

# 使用 MATLAB 进行强化学习

了解奖励和策略结构

# 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](#)。本电子书探讨**奖励和策略结构**。  
最后一本电子书介绍[训练和部署](#)。

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。

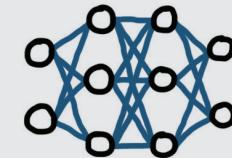
*reward*



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。

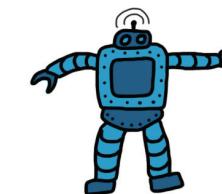
*training*



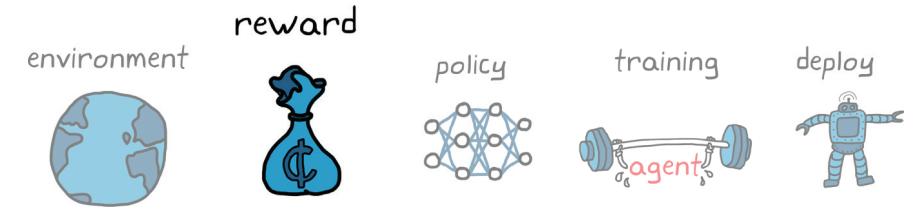
5

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。

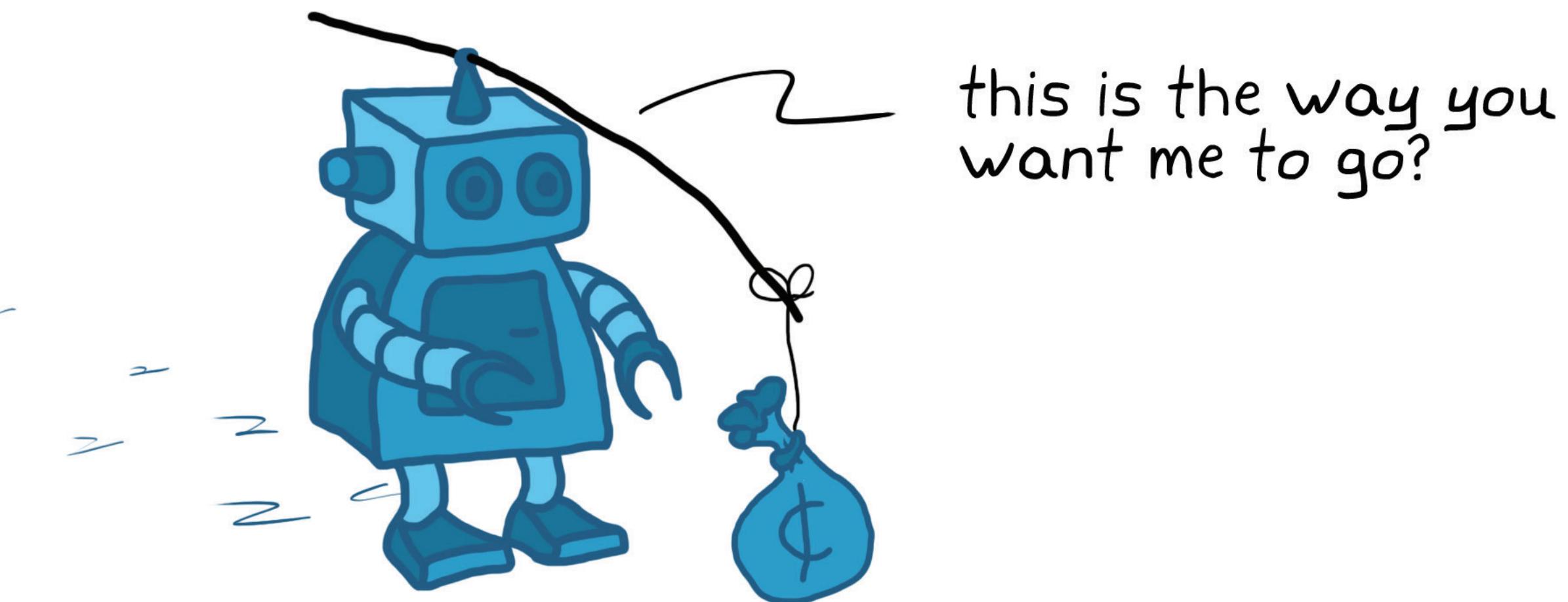
*deploy*



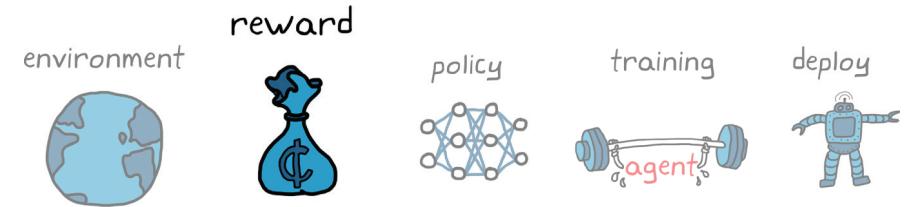
# 奖励



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。

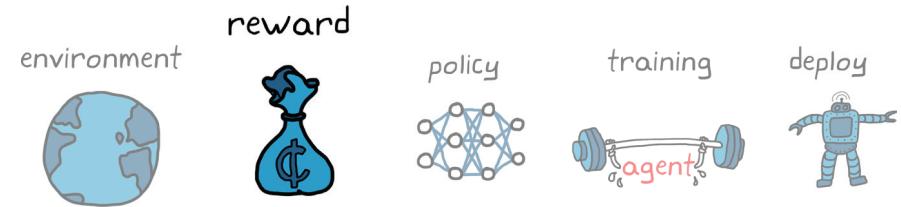
# 稀疏奖励

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。

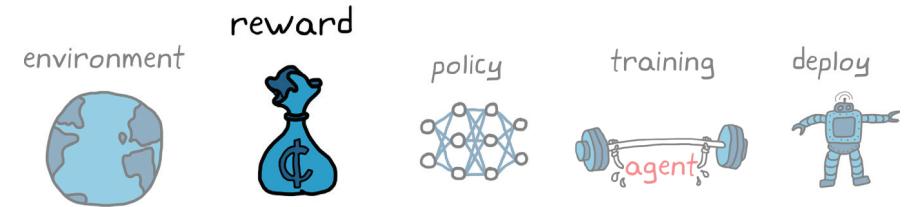


$$\text{reward} = \begin{cases} 1 & \text{for } \text{state} = 10 \text{ meters} \\ 0 & \text{for } \text{state} \approx 10 \text{ meters} \end{cases}$$

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



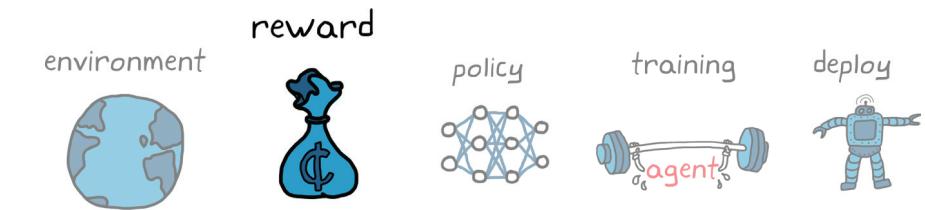
可通过奖励重塑改进稀疏奖励，即提供较小的中间奖励，引导代理沿正确的道路前进。



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

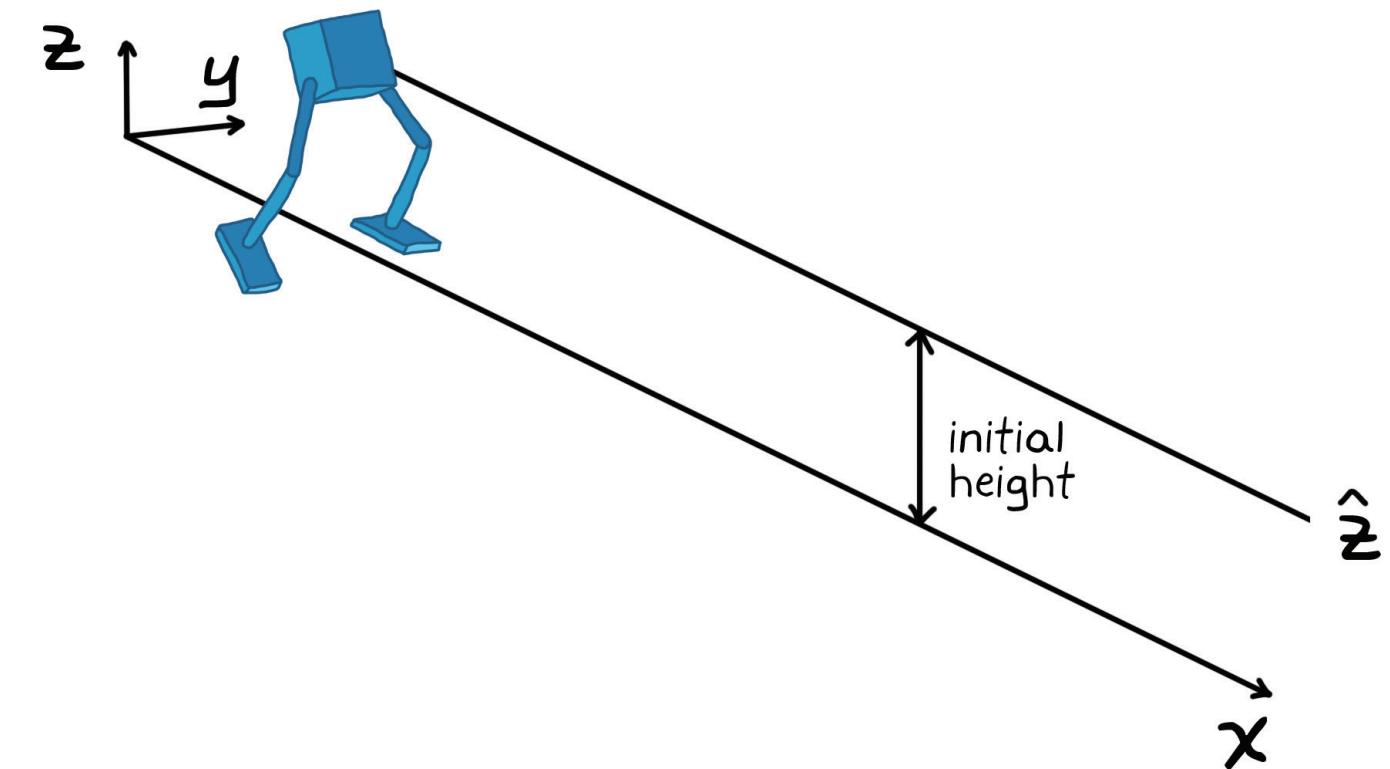


奖励重塑并不总是用于代替稀疏奖励。它还是工程师将特定领域的知识注入代理的一种方法。例如，如果您确定想让机器人行走，而不是沿地面爬行，您可以奖励代理，让机器人的躯干保持在行走高度。您还可以奖励作动器工作量的降低，更长时间地站立，不偏离预定路线。

