Chapter 18 强化学习

强化学习（Reinforcement learning，RL） 是当今机器学习中最令人兴奋的领域之一，也是最古老的领域之一。自 1950 年代以来，它一直存在，多年来产生了许多有趣的应用程序，特别是在游戏（例如 TD-Gammon，一种西洋双陆棋游戏程序）和机器控制方面，但很少成为头条新闻。然而，2013 年发生了一场革命，当时一家名为 DeepMind 的英国初创公司的研究人员展示了一个系统，该系统可以从头开始学习玩几乎所有 Atari 游戏，最终在大多数游戏中都优于人类，仅使用原始像素作为输入并且没有任何先前对游戏规则的了解。这是一系列惊人壮举中的第一个，最终他们的系统 AlphaGo 在 2016 年 3 月战胜了传奇的围棋职业选手李世石，并在 2017 年 5 月战胜了世界冠军柯洁。没有任何程序能够击败这个游戏的高手，更不用说世界冠军了。今天，整个 RL 领域都充满了新的想法，有着广泛的应用。

那么 DeepMind（2014 年被谷歌以超过 5 亿美元收购）是如何实现这一切的呢？ 事后看来，这似乎很简单：他们将深度学习的力量应用于强化学习领域，而且效果超出了他们最疯狂的梦想。在本章中，我将首先解释强化学习是什么以及它擅长什么，然后介绍深度强化学习中最重要的两种技术：**策略梯度（policy gradients）** 和 **深度 Q 网络（deep Q-networks）**，包括对马尔可夫决策过程的讨论。让我们开始吧！

Setup

In [1]:
```
import sys

assert sys.version_info >= (3, 7)
```

In [2]:
```
from packaging import version
import tensorflow as tf

assert version.parse(tf.__version__) >= version.parse("2.8.0")
```

In [3]:
```
import matplotlib.animation
import matplotlib.pyplot as plt

plt.rc('font', size=14)
plt.rc('axes', labelsize=14, titlesize=14)
plt.rc('legend', fontsize=14)
plt.rc('xtick', labelsize=10)
plt.rc('ytick', labelsize=10)
plt.rc('animation', html='jshtml')
```

In [4]:
```
from pathlib import Path

IMAGES_PATH = Path() / "images" / "rl"
IMAGES_PATH.mkdir(parents=True, exist_ok=True)

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = IMAGES_PATH / f"{fig_id}.{fig_extension}"
```

```
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

In [5]:
```
if not tf.config.list_physical_devices('GPU'):
    print("No GPU was detected. Neural nets can be very slow without a GPU.")
    if "google.colab" in sys.modules:
        print("Go to Runtime > Change runtime and select a GPU hardware "
              "accelerator.")
    if "kaggle_secrets" in sys.modules:
        print("Go to Settings > Accelerator and select GPU.")
```

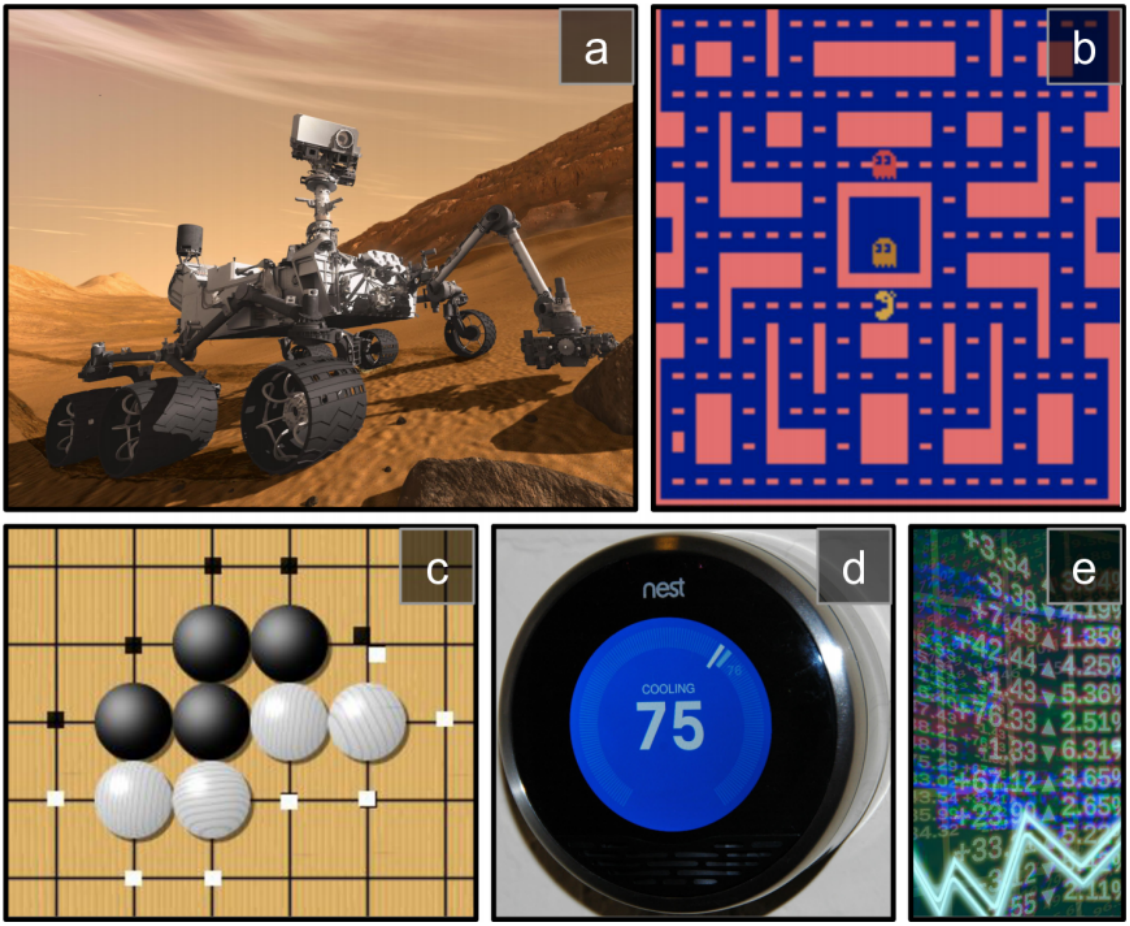
让我们安装 gym 库，它为强化学习提供了许多环境。 我们还将安装经典控制环境（包括我们将很快使用的 CartPole）以及练习所需的 Box2D 和 Atari 环境所需的额外库。

In [6]:
```
if "google.colab" in sys.modules or "kaggle_secrets" in sys.modules:
    %pip install -q -U gym
    %pip install -q -U gym[classic_control,box2d,atari,accept-rom-license]
```

1. 学习优化奖励（Learning to Optimize Rewards）

在强化学习中，软件 **代理（agent）** 在 **环境（environment）** 中进行 **观察（observations）** 并采取 **行动（actions）**，作为回报，它会从环境中获得 **奖励（rewards）**。 它的目标是学习以一种随着时间的推移最大化其预期回报的方式行事。如果你不介意有点拟人化，你可以将正面奖励视为快乐，将负面奖励视为痛苦（在这种情况下，“奖励”一词有点误导）。简而言之，智能体在环境中行动并通过反复试验来学习以最大化其快乐并最小化其痛苦。

这是一个相当广泛的设置，可以适用于各种各样的任务。下面是几个示例（参见下图）：



a. 代理可以是控制机器人的程序。在这种情况下，环境就是真实世界，智能体通过一组传感器（如摄像头和触摸传感器）观察环境，其动作包括发送信号以启动电机。它可能被编程为在接近目标目的地时获得正奖励，而在浪费时间或走错方向时获得负奖励。

b. 代理可以是控制吃豆人的程序。在这种情况下，环境是 Atari 游戏的模拟，动作是九个可能的操纵杆位置（左上、下、中心等），观察是屏幕截图，奖励只是游戏点数。

c. 类似地，代理也可以是玩棋盘游戏的程序，比如围棋。它只有赢了才会得到奖励。

d. 代理不必控制物理（或虚拟）移动的物体。例如，它可以是一个智能恒温器，当它接近目标温度并节省能源时获得正奖励，而当人类需要调整温度时获得负奖励，因此智能体必须学会预测人类的需求。

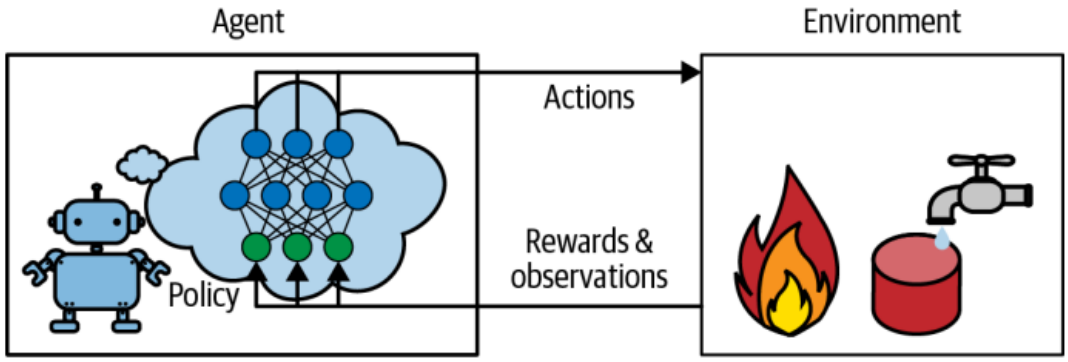
e. 代理可以观察股票市场价格，并决定每秒钟买卖多少钱。回报显然是货币上的得失。

请注意，可能根本没有任何积极的奖励；例如，agent 可能在迷宫中四处走动，每一步都得到负奖励，所以它最好尽快找到出口！还有许多其他强化学习非常适合的任务示例，例如自动驾驶汽车、推荐系统、在网页上放置广告或控制图像分类系统应将注意力集中在何处。

2. 策略搜索（Policy Search）

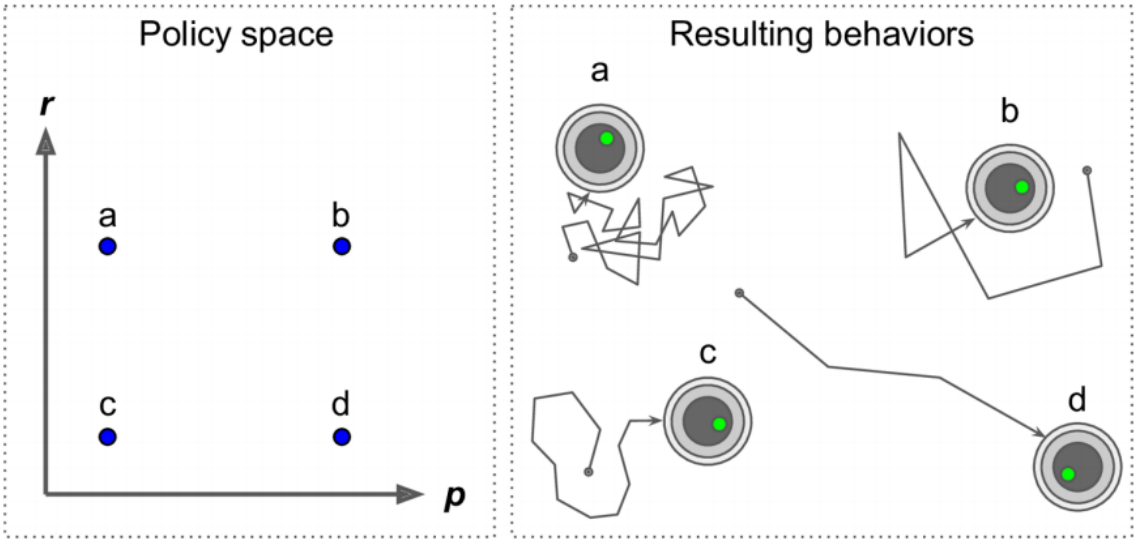
软件代理用来确定其行为的算法称为其 **策略（policy）**。该策略可以是一个神经网络，将观

察结果作为输入并输出要采取的行动（见下图）。



该策略可以是您能想到的任何算法，并且不必是确定性的。事实上，在某些情况下它甚至不必观察环境！例如，考虑一个机器人真空吸尘器，它的奖励是它在 30 分钟内吸起的灰尘量。它的策略可以是每秒以某个概率 p 向前移动，或者以概率 $1-p$ 随机向左或向右旋转。旋转角度将是 $-r$ 和 $+r$ 之间的随机角度。由于该策略涉及一些随机性，因此称为 **随机策略（stochastic policy）**。机器人将有一个不稳定的轨迹，这保证了它最终会到达它可以到达的任何地方并捡起所有灰尘。问题是，它会在 30 分钟内吸起多少灰尘？

你会如何训练这样的机器人？只有两个策略参数可以调整：概率 p 和角度范围 r。一种可能的学习算法是为这些参数尝试许多不同的值，然后选择性能最好的组合（见下图）。



这是 **策略搜索（policy search）** 的示例，在本例中使用了蛮力方法。当 **策略空间（policy space）** 太大时（通常是这种情况），以这种方式找到一组好的参数就像大海捞针一样。

探索策略空间的另一种方法是使用 **遗传算法（genetic algorithms）**。例如，您可以随机创建第一代 100 个策略并进行试验，然后“杀死”80 个最差的策略，并让 20 个幸存者各产生 4 个后代。后代是其父代的副本加上一些随机变异。幸存的策略加上他们的后代一起构成了第二代。您可以继续以这种方式迭代几代，直到找到一个好的策略。

另一种方法是使用优化技术，通过评估与策略参数相关的奖励梯度，然后通过遵循更高奖励的梯度来调整这些参数。我们将讨论这种称为 **策略梯度（policy gradients，PG）** 的方法，

本章后面会有更多详细信息。回到吸尘器机器人，你可以稍微增加 p 并评估这样做是否会增加机器人在 30 分钟内吸起的灰尘量； 如果是，则将 p 增加一些，否则减少 p。我们将使用 TensorFlow 实现一种流行的 PG 算法，但在此之前，我们需要为代理创建一个生存环境——因此是时候引入 OpenAI Gym 了。

3. 引入 OpenAI Gym（Introduction to OpenAI Gym）

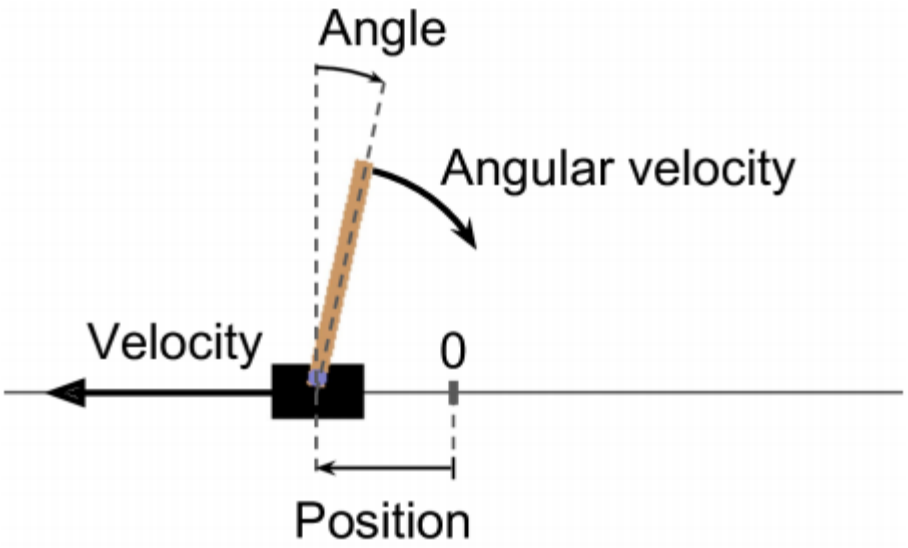
强化学习的挑战之一是，为了训练代理，您首先需要有一个工作环境。如果您想编写一个可以学习玩 Atari 游戏的代理程序，您将需要一个 Atari 游戏模拟器。如果你想编写一个步行机器人，那么环境就是现实世界，你可以直接在那个环境中训练你的机器人。但是，这有其局限性：如果机器人从悬崖上掉下来，您不能只单击撤消。你也不能加快时间——增加更多的计算能力不会让机器人移动得更快——而且并行训练 1000 个机器人通常太昂贵了。简而言之，现实世界中的训练既困难又缓慢，因此您通常至少需要一个 **模拟环境（simulated environment）** 来进行 bootstrap 训练。例如，您可以使用 **PyBullet** 或 **MuJoCo** 等库进行 3D 物理模拟。

OpenAI Gym 是一个提供各种模拟环境（Atari 游戏、棋盘游戏、2D 和 3D 物理模拟等）的工具包，您可以使用它们来训练代理、比较它们或开发新的 RL 算法。

让我们导入它并创建一个环境：

In [8]:
```
import gym

