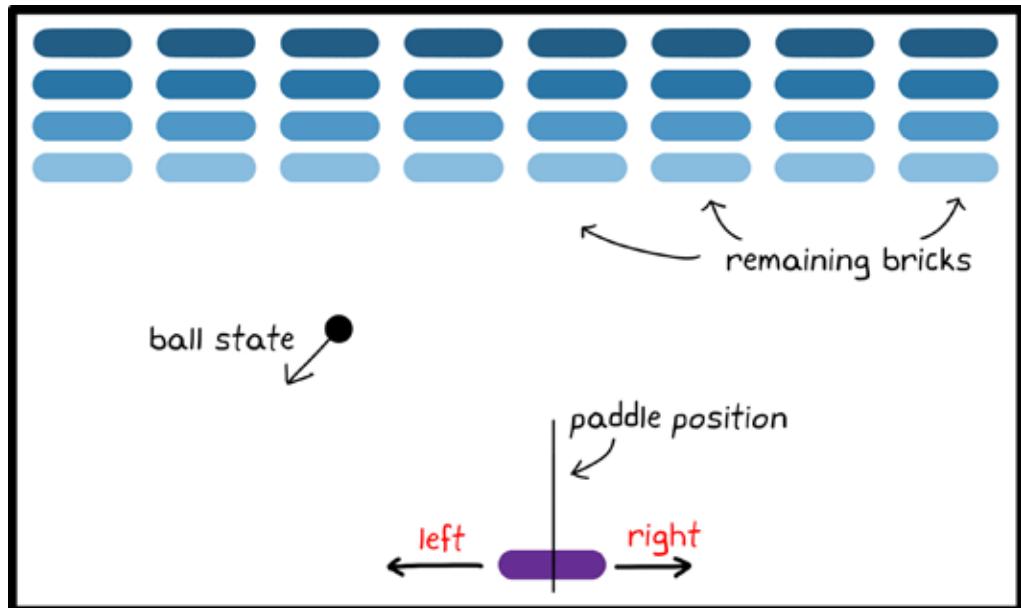
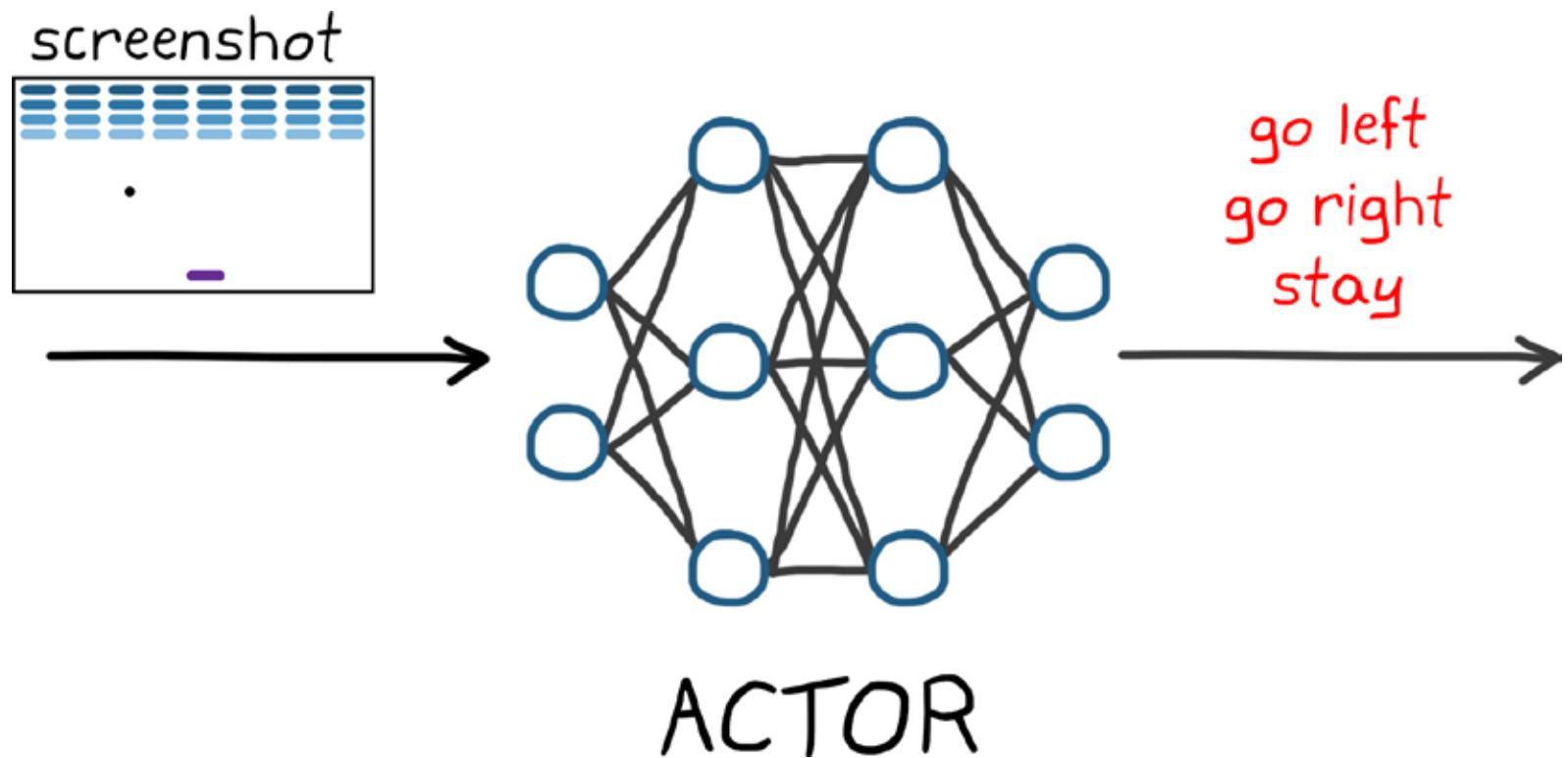
policy function-based learning



学习打砖块的策略方法



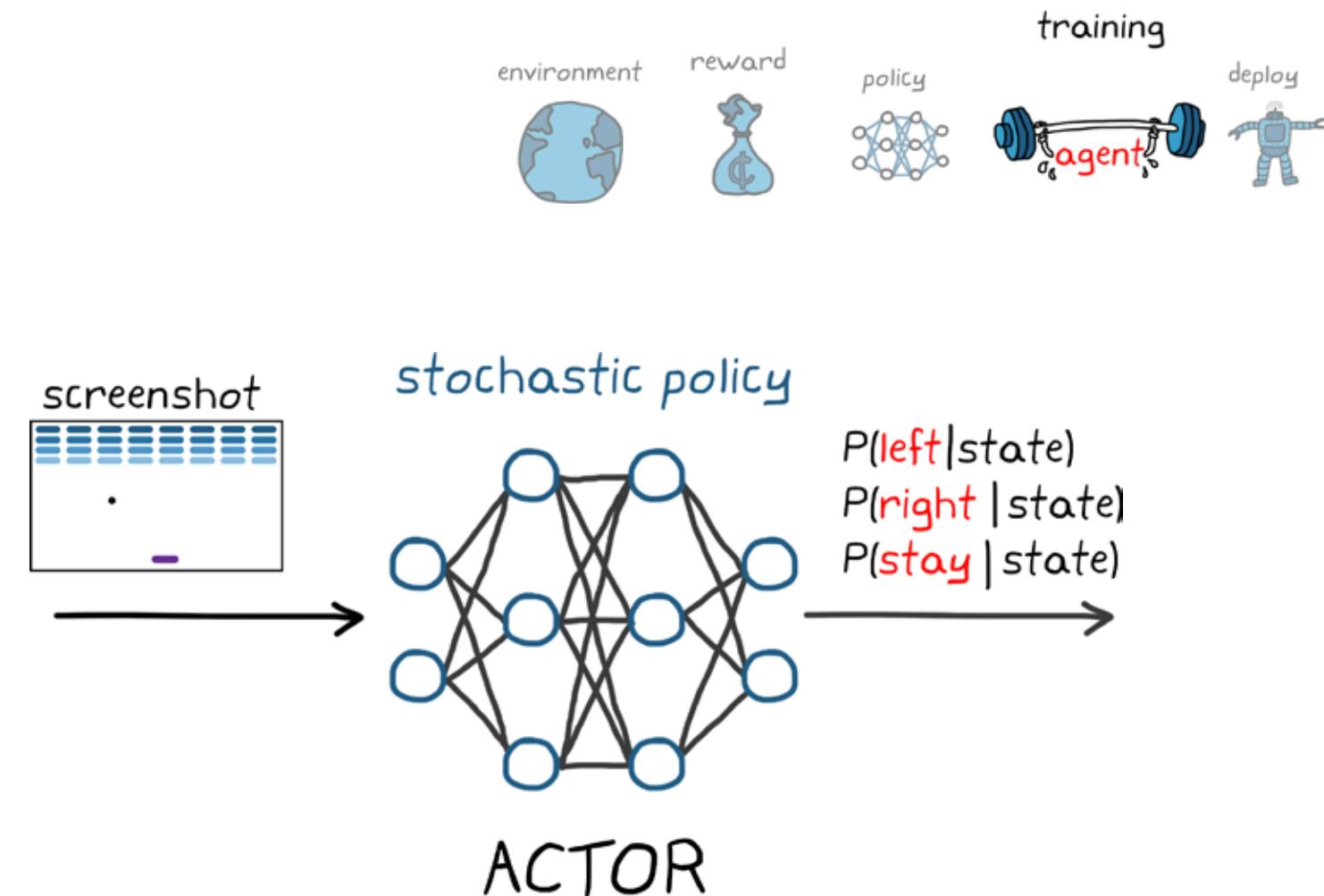
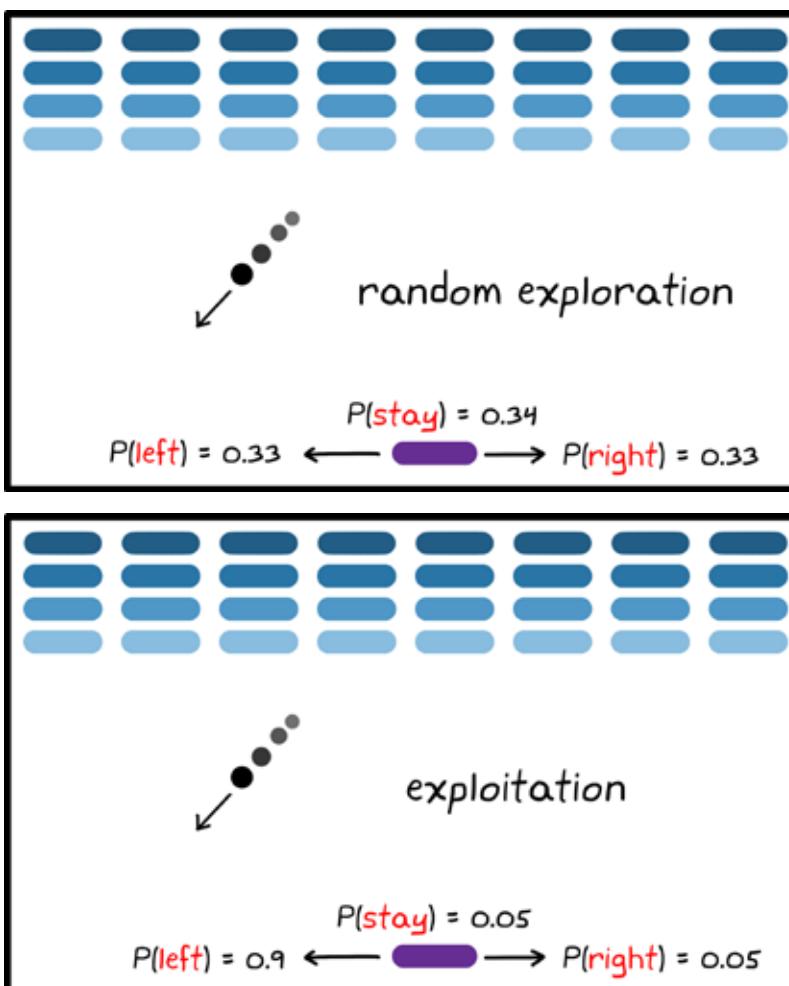
在打砖块这个游戏中，您需要使用球拍击打弹力球尽量消除砖块。这款游戏共包含三个动作，左移球拍、右移或保持不动，同时还有一个近乎连续的状态空间，其中包括球拍位置、弹力球位置和速度及剩余砖块位置。



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。

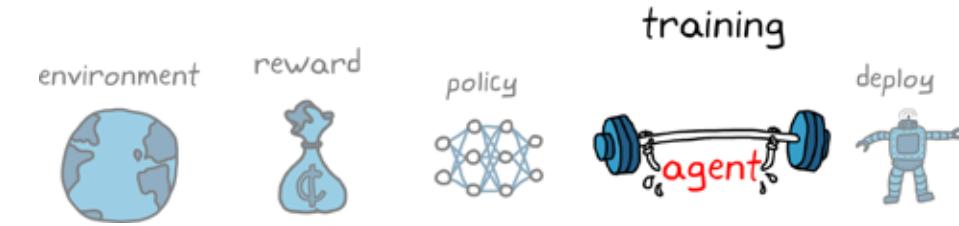


智能体是否应该利用环境，选择已知可以得到最多奖励的那些动作？还是应该选择探索环境中仍然未知部分的动作？

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。

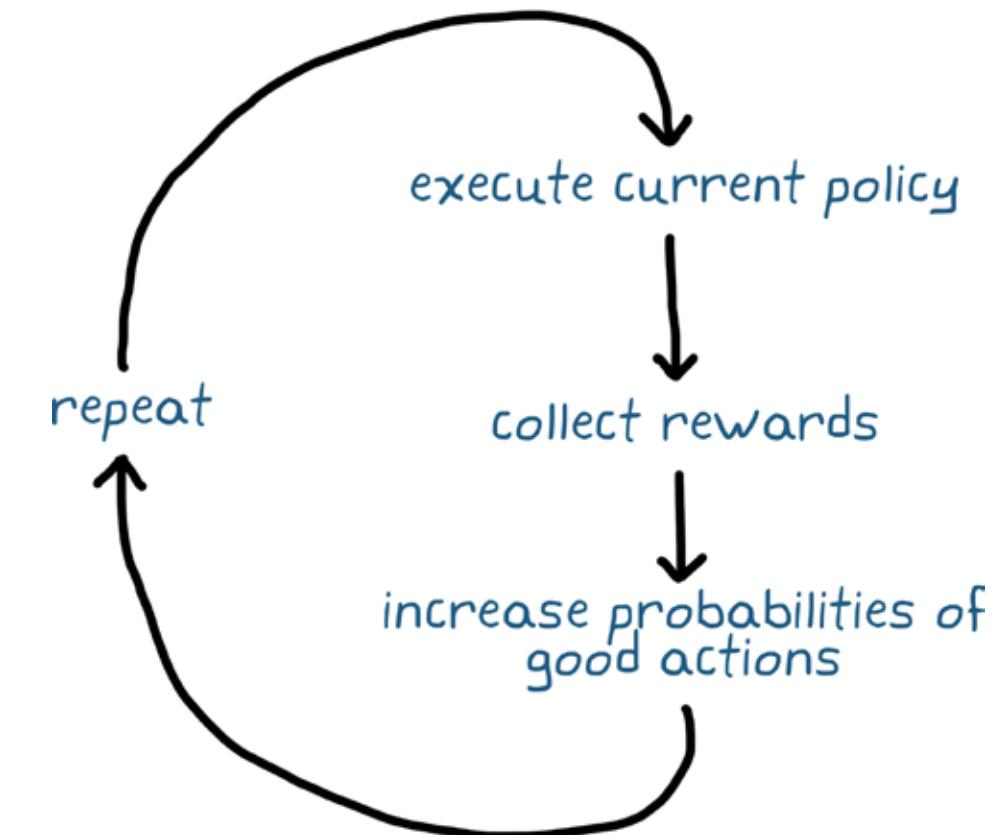
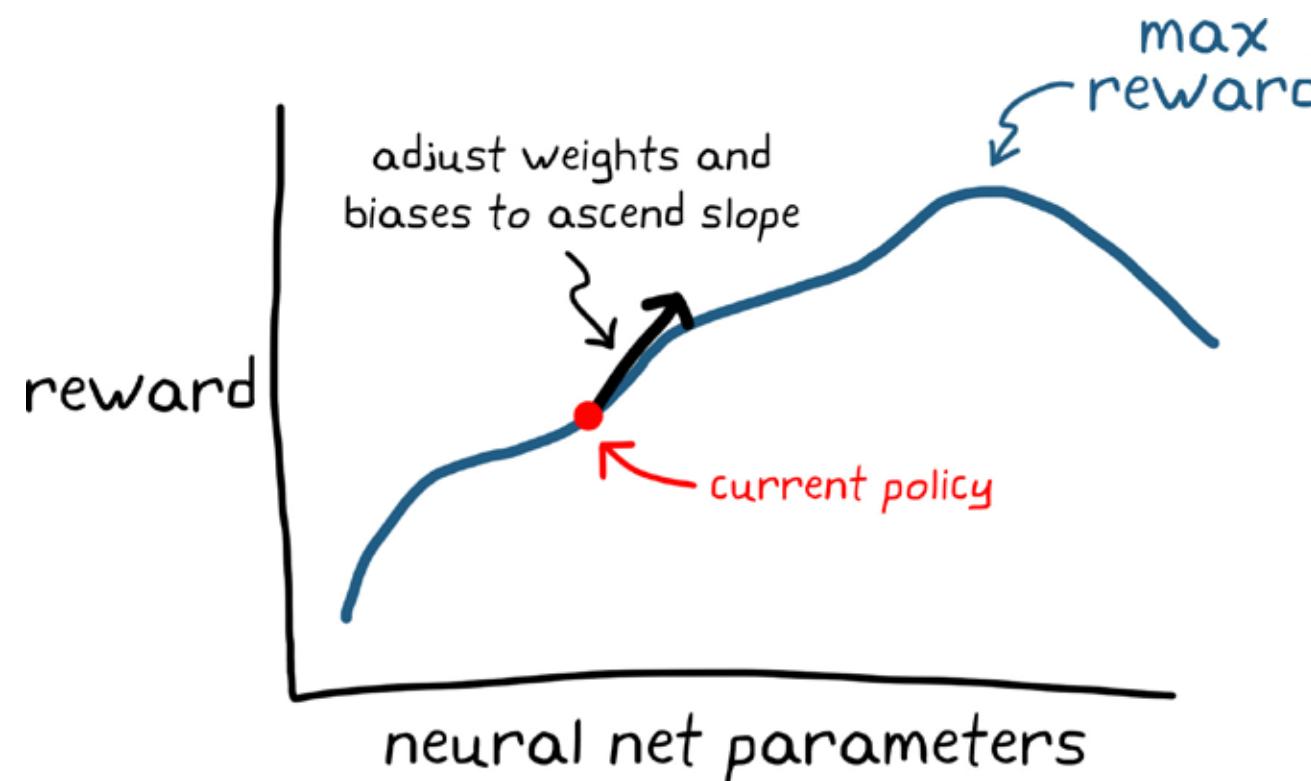
久而久之，智能体将沿回报最高的方向微调三个动作的概率。最终，每个状态对应的最有利动作的概率将最高，从而智能体将始终采取该动作。