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

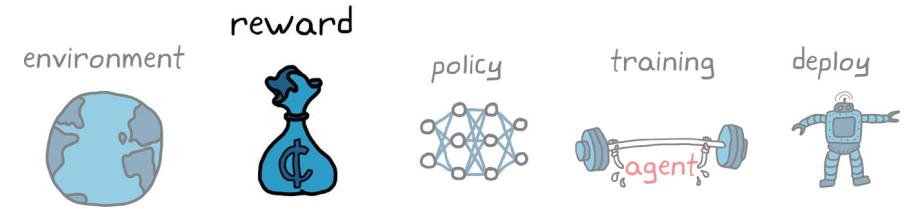
Annotations for the reward function components:

- $v_x$ : 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

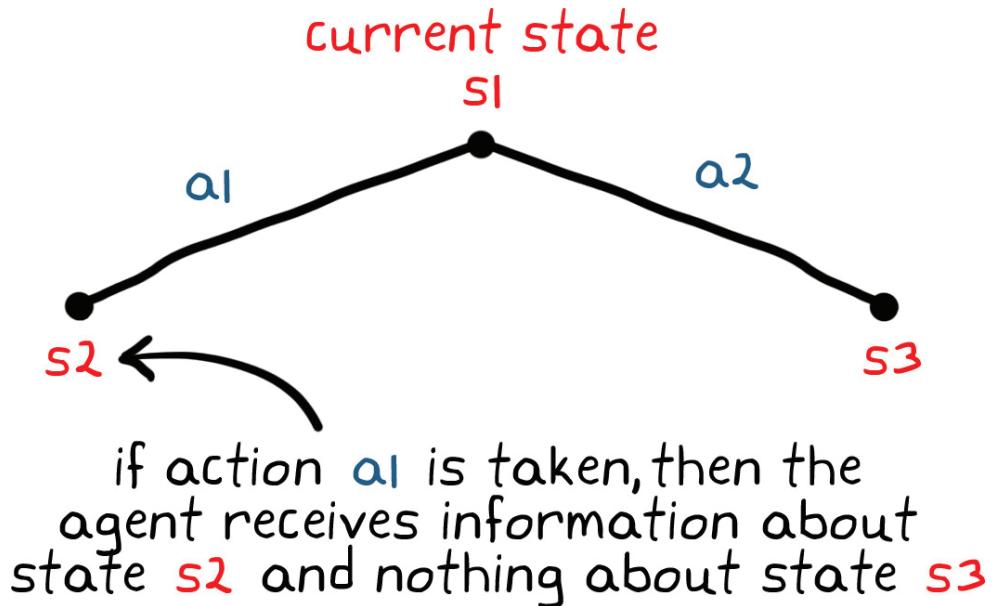


这不是说设计奖励函数就很容易，使之正确运作可能是强化学习中比较难的任务之一。例如，直到您花了很长时间训练代理，而它无法产生您所寻求的结果之后，您才可能知道奖励函数是不是设计得不好。但是，有了这个总体的认知，您至少能够更好地理解哪些事情需要注意并且可能使设计奖励函数容易一点。

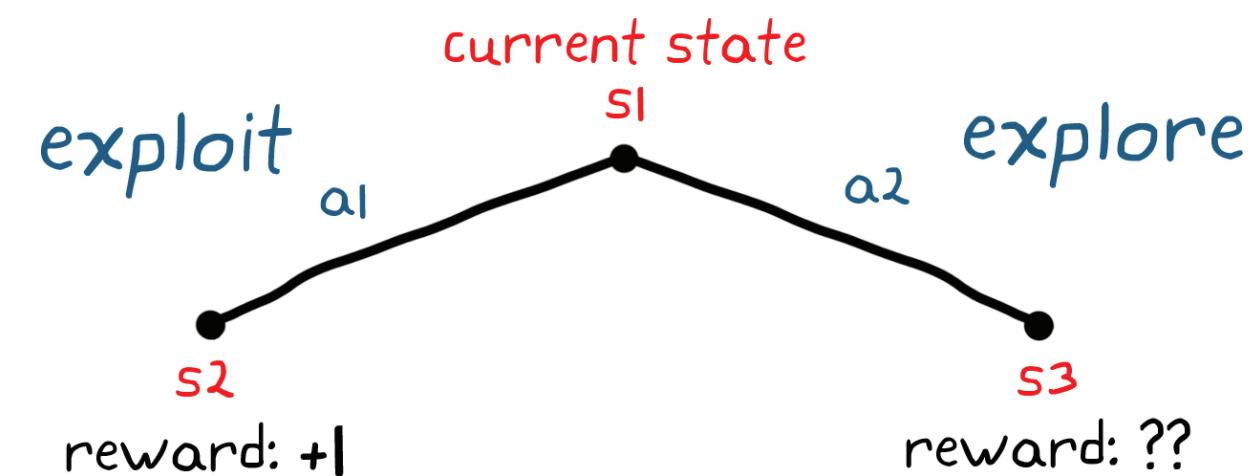
# 探索 (Exploration) 与利用 (exploitation)



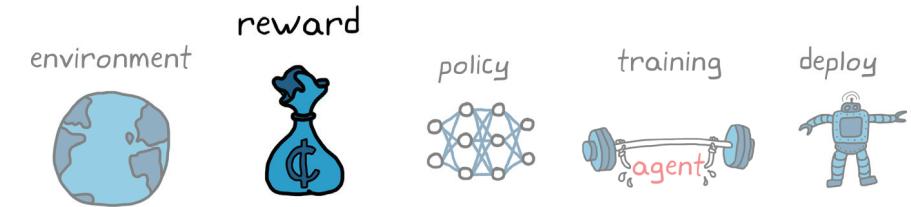
强化学习的一个重要方面是在代理与环境交互时权衡探索与利用之间的利弊。强化学习时需要做这个决定的原因在于学习是在线实现的。不是利用静态数据集，而是由代理的动作决定从环境中返回哪些数据。代理做出的选择决定了它会接收到的信息，以及因此它可从中学习的信息。



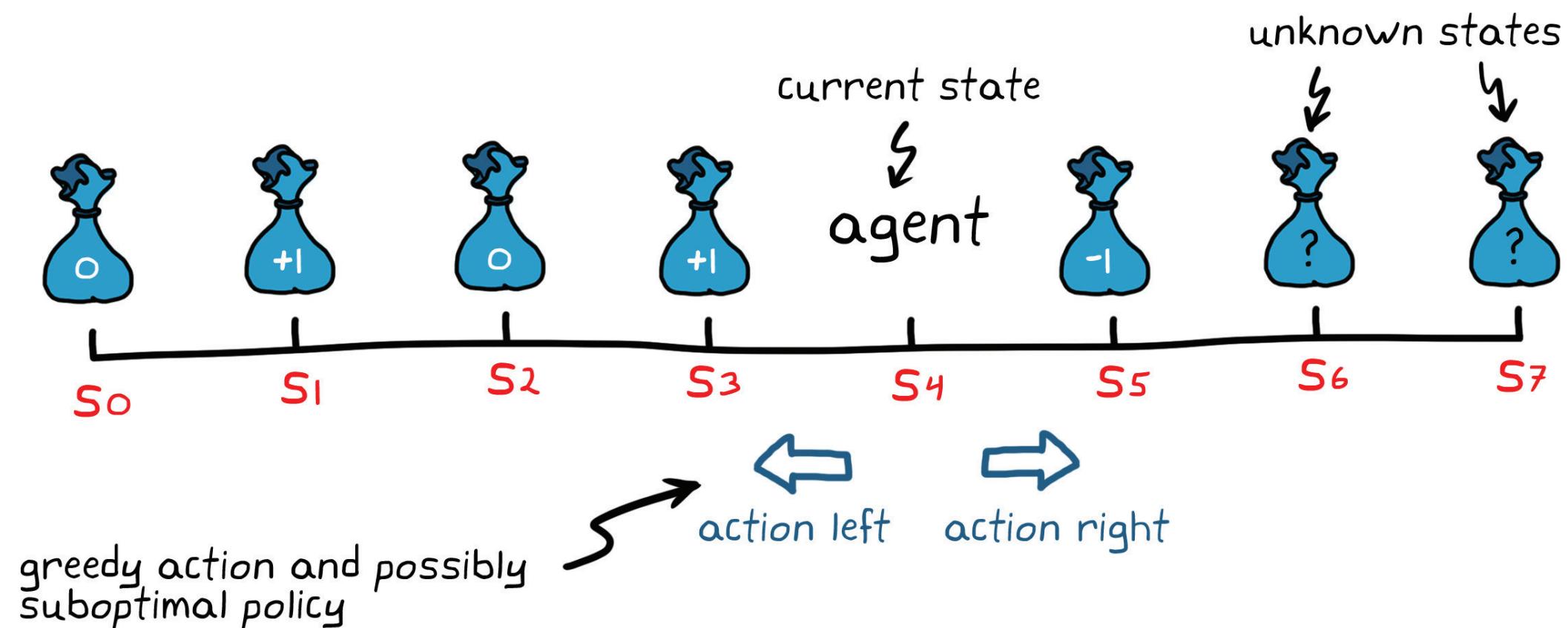
这里的构想是：代理是否应该利用环境，选择收取它已经知道的最多奖励的那些动作？还是应该选择探索环境中仍然未知部分的动作？