env = gym.make("CartPole-v1", render_mode="rgb_array")
```

在这里，我们创建了一个 CartPole 环境。这是一个 2D 模拟，其中小车可以向左或向右加速以平衡放置在其顶部的杆（见下图）。这是一个经典的控制任务。



注意：**gym.envs.registry** 字典包含了所有可用环境的名称和规格。

In [9]:
```
# extra code – shows the first few environments
envs = gym.envs.registry
sorted(envs.keys())[:5] + ["..."]
```

Out[9]:
```
['ALE/Adventure-ram-v5',
 'ALE/Adventure-v5',
 'ALE/AirRaid-ram-v5',
 'ALE/AirRaid-v5',
 'ALE/Alien-ram-v5',
 '...']
```

In [10]:
```
# extra code – shows the specification for the CartPole-v1 environment
envs["CartPole-v1"]
```

Out[10]:
```
EnvSpec(id='CartPole-v1', entry_point='gym.envs.classic_control.cartpole:CartPoleEnv', reward_threshold=475.0, nondeterministic=False, max_episode_steps=500, order_enforce=True, autoreset=False, disable_env_checker=False, apply_api_compatibility=False, kwargs={}, namespace=None, name='CartPole', version=1)
```

创建环境后，您必须使用 **reset()** 方法对其进行初始化，可选择指定随机种子。这将返回第一个观察结果。观察结果取决于环境类型。对于 **CartPole** 环境，每个观察值都是一个一维 NumPy 数组，其中包含四个浮点数，代表小车的水平位置（0.0 = center）、速度（正数表示向右）、杆的角度（0.0 = vertical）及其角速度（正数表示顺时针）。**reset()** 方法还返回一个字典，其中可能包含额外的特定环境的信息。这对于调试或训练很有用。例如，在许多 Atari 环境中，它包含剩余生命数。然而，在 CartPole 环境中，这个字典是空的。

In [11]:
```
obs, info = env.reset(seed=42)

obs
```

Out[11]:
```
array([ 0.0273956 , -0.00611216,  0.03585979,  0.0197368 ], dtype=float32)
```

In [12]:
```
info
```

Out[12]:
```
{}
```

让我们调用 **render()** 方法将此环境渲染为图像。由于我们在创建环境时设置了 **render_mode="rgb_array"**，图像将作为 NumPy 数组返回：

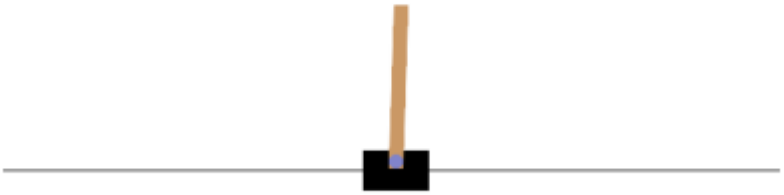
In [13]:
```
img = env.render()
img.shape  # height, width, channels (3 = Red, Green, Blue)
```

Out[13]:
```
(400, 600, 3)
```

然后，您可以像往常一样使用 Matplotlib 的 **imshow()** 函数来显示此图像。

In [14]:
```
# extra code – creates a little function to render and plot an environment

def plot_environment(env, figsize=(5, 4)):
    plt.figure(figsize=figsize)
    img = env.render()
    plt.imshow(img)
    plt.axis("off")
```

```
    return img

plot_environment(env)
plt.show()
```



现在让我们问问环境，可以采取哪些行动：

In [15]:
```
env.action_space
```

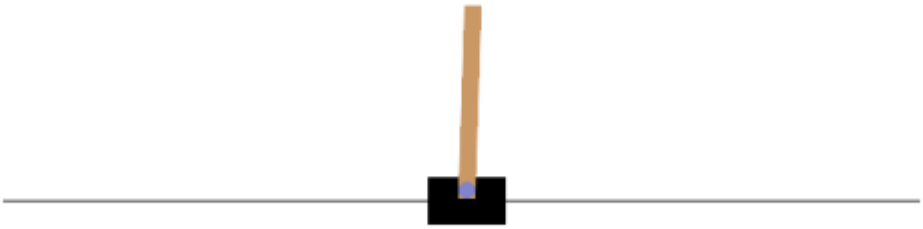
Out[15]:
```
Discrete(2)
```

Discrete(2) 表示可能的动作是整数 0 和 1，代表向左或向右加速。其他环境可能有额外的离散动作，或其他类型的动作（例如，连续的）。由于杆子向右倾斜 (**obs[2] > 0**)，让我们向右加速小车：

In [16]:
```
action = 1  # accelerate right

obs, reward, done, truncated, info = env.step(action)

obs
```

Out[16]:
```
array([ 0.02727336,  0.18847767,  0.03625453, -0.26141977], dtype=float32)
```

In [17]:
```
# extra code – displays the environment
plot_environment(env)
save_fig("cart_pole_plot")
plt.show()
```



In [18]:
```
reward
```

Out[18]:
```
1.0
```

In [19]:
```
done
```

Out[19]:
```
False
```

In [20]:
```
truncated
```

Out[20]:
```
False
```

In [21]:
```
info
```

Out[21]:
```
{}
```

In [22]:
```
if done or truncated:
    obs, info = env.reset()
```

step() 方法执行所需的操作，并返回五个值：

- **obs**：这是新的观察。小车现在向右移动 (**obs[1] > 0**)。轴仍然向右倾斜 (**obs[2] > 0**)，但它的角速度现在为负 (**obs[3] < 0**)，因此在下一步之后它可能会向左倾斜。
- **reward**：在这种环境下，无论你做什么，你每一步都能得到 1.0 的奖励，所以目标是让这种情节尽可能长时间地运行。
- **done**：当情节结束时，这个值将为真。当轴倾斜太多，或偏离屏幕，或在 200 步之后（在最后一种情况下，你赢了）。之后，必须重置环境，才能重新使用。
- **truncated**：当情节提前中断时，此值将为 True ，例如，环境包装器强加了每个情节的最大步数（有关环境包装器的更多详细信息，请参阅 Gym 的文档）。一些 RL 算法处理中断情节与正常完成的情节不同（即，当 **done** 为 **True** 时），但在本章中，我们将以相同的方式对待它们。

- **info**：这个特定环境的字典可能会提供额外的信息，就像 **reset()** 方法返回的信息一样。

注意：一旦您使用完一个环境，您就应该调用其 **close()** 方法来释放资源。

让我们硬编码一个简单的策略，当杆子向左倾斜时向左加速，当杆子向右倾斜时向右加速。我们将运行此策略以查看它获得超过 500 个情节的平均奖励：

In [23]:
```
def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1

totals = []

for episode in range(500):
    episode_rewards = 0
    obs, info = env.reset(seed=episode)
    for step in range(200):
        action = basic_policy(obs)
        obs, reward, done, truncated, info = env.step(action)
        episode_rewards += reward
        if done or truncated:
            break