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。

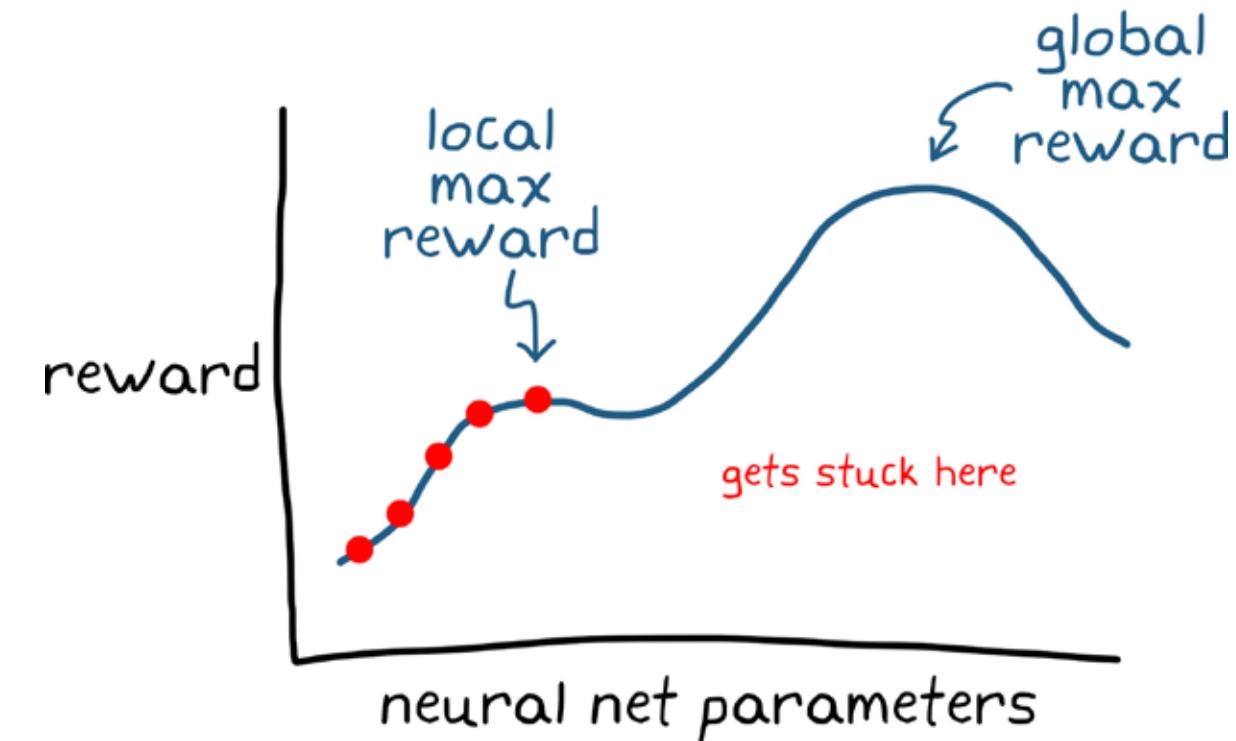
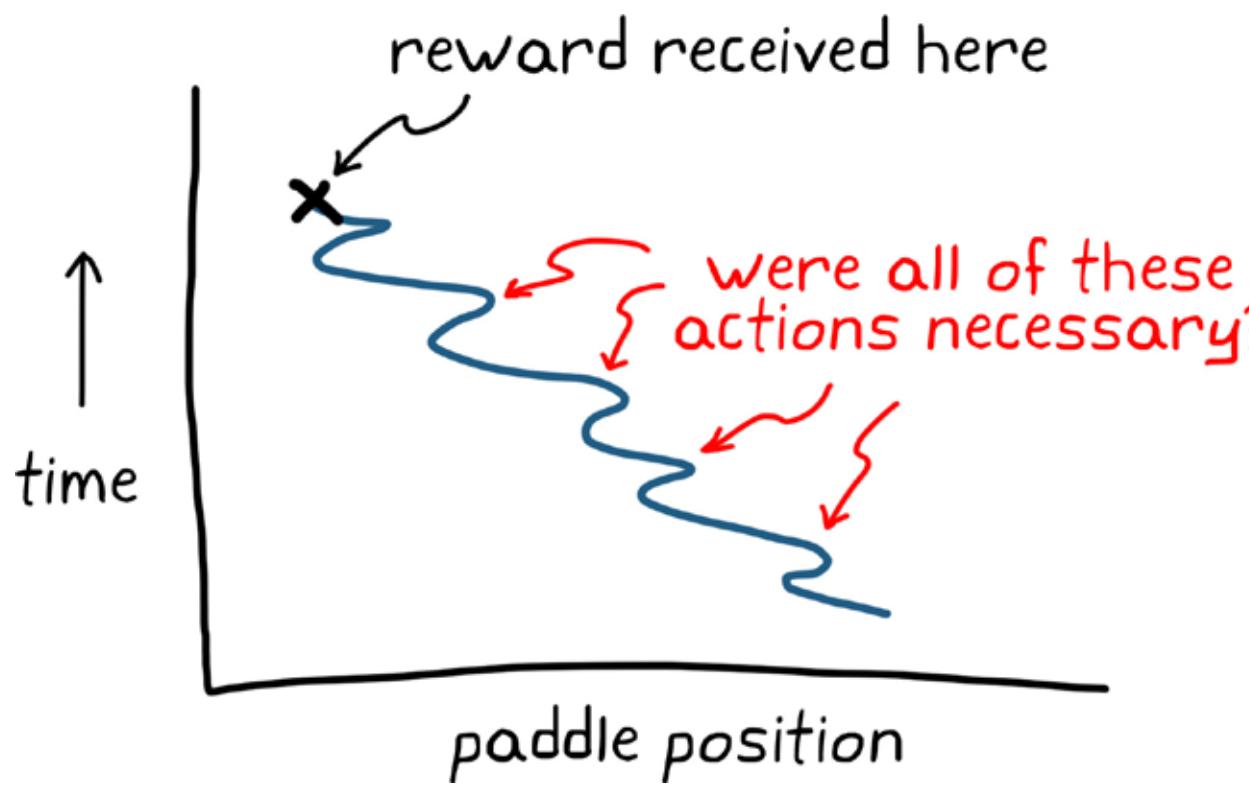
如果左移球拍未接住球并生成负奖励,则更改神经网络,提高智能体下次处于该状态时右移球拍的概率。



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。

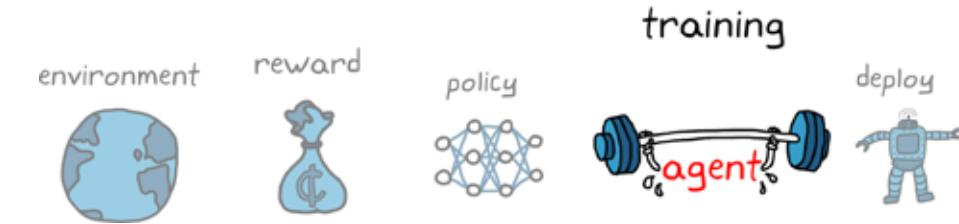
策略梯度法的缺点

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。

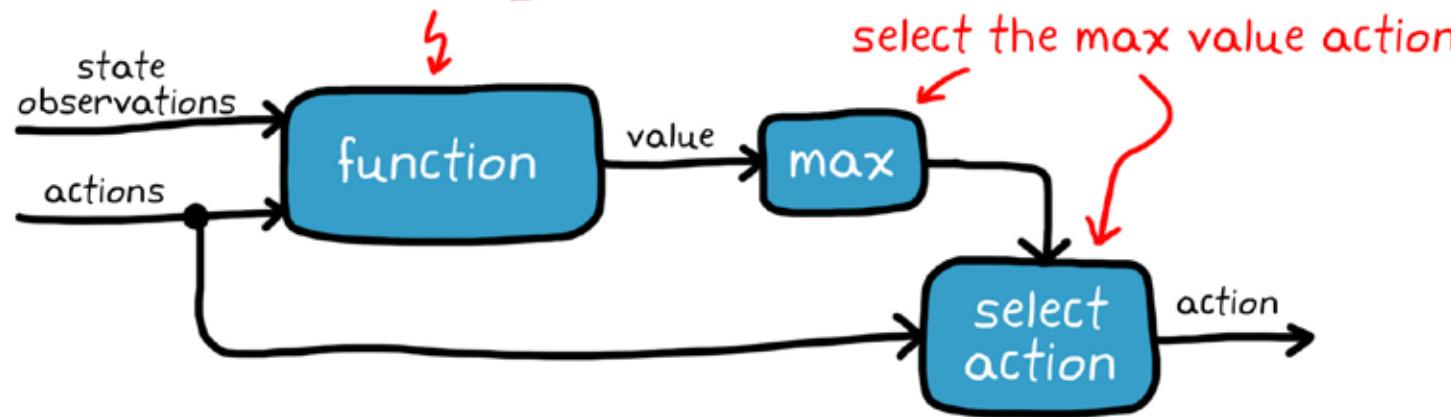
基于价值函数的学习



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。

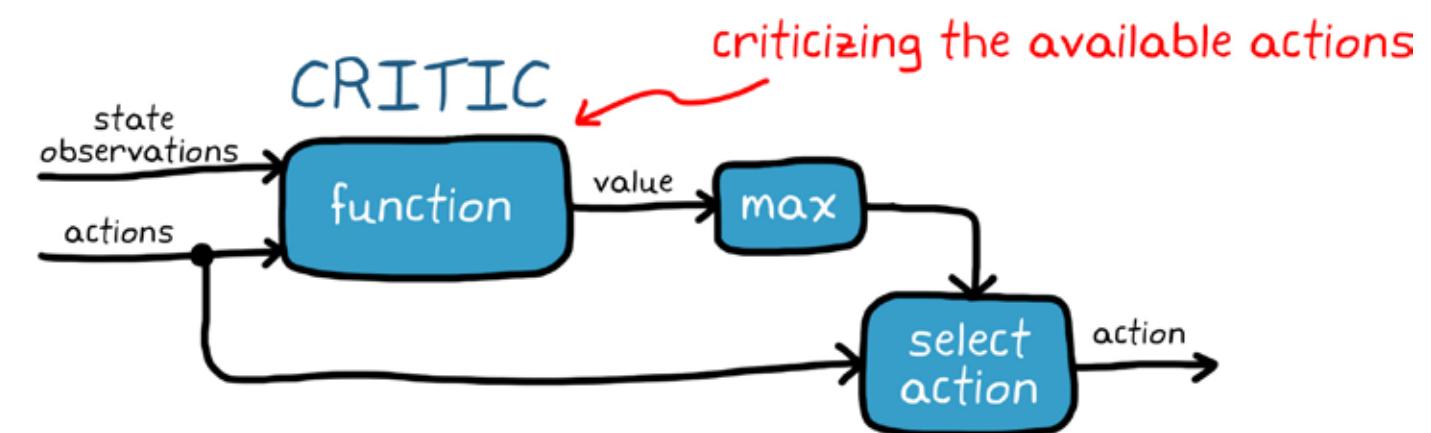
what is the current state?
value = function(state observations, action)
how good is the action from this state?

check the value of every action

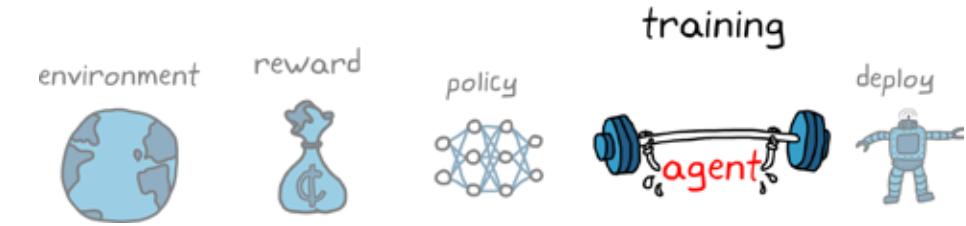


单纯凭借这个函数不足以表示策略，因为它输出的是价值，而策略需要输出动作。因此，策略是使用此函数检查给定状态下的每个可能动作的价值，然后选择具有最高价值的动作。

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。



价值函数与网格世界



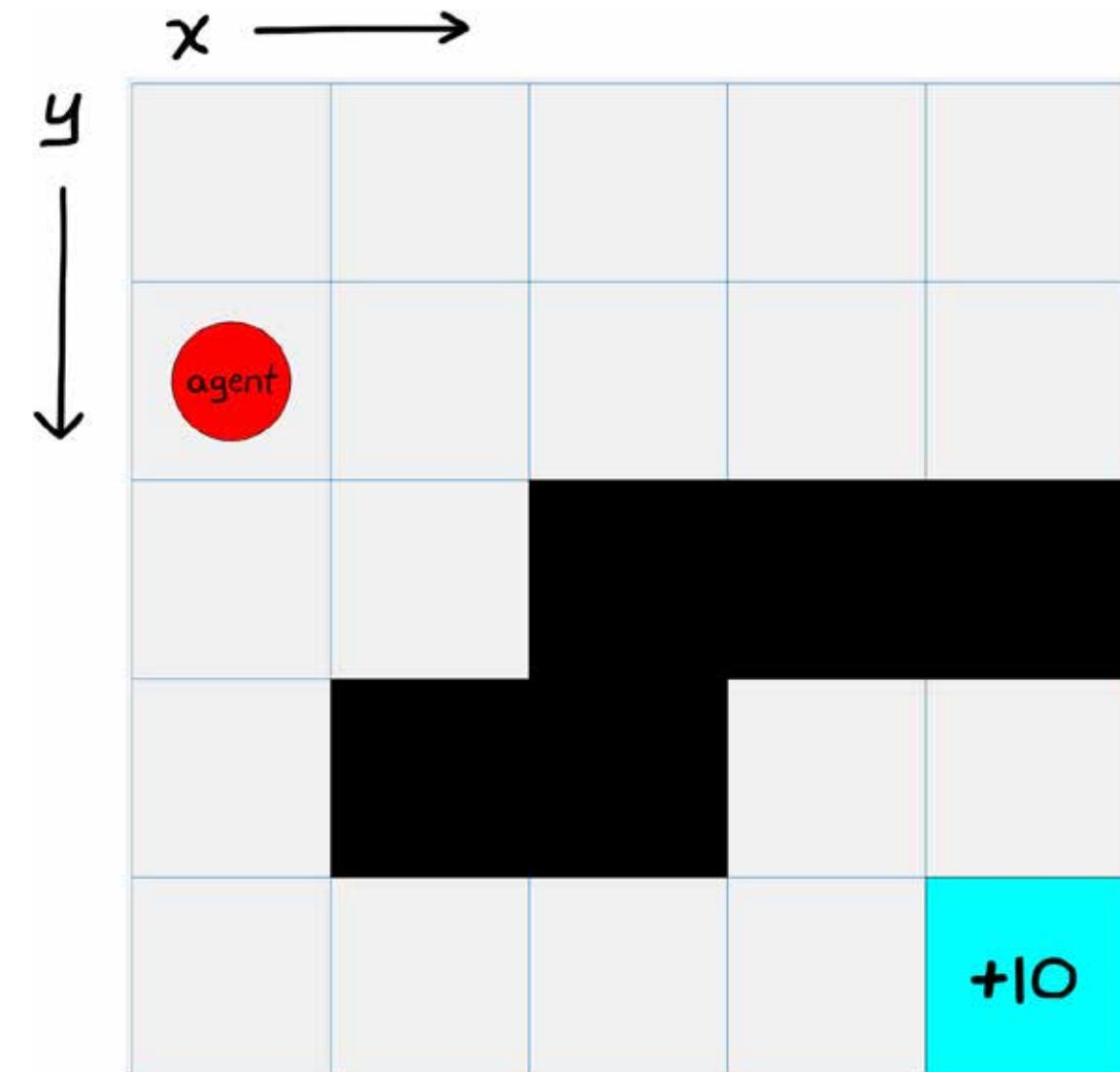
为了了解该函数是如何实际工作的,我们来看一个使用网格世界环境的示例。

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。

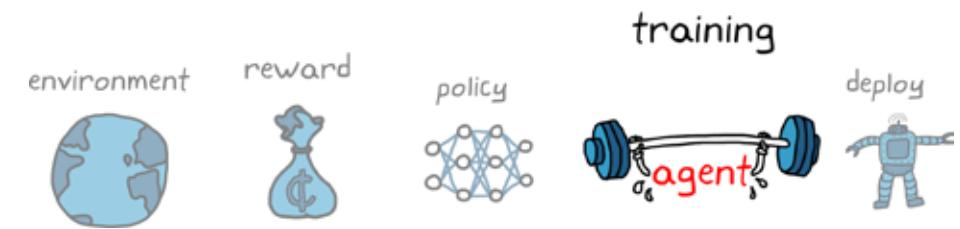
如果智能体试图脱离网格或进入黑色障碍区,则不会进入新状态,但仍会产生 -1 奖励。这样,智能体会因碰壁而受到惩罚,也不会取得任何实质性进展。