# 纯利用面临的问题



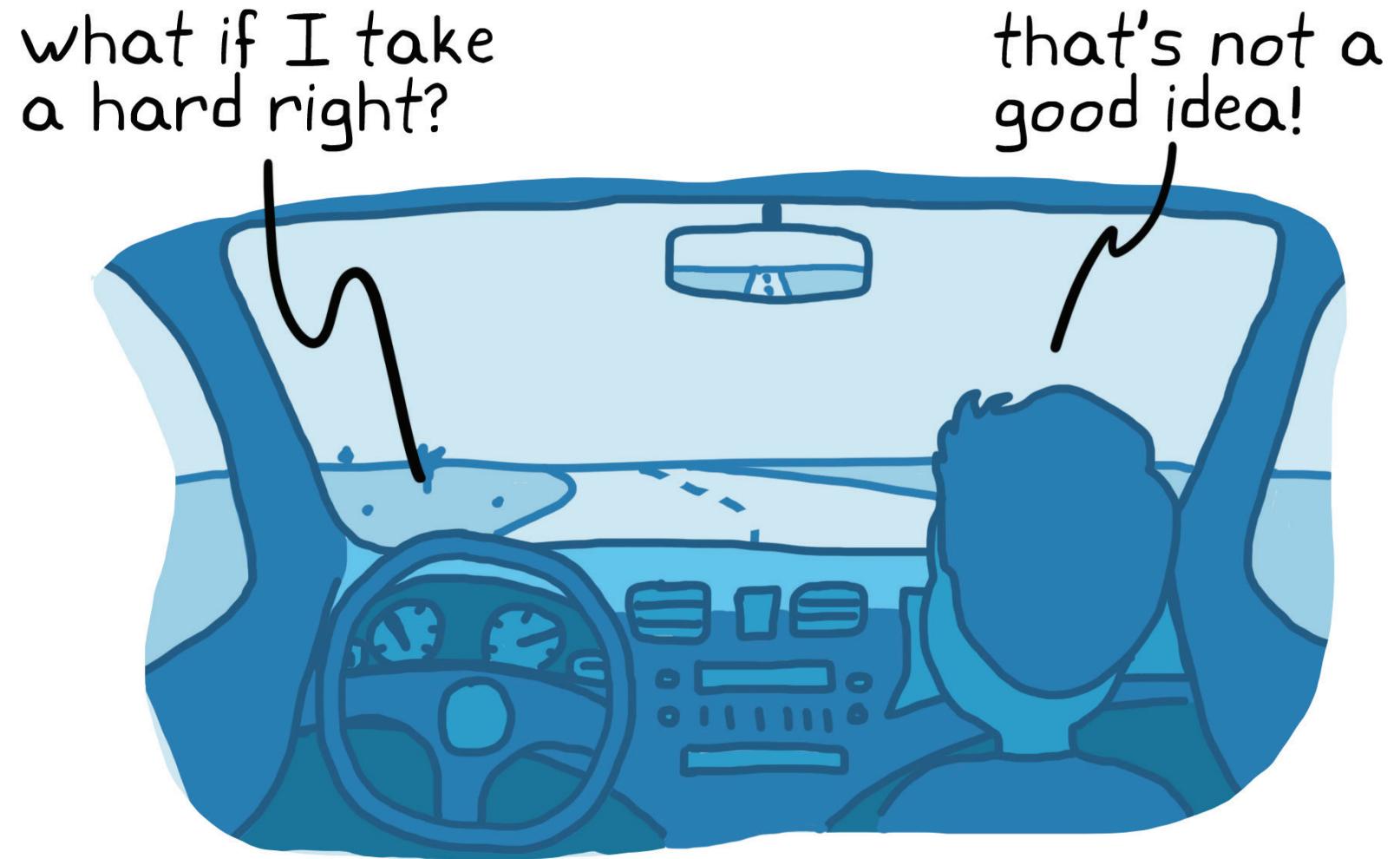
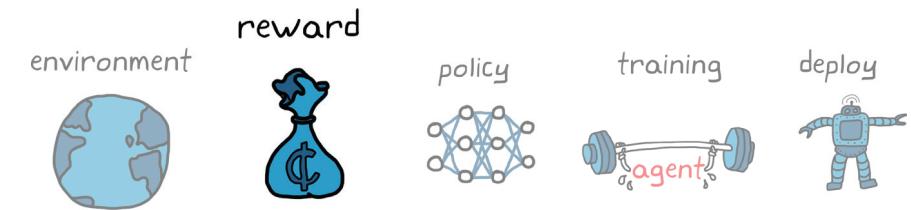
比如说，代理处于特定的状态，它可以采取两个动作之一：向左或向右。它知道，向左会产生 +1 奖励，向右会产生 -1 奖励。代理对于初始低奖励状态右侧的环境一无所知。如果代理采取贪婪的方法，总是利用环境，那么它会选择向左并收取它知道的最高奖励，而完全忽略其他状态。



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。

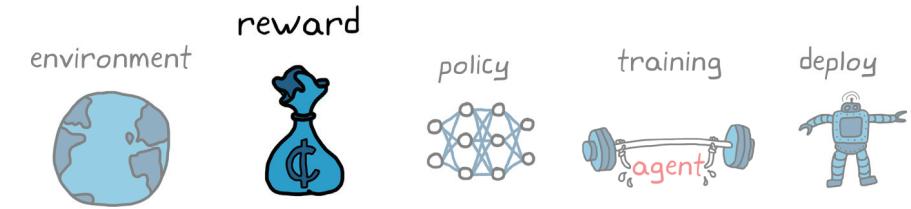
# 纯探索面临的问题

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。

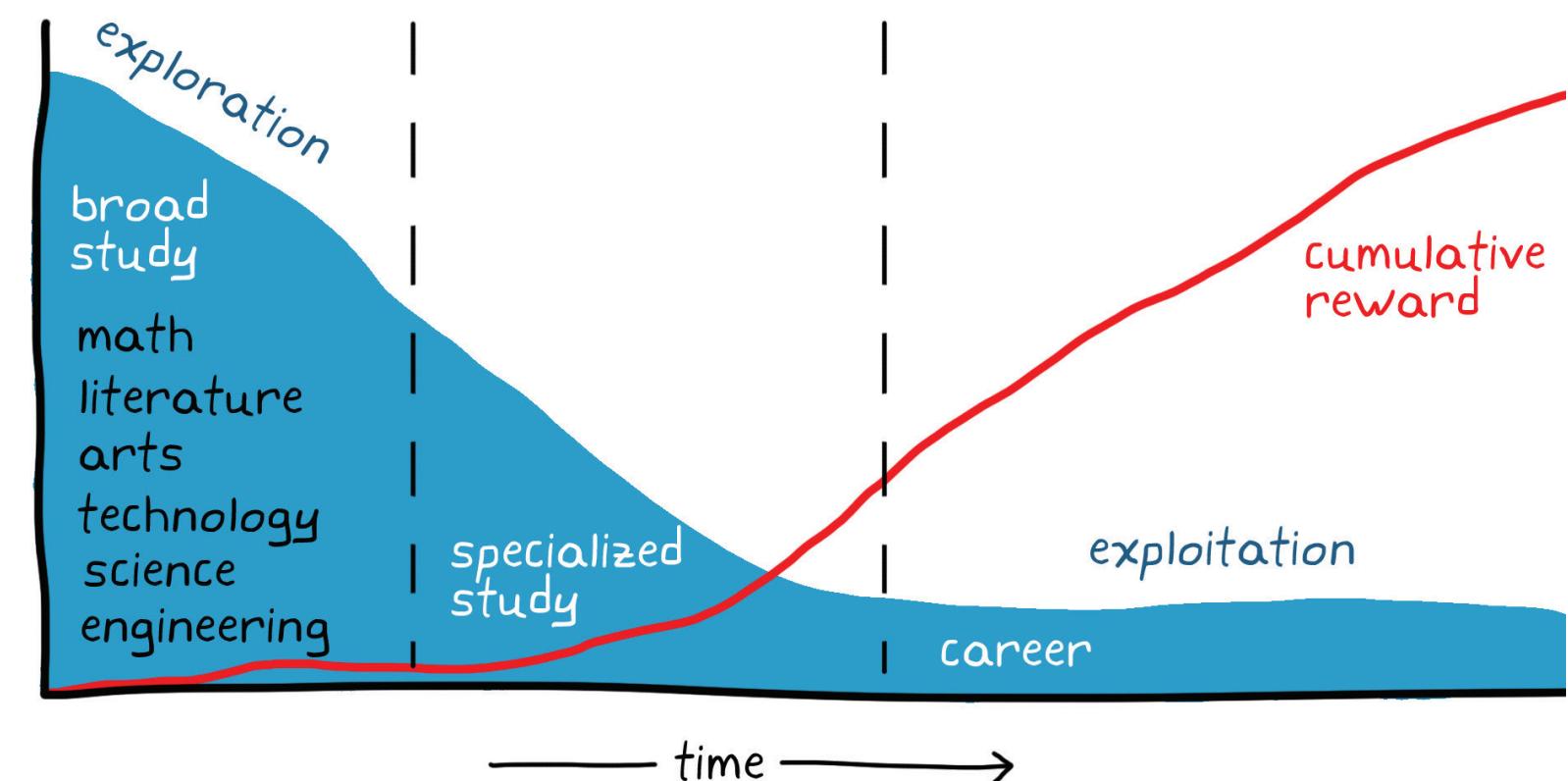
# 探索与利用的平衡



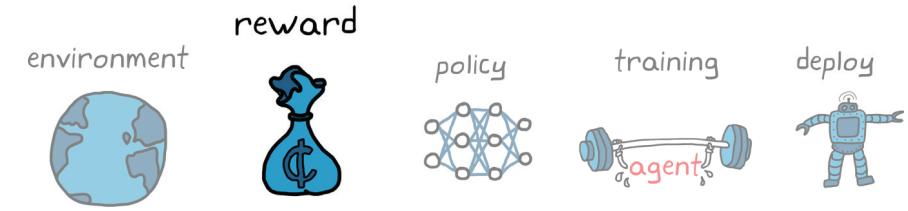
考虑一下学生在选择职业道路时可能采取的做法。学生们在低年级时会去探索不同的学科和课程，对新的体验一般持开放态度。在进行一番探索后，他们可能趋向于更多地学习某一专业学科，然后最终趋向于他们认为财务收益和工作满意度（奖励）最大化的职业。