    totals.append(episode_rewards)
```

这个代码是不言自明的。让我们来看看其结果：

In [24]:
```
import numpy as np

np.mean(totals), np.std(totals), min(totals), max(totals)
```

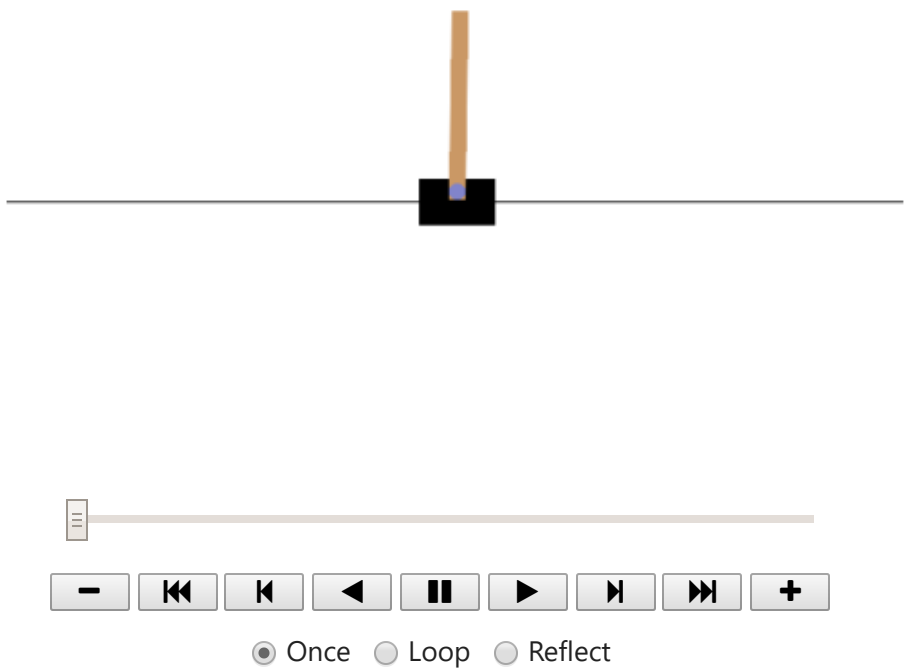
Out[24]:
```
(41.698, 8.389445512070509, 24.0, 63.0)
```

即使进行了 500 次尝试，该策略也从未成功将杆子保持直立超过 63 步。不是很好。如果你查看本章笔记本中的模拟，你会发现小车左右摆动的幅度越来越大，直到杆子倾斜太多。让我们看看神经网络是否可以提出更好的策略。

In [25]:
```
# extra code – this cell displays an animation of one episode

def update_scene(num, frames, patch):
    patch.set_data(frames[num])
    return patch,

def plot_animation(frames, repeat=False, interval=40):
    fig = plt.figure()
    patch = plt.imshow(frames[0])
    plt.axis('off')
    anim = matplotlib.animation.FuncAnimation(
        fig, update_scene, fargs=(frames, patch),
        frames=len(frames), repeat=repeat, interval=interval)
    plt.close()
    return anim
```

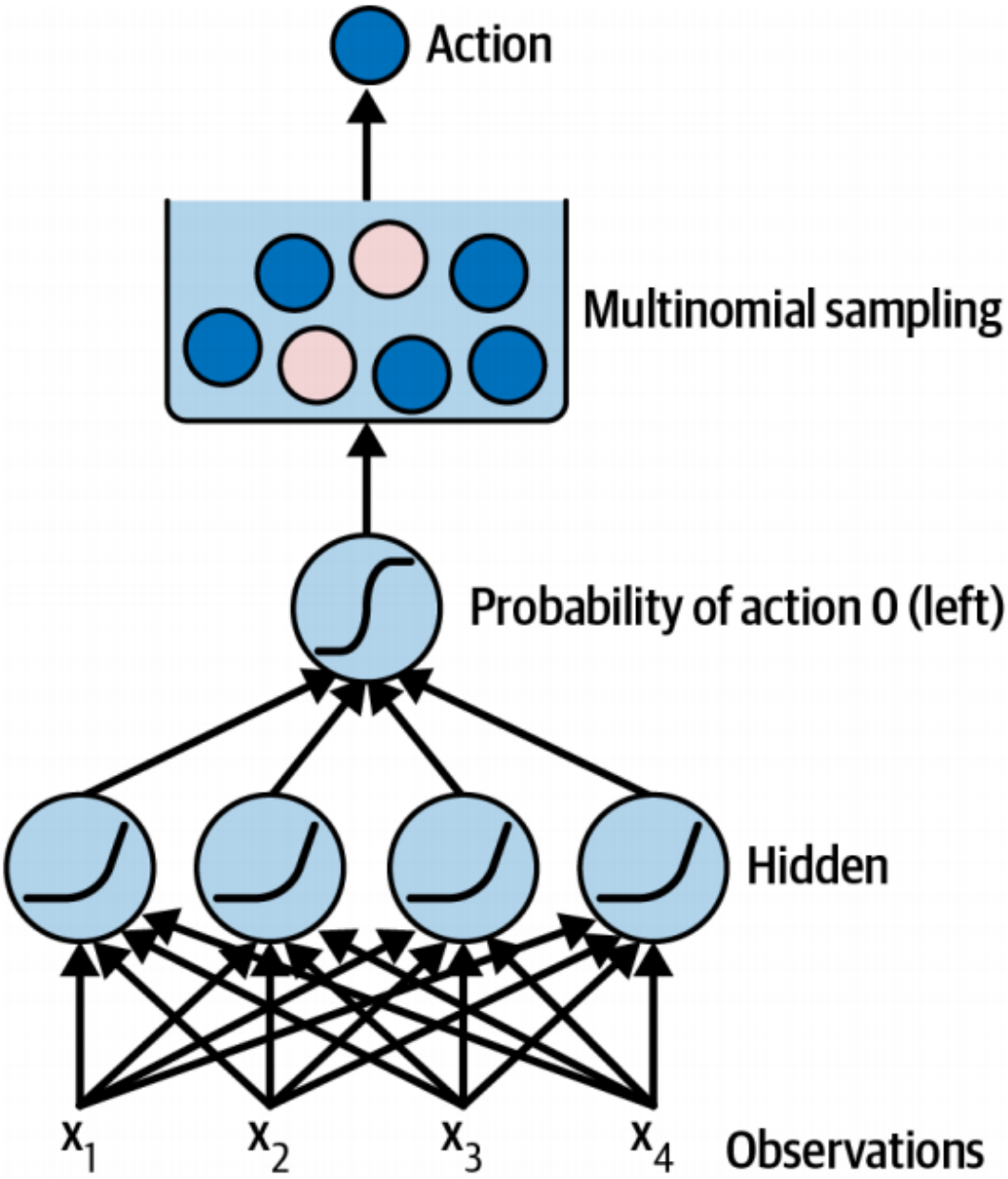
```
def show_one_episode(policy, n_max_steps=200, seed=489):
    frames = []
    env = gym.make("CartPole-v1", render_mode="rgb_array")
    np.random.seed(seed)
    obs, info = env.reset(seed=seed)
    for step in range(n_max_steps):
        frames.append(env.render())
        action = policy(obs)
        obs, reward, done, truncated, info = env.step(action)
        if done or truncated:
            break
    env.close()
    return plot_animation(frames)

show_one_episode(basic_policy)
```

Out[25]:



4. 神经网络策略（Neural Network Policies）

让我们创建一个神经网络策略。这个神经网络将观察作为输入，它会输出要执行的动作，就像我们之前硬编码的策略一样。更准确地说，它将为每个动作估计一个概率，然后我们将根据估计的概率随机选择一个动作（见下图）。



在 CartPole 环境中，只有两种可能的动作（左或右），因此我们只需要一个输出神经元。它将输出动作 0（左）的概率 p，当然动作 1（右）的概率将是 $1-p$。例如，如果它输出 0.7，那么我们将以 70% 的概率选择动作 0，或者以 30% 的概率选择动作 1。

你可能想知道为什么我们要根据神经网络给出的概率选择一个随机动作，而不是只选择得分最高的动作。这种方法让代理在 **探索（exploring）** 新动作和 **利用（exploiting）** 已知有效的动作之间找到适当的平衡。打个比方：假设你第一次去一家餐馆，所有的菜看起来都一样诱人，所以你随便挑了一个。如果好吃，可以增加下次点的概率，但概率不能增加到100%，否则你永远不会尝试其他菜品，其中一些可能甚至比你试过的更好。这种 **探索/利用困境（exploration/exploitation dilemma）** 是强化学习的核心。

还要注意，在这个特定环境中，可以安全地忽略过去的动作和观察，因为每个观察都包含环境的完整状态。如果存在某种隐藏状态，那么您可能还需要考虑过去的行为和观察。例如，如果环境仅显示小车的位置而不显示其速度，则您不仅要考虑当前观察结果，还要考虑之前

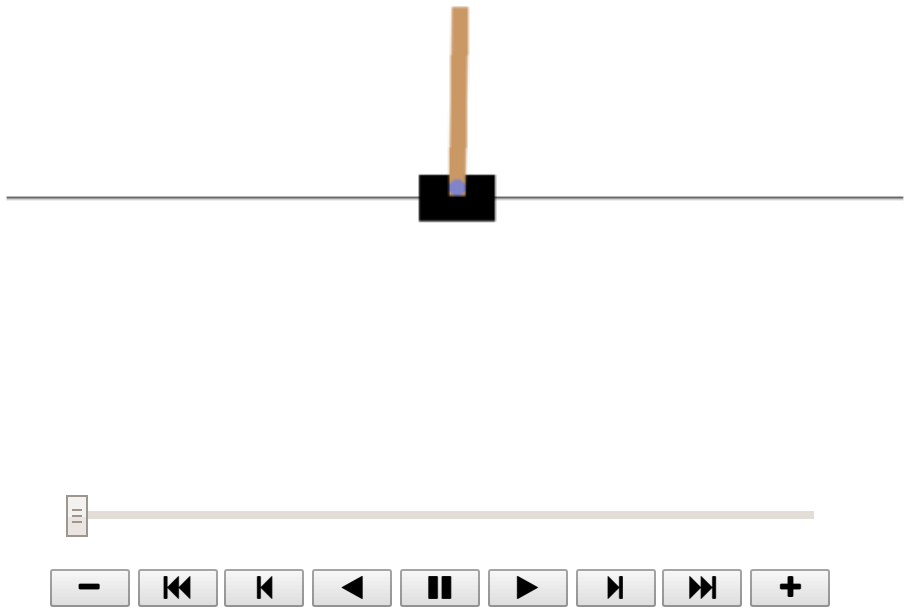
的观察结果以估计当前速度。另一个例子是当观察结果嘈杂时；在这种情况下，您通常希望使用过去的几次观察来估计最有可能的当前状态。因此，CartPole 问题非常简单；观察结果没有噪音，并且包含环境的完整状态。

下面是使用 Keras 来构建一个基本的神经网络策略的代码：

In [26]:
```
import tensorflow as tf

tf.random.set_seed(42)  # extra code – ensures reproducibility on the CPU

model = tf.keras.Sequential([
    tf.keras.layers.Dense(5, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid"),
])
```

In [27]:
```
# extra code – a function that creates an animation for a given policy model

def pg_policy(obs):
    left_proba = model.predict(obs[np.newaxis], verbose=0)
    return int(np.random.rand() > left_proba)