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](#)

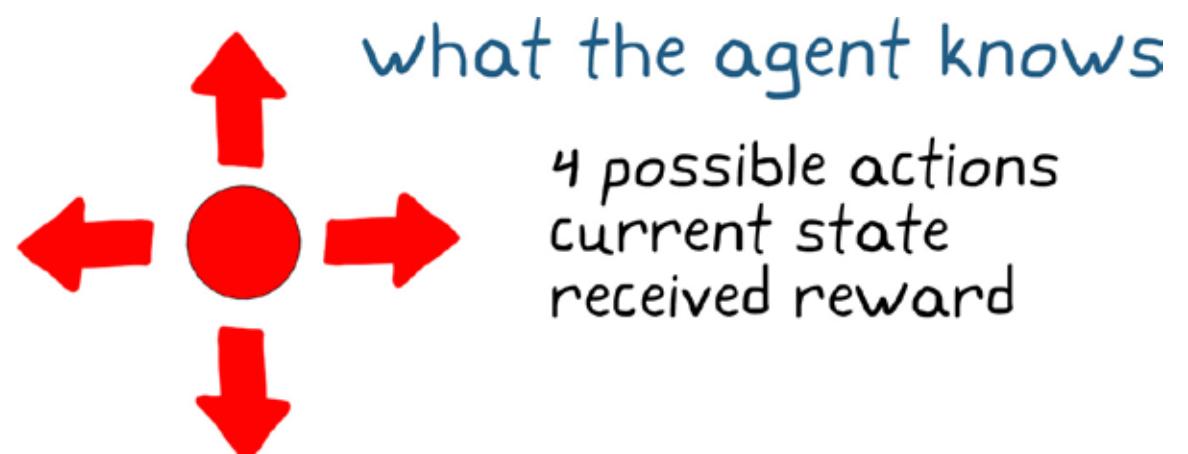
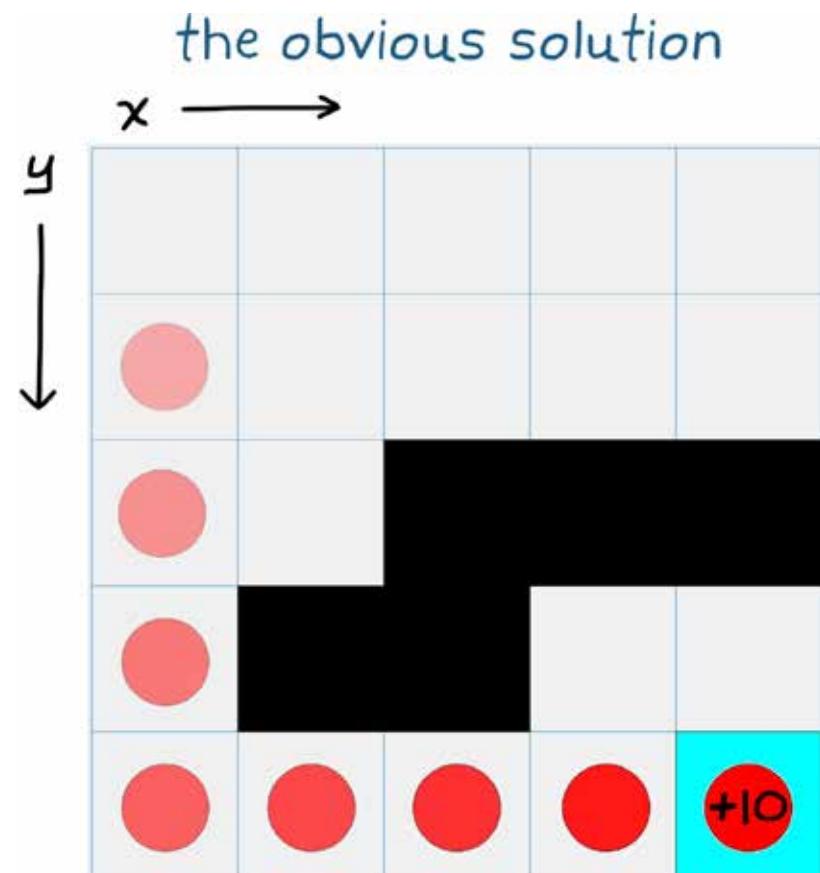


价值函数与网格世界 (续)

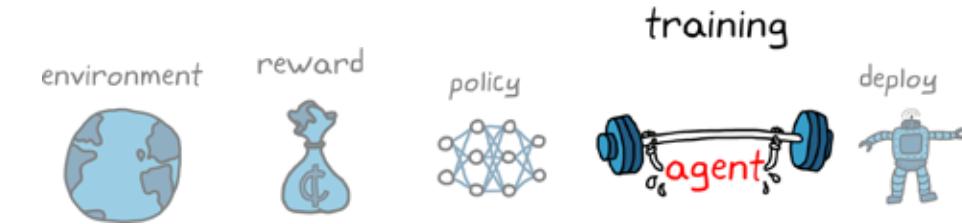


确定走哪一条路线才能得到奖励看起来可能很容易。

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。

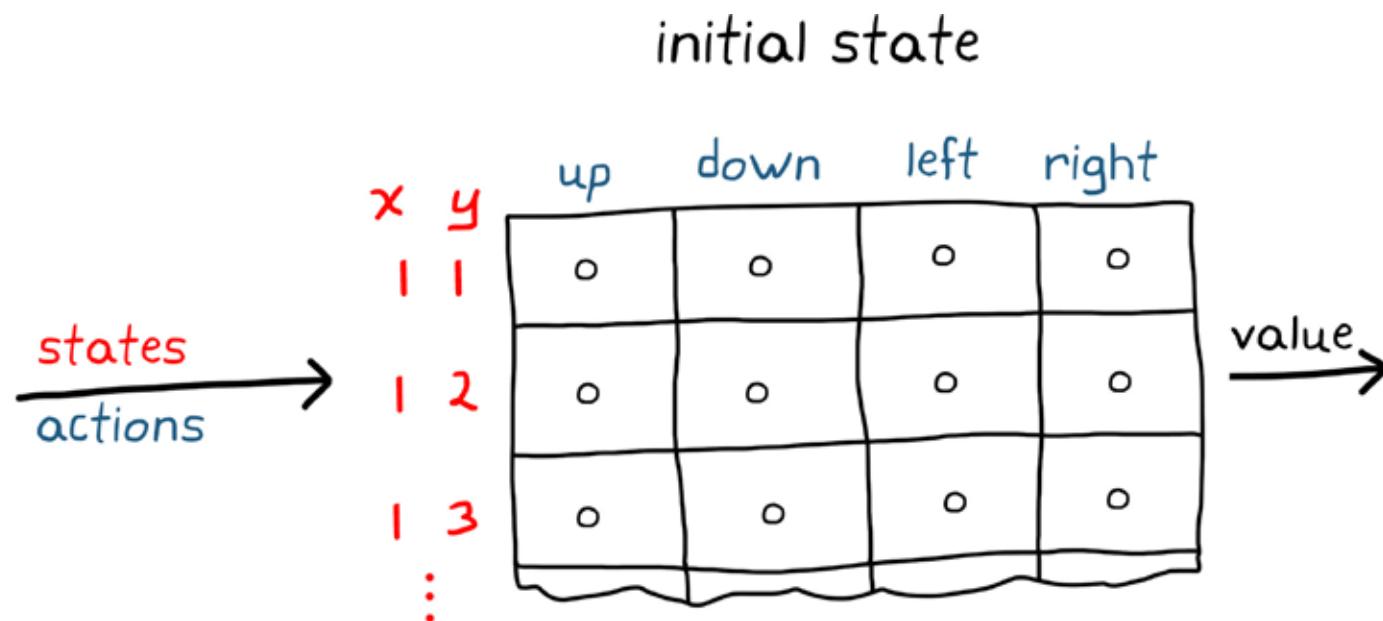
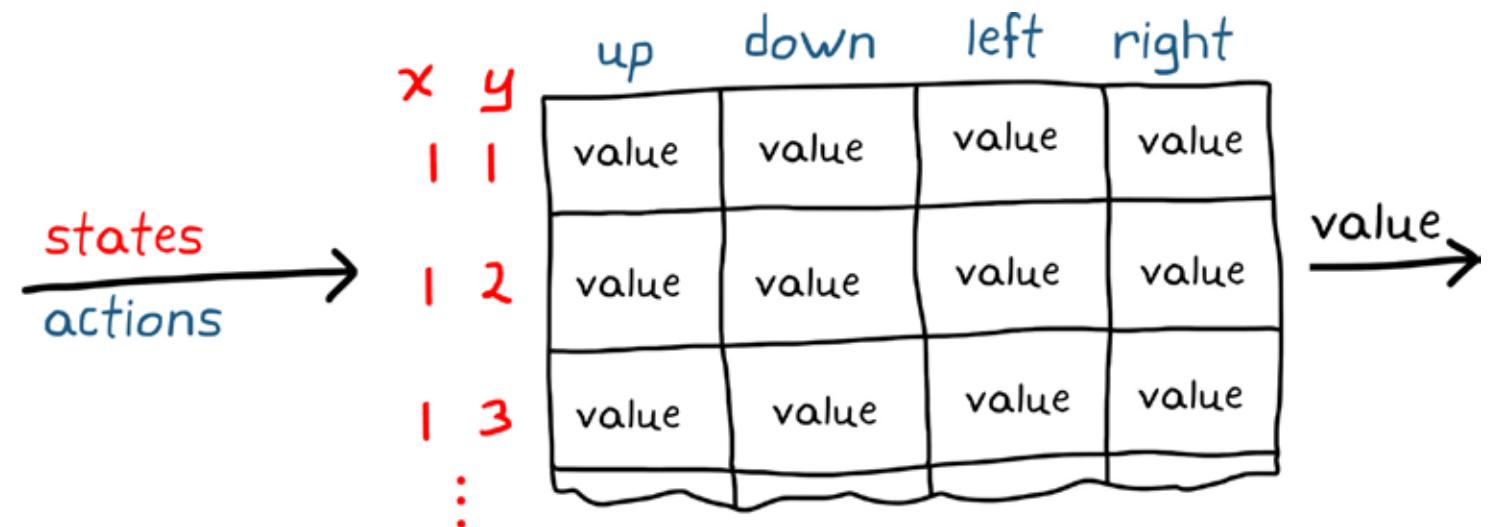


使用 Q-table 求解网格世界



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。

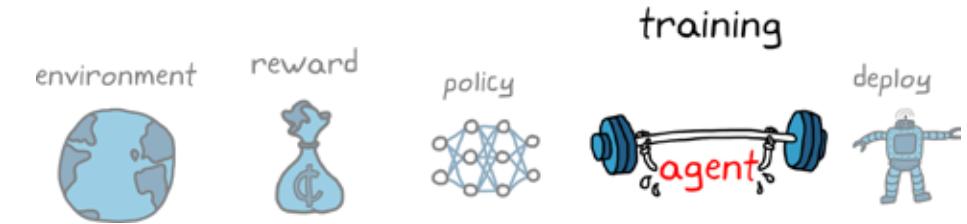
那么，智能体如何认识到这些价值呢？通过所谓的 Q-learning 过程。



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。

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

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

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。

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

↑
maximum expected value from state, s'

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。

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

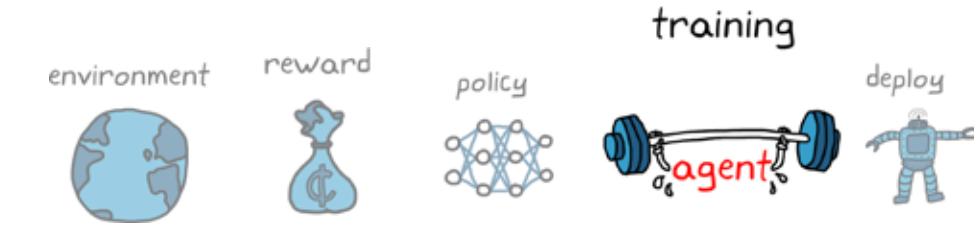
↑
discount future rewards

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。



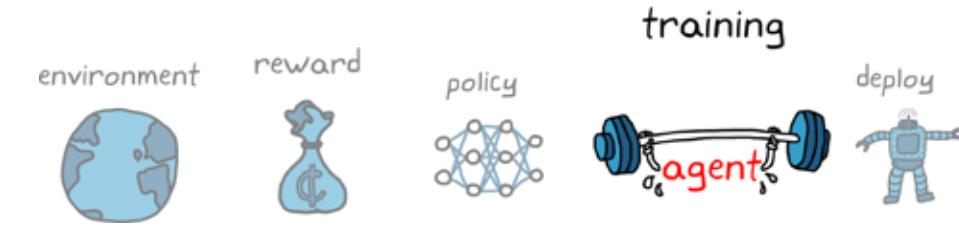
$$\text{new } Q(s, a) = Q(s, a) + \alpha \left[\underbrace{R(s, a) + \gamma \cdot \max Q'(s', a')}_{\text{new best estimate of value}} - \overbrace{Q(s, a)}^{\text{previous estimate of value}} \right]$$

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

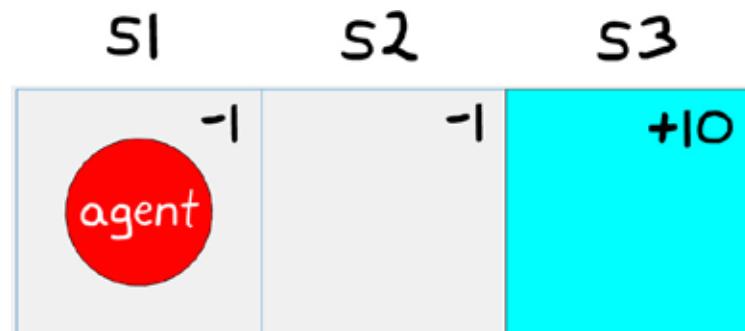
$$\text{new } Q(s, a) = \underbrace{Q(s, a)}_{\uparrow \text{delta value is added to old estimate}} + \alpha \left[R(s, a) + \gamma \cdot \max Q'(s', a') - Q(s, a) \right]$$

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。

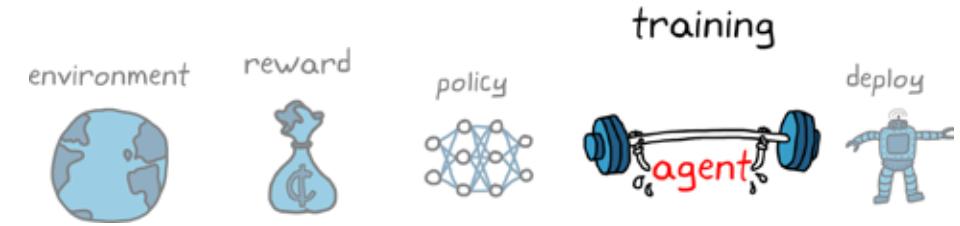


片段	步骤	状态	当前 Q(s, a)	动作	R(s, a)	新 Q(s, a)																								
1	1	S1	<table border="1"><tr><td></td><td>S1</td><td>S2</td><td>S3</td></tr><tr><td>左</td><td>0</td><td>0</td><td>0</td></tr><tr><td>右</td><td>0</td><td>0</td><td>0</td></tr></table>		S1	S2	S3	左	0	0	0	右	0	0	0	右 (随机)	-1	<table border="1"><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																											
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																											

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)

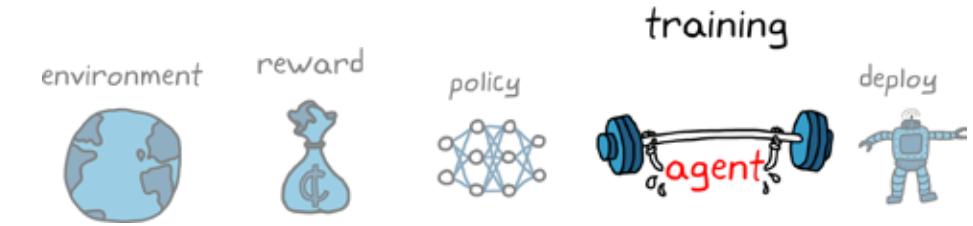


片段	步骤	状态	当前 Q(s, a)	动作	R(s, a)	新 Q(s, a)																								
2	1	S1	<table border="1"> <thead> <tr> <th></th><th>S1</th><th>S2</th><th>S3</th></tr> </thead> <tbody> <tr> <td>左</td><td>0</td><td>0</td><td>0</td></tr> <tr> <td>右</td><td>-1</td><td>10</td><td>0</td></tr> </tbody> </table>		S1	S2	S3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