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

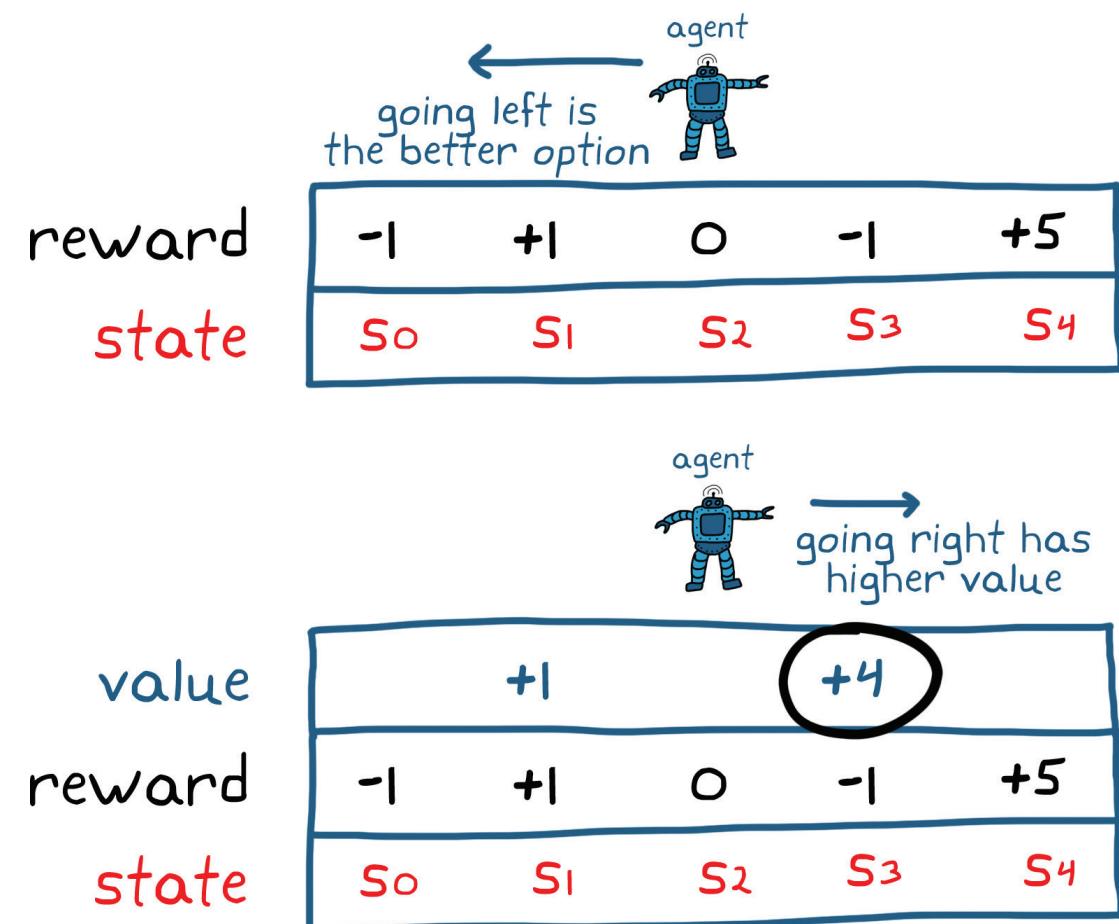


强化学习的第二个重要方面是**价值**的概念。评估一个状态或动作的价值，而不是奖励，可以帮助代理选择将会在一段时间内收取最多奖励（而不是短期利益）的动作。

**奖励**: 处于某一状态或采取特定动作的即时收益

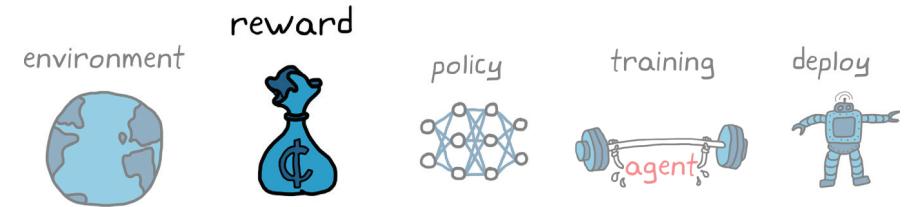
**价值**: 代理预期从某一状态和往后将会获得的总回报

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。



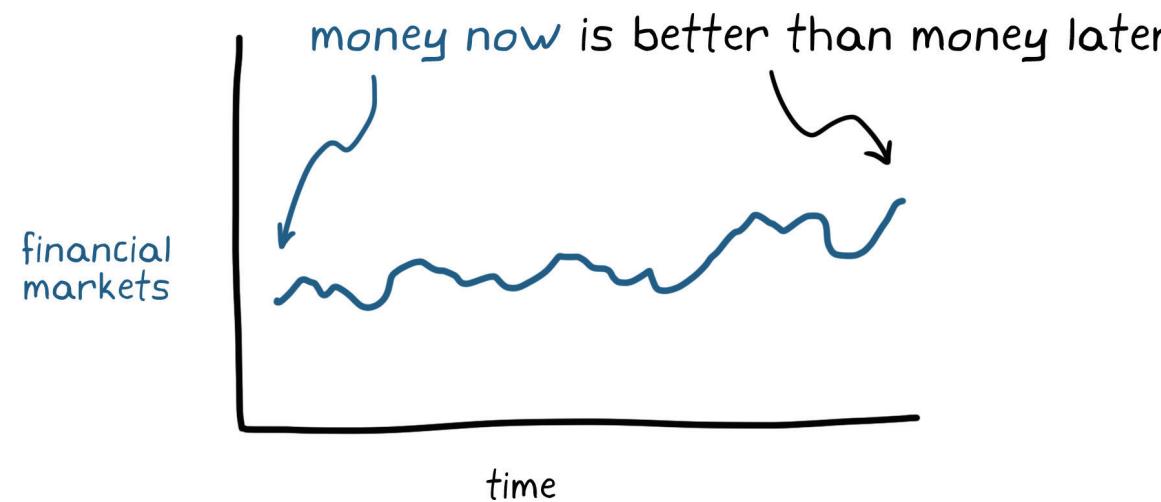
但是，如果代理能够评估状态的价值，它就会看到，向右走比向左走有更高的价值，即使奖励较低。使用价值作为其向导，最后代理就会得到 +4 的总回报。

# 短视的好处

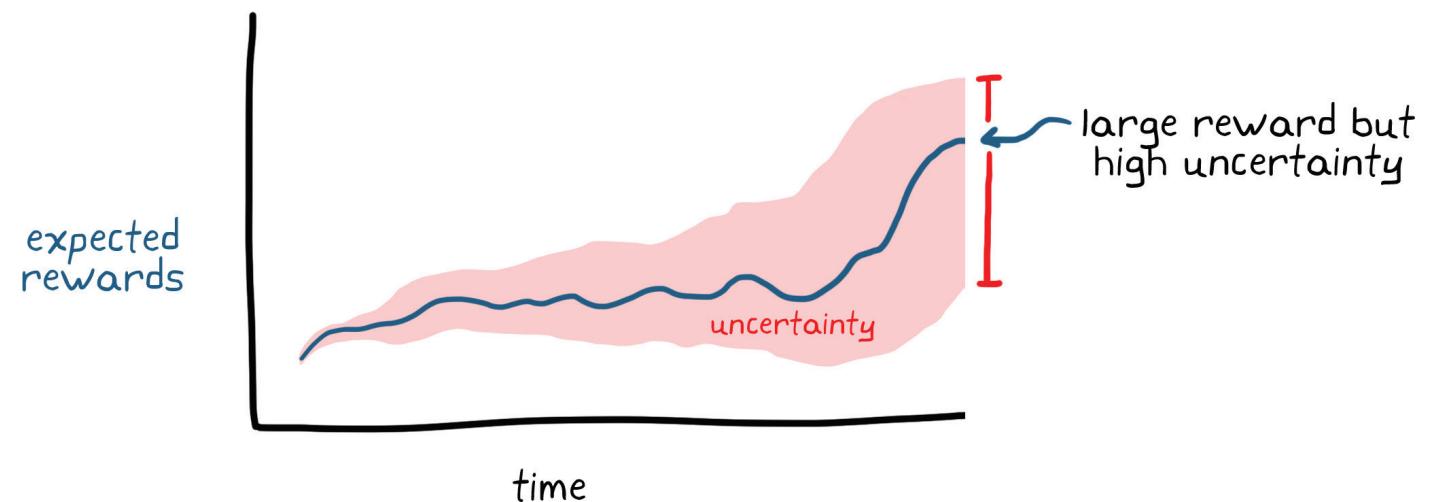


当然，承诺在多个连续动作后得到高奖励并不意味着第一个动作肯定最好；这里至少有两个好理由。

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。



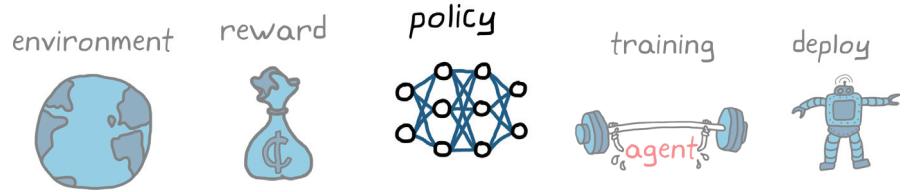
其次，对更远的将来所能获取的奖励的预测变得不大可靠；因此，等到那时，该奖励可能已不存在了。