np.random.seed(42)
show_one_episode(pg_policy)
```

Out[27]:



我们使用顺序模型来定义策略网络。输入的数量是观察空间的大小——在 CartPole 的情况下是——我们只有五个隐藏单元，因为这是一个相当简单的任务。最后，我们想要输出一个单一的概率——向左走的概率——所以我们有一个使用 sigmoid 激活函数的单一输出神经元。如果有两个以上的可能动作，每个动作将有一个输出神经元，我们将改用 softmax 激活函数。

好的，我们现在有了一个神经网络策略，它将进行观察和输出动作概率。但是我们该如何训练它呢？

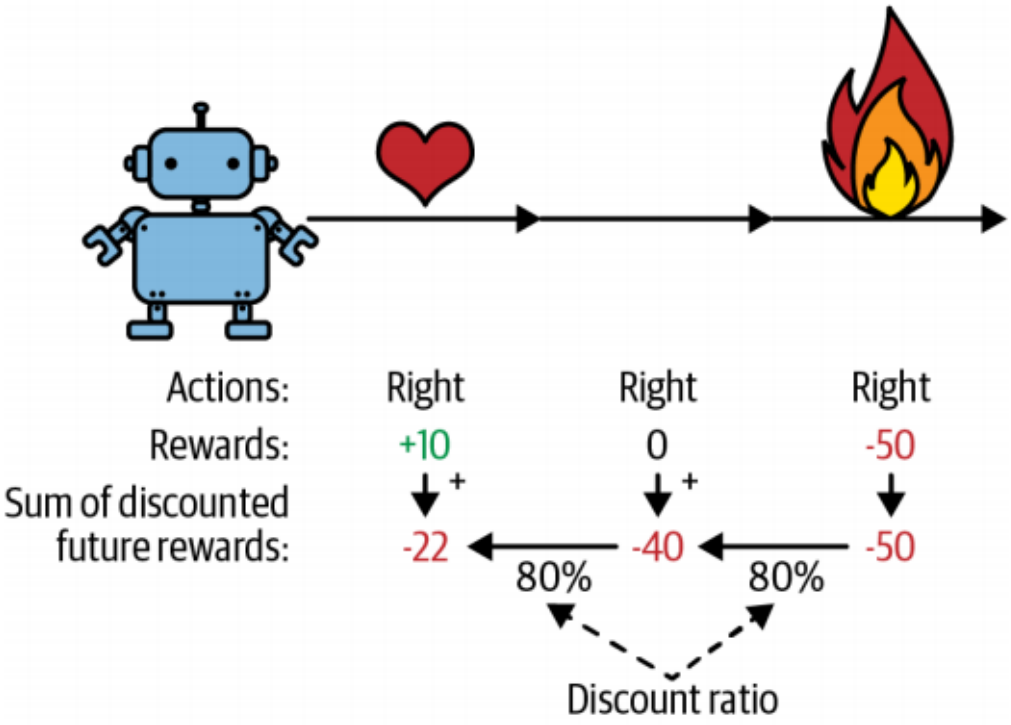
5. 评估行动：信用分配问题（Evaluating Actions: The Credit Assignment Problem）

如果我们知道每一步的最佳动作是什么，我们就可以像往常一样训练神经网络，方法是最小化估计概率分布和目标概率分布之间的交叉熵。这只是常规的监督学习。 然而，在强化学习中，代理获得的唯一指导是通过奖励，而奖励通常是稀疏和延迟的。例如，如果代理设法平衡杆子 100 步，它怎么知道它采取的 100 个动作中哪些是好的，哪些是坏的？它只知道杆子在最后一个动作后掉了下来，但肯定这最后一个动作不完全负责。这被称为 **信用分配问题**

（**credit assignment problem**）：当代理获得奖励时，它很难知道哪些行为应该被记入（或归咎于）它。想想一只狗，它在表现良好数小时后获得奖励；它会明白它的奖励是什么吗？

为了解决这个问题，一个常见的策略是根据一个动作之后所有奖励的总和来评估一个动作，通常在每一步都应用一个 **折扣因子（discount factor）** γ（gamma）。这笔折扣奖励的总和称为行动的 **回报（return）**。

考虑下图中的示例。



如果代理决定连续三次向右走，并在第一步后获得 +10 奖励，在第二步后获得 0，最后在第三步后获得 –50，那么假设我们使用折扣因子 $\gamma = 0.8$，第一步行动的回报为 $10 + \gamma \times 0 + \gamma^2 \times (-50) = -22$。如果折扣因子接近于 0，那么与即时奖励相比，未来的奖励将没有多大意义。相反，如果折扣因子接近 1，那么遥远未来的奖励几乎与即时奖励一样重要。典型的折扣系数从 0.9 到 0.99 不等。折扣系数为 0.95 时，未来 13 步的奖励大约是即时奖励的一半（因为 0.9513 ≈ 0.5），而贴现系数为 0.99 时，未来 69 步的奖励是即时奖励的一半。在 CartPole 环境中，操作具有相当短期的效果，因此选择 0.95 的折扣因子似乎是合理的。

当然，一个好的动作后面可能会有几次坏动作，导致杆子很快掉下来，导致好的动作得到的回报很低。同样，一个好演员有时可能会出演一部糟糕的电影。然而，如果我们玩这个游戏的次数足够多，平均而言，好的行为会比坏的行为得到更高的回报。我们想估计一个动作与其他可能的动作相比平均好坏多少。这称为 **行动优势（action advantage）**。为此，我们必须运行许多情节并通过减去均值并除以标准差来归一化所有的动作回报。之后，我们可以合理地假设具有负面优势的行为是坏的，而具有正面优势的行为是好的。好的，既然我们有了评估每个动作的方法，我们就可以准备使用策略梯度来训练我们的第一个代理了。让我们看看如何实现。

6. 策略梯度（Policy Gradients）

如前所述，PG 算法通过遵循更高奖励的梯度来优化策略的参数。一种流行的 PG 算法，称为 **REINFORCE** 算法，由 Ronald Williams 于 1992 年引入。这是一种常见的变体：

1. 首先，让神经网络策略玩几次游戏，在每一步计算出使选择的动作更有可能的梯度，但还没有应用这些梯度。
2. 运行了几个情节后，请使用上一节中描述的方法计算每个行动优势。
3. 如果一个行动优势是正的，这意味着动作可能是好的，并且您希望应用之前计算的梯度，使动作更有可能在未来被选择。然而，如果行动优势是负的，这就意味着动作可能是坏的，你想要应用相反的梯度，使这个动作在未来的可能性稍小一些。解决方案是将每个梯度向量乘以相应的行动优势。
4. 最后，计算所有结果梯度向量的平均值，并使用它来执行一个梯度下降步骤。

让我们使用 Keras 来实现这个算法。我们将训练我们之前构建的神经网络策略，以便它学会平衡手推车上的杆子。首先，我们需要一个玩一步的函数。我们现在假设它采取的任何行动都是正确的，以便我们可以计算损失及其梯度。这些梯度只会保存一段时间，我们稍后会根据动作的好坏来修改它们：

In [28]:
```
def play_one_step(env, obs, model, loss_fn):
    with tf.GradientTape() as tape:
        left_proba = model(obs[np.newaxis])
        action = (tf.random.uniform([1, 1]) > left_proba)
        y_target = tf.constant([[1.]]) - tf.cast(action, tf.float32)
        loss = tf.reduce_mean(loss_fn(y_target, left_proba))

    grads = tape.gradient(loss, model.trainable_variables)
    obs, reward, done, truncated, info = env.step(int(action))
    return obs, reward, done, truncated, grads
```

上述代码中，

- 在 **GradientTape** 块中（见第 12 章），我们首先调用模型，给它一个单独的观察值。我们重塑观察，使其成为包含单个实例的批次，因为模型需要一个批次。这输出向左走的概率。
- 接下来，我们采样一个介于 0 和 1 之间的随机浮点数，并检查它是否大于 **left_proba**。该动作将以概率 **left_proba** 为假，或以概率 **1–left_proba** 为真。一旦我们将此布尔值转换为整数，动作将是 0（左）或 1（右）并具有适当的概率。
- 我们现在定义向左走的目标概率：它是 1 减去动作（转换为浮点数）。如果动作是 0（左），那么向左走的目标概率就是 1。如果动作是 1（右），那么目标概率就是 0。
- 然后我们使用给定的损失函数来计算损失，并我们使用 tape 来计算关于模型的可训练变量的损失梯度。同样，在我们应用它们之前，这些梯度将会被调整，这取决于动作的好坏。
- 最后，我们执行选定的动作，并返回新的观察结果、奖励、情节是否结束、是否被中断，当然还有我们刚刚计算的梯度。

现在让我们创建另一个函数，它将依赖于 **play_one_step()** 函数来播放多个情节，返回每一个

情节和每一步的所有奖励和梯度：

In [29]:
```
def play_multiple_episodes(env, n_episodes, n_max_steps, model, loss_fn):
    all_rewards = []
    all_grads = []

    for episode in range(n_episodes):
        current_rewards = []
        current_grads = []
        obs, info = env.reset()
        for step in range(n_max_steps):
            obs, reward, done, truncated, grads = play_one_step(
                env, obs, model, loss_fn)
            current_rewards.append(reward)
            current_grads.append(grads)
            if done or truncated:
                break

        all_rewards.append(current_rewards)
        all_grads.append(current_grads)

    return all_rewards, all_grads
```

此代码返回奖励列表列表：每个剧情一个奖励列表，每步包含一个奖励。它还返回一个梯度列表列表：每个情节一个梯度列表，每个梯度列表包含每个步骤的一个梯度元组，每个元组包含每个可训练变量的一个梯度张量。

该算法将使用 **play_multiple_episodes()** 函数多次玩游戏（例如 10 次），然后它会返回并查看所有奖励，对它们进行打折并归一化。为此，我们需要更多函数；第一个将计算每一步的未来折扣奖励总和，第二个将通过减去平均值并除以标准差来标准化所有这些折扣奖励（即回报）：

In [30]:
```
def discount_rewards(rewards, discount_factor):
    discounted = np.array(rewards)
    for step in range(len(rewards) - 2, -1, -1):
        discounted[step] += discounted[step + 1] * discount_factor
    return discounted

def discount_and_normalize_rewards(all_rewards, discount_factor):
    all_discounted_rewards = [discount_rewards(rewards, discount_factor)
                              for rewards in all_rewards]
    flat_rewards = np.concatenate(all_discounted_rewards)
    reward_mean = flat_rewards.mean()
    reward_std = flat_rewards.std()
    return [(discounted_rewards - reward_mean) / reward_std
            for discounted_rewards in all_discounted_rewards]
```

让我们来看看这个方法是否有效：

In [31]:
```
discount_rewards([10, 0, -50], discount_factor=0.8)
```

Out[31]:
```
array([-22, -40, -50])
```

In [32]:
```
discount_and_normalize_rewards([[10, 0, -50], [10, 20]],
                               discount_factor=0.8)
```

Out[32]:
```
[array([-0.28435071, -0.86597718, -1.18910299]),
 array([1.26665318, 1.0727777 ])]
```

对 **discount_rewards()** 的调用返回的正是我们所期望的。您可以验证函数 **discount_and_normalize_rewards()** 确实返回了两个情节中每个动作的标准化动作优势。注意第一集比第二集差很多，所以它的归一化优势都是负的； 第一集中的所有行为都被认为是坏的，相反，第二集中的所有行为都被认为是好的。

我们几乎准备好运行算法了！现在让我们定义超参数。我们将运行 150 次训练迭代，每次迭代播放 10 集，每集最多持续 200 步。我们将使用 0.95 的折扣系数：

In [33]:
```
n_iterations = 150
n_episodes_per_update = 10
n_max_steps = 200
discount_factor = 0.95
```

我们还需要优化器和损失函数。学习率为 0.01 的常规 Nadam 优化器就可以了，我们将使用二元交叉熵损失函数，因为我们正在训练一个二元分类器（有两种可能的动作——左或右）：

In [34]:
```
# extra code – let's create the neural net and reset the environment, for
#              reproducibility

tf.random.set_seed(42)

model = tf.keras.Sequential([
    tf.keras.layers.Dense(5, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid"),
])