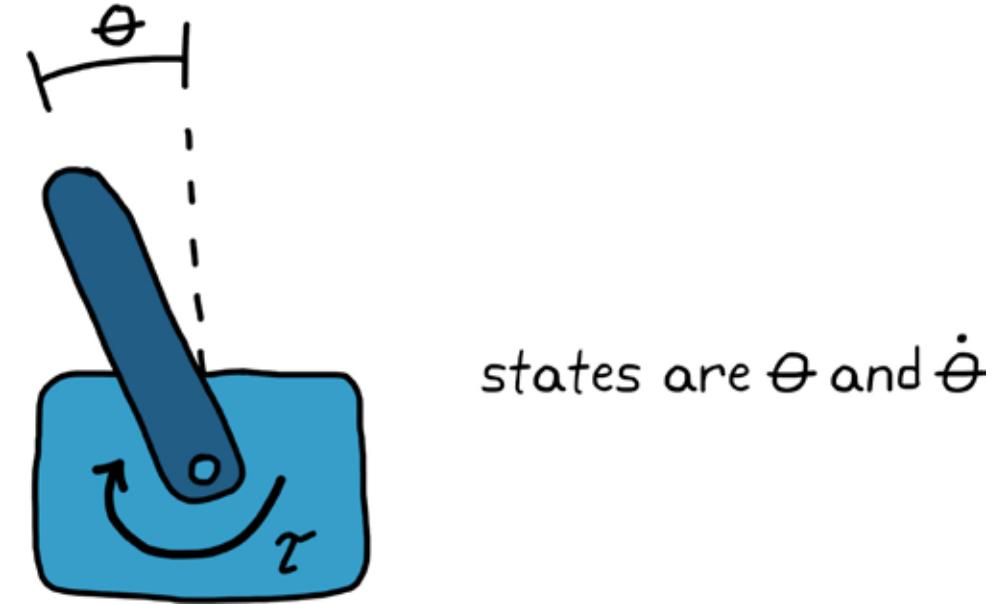
片段结束

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。

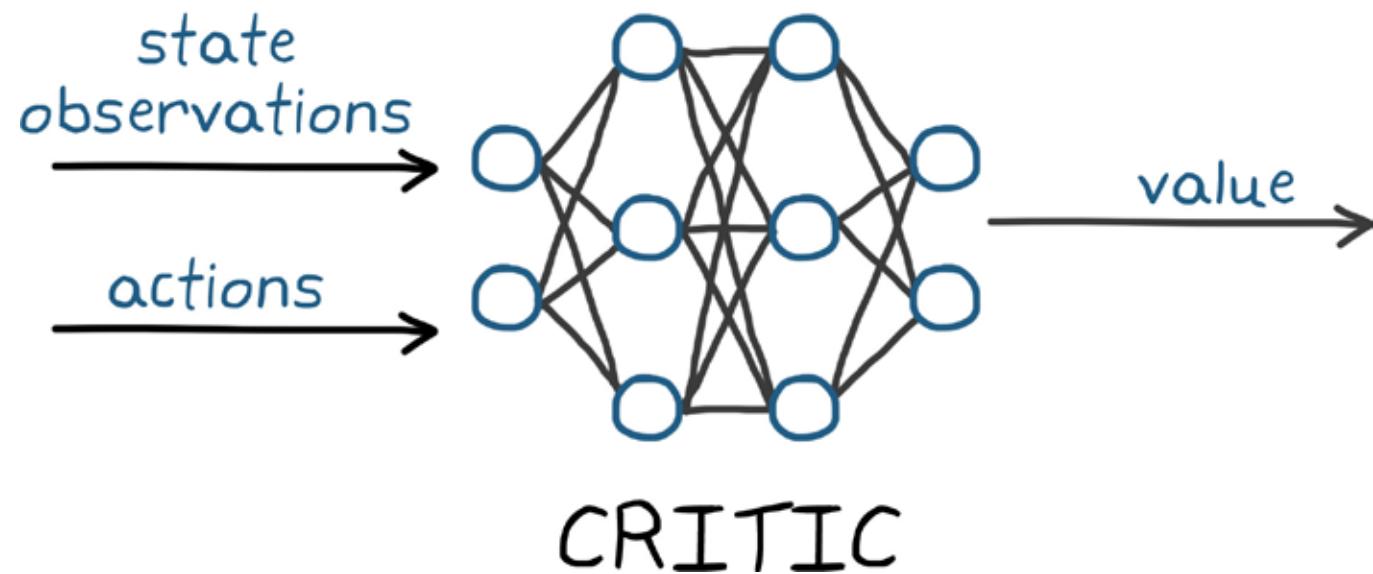
基于神经网络的评价器



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



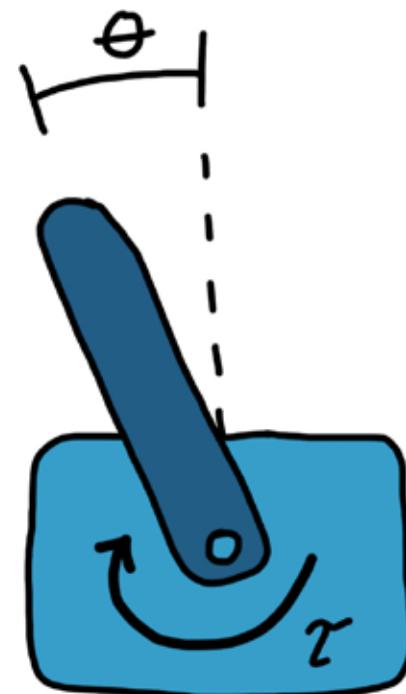
用神经网络表示价值函数（评价器）。该构想与使用表格相同：输入状态观测量和动作，神经网络返回状态/动作对的价值，策略将选择具有最高价值的动作。

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

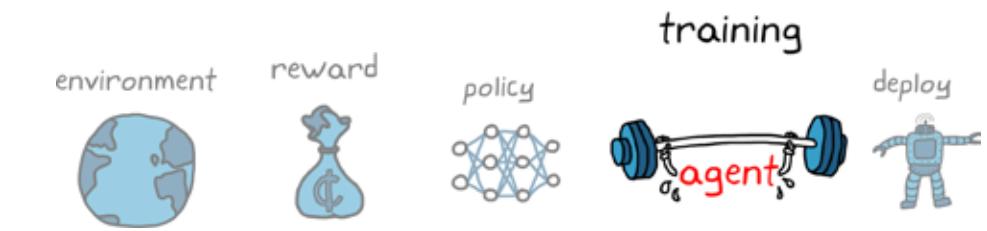
您可以使用神经网络为连续状态空间定义价值函数。如果倒立摆具有离散动作空间，则可逐一将离散动作馈送至评价器网络。

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: $\mathcal{T} = [-2, -1, 0, 1, 2] \text{ Nm}$

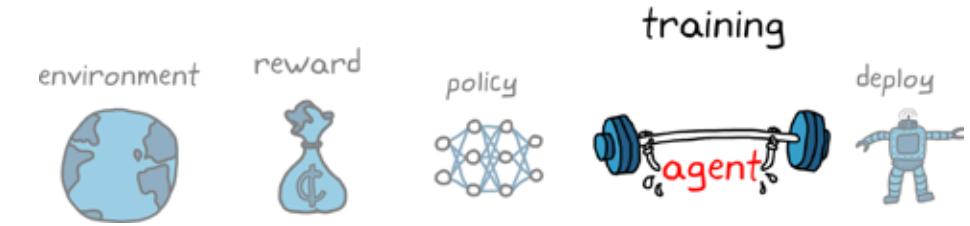


那么, 您可以怎么做?

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。

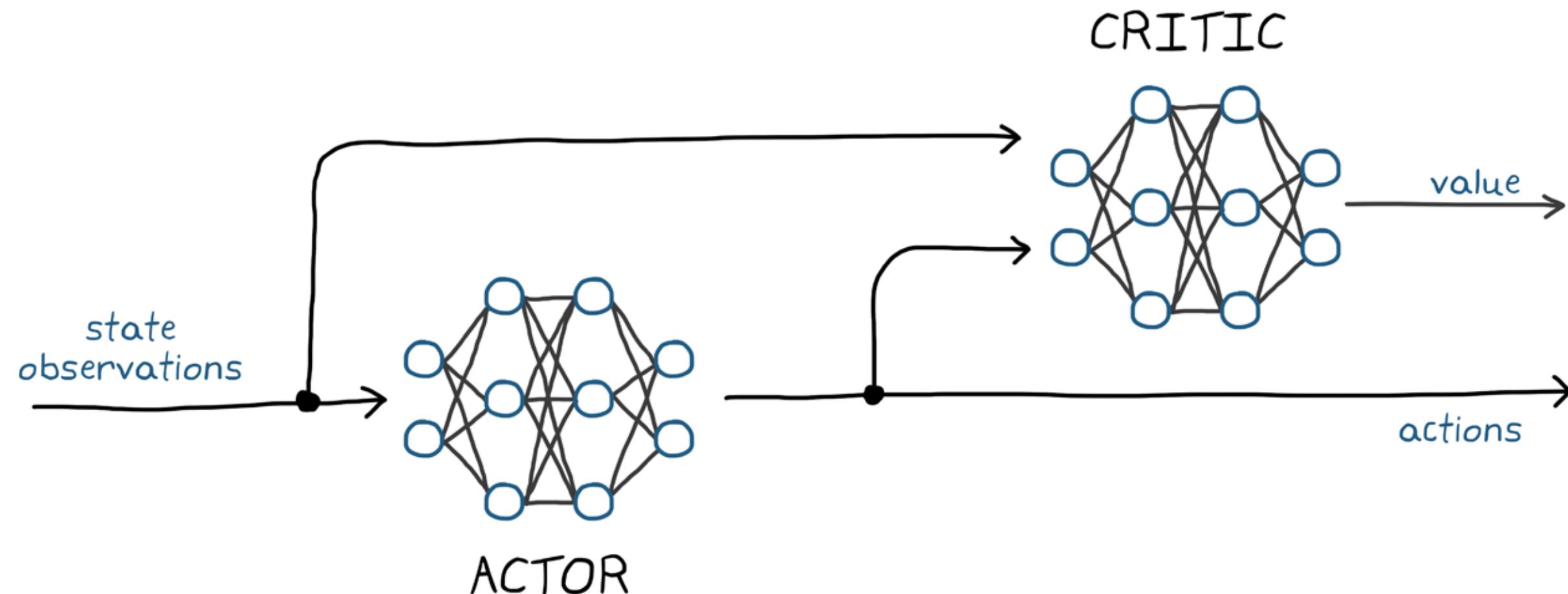
» [了解如何在 MATLAB 中训练执行器-评价器智能体保持倒立摆平衡](#)

执行器-评价器方法

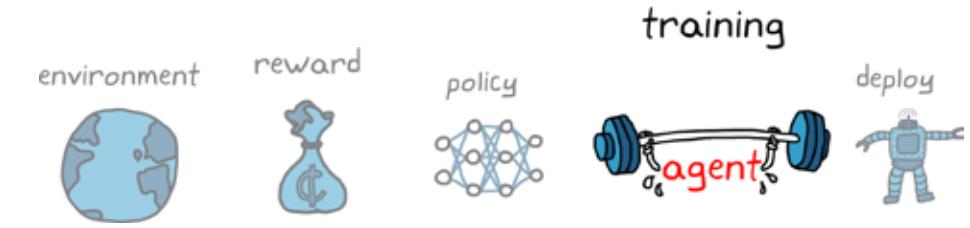


执行器是一个网络，用于在当前状态给定的情况下尝试采取自认为最佳的动作，如策略函数方法所示。评价器是另一个网络，用于根据状态和执行器采取的动作估计价值，如价值函数方法所示。此方法适用于连续动作空间，因为评价器只需查看执行器采取的单个动作，无需评估所有动作来尝试确定最佳动作。

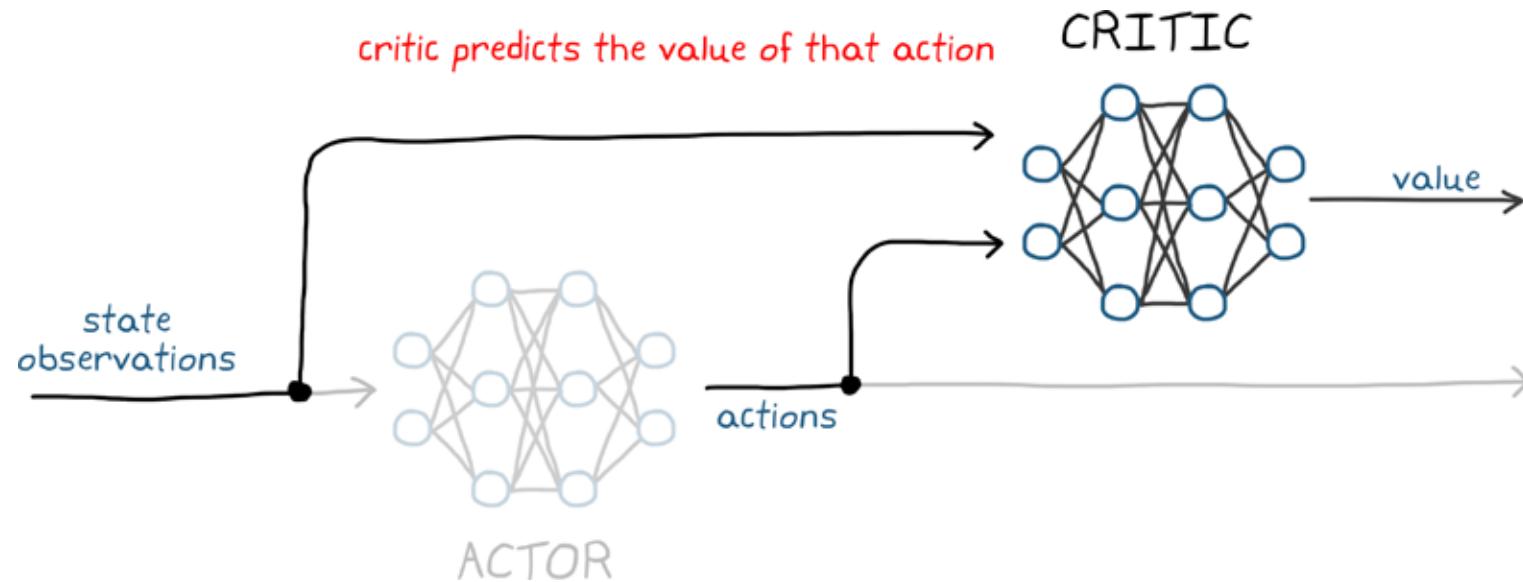
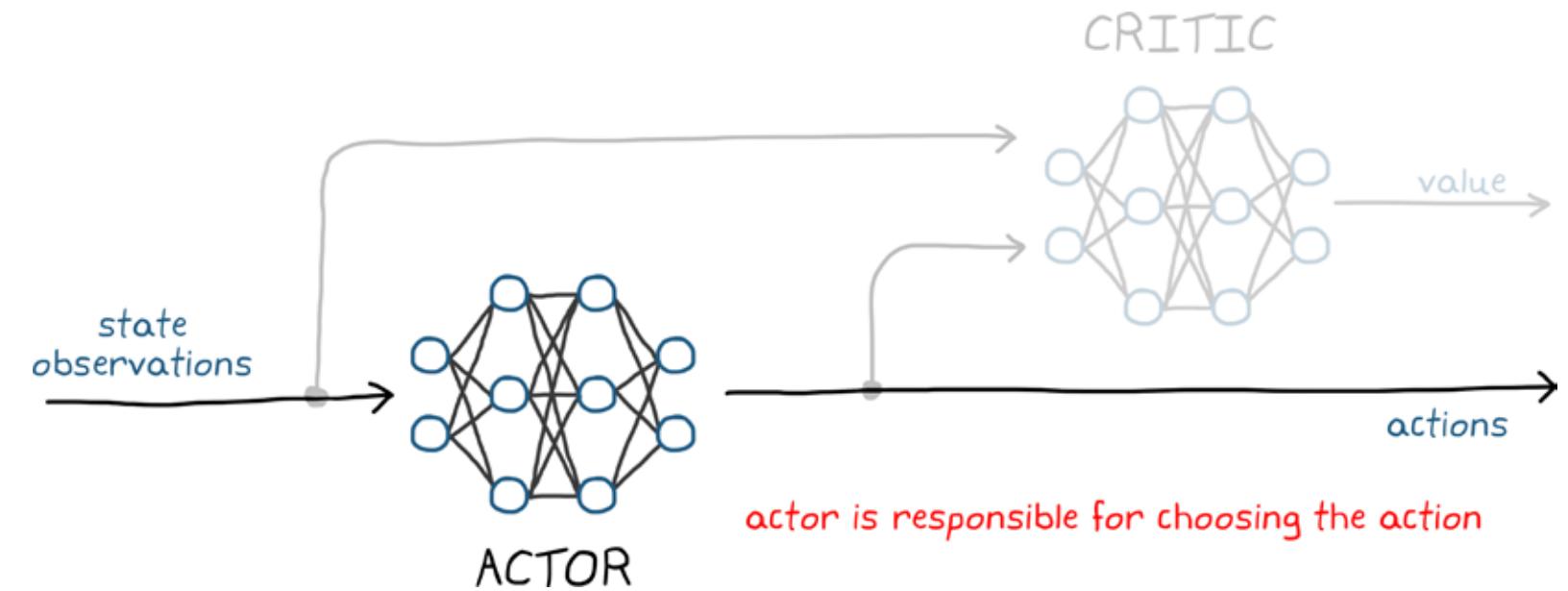
actor-critic learning algorithms