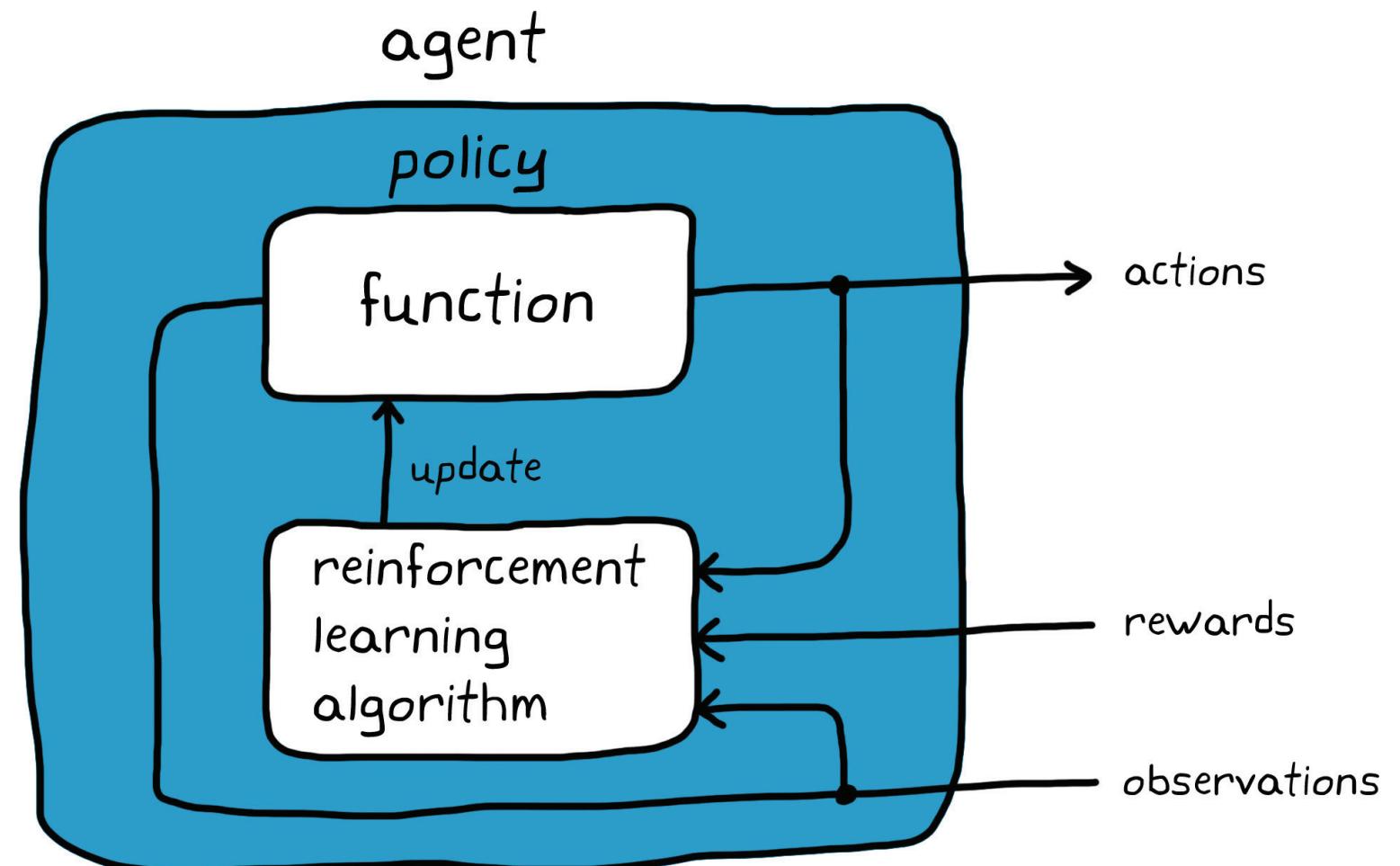
在这两种情况下，在评估价值时多一点短视比较有利。在强化学习中，通过对奖励打折，越远的未来折扣越大，您可以设置想让代理短视的程度。具体实现方式是设置折扣系数 gamma，介于 0 到 1 之间。

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

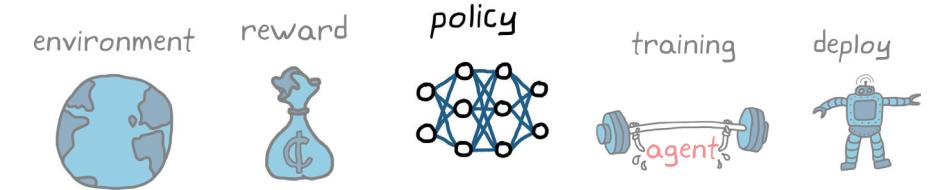
# 什么是策略?



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)*

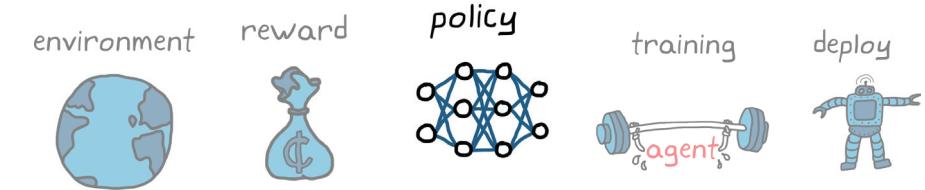
一般来说，有两种构造策略函数的方法：

- 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。<sup>\*</sup>

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。

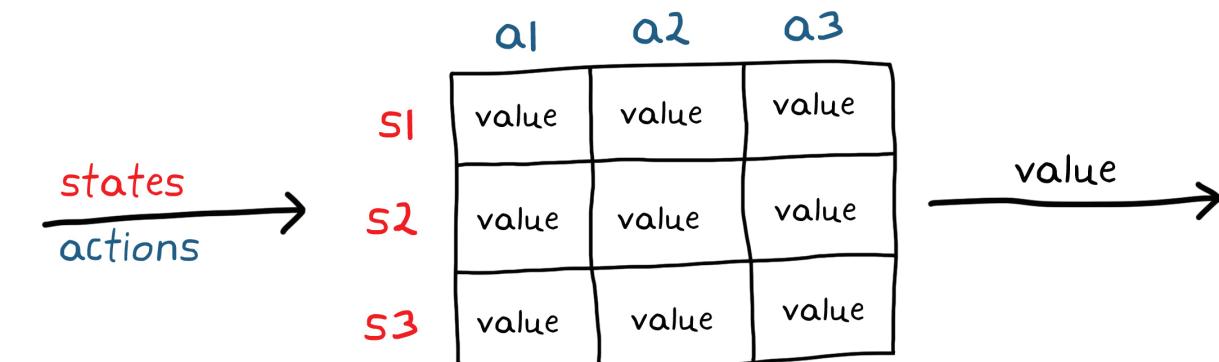
<sup>\*</sup> 事先提醒！您可以在名为 *Actor-Critic* 的第三种方法中结合直接策略映射和基于价值映射的好处，稍后介绍这种方法。

# 用表格表示策略

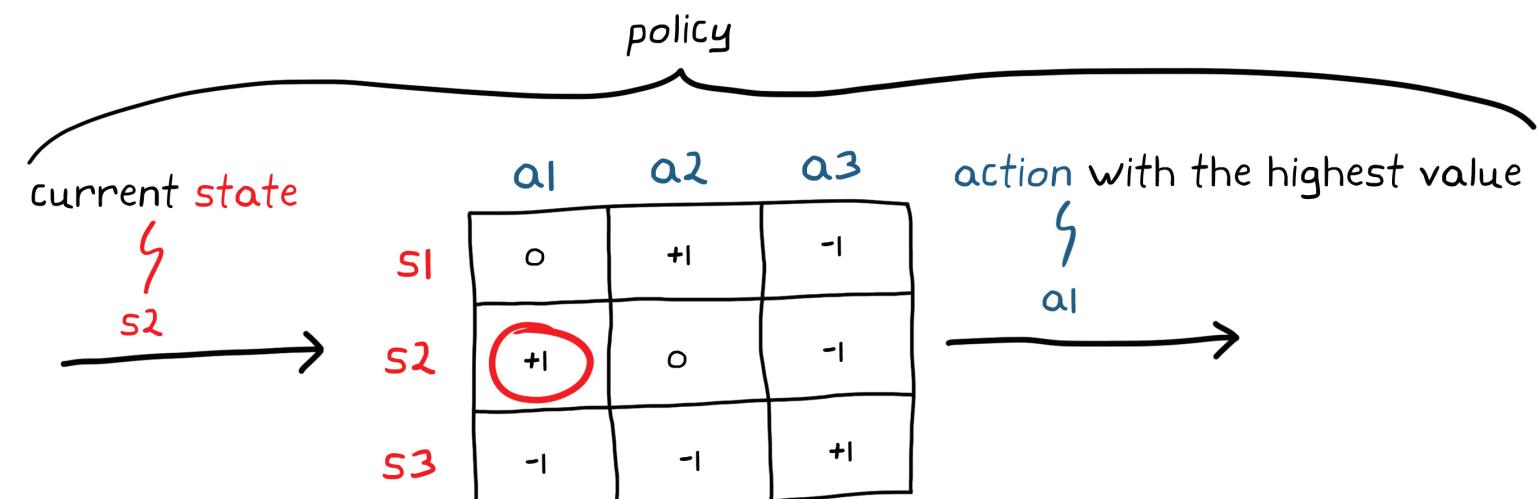


如果环境的状态和动作空间离散，且数量少，则可以使用简单表格来表示策略。

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。

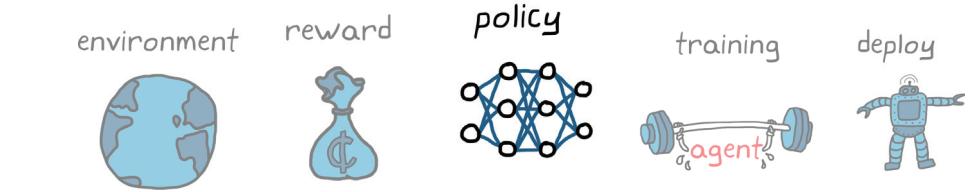
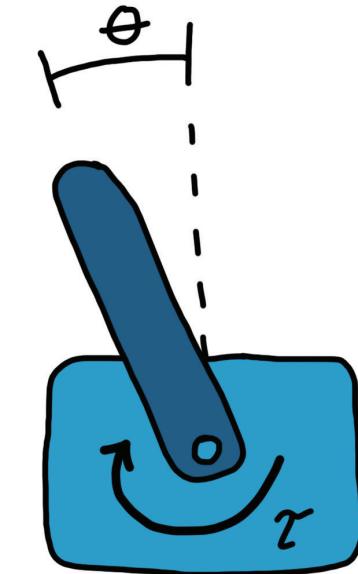


使用Q-table，策略会在当前状态给定的情况下检查每个可能动作的价值，然后选择具有最高价值的动作。使用 Q-table 训练代理将包括确定表格中每个状态/动作对的正确价值。在表格完全填充正确的值之后，选择将会产生最多长期奖励回报的动作就相当直接了。



# 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  $\pi$  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。

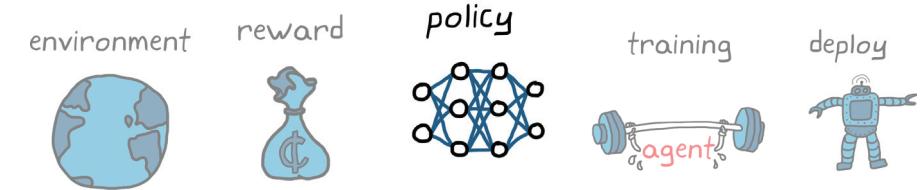
$$\text{value} = -(\dot{\theta}^2 + \theta^2) + \tau ?$$