obs, info = env.reset(seed=42)
```

In [35]:
```
optimizer = tf.keras.optimizers.Nadam(learning_rate=0.01)
loss_fn = tf.keras.losses.binary_crossentropy
```

我们现在已经准备好建立和运行训练循环了！

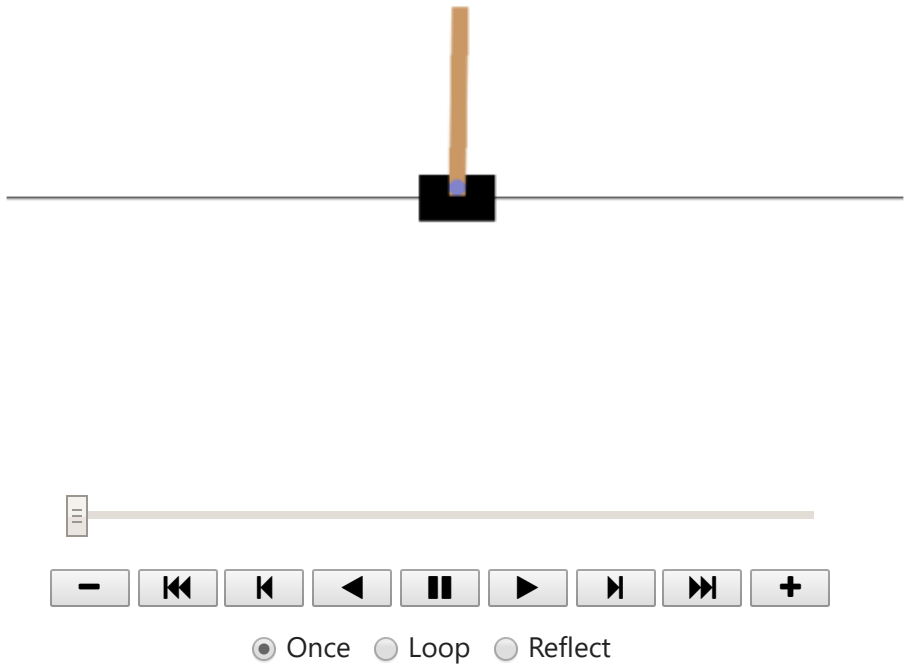
In [35]:
```
for iteration in range(n_iterations):
    all_rewards, all_grads = play_multiple_episodes(
        env, n_episodes_per_update, n_max_steps, model, loss_fn)

    # extra code – displays some debug info during training
    total_rewards = sum(map(sum, all_rewards))
    print(f"\rIteration: {iteration + 1}/{n_iterations},"
          f" mean rewards: {total_rewards / n_episodes_per_update:.1f}", end="")

    all_final_rewards = discount_and_normalize_rewards(all_rewards,
                                                       discount_factor)
    all_mean_grads = []
```

```
    for var_index in range(len(model.trainable_variables)):
        mean_grads = tf.reduce_mean(
            [final_reward * all_grads[episode_index][step][var_index]
             for episode_index, final_rewards in enumerate(all_final_rewards)
                 for step, final_reward in enumerate(final_rewards)], axis=0)
        all_mean_grads.append(mean_grads)

    optimizer.apply_gradients(zip(all_mean_grads, model.trainable_variables))
```

```
Iteration: 150/150, mean rewards: 186.7
```

In [36]:
```
# extra code – displays the animation
np.random.seed(42)
show_one_episode(pg_policy)
```

Out[36]:



如上代码中，

- 在每次训练迭代中，此循环调用 **play_multiple_episodes()** 函数，该函数播放 10 集并返回每集中每一步的奖励和梯度。
- 然后我们调用 **discount_and_normalize_rewards()** 函数来计算每个动作的归一化优势，在这段代码中称为 **final_reward**。事后看来，这提供了衡量每个行动实际上有多好或有多坏的衡量标准。

- 接下来，我们遍历每个可训练变量，并为每个变量计算该变量在所有情节和所有步骤上的梯度加权平均值，并由 **final_reward** 加权。
- 最后，我们使用优化器应用这些平均梯度：模型的可训练变量将被调整，希望策略会更好一些。

我们完成了！这段代码将训练神经网络策略，它会成功学会平衡手推车上的杆子。 每集的平均奖励将非常接近 200。默认情况下，这是该环境的最大值。成功！

我们刚刚训练的简单策略梯度算法解决了 CartPole 任务，但它不能很好地扩展到更大更复杂的任务。事实上，它的 **样本效率（sample inefficient）** 非常低，这意味着它需要探索游戏很长时间才能取得重大进展。这是因为它必须运行多个情节来估计每个动作的优势，正如我们所见。然而，它是更强大算法的基础，例如 actor-critic 算法（我们将在本章末尾简要讨论）。

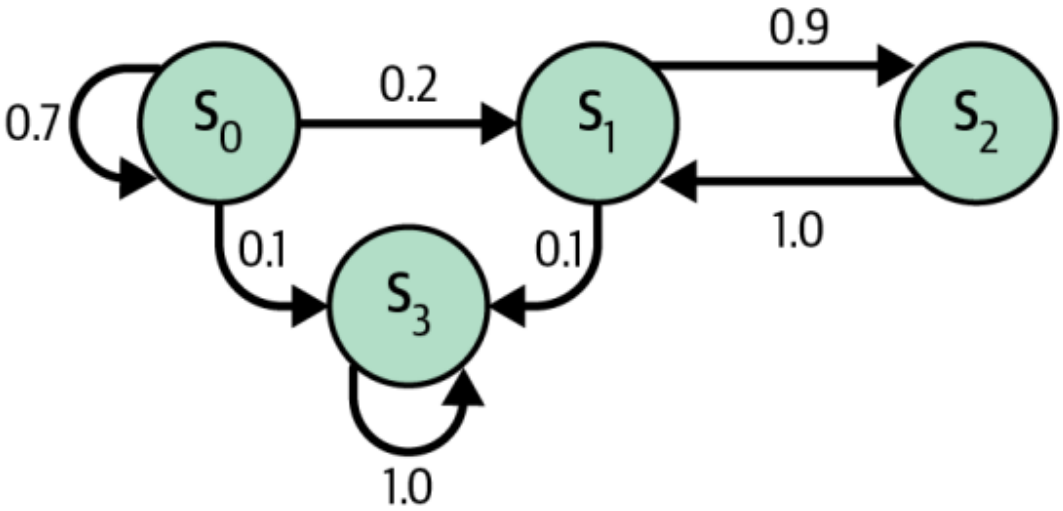
注意：研究人员试图找到即使在代理最初对环境一无所知的情况下也能正常工作的算法。然而，除非你正在写论文，否则你应该毫不犹豫地将先验知识注入代理，因为它会大大加快训练速度。例如，既然你知道杆子应该尽可能垂直，你可以添加与杆子角度成比例的负奖励。这将使奖励更加稀疏并加快训练速度。此外，如果您已经有一个相当好的策略（例如，硬编码），您可能希望在使用策略梯度改进它之前训练神经网络模仿它。

我们现在将研究另一个流行的算法系列。PG 算法直接尝试优化策略以增加奖励，而我们现在要探索的算法则不那么直接：代理学习估计每个状态或每个状态下的每个动作的预期回报，然后它使用这些知识来决定如何行动。要理解这些算法，我们必须首先考虑 **马尔可夫决策过程（Markov decision processes，MDP）**。

7. 马尔可夫决策过程（Markov Decision Processes）

20 世纪初，数学家 Andrey Markov 研究了没有记忆的随机过程，称为 **马尔可夫链（Markov chains）**。这样的过程具有固定数量的状态，并且在每一步随机地从一种状态演化到另一种状态。它从状态 s 演化到状态 s' 的概率是固定的，并且仅取决于对 (s, s')，而不取决于过去的状态。这就是为什么我们说系统没有记忆。

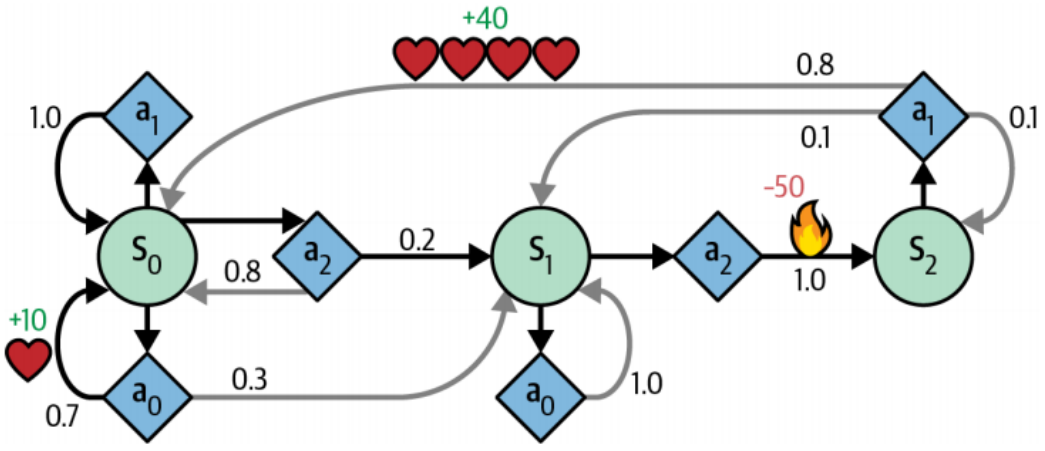
下图显示了一个具有四种状态的马尔可夫链的例子：



假设过程以状态 s_0 开始，并且有 70% 的机会在下一步中保持该状态。最终它必然会离开该状态并且永远不会回来，因为没有其他状态指向 s_0 。如果它进入状态 s_1 ，那么它很可能会进入状态 s_2 （90% 的概率），然后立即返回状态 s_1 （100% 的概率）。它可能会在这两种状态之间交替多次，但最终它会落入状态 s_3 并永远停留在那里，因为没有出路：这称为终止状态。马尔可夫链可以有非常不同的动力学，它们被大量用于热力学、化学、统计学等等。

Richard Bellman 在 1950 年代首次描述了马尔可夫决策过程。它们类似于马尔可夫链，但有一个转折点：在每一步，代理可以从几个可能的动作中选择一个，并且转换概率取决于所选动作。此外，一些状态转换会返回一些奖励（正面或负面），而代理的目标是找到一个随着时间的推移将奖励最大化的策略。

例如，下图中表示的 MDP 有三个状态（用圆圈表示），在每个步骤中最多有三个可能的离散操作（用菱形表示）。



如果它从状态 s_0 开始，代理可以在动作 a_0、a_1 或 a_2 之间进行选择。如果它选择动作 a_1，它只会确定地保持在状态 s_0，并且没有任何奖励。因此，如果它愿意，它可以决定永远留在那里。但如果它选择动作 a_0，它有 70% 的概率获得 +10 的奖励并保持状态 s0。然后它可以一次又一次地尝试获得尽可能多的奖励，但在某一时刻它会以状态 s_1 结束。在状态 s_1 中，它只有两个可能的动作：a_0 或 a_2 。它可以通过重复选择动作 a_0 来选择留在原地，也可以选择继续进入状态 s_2 并获得 –50 的负奖励。在状态 s_2 它别无选择，只能采取行动 a_1 ，这很可

能会导致它回到状态 s_0，并在途中获得 +40 的奖励。你明白了。通过查看这个 MDP，你能猜出哪个策略会随着时间的推移获得最多的回报吗？在状态 s_0 中，行动 a_0 显然是最佳选择，在状态 s_2 中，智能体别无选择，只能采取行动 a_1，但在状态 s_1 中，智能体应该留在原地（a_0）还是穿过火海并不明显（a_2）。

Bellman 找到了一种方法来估计任何状态 s 的 **最优状态值（optimal state value）**，记为 $V^*(s)$，它是代理在达到该状态后平均期望的所有折扣未来奖励的总和，假设它的行为是最优的。他表明，如果代理行为最优，则适用 **贝尔曼最优方程（Bellman optimality equation）**（参见如下方程）。这个递归方程表示，如果代理采取最优行动，那么当前状态的最优值等于它在采取一个最优行动后平均获得的奖励，加上该行动可能导致的所有可能的下一个状态的预期最优值。

贝尔曼最优方程：

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \cdot V^*(s')] \ for\ all\ s$$

方程中，

- $T(s,a,s')$ 是从状态 s 到状态 s' 的转移概率，给定代理选择了动作 a。例如，在上图中，$T(s_2,a_1,s_0) = 0.8$。
- $R(s,a,s')$ 是当代理从状态 s 转到状态 s' 时获得的奖励，且代理选择了动作 a。例如，在上图中，$R(s_2,a_1,s_0) = +40$。
- γ 是折扣因子。

这个等式直接引出了一种算法，可以精确估计每个可能状态的最优状态值：首先将所有状态值估计初始化为零，然后使用值迭代算法迭代更新它们（见如下公式）。一个显着的结果是，如果有足够的时间，这些估计可以保证收敛到最优状态值，对应于最优策略。

数值迭代算法：

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \cdot V_k(s')] \ for\ all\ s$$

在这个方程中，$V_k(s)$ 是算法在第 k 次迭代时的状态 s 的估计值。

注意：该算法是 **动态规划（dynamic programming）** 的一个例子，它将一个复杂的问题分解为可处理的子问题，可以迭代处理。

了解最佳状态值可能很有用，尤其是对评估策略而言，但它并不能为我们提供代理的最佳策略。幸运的是，Bellman 找到了一种非常相似的算法来估计最佳 **状态-动作值（state-action values）**，通常称为 **Q-values（quality values）**。状态-动作对 (s,a) 的最佳 Q-value，记为 $Q^*(s,a)$，是代理在达到状态 s 并选择动作 a 后平均期望的折扣未来奖励之和，但是在它看到这个动作的结果之前，假设它在那个动作之后表现最佳。

让我们看看它是如何工作的。同样，您首先将所有 Q-value 估计值初始化为零，然后使用 **Q-**

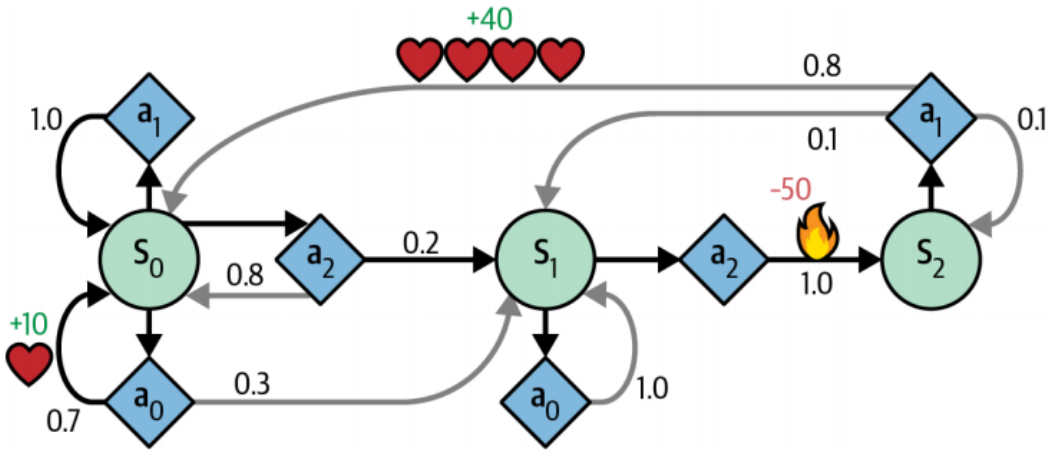
value 迭代（Q-value iteration） 算法更新它们（请参见如下公式）。

Q-value 迭代算法：

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \cdot \max_{a'} Q_k(s',a')] \; for \; all \; (s,a)$$

一旦有了最佳 Q-values，定义最佳策略（记为 $\pi^*(s)$）就很简单了；当代理处于状态 s 时，它应该为该状态选择具有最高 Q-value 的动作：$\pi^*(s) = \arg\max_a Q^*(s,a)$ 。

让我们将此算法应用于下图中所示的 MDP 。



首先，我们需要定义 MDP ：

In [36]:
```
transition_probabilities = [  # shape=[s, a, s']
    [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
    [[0.0, 1.0, 0.0], None, [0.0, 0.0, 1.0]],
    [None, [0.8, 0.1, 0.1], None]
]