执行器-评价器学习周期

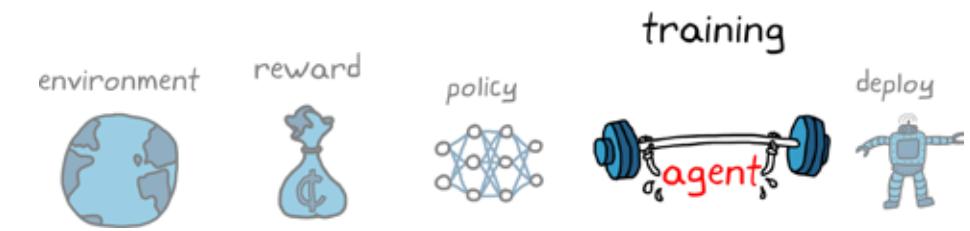


执行器采用与策略函数算法相同的方式选择动作，并将其应用于环境。

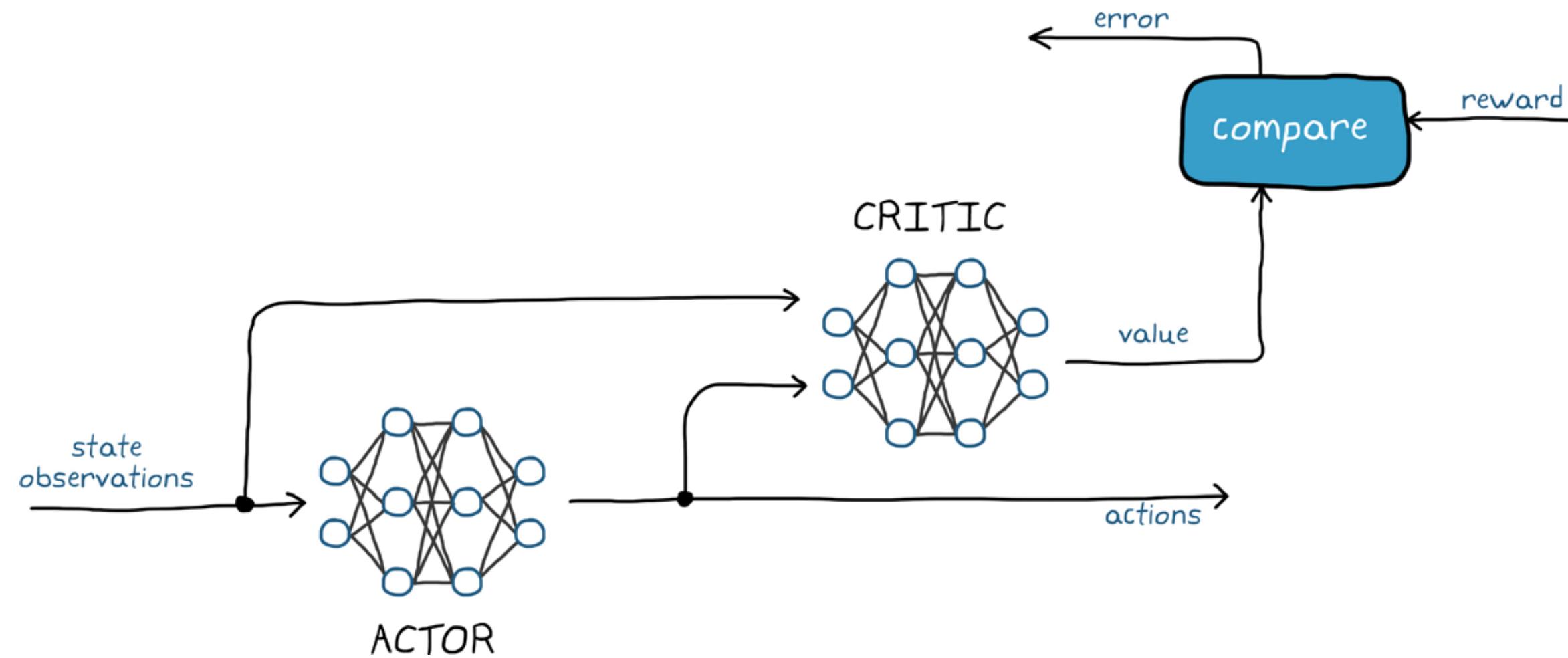


评价器根据当前状态和动作对预测相应动作的价值。

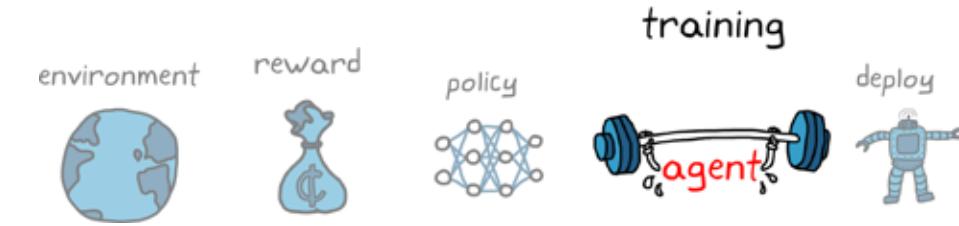
执行器-评价器学习周期 (续)



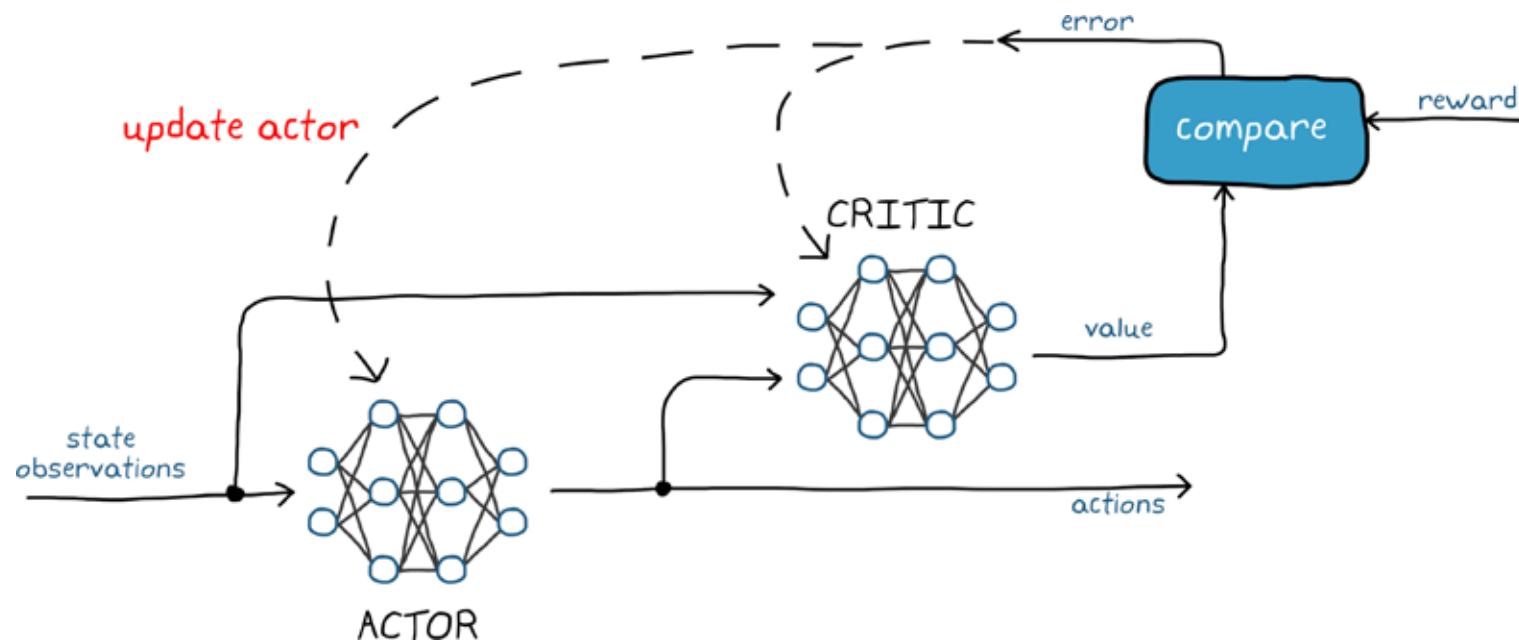
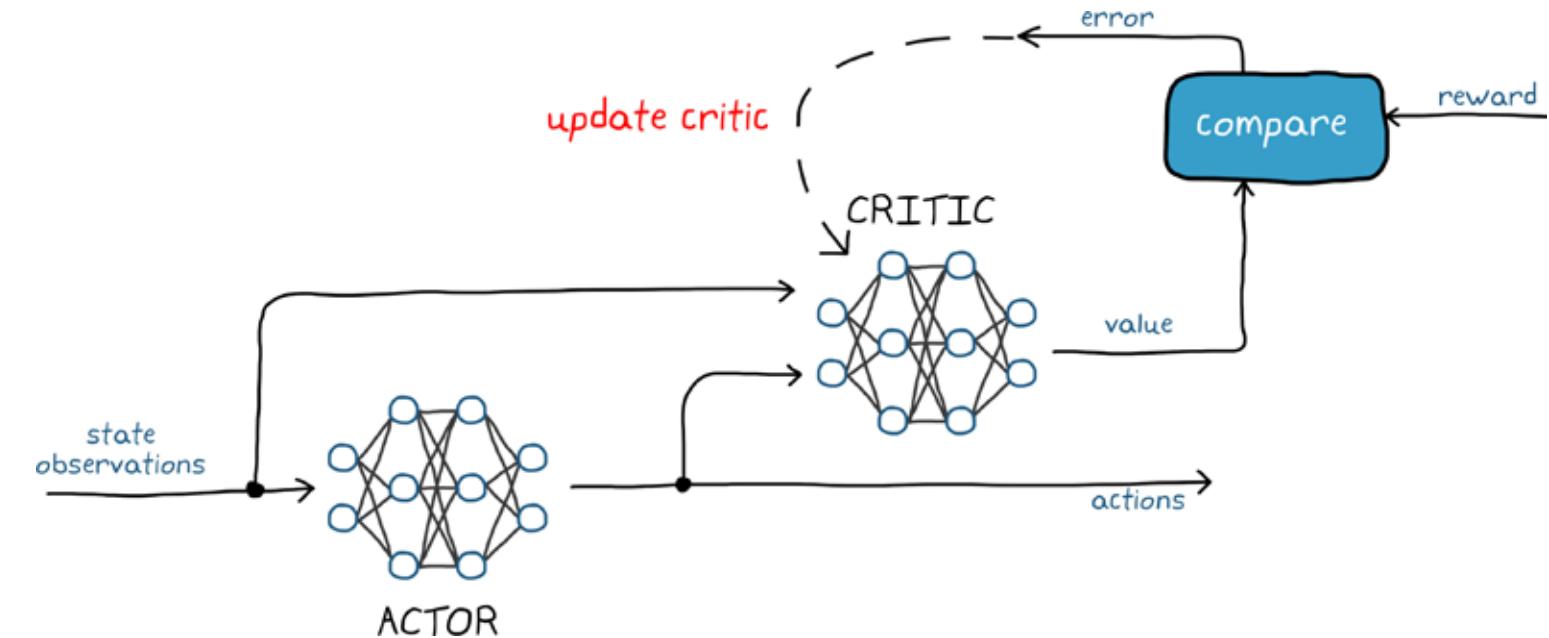
而后,评价器使用环境奖励确定其价值预测的准确性。误差是对前一状态的价值的新估计值和评价器网络给出的旧估计值之间的差异。价值的新估计值基于得到的奖励和当前状态的折扣价值得出。评价器可通过误差判断结果好于预期还是逊于预期。



执行器-评价器学习周期 (续)



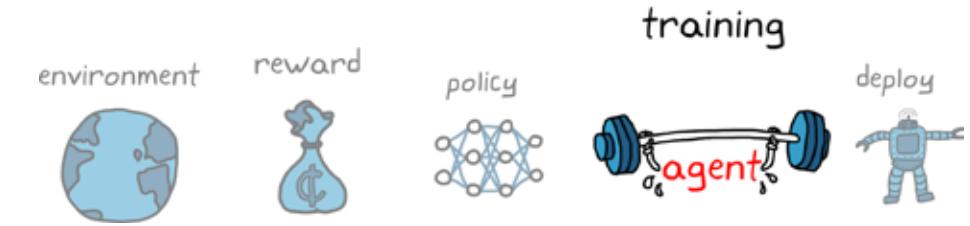
评价器使用此误差自我更新,采用的方式与价值函数完全相同,以便下次进入此状态时更好地进行预测。



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

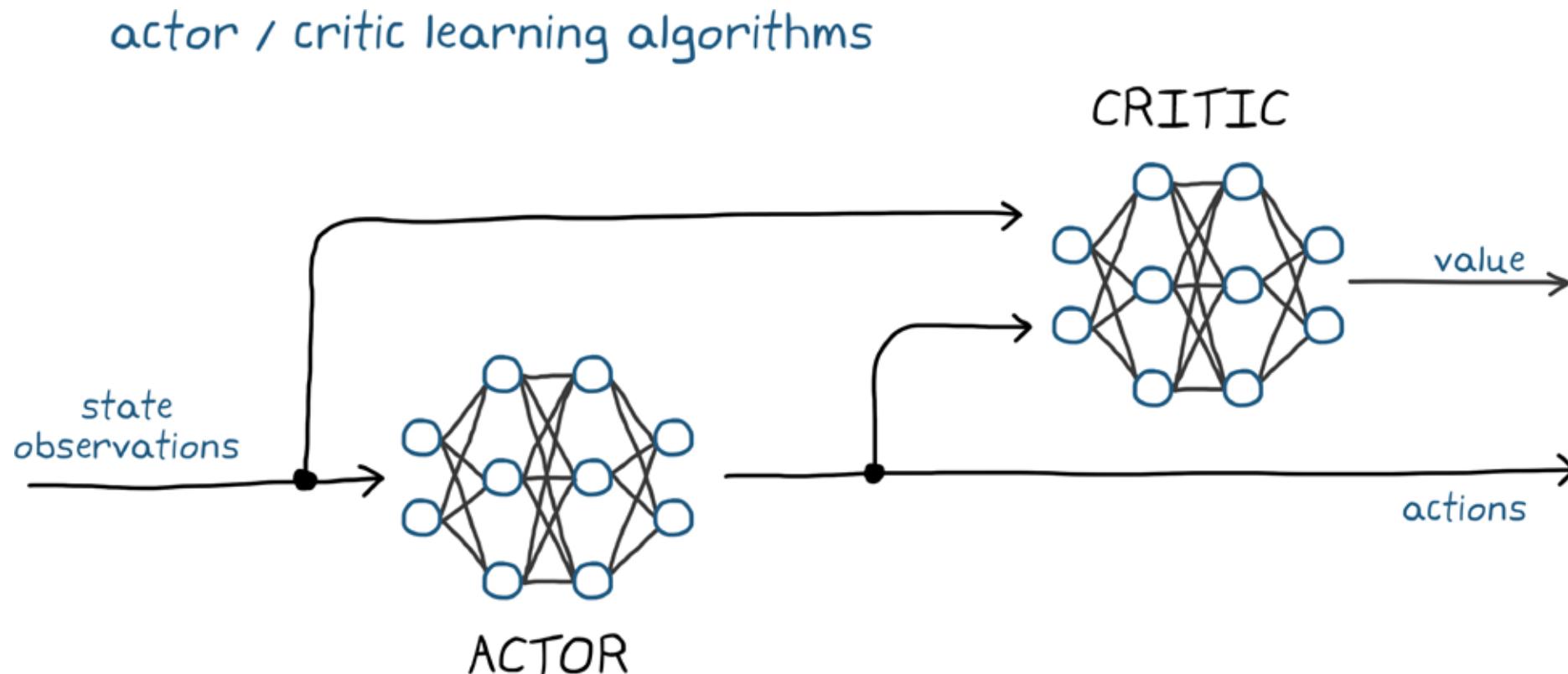


许多不同类型的学习算法使用执行器-评价器策略；本电子书对这些概念进行了归纳，不特定于某种算法。

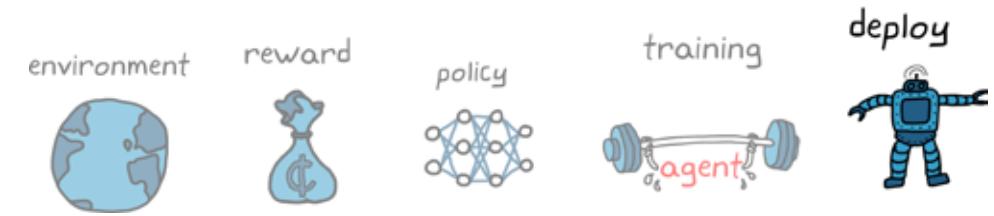
执行器和评价器均为尝试学习最优行为的神经网络。执行器使用评价器的反馈来确定动作好坏，从而学习需要采取的正确行为；评价器通过得到的奖励学习价值函数，以便正确评价执行器采取的动作。对于执行器-