*we need to define the logical structure*

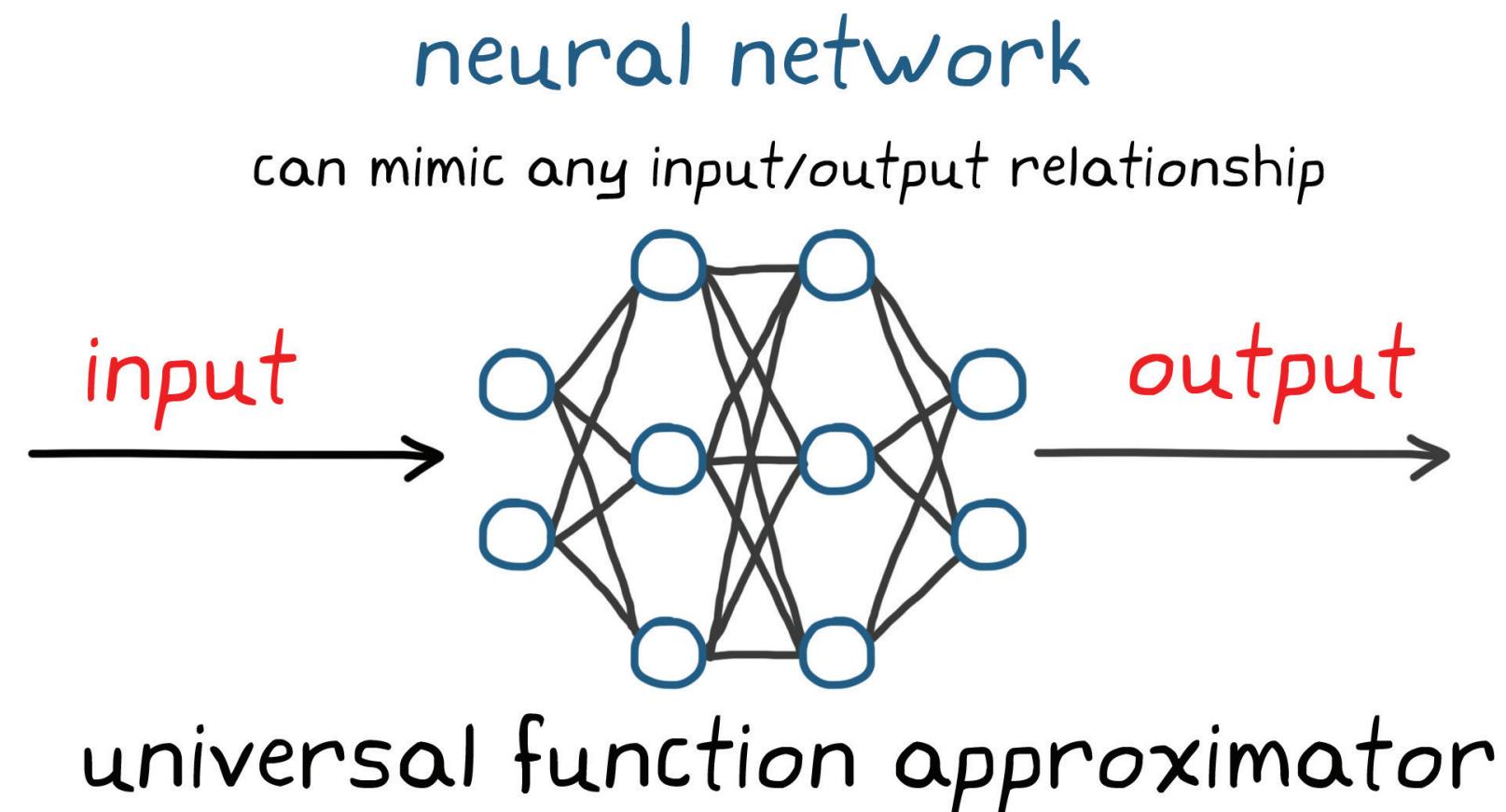
$$\text{value} = \dot{\theta} \sin(\theta) + \tau ?$$

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

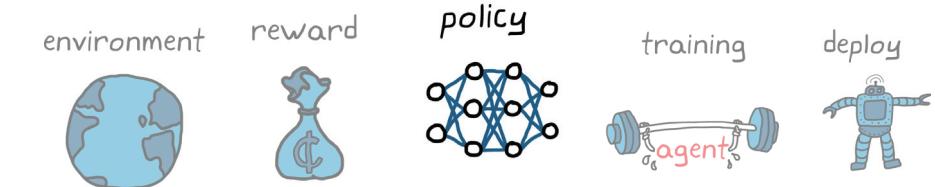


神经网络是一组节点或人工神经元，采用一种能够使其成为通用函数逼近器的方式连接。这意味着，给出节点和连接的正确组合，您可以设置该网络，模仿任何输入与输出关系。尽管函数可能极其复杂，神经网络的通用性质可以确保有某种神经网络可以实现目标。



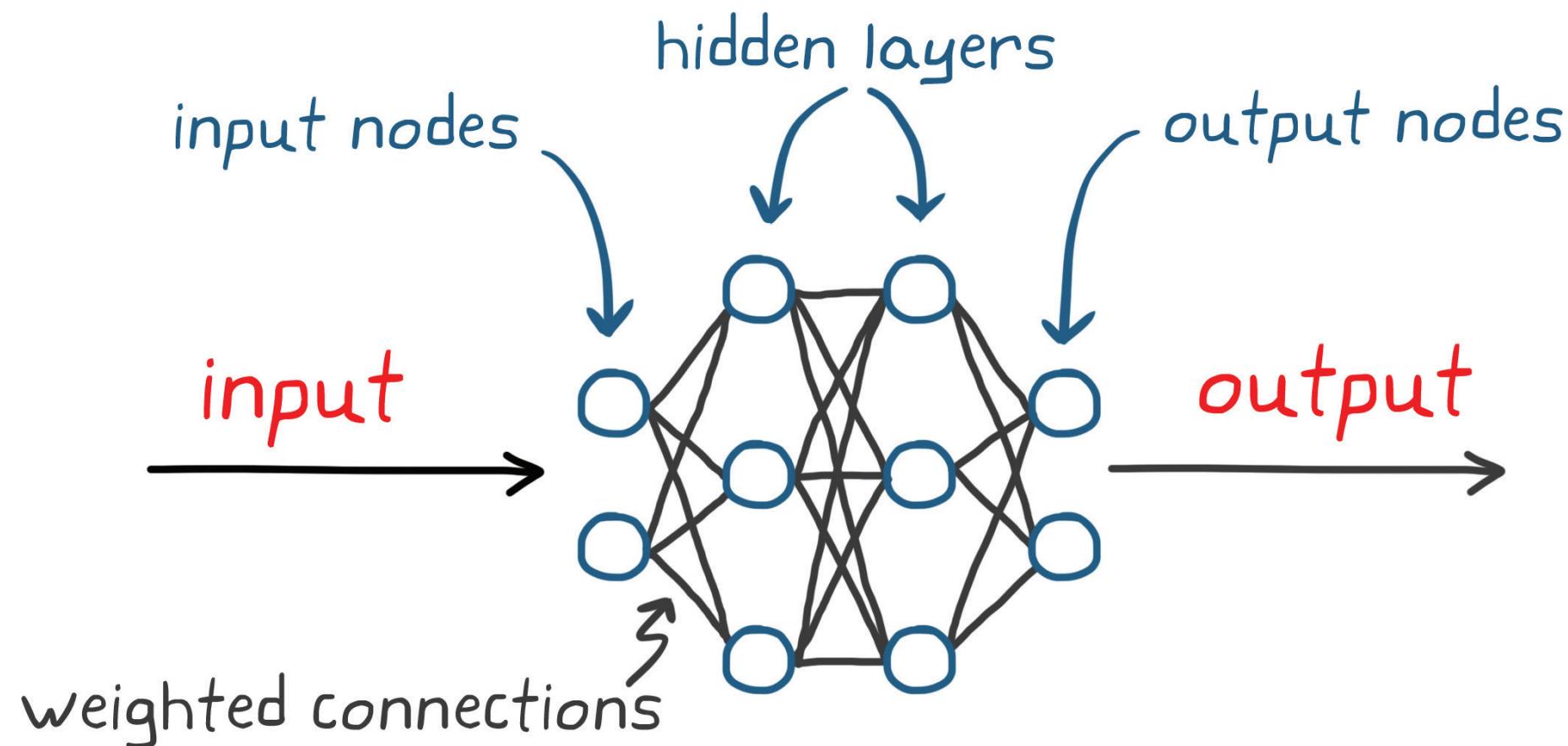
所以，与其尝试寻找适合特定环境的完美非线性函数结构，不如使用神经网络，这样就可以在许多不同环境中使用相同的节点和连接组合。唯一的区别在于参数自身。学习过程将包括系统地调节参数，找到最优输入/输出关系。

# 什么是神经网络?

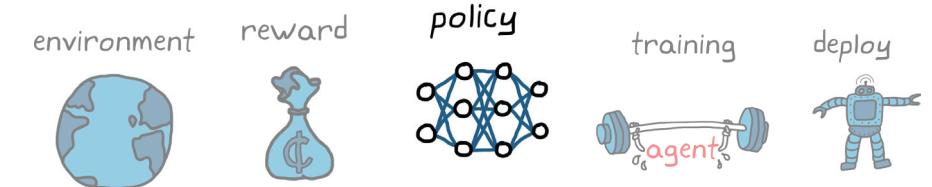


这里不会深入探讨神经网络的数学原理。但强调一些事情十分重要，方便解释稍后建立策略时的一些决策。

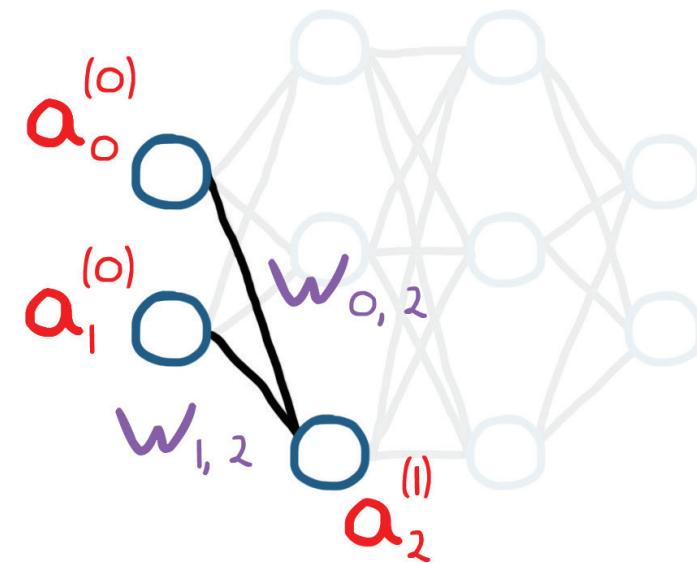
左边是输入节点，一个节点对应函数的一个输入，右边是输出节点。中间是称为隐藏层的节点列。此网络有 2 个输入、2 个输出和 2 个隐藏层，每层 3 个节点。对于全连接的网络，存在从每个输入节点到下一层中每个节点的加权连接，然后是从这些节点连接到后面一层，直到输出节点为止。