rewards = [  # shape=[s, a, s']
    [[+10, 0, 0], [0, 0, 0], [0, 0, 0]],
    [[0, 0, 0], [0, 0, 0], [0, 0, -50]],
    [[0, 0, 0], [+40, 0, 0], [0, 0, 0]]
]

possible_actions = [[0, 1, 2], [0, 2], [1]]
```

例如，要知道在执行动作 a_1 后从 s_2 到 s_0 的转移概率，我们将查找 `transition_probabilities[2][1][0]`（即 0.8）。同样，要获得相应的奖励，我们将查找 `rewards[2][1][0]`（即 +40）。为了获得 s_2 中可能的动作列表，我们将查找 `possible_actions[2]`（在这种情况下，只有动作 a_1 是可能的）。接下来，我们必须将所有 Q-values 初始化为零（除了不可能的动作，我们将 Q-values 设置为 $-\infty$）：

In [37]:
```
Q_values = np.full((3, 3), -np.inf)  # -np.inf for impossible actions
```

```
for state, actions in enumerate(possible_actions):
    Q_values[state, actions] = 0.0  # for all possible actions
```

现在让我们运行 Q-value 迭代算法。它对所有 Q-values、每个状态和每个可能的动作重复应用公式：

In [38]:
```
gamma = 0.90  # the discount factor

history1 = []  # extra code – needed for the figure below

for iteration in range(50):
    Q_prev = Q_values.copy()
    history1.append(Q_prev) # extra code
    for s in range(3):
        for a in possible_actions[s]:
            Q_values[s, a] = np.sum([
                    transition_probabilities[s][a][sp]
                    * (rewards[s][a][sp] + gamma * Q_prev[sp].max())
                for sp in range(3)])

history1 = np.array(history1)  # extra code
```

就这样！所得到的 Q-values 如下所示：

In [39]:
```
Q_values
```

Out[39]:
```
array([[18.91891892, 17.02702702, 13.62162162],
       [ 0.        ,        -inf, -4.87971488],
       [       -inf, 50.13365013,        -inf]])
```

例如，当代理处于状态 s_0，并且它选择动作 a_1 时，折扣未来奖励的期望和约为 17.0。

对于每个状态，我们都可以找到具有最高 Q-value 的动作：

In [40]:
```
Q_values.argmax(axis=1)
```

Out[40]:
```
array([0, 0, 1], dtype=int64)
```

当使用 0.90 的折扣因子时，这为我们提供了此 MDP 的最优策略：在状态 s_0 中选择动作 a_0，在状态 s_1 中选择动作 a_0（即保持原状），在状态 s_2 中选择动作 a_1（唯一可能的动作）。有趣的是，如果我们将折扣因子增加到 0.95，最优策略会发生变化：在状态 s_1 中，最佳动作变为 a_2（穿越火海！）。这是有道理的，因为你越看重未来的回报，你就越愿意为了未来的幸福而忍受现在的一些痛苦。

8. 时间差分学习（Temporal Difference Learning）

具有离散动作的强化学习问题通常可以建模为马尔可夫决策过程，但代理最初并不知道转移概率是多少（它不知道 $T(s, a, s')$），也不知道奖励将是什么（它不知道 $R(s, a, s')$）。它必须至少经历一次每个状态和每个转换才能知道奖励，如果要对转换概率有一个合理的估计，它必须经历多次。

时间差分（temporal difference，TD） 学习算法与 Q-value 迭代算法非常相似，但进行了调整以考虑到代理仅具有 MDP 的部分知识这一事实。一般来说，我们假设代理最初只知道可能的状态和动作，仅此而已。代理使用 **探索策略（exploration policy）**（例如，纯随机策略）来探索 MDP，随着它的进展，TD 学习算法会根据实际观察到的转换和奖励更新状态值的估计值（请参见下式).

TD 学习算法：

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$$

或等价地，

$$V_{k+1}(s) \leftarrow V_k(s) + \alpha \cdot \delta_k(s,r,s') \text{ , } with\ \delta_k(s,r,s') = r + \gamma \cdot V_k(s') - V_k(s)$$

方程中，

- α 是学习率（例如，0.01）。
- $r + \gamma \cdot V_k(s')$ 称为 TD target。
- $\delta_k(s,r,s')$ 称为 TD error。

写这个方程的第一种形式的一种更简洁的方法是使用符号 $a \leftarrow_\alpha b$ ，即 $a_{k+1} \leftarrow (1-\alpha) \cdot a_k + \alpha \cdot b_k$ 。所以，方程的第一行可以这样重写：$V(s) \leftarrow_\alpha r + \gamma \cdot V(s')$ 。

注意：TD 学习与随机梯度下降有许多相似之处，包括它一次处理一个样本这一事实。而且，就像 SGD 一样，只有逐渐降低学习率，它才能真正收敛；否则，它会一直在最佳 Q-values 附近跳动。

对于每个状态 s，该算法跟踪代理在离开该状态时获得的即时奖励的运行平均值，加上它期望稍后获得的奖励，假设它表现最佳。

9. Q-Learning

类似地，Q-learning 算法是 Q-value 迭代算法的一种改编，适用于转移概率和奖励最初未知的情况（请参见如下公式）。Q-learning 的工作原理是观察代理玩（例如，随机）并逐渐改进其对 Q-values 的估计。一旦它有了准确的 Q-values 估计（或足够接近），那么最优策略就是选择具有最高 Q-value 的动作（即贪心策略）。

Q-learning 算法：

$$Q(s,a) \leftarrow_\alpha r + \gamma \cdot \max_{a'} Q(s',a')$$

对于每个状态-动作对 (s,a)，该算法会跟踪代理在使用动作 a 离开状态 s 时获得的奖励 r 的运行平均值，加上它期望获得的折扣未来奖励的总和。为了估计这个总和，我们采用下一个状态 s' 的 Q-value 估计的最大值，因为我们假设目标策略从那时起将采取最佳行动。

让我们来实现 Q-learning 算法。首先，我们需要让代理探索环境。为此，我们需要一个阶跃函数，以便代理可以执行一个动作并获得结果状态和奖励：

In [41]:
```
def step(state, action):
    probas = transition_probabilities[state][action]
    next_state = np.random.choice([0, 1, 2], p=probas)
    reward = rewards[state][action][next_state]
    return next_state, reward
```

现在让我们实施代理的探索策略。由于状态空间非常小，一个简单的随机策略就足够了。如果我们运行该算法足够长的时间，代理将多次访问每个状态，并且还会多次尝试每个可能的操作：

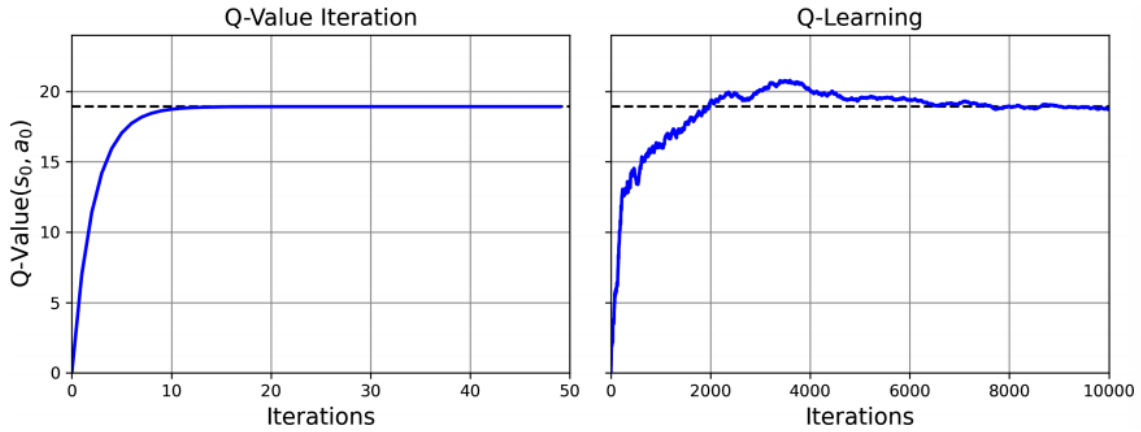
In [42]:
```
def exploration_policy(state):
    return np.random.choice(possible_actions[state])
```

接下来，在我们像之前一样初始化 Q-values 之后，我们准备运行学习率衰减的 Q-learning 算法（使用功率调度，在第 11 章中介绍）：

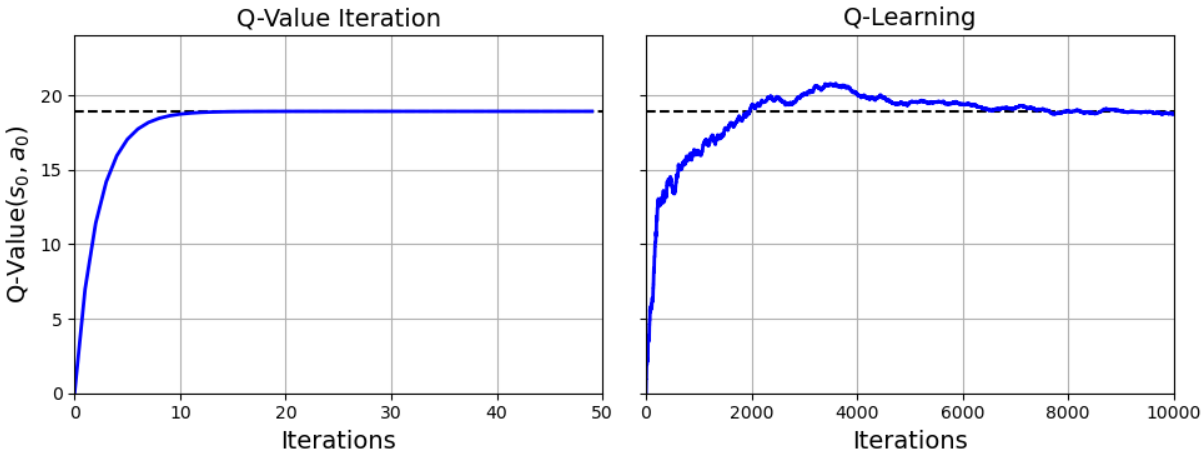
In [43]:
```
# extra code – initializes the Q-Values, just like earlier
np.random.seed(42)
Q_values = np.full((3, 3), -np.inf)
for state, actions in enumerate(possible_actions):
    Q_values[state][actions] = 0
```

In [44]:
```
alpha0 = 0.05  # initial learning rate
decay = 0.005  # learning rate decay
gamma = 0.90  # discount factor
state = 0  # initial state
history2 = []  # extra code – needed for the figure below