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。

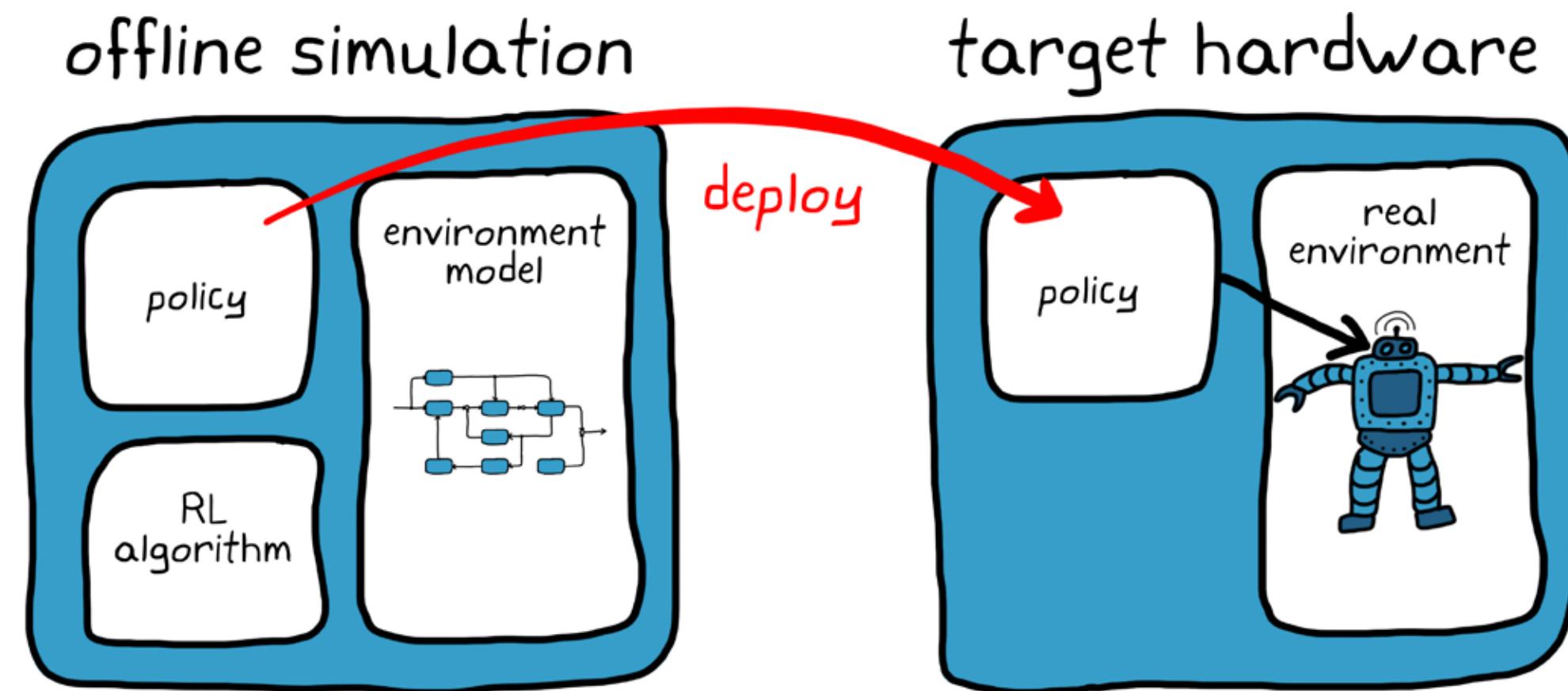


策略部署

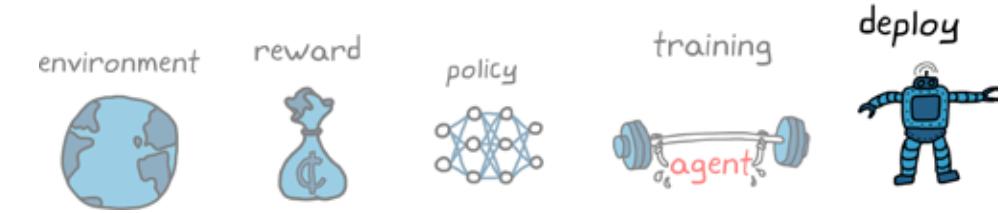


强化学习工作流程的最后一步是部署策略。

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。



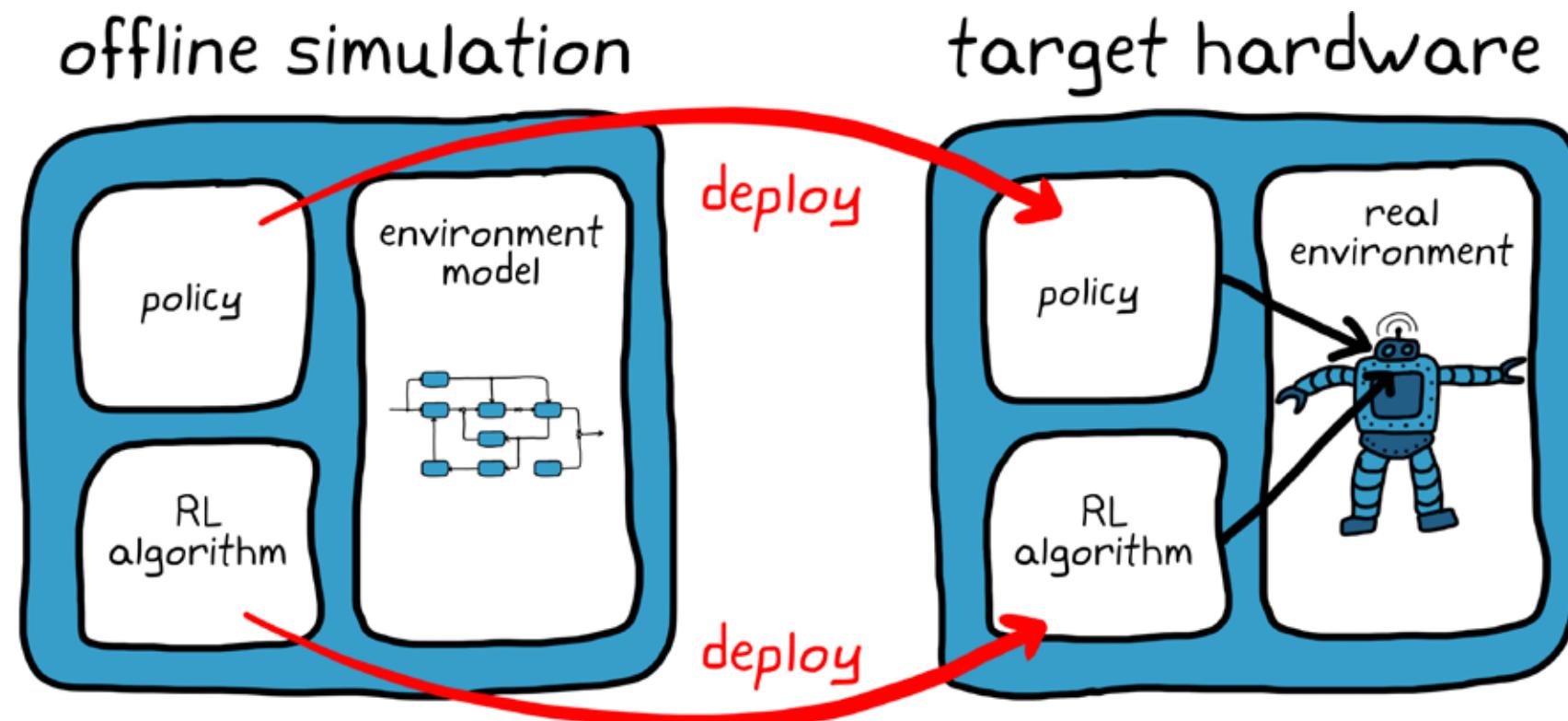
部署学习算法



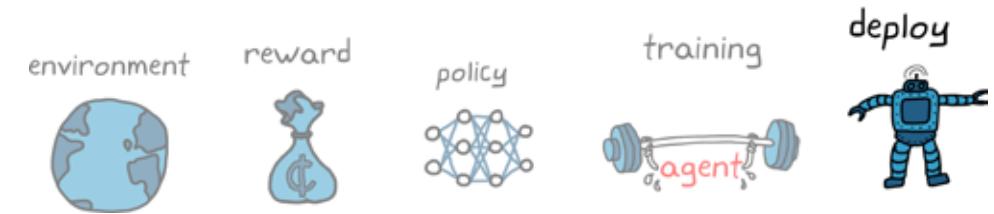
即便大部分学习在仿真环境下离线完成，部署后仍然可能需要使用真实物理硬件继续开展学习。

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。

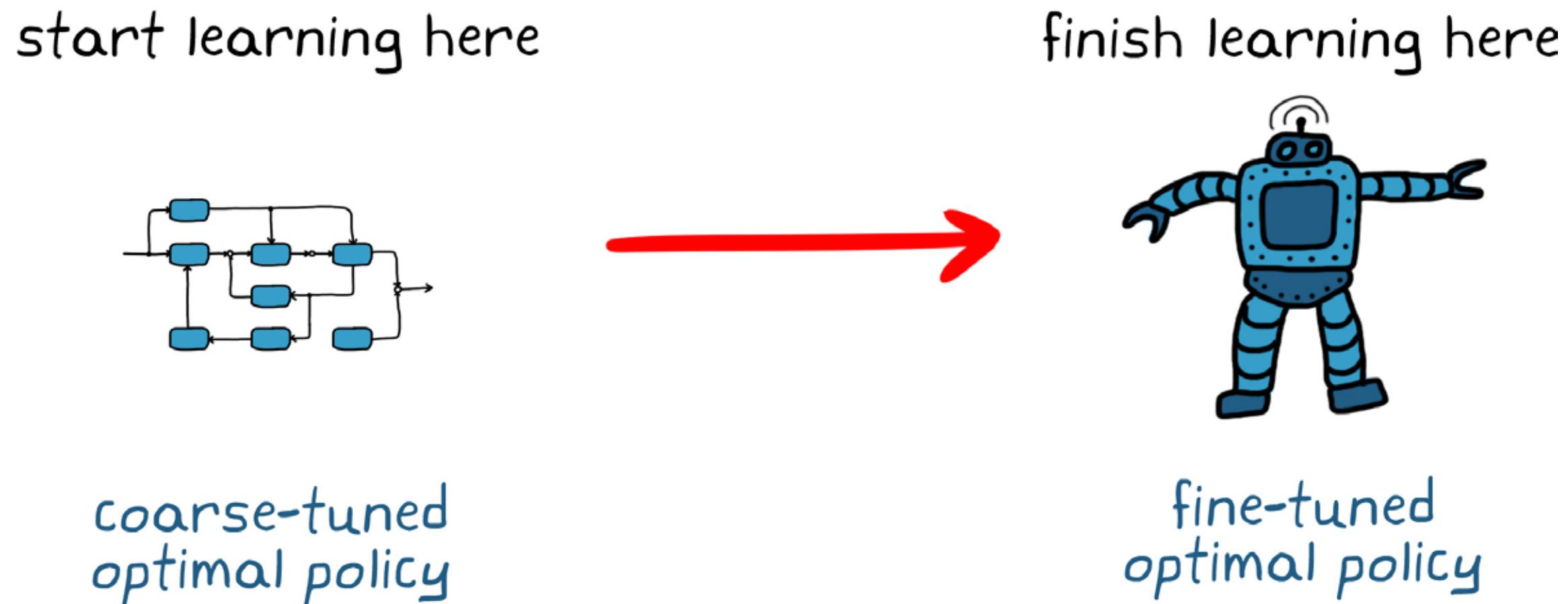
出于这些原因，您需要将静态策略和学习算法都部署到目标。对于此设置，您可以选择执行静态策略（停止学习）或继续更新策略（启动学习）。



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。

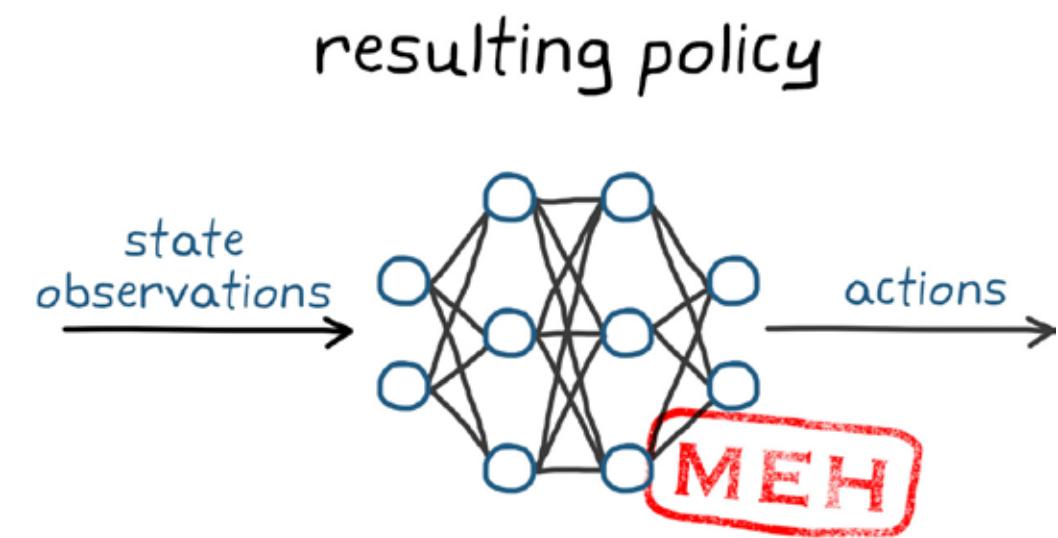
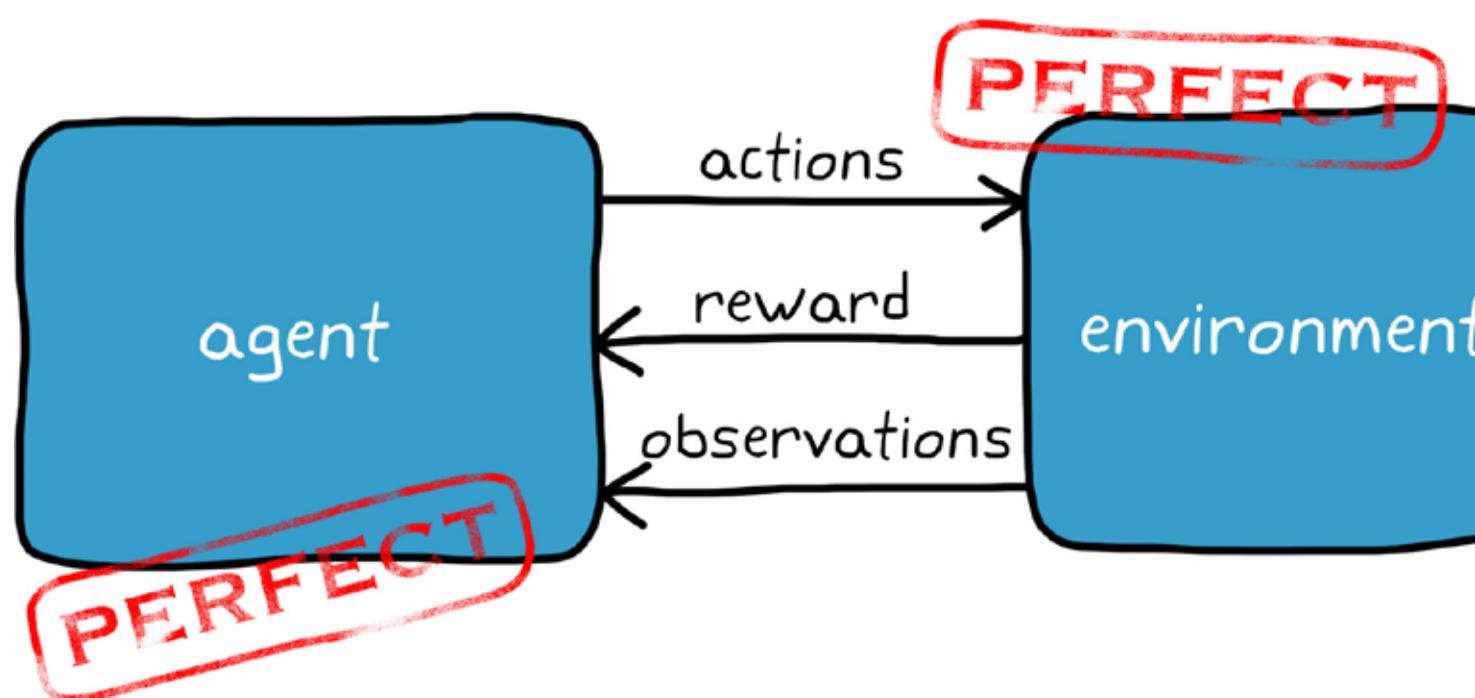