# 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  $a_2^{(1)}$  =  $w_{o,2} \cdot a_o^{(0)}$  +  $w_{i,2} \cdot a_i^{(0)}$  +  $b_2^{(1)}$

layer  
node position

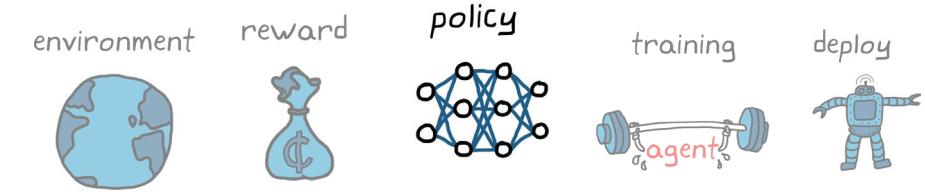
您可以对某个层中的每个节点执行此计算，以紧凑的矩阵形式写出来，作为一个线性方程组。这组矩阵运算实质上是将一层节点的数值变换为下一层节点的值。

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

$$a^{(1)} = W_o a^{(0)} + b^{(1)}$$
$$a^{(2)} = W_1 a^{(1)} + b^{(2)}$$
$$a^{(3)} = W_2 a^{(2)} + b^{(3)}$$

matrices

# 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。

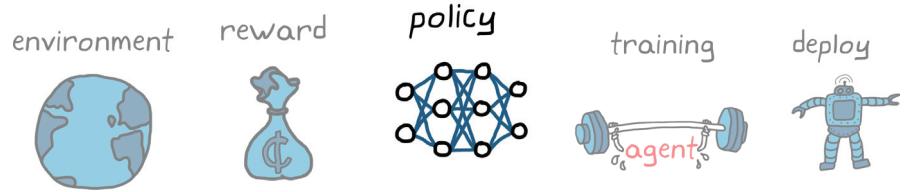
$$a^{(l)} = \text{act} [ W_o a^{(o)} + b^{(l)} ]$$

*activation function is applied after linear operations*

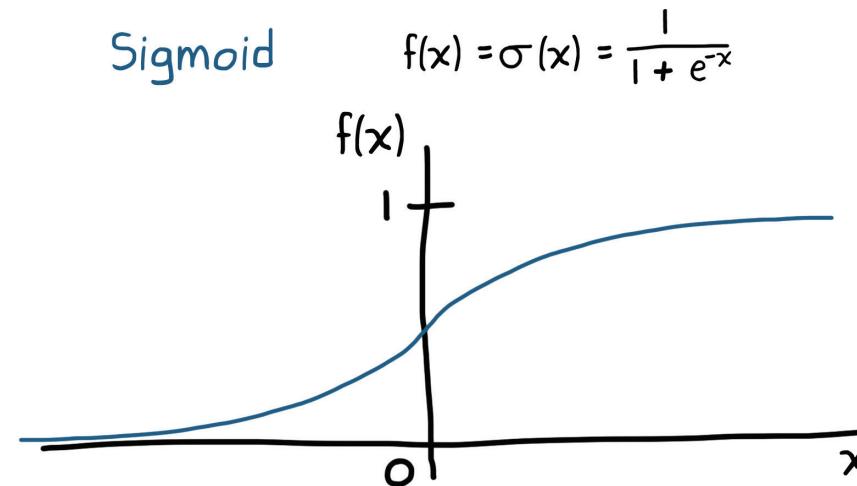
有若干不同的激活函数。它们共同的特点是非线性, 这对于构造可以逼近任何函数的网络至关重要。为什么出现这种情况? 因为许多非线性函数可以分解成加权的激活函数输出组合。

有关详细信息, 请阅读[通用逼近定理可视化](#)。

# ReLU 和 Sigmoid 激活

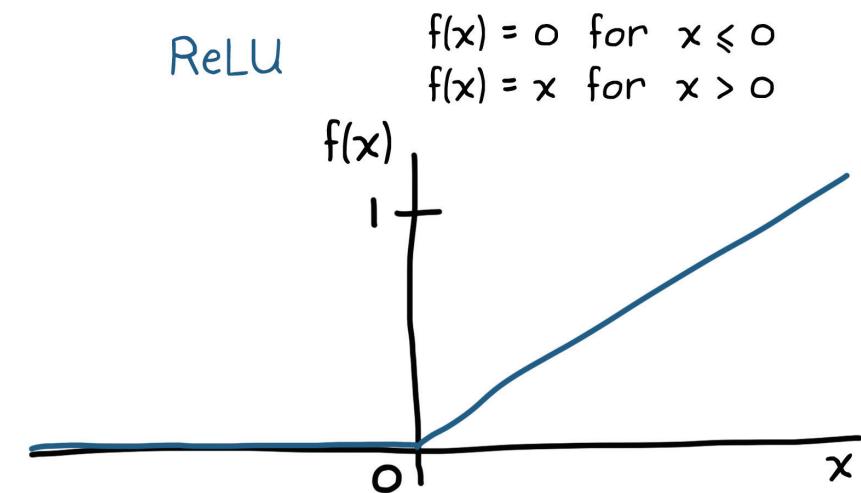


sigmoid 激活函数会生成一条平滑的曲线，使介于负无穷大和正无穷大之间的任何输入都被压缩到 0 到 1 之间。



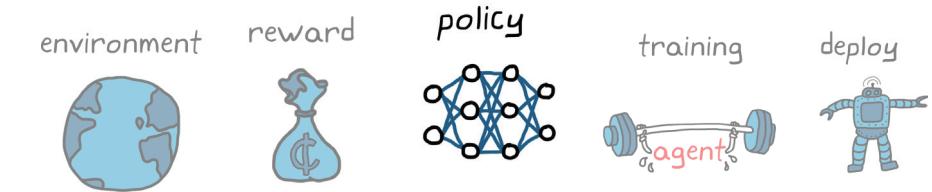
举个例子，预激活节点值 -2 使用 sigmoid 激活将变为 0.12，使用 ReLU 激活将变为 0。

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。

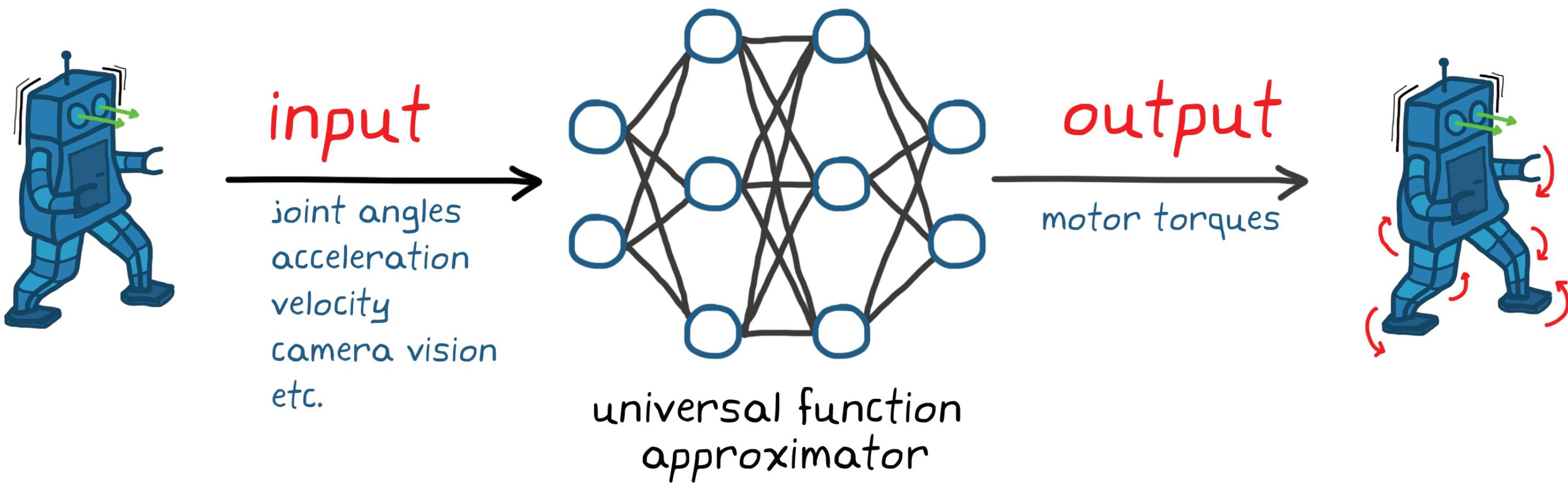


preactivation node value	postactivation sigmoid	ReLU
-2	0.12	0
-1	0.27	0
1	0.73	1
2	0.88	2

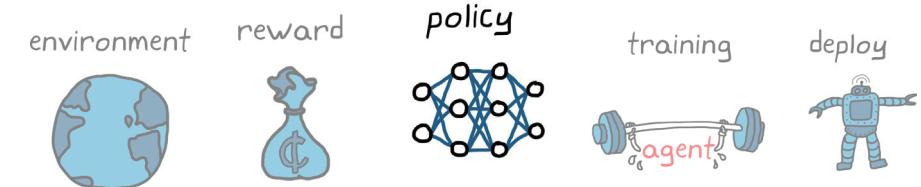
# 用神经网络表示策略



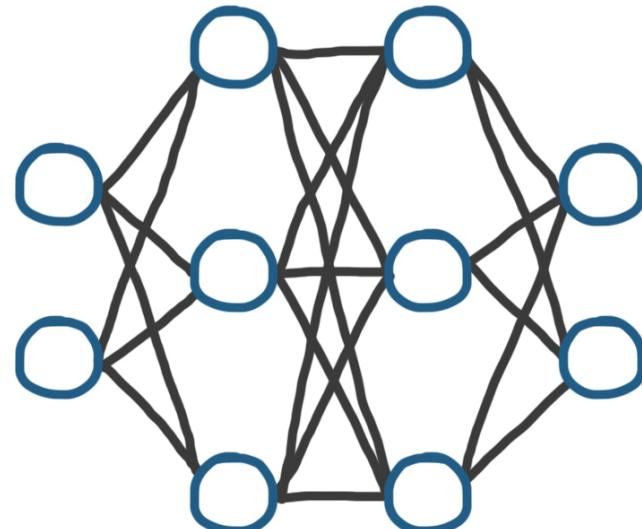
我们先归纳要点。您想找到一个函数，以大量的观测量为输入，并将其变换为能控制一些非线性环境的一组动作。由于此函数的结构通常过于复杂，难以直接求解，您想借助神经网络进行近似，随时间推移，学习该函数。人们倾向于认为，可以接入任何神经网络，然后放手让强化学习算法自行去寻找合适的权重和偏置的组合，任务就完成了。遗憾的是，事情往往不是这样。