for iteration in range(10_000):
    history2.append(Q_values.copy())  # extra code

    action = exploration_policy(state)
    next_state, reward = step(state, action)
    next_value = Q_values[next_state].max()  # greedy policy at the next step
    alpha = alpha0 / (1 + iteration * decay)
    Q_values[state, action] *= 1 - alpha
    Q_values[state, action] += alpha * (reward + gamma * next_value)
    state = next_state

history2 = np.array(history2)  # extra code
```

该算法将收敛到最佳 Q-values，但需要多次迭代，并且可能需要进行大量超参数调整。如下图所示，Q-value 迭代算法（左）收敛速度非常快，迭代次数少于 20 次，而 Q-learning 算法（右）需要大约 8000 次迭代才能收敛。显然，不知道转移概率或奖励会使找到最优策略变得更加困难！



In [45]:
```
# extra code – this cell generates and saves Figure 18–9

true_Q_value = history1[-1, 0, 0]

fig, axes = plt.subplots(1, 2, figsize=(10, 4), sharey=True)
axes[0].set_ylabel("Q-Value$(s_0, a_0)$", fontsize=14)
axes[0].set_title("Q-Value Iteration", fontsize=14)
axes[1].set_title("Q-Learning", fontsize=14)
for ax, width, history in zip(axes, (50, 10000), (history1, history2)):
    ax.plot([0, width], [true_Q_value, true_Q_value], "k--")
    ax.plot(np.arange(width), history[:, 0, 0], "b-", linewidth=2)
    ax.set_xlabel("Iterations", fontsize=14)
    ax.axis([0, width, 0, 24])
    ax.grid(True)

save_fig("q_value_plot")
plt.show()
```



Q-learning 算法被称为 **off-policy** 算法，因为被训练的策略不一定是训练期间使用的策略。例如，在我们刚刚运行的代码中，正在执行的策略（探索策略）是完全随机的，而从未使用过正在训练的策略。训练后，最优策略对应于系统地选择具有最高 Q 值的动作。相反，策略梯度算法是一种 **on-policy** 算法：它使用正在训练的策略来探索世界。令人惊讶的是，Q-learning 能够通过观察代理的随机行为来学习最优策略。想象一下，当你的老师是一只蒙着眼睛的猴子时而你正在学习打高尔夫球。我们能做得更好吗？

9.1 探索策略（Exploration Policies）

当然，只有探索策略对 MDP 的探索足够彻底，Q-learning 才能发挥作用。虽然一个纯随机的策略保证最终会多次访问每个状态和每个转换，但这样做可能需要非常长的时间。因此，更好的选择是使用 **ε-greedy** 策略（ϵ 是 epsilon）：在每一步它以概率 ϵ 随机行动，或者以概率 $1-\epsilon$ 贪婪行动（即选择具有最高 Q-value 的行动）。**ε-greedy** 策略（相对于完全随机的策略）的优势在于它会花费越来越多的时间去探索环境中有趣的部分，随着 Q-value 估计越来越好，同时仍然花一些时间去访问 MDP 的未知区域。从 ϵ 的高值（例如 1.0）开始，然后逐渐降低它（例如，降至 0.05）是很常见的。

或者，另一种方法不是仅依靠机会进行探索，而是鼓励探索策略尝试以前没有尝试过的动作。这可以作为添加到 Q-value 估计值的奖励来实现，如下公式所示。

使用探索函数进行 Q-learning：

$$Q(s,a) \leftarrow_{\alpha} r + \gamma \cdot \max_{a'} f(Q(s',a'), N(s',a'))$$

方程中，

- $N(s',a')$ 计算在状态 s' 中选择动作 a' 的次数。
- $f(Q,N)$ 是一个 **探索函数（exploration function）**，如 $f(Q,N) = Q + \kappa/(1+N)$，其中 κ 是一个好奇超参数，衡量代理对未知的吸引程度。

9.2 近似 Q-learning 和深度 Q-learning（Approximate Q-Learning and Deep Q-Learning）

Q-learning 的主要问题是它不能很好地扩展到具有许多状态和动作的大型（甚至中型）MDPs。例如，假设您想使用 Q-learning 来训练代理玩吃豆人。吃豆人可以吃掉大约 150 个小球，每个小球都可以存在或不存在（即已被吃掉）。因此，可能状态的数量大于 $2^{150} \approx 10^{45}$。如果你把所有幽灵和吃豆人的所有可能位置组合加起来，可能状态的数量就会比我们星球上的原子数量还要多，所以你绝对无法跟踪每一个 Q-value。

解决方案是找到一个函数 $Q_{\boldsymbol{\theta}}(s,a)$，它使用可管理数量的参数（由参数向量 $\boldsymbol{\theta}$ 给出）来近似任何状态-动作对 (s,a) 的 Q-value。这称为 **近似 Q-learning（approximate Q-learning）**。多年来，一直建议使用从状态中提取的手工特征的线性组合（例如，最近的幽灵的距离、它们的方向等）来估计 Q-values，但在 2013 年，DeepMind 表明使用深度神经网络可以工作得更好，特别是对于复杂的问题，而且它不需要任何特征工程。用于估计 Q-values 的 DNN 称为 **深度 Q-network（deep Q-network，DQN）**，使用 DQN 进行近似 Q-learning 称为 **深度 Q-learning（deep Q-learning）**。

现在，我们如何训练 DQN？那么，考虑一下 DQN 为给定的状态-动作对 (s,a) 计算的近似 Q-value。感谢 Bellman，我们知道我们希望这个近似 Q-value 尽可能接近我们在状态 s 中执行动作 a 后实际观察到的奖励 r，再加上从那时起最佳播放的折扣值。为了估计未来折扣奖励的总和，我们可以在下一个状态 s' 上对所有可能的动作 a' 执行 DQN。我们为每个可能的动作得到一个近似的未来 Q-value 。然后我们选择最高的（因为我们假设我们会玩得最好）

并对它打折，这给了我们对未来打折奖励总和的估计。通过将奖励 r 和未来折扣值估计相加，我们得到状态-动作对 (s, a) 的目标 Q-value $y(s, a)$，如下公式所示。

目标 **Q-value**：

$$y(s, a) = r + \gamma \cdot \max_{a'} Q_{\boldsymbol{\theta}}(s', a')$$

有了这个目标 Q-value，我们可以使用任何梯度下降算法运行训练步骤。具体来说，我们一般会尝试最小化估计 Q -value $Q_{\boldsymbol{\theta}}(s, a)$ 和目标 Q-value $y(s, a)$ 之间的平方误差，或者 Huber 损失，以降低算法对大误差的敏感性。这就是深度 Q-learning 算法！让我们看看如何实现它来解决 CartPole 环境。

10. 实施深度 Q-Learning（Implementing Deep Q-Learning）

我们首先需要的是深度 Q-network。理论上，我们需要一个以状态-动作对作为输入并输出近似 Q-value 的神经网络。然而，在实践中，使用仅将状态作为输入并为每个可能的动作输出一个近似 Q-value 的神经网络效率要高得多。要解决 CartPole 环境，我们不需要非常复杂的神经网络；几个隐藏层会有效：

In [46]:
```
tf.random.set_seed(42)  # extra code – ensures reproducibility on the CPU

input_shape = [4]  # == env.observation_space.shape
n_outputs = 2  # == env.action_space.n

model = tf.keras.Sequential([
    tf.keras.layers.Dense(32, activation="elu", input_shape=input_shape),
    tf.keras.layers.Dense(32, activation="elu"),
    tf.keras.layers.Dense(n_outputs)
])
```

要使用此 DQN 选择动作，我们选择具有最大预测 Q-value 的动作。为确保代理探索环境，我们将使用 **ε-greedy** 策略（即，我们将选择概率为 ϵ 的随机动作）：

In [47]:
```
def epsilon_greedy_policy(state, epsilon=0):
    if np.random.rand() < epsilon:
        return np.random.randint(n_outputs)  # random action
    else:
        Q_values = model.predict(state[np.newaxis], verbose=0)[0]
        return Q_values.argmax()  # optimal action according to the DQN
```

我们不会仅根据最新经验来训练 DQN，而是将所有经验存储在 回放缓冲区（**replay buffer**） 或 回放内存（**replay memory**） 中，并且我们将在每次训练迭代时从中抽取随机训练批次。这有助于减少训练批次中经验之间的相关性，从而极大地帮助训练。为此，我们将只使用双端队列 (deque)：

In [48]:
```
from collections import deque
```