RL 的缺点

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。

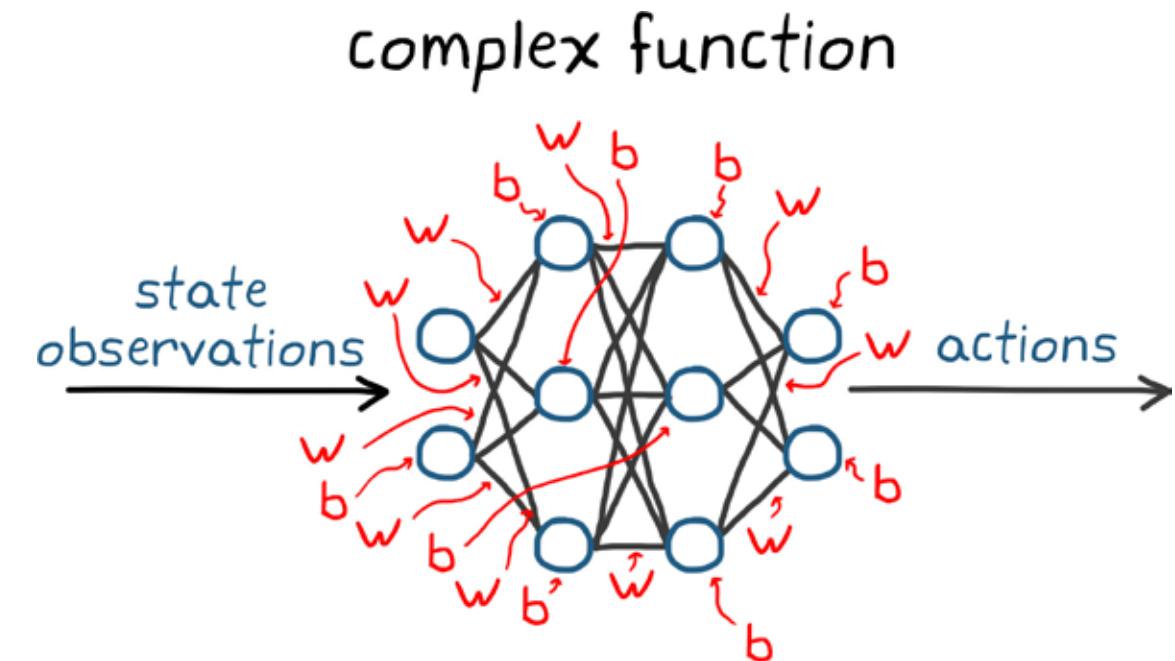
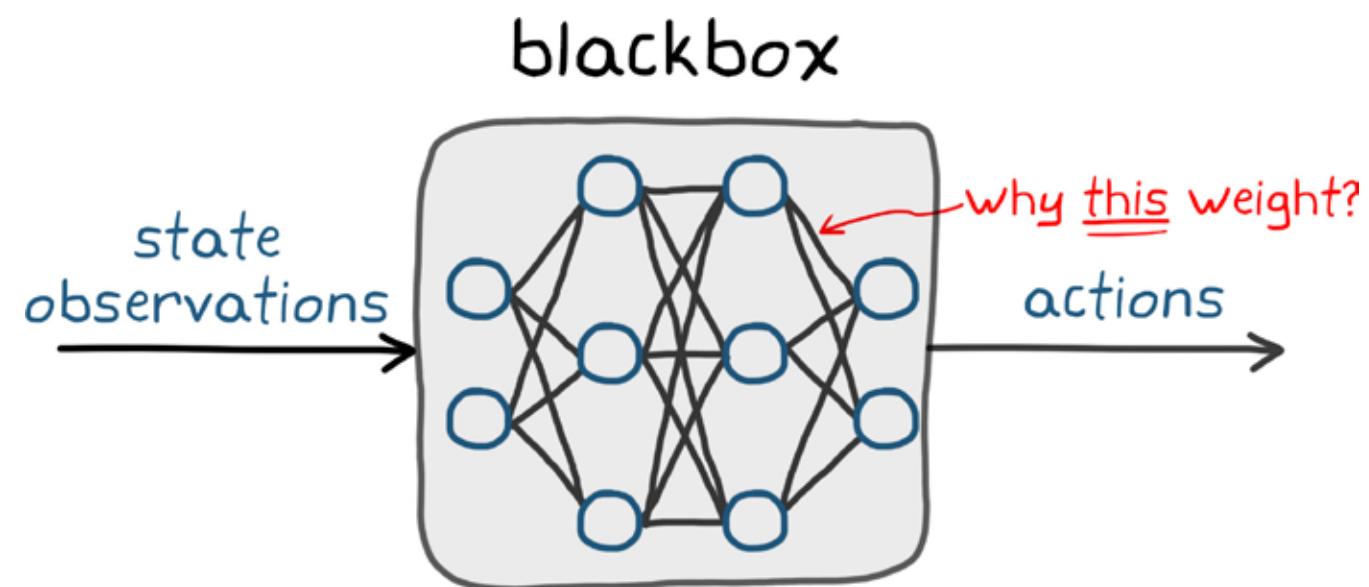
这些挑战可归结为两大问题:

- 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。



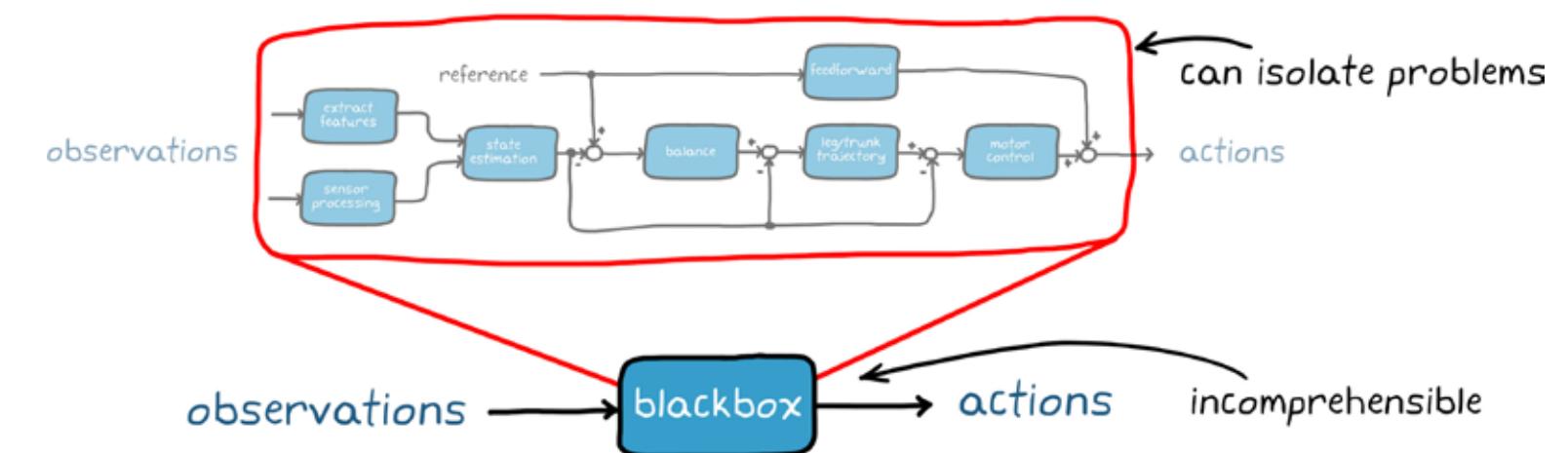
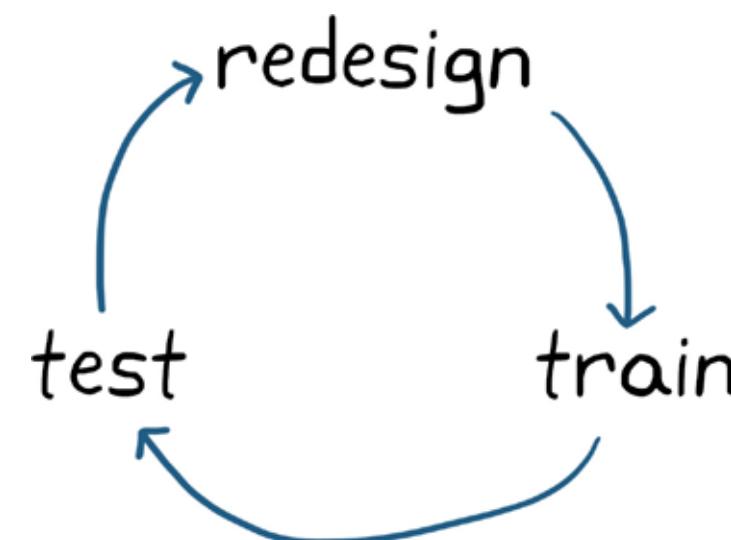
对于设计者来说,此函数是一个“黑盒”。您可能对此函数的运行原理及此网络已识别的隐藏特征具有直观的认识,但恐怕还不了解给定权重或偏置值背后的原因。因此,如果策略不符合规范或操作环境发生变化,势必不知如何调整策略才能解决问题。

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。

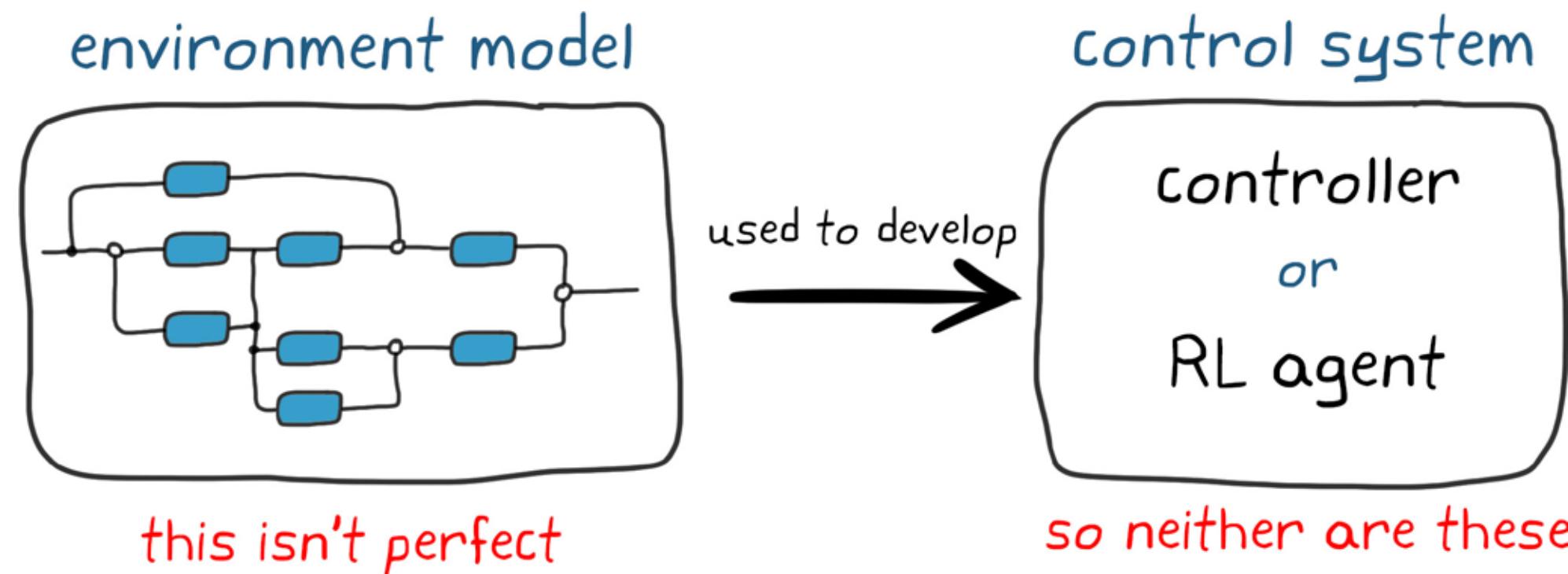


如果解决方案是一个庞大的神经元、权重和偏置的集合，则很难隔离问题。因此，如果最终得出的策略不太适当，而又无法单独修改出问题的策略部分，则必须重新设计智能体或环境模型，然后再次进行训练。重新进行一轮设计、训练和测试可能极为耗时。

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。



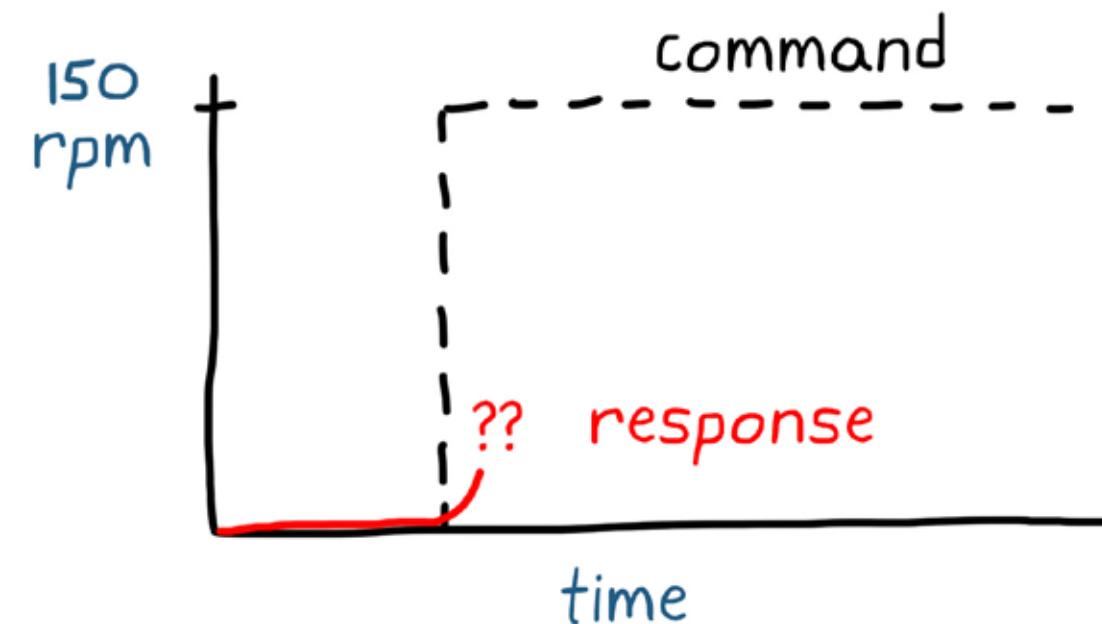
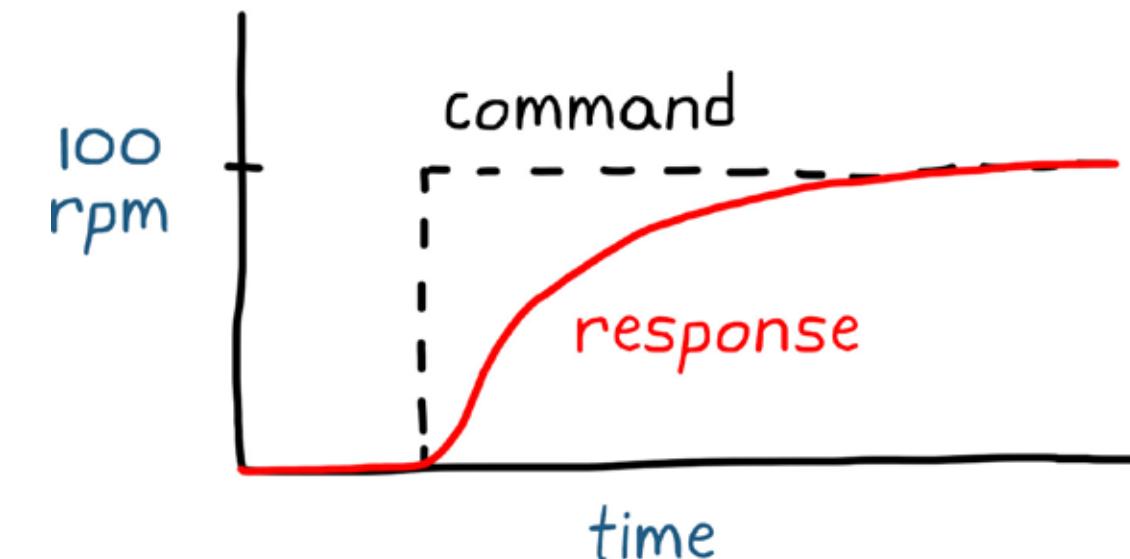
如果使用模型设计传统控制系统，这将不再是问题，因为您可以理解函数并调整控制器。但是，对于神经网络策略，则没有那么好办。鉴于永远无法构建绝对真实的模型，因此使用该模型训练的所有智能体都会存在轻微错误。为了解决这个问题，唯一方法是基于物理硬件训练智能体，这本身就颇具挑战性。

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

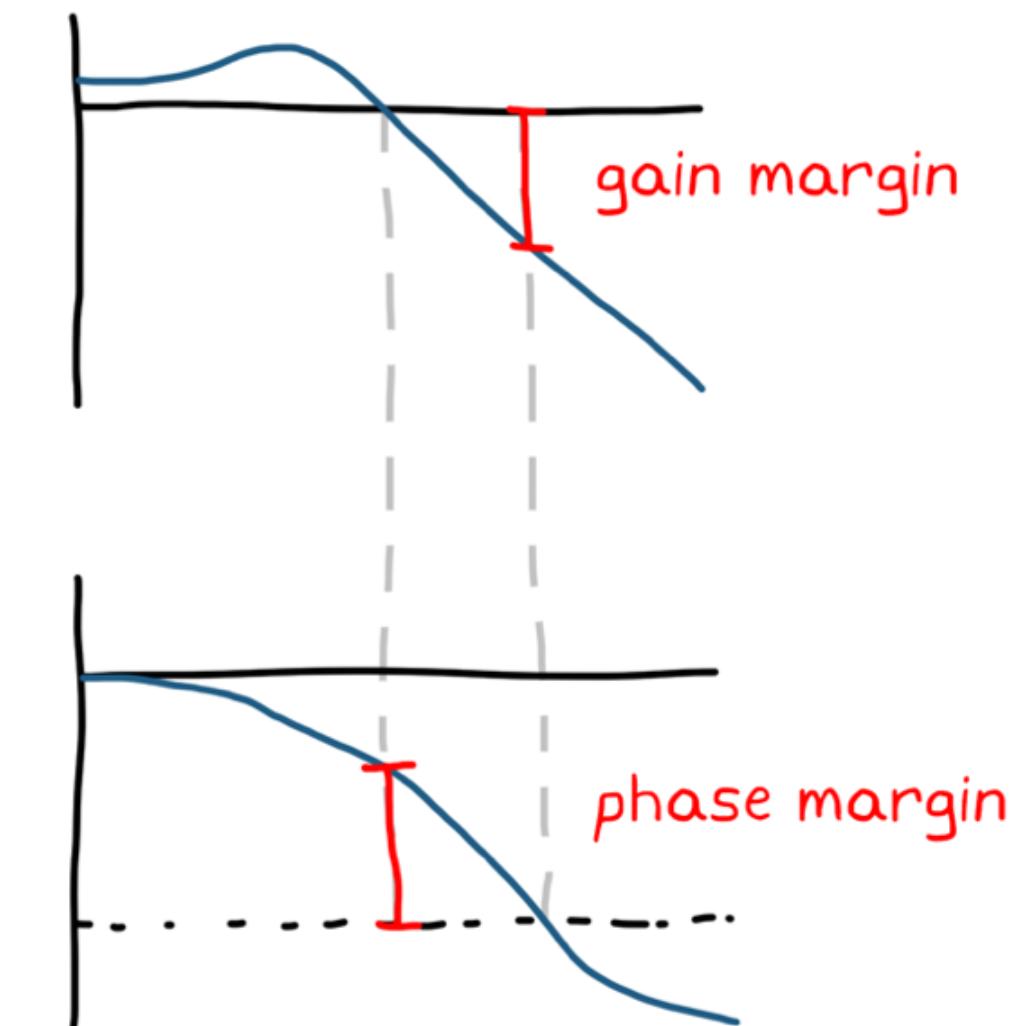
学习的神经网络同样会增加形式化验证的难度。此类方法包括通过提供形式化证明(而不是采用测试)保证满足某项条件。例如,如果在软件中对信号进行了绝对值运算,则无需通过测试确保该信号始终为非负信号。只需检查代码并证明始终符合条件即可完成验证。其他类型的形式化验证包括计算鲁棒性和稳定性系数,如幅值裕度和相位裕度。