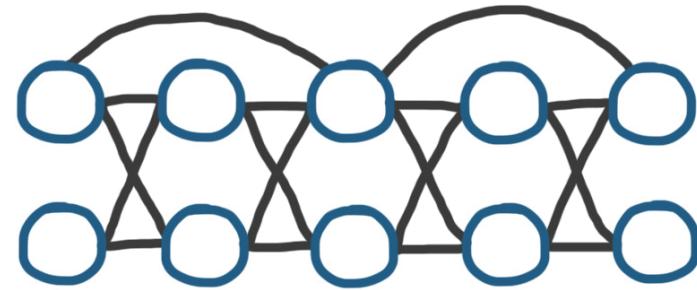
# 神经网络结构



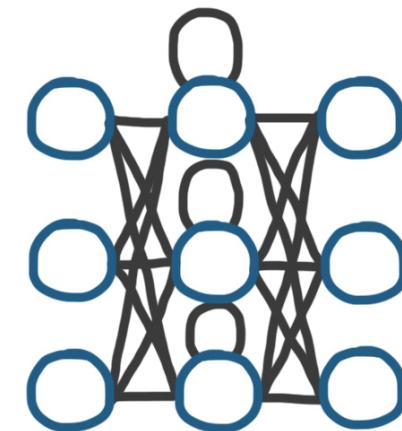
您必须提前对神经网络做出一些选择，确保它的复杂度足以近似您所寻求的函数，但也不要复杂得无法进行训练或慢得不可想像。例如，如您所见，您需要选择激活函数、隐藏层的数量以及每层的神经元数量。但除此之外，您还可以控制网络的内部结构。是否应该像您一开始使用的网络那样全连接？还是说应该像残差神经网络中那样，连接时跳过一些层？使用循环神经网络，利用自身环路形成内部记忆？各组神经元是否应该像卷积神经网络那样共同工作？



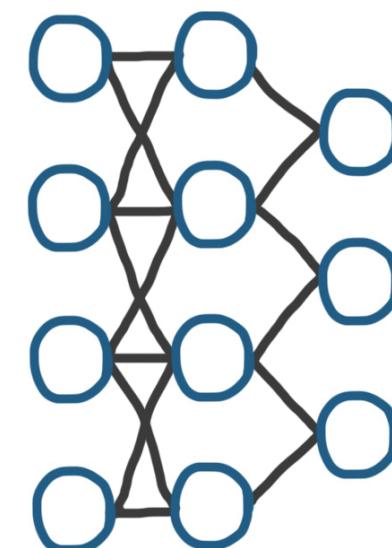
fully connected



residual



recurrent



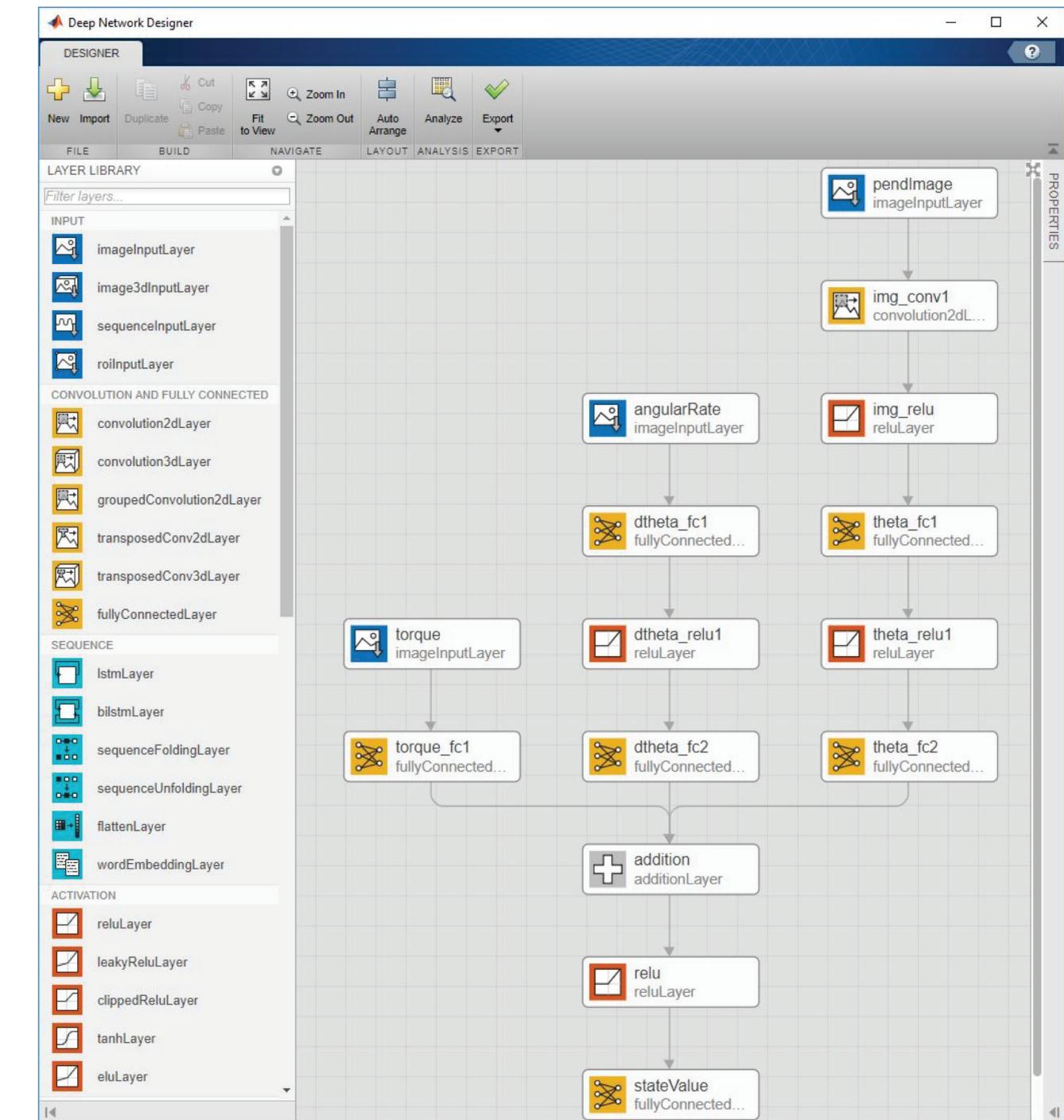
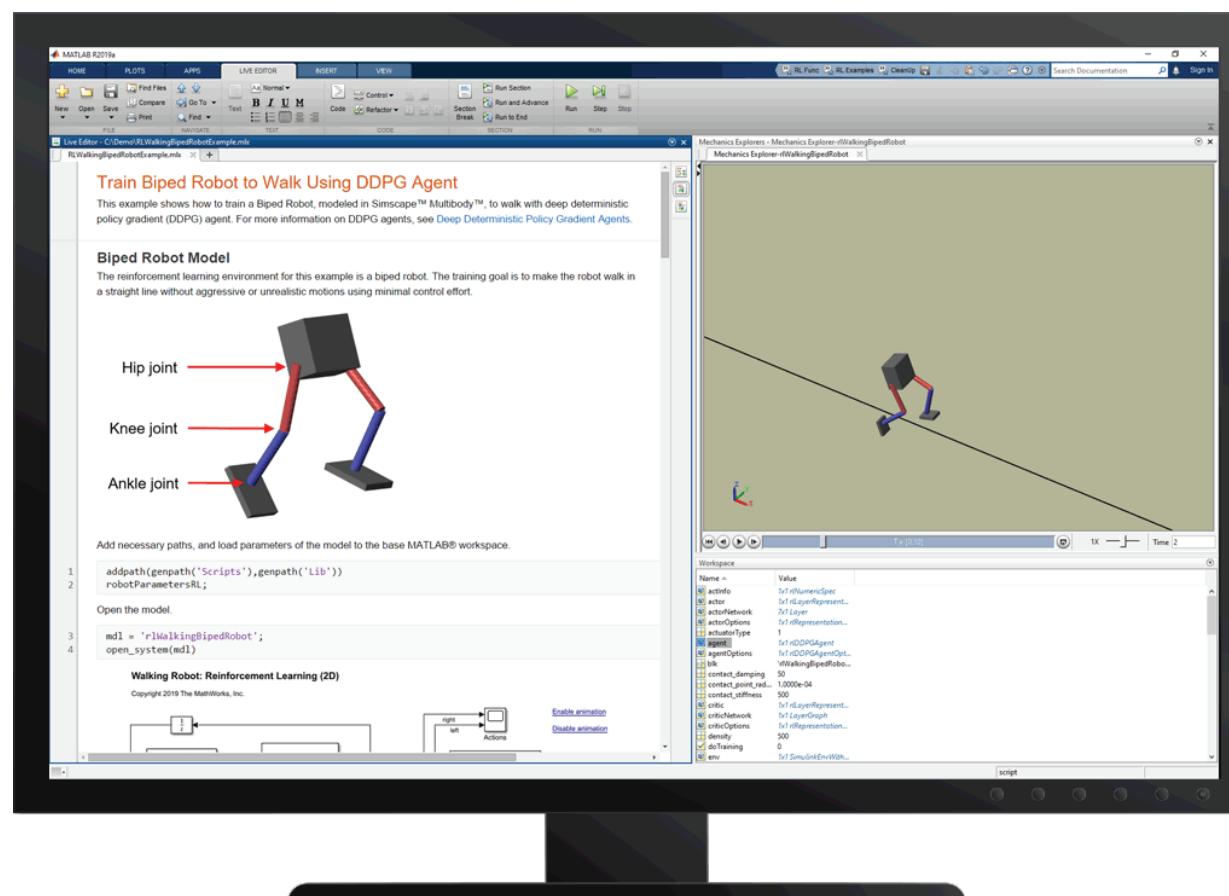
convolutional

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