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

verify this
is positive

code inspection shows
this is true

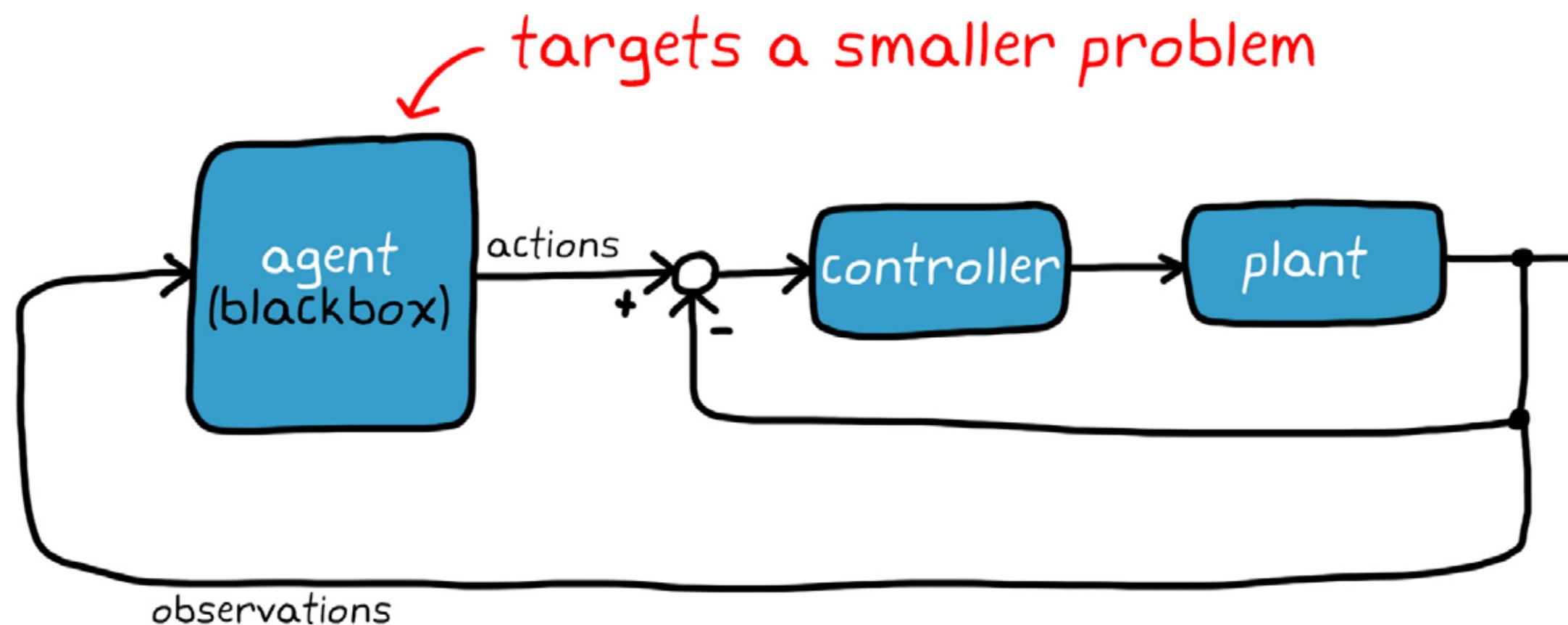
对于神经网络,这种形式化验证难度更大。我们说过,要检查代码并对其行为做出任何保证是很难的。也没有方法可以确定其鲁棒性或稳定性。一切都归结于一个事实:无法从内部解释函数。



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	.
.
.
.

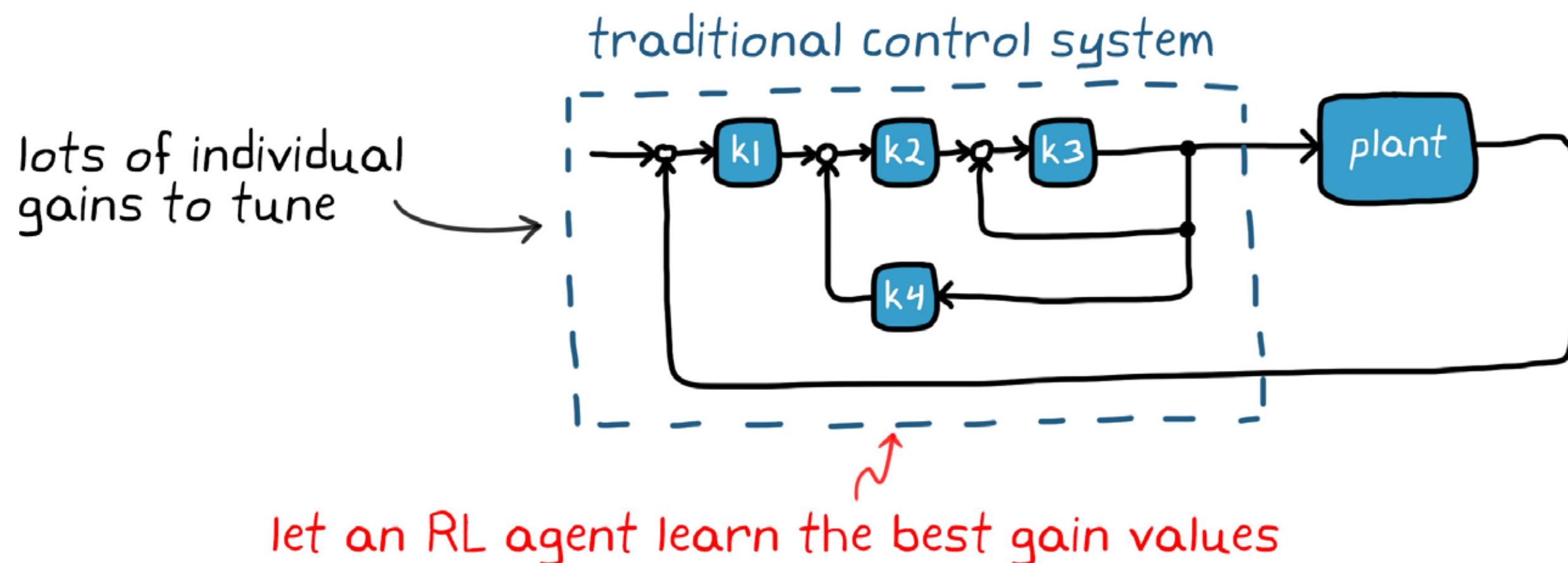
```
% software monitor  
if abs(body_angle) > 45; % monitor for falling  
    mode = "safe"; % set safe mode  
    extend_arms(); % prepare for impact  
end
```

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。

解决不同问题

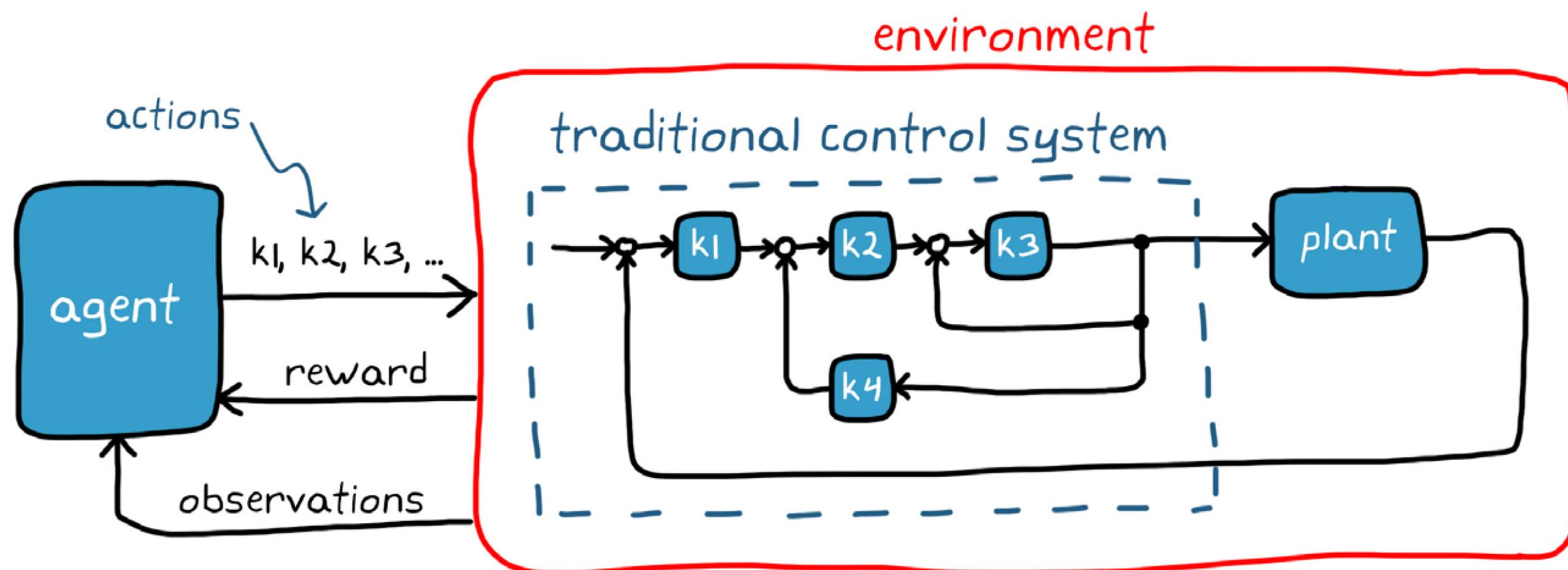
替代方法很好，但也可以干脆通过解决另一个问题来直接解决问题。您可以将强化学习作为工具，用于优化基于传统架构控制系统的控制器增益。想象一下设计包含数十个嵌套环路和控制器的架构，每个嵌套环路和控制器都有若干增益。最终可能会遇到一种情况：需要调整一百个乃至更多的单独增益值。您可以设置 RL 智能体，同时学习所有增益的最佳值，而不是尝试手动调整每一个增益。



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）。

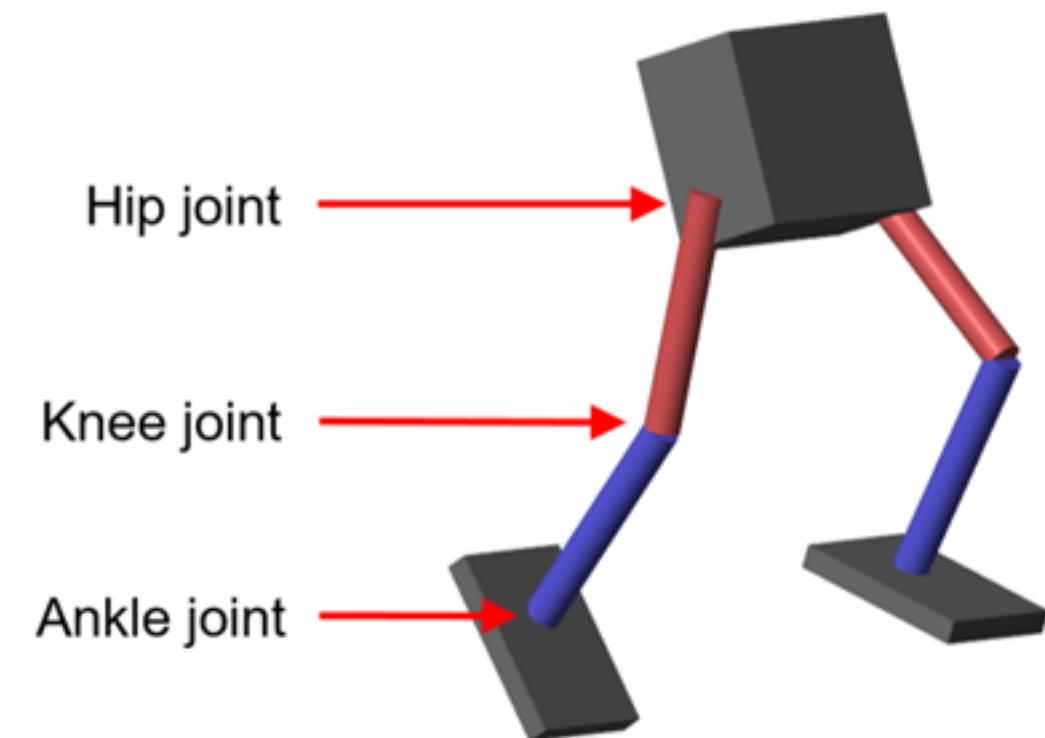
采用这种方法，则可以两全其美。无需部署和验证任何神经网络，也不必担心被迫进行更改；只需将最终静态增益值编码到控制器中。这样，您仍然采用传统架构系统，也就是可以在硬件上进行验证和手动调整，但其中填入了使用强化学习优选出的增益值。



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。

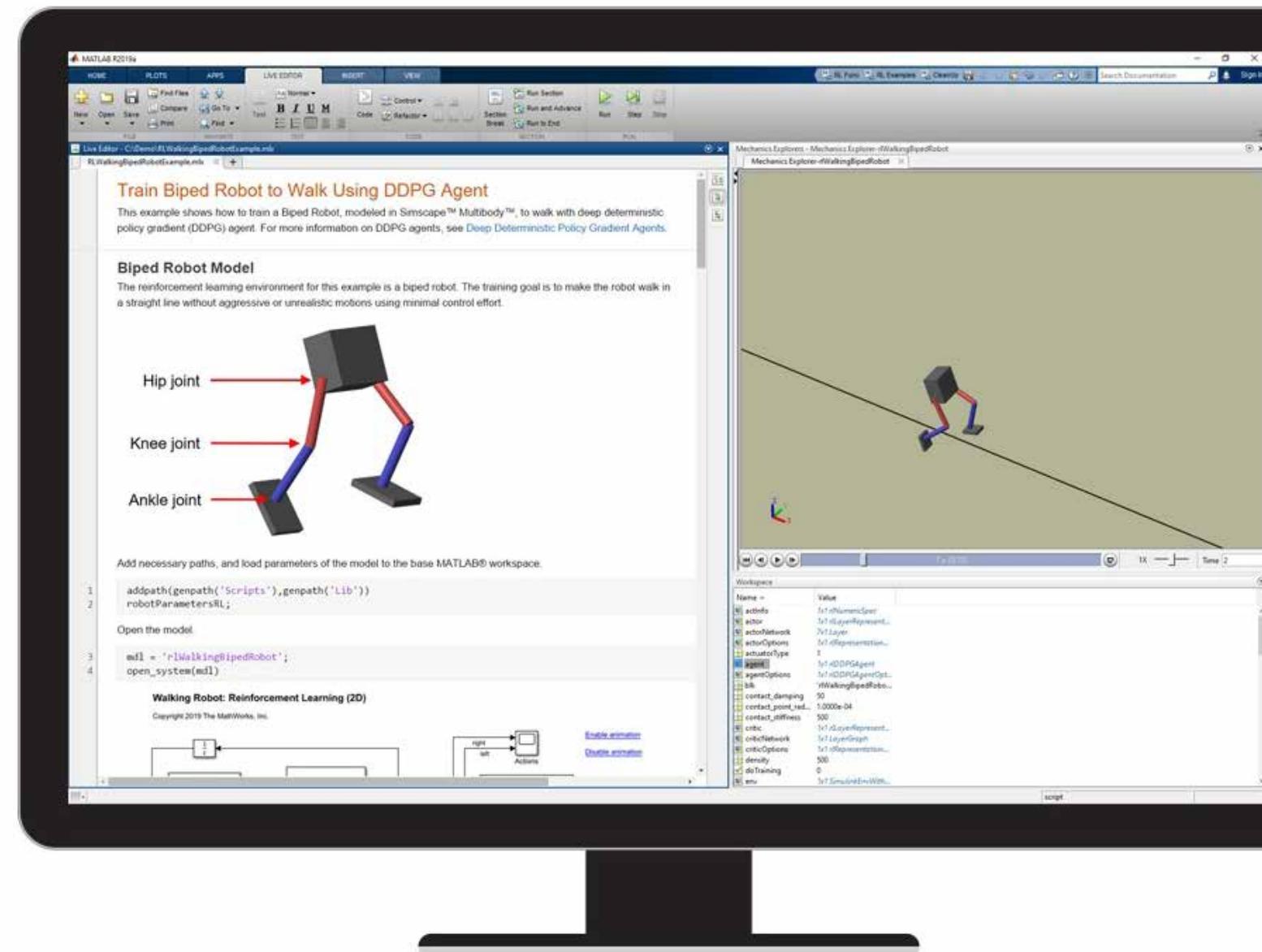
学习算法、强化学习设计工具(如 MATLAB 和 Reinforcement Learning Toolbox™)及验证方法一直在不断进步。



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](#)