```
replay_buffer = deque(maxlen=2000)
```

注意：deque 是一个可以在两端有效地添加或删除元素的队列。从队列末端插入和删除项目非常快，但是当队列变长时，随机访问可能会很慢。如果你需要一个非常大的重放缓冲区，你应该使用一个循环缓冲区（参见 notebook 的实现），或者查看 DeepMind 的 Reverb 库。

In [49]:
```
# extra code – A basic circular buffer implementation

class ReplayBuffer:
    def __init__(self, max_size):
        self.buffer = np.empty(max_size, dtype=object)
        self.max_size = max_size
        self.index = 0
        self.size = 0

    def append(self, obj):
        self.buffer[self.index] = obj
        self.size = min(self.size + 1, self.max_size)
        self.index = (self.index + 1) % self.max_size

    def sample(self, batch_size):
        indices = np.random.randint(self.size, size=batch_size)
        return self.buffer[indices]
```

每个经验将由六个元素组成：状态 s、代理采取的动作 a、结果奖励 r、它到达的下一个状态 s'、指示情节是否在该点结束（完成）的布尔值，以及最后另一个布尔值，指示情节是否在该点被截断。我们将需要一个小函数来从重放缓冲区中随机抽取一批经验。它将返回对应于六个经验元素的六个 NumPy 数组：

In [50]:
```
def sample_experiences(batch_size):

    indices = np.random.randint(len(replay_buffer), size=batch_size)
    batch = [replay_buffer[index] for index in indices]

    return [
        np.array([experience[field_index] for experience in batch])
        for field_index in range(6)
    ]  # [states, actions, rewards, next_states, dones, truncateds]
```

我们还创建一个函数，它将使用 **ε-greedy** 策略播放单个步骤，然后将结果经验存储在重播缓冲区中：

In [51]:
```
def play_one_step(env, state, epsilon):
    action = epsilon_greedy_policy(state, epsilon)
    next_state, reward, done, truncated, info = env.step(action)
    replay_buffer.append((state, action, reward, next_state, done, truncated))
    return next_state, reward, done, truncated, info
```

最后，让我们创建最后一个函数，它将从回放缓冲区中抽取一批经验，并通过对该批次执行单个梯度下降步骤来训练 DQN：

In [52]:
```
# extra code – for reproducibility, and to generate the next figure
env.reset(seed=42)
np.random.seed(42)
tf.random.set_seed(42)
rewards = []
best_score = 0
```

In [53]:
```
batch_size = 32
discount_factor = 0.95
optimizer = tf.keras.optimizers.Nadam(learning_rate=1e-2)
loss_fn = tf.keras.losses.mean_squared_error

def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones, truncateds = experiences
    next_Q_values = model.predict(next_states, verbose=0)
    max_next_Q_values = next_Q_values.max(axis=1)
    runs = 1.0 - (dones | truncateds)  # episode is not done or truncated
    target_Q_values = rewards + runs * discount_factor * max_next_Q_values
    target_Q_values = target_Q_values.reshape(-1, 1)
    mask = tf.one_hot(actions, n_outputs)
    with tf.GradientTape() as tape:
        all_Q_values = model(states)
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
        loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))

    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

如上代码，

- 首先我们定义一些超参数，然后创建优化器和损失函数。
- 然后我们创建 **training_step()** 函数。它首先对一批经验进行采样，然后使用 DQN 预测每个经验下一个状态下每个可能动作的 Q-value。由于我们假设代理将以最佳方式进行游戏，因此我们只保留每个下一个状态的最大 Q-value。接下来，我们使用公式计算每个经验的状态-动作对的目标 Q-value。
- 我们想使用 DQN 来计算每个经历过的状态-动作对的 Q-value，但 DQN 也会输出其他可能动作的 Q-value，而不仅仅是代理实际选择的动作。因此，我们需要屏蔽掉所有不需要的 Q-value。**tf.one_hot()** 函数可以将动作索引数组转换为此类掩码。例如，如果前三个经验分别包含动作 1、1、0，则掩码将以 [[0, 1], [0, 1], [1, 0],...] 开头。然后我们可以将 DQN 的输出与这个掩码相乘，这会将我们不需要的所有 Q-value 清零。然后我们对轴 1 求和以去除所有零，只保留经历过的状态-动作对的 Q-values。这为我们提供了 **Q_values** 张量，其中包含批次中每个经验的一个预测 Q 值。
- 接下来，我们计算损失：它是经验状态动作对的目标 Q-values 和预测 Q-values 之间的均方误差。
- 最后，我们执行梯度下降步骤以最小化模型可训练变量的损失。

这是最难的部分。 现在训练模型很简单：

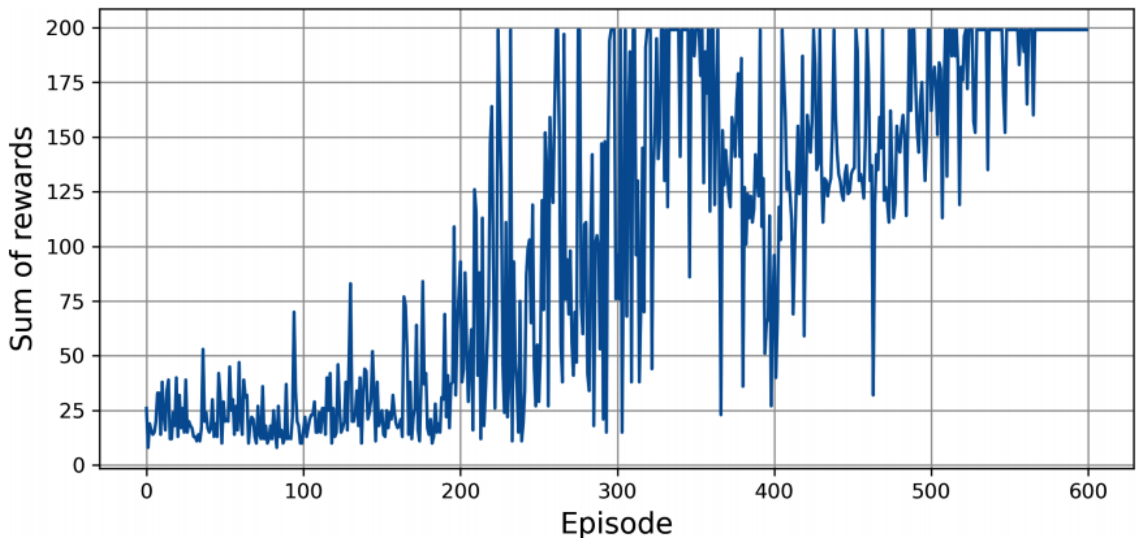
In [56]:
```
for episode in range(600):
    obs, info = env.reset()
```

```
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
        obs, reward, done, truncated, info = play_one_step(env, obs, epsilon)
        if done or truncated:
            break

    # extra code – displays debug info, stores data for the next figure, and
    #              keeps track of the best model weights so far
    print(f"\rEpisode: {episode + 1}, Steps: {step + 1}, eps: {epsilon:.3f}",
          end="")
    rewards.append(step)
    if step >= best_score:
        best_weights = model.get_weights()
        best_score = step

    if episode > 50:
        training_step(batch_size)

model.set_weights(best_weights)  # extra code – restores the best model weights
```

```
Episode: 600, Steps: 144, eps: 0.010
```

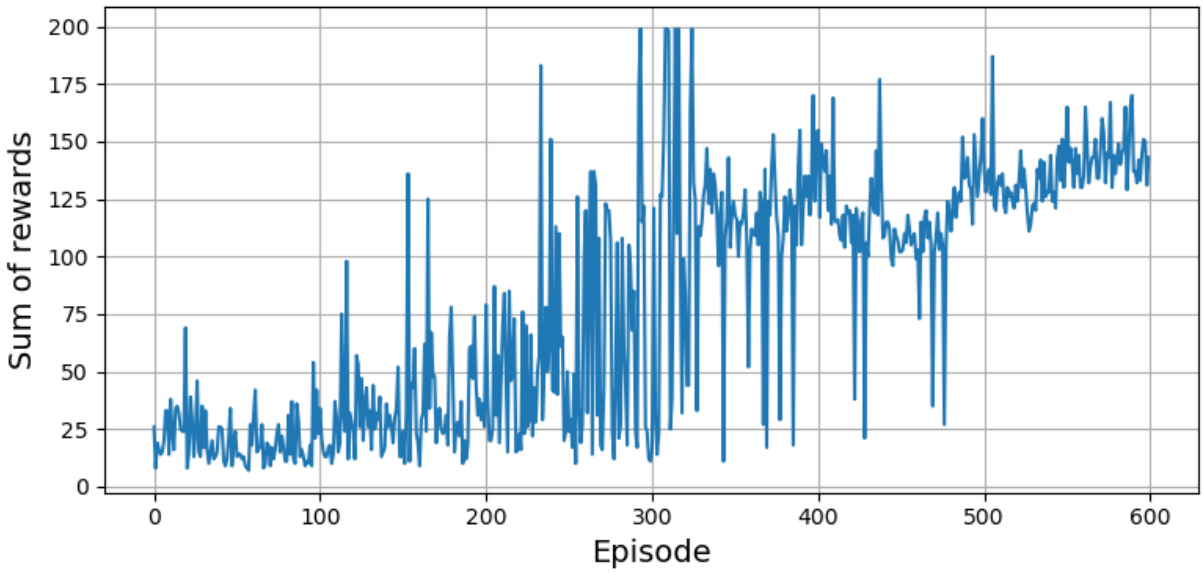
我们运行 600 集，每集最多 200 步。在每一步，我们首先计算 **ε-greedy** 策略的 **epsilon** 值：它将在不到 500 集的时间内从 1 线性下降到 0.01。然后我们调用 **play_one_step()** 函数，这将使用 **ε-greedy** 策略来选择一个动作，然后执行它并将经验记录在重放缓冲区中。如果剧集完成或被截断，我们将退出循环。最后，如果我们已经过了第 50 集，我们将调用 **training_step()** 函数在从回放缓冲区中采样的一批上训练模型。我们在没有训练的情况下播放许多剧集的原因是为了给回放缓冲区一些时间来填满（如果我们等待的时间不够，那么回放缓冲区中就不会有足够的多样性）。就是这样：我们刚刚实现了深度 Q-learning 算法！

下图显示了代理在每一集中获得的总奖励。



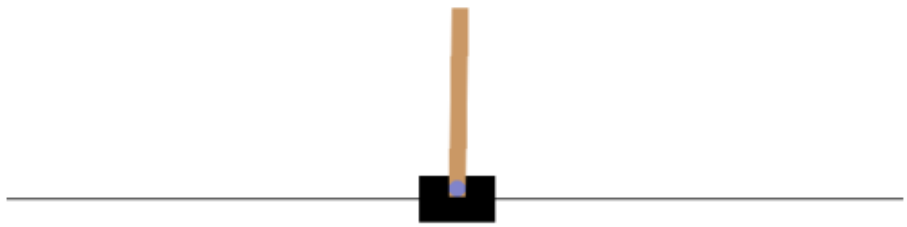
In [57]:
```
# extra code – this cell generates and saves Figure 18–10
plt.figure(figsize=(8, 4))
plt.plot(rewards)
plt.xlabel("Episode", fontsize=14)
plt.ylabel("Sum of rewards", fontsize=14)
```

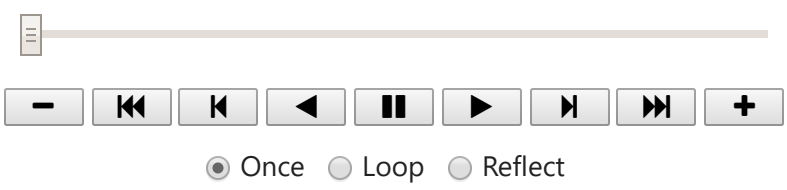
```
plt.grid(True)
save_fig("dqn_rewards_plot")
plt.show()
```



In [58]:
```
# extra code – shows an animation of the trained DQN playing one episode
show_one_episode(epsilon_greedy_policy)
```

Out[58]:





如您所见，该算法需要一段时间才能开始学习所有东西，部分原因是开始时 ϵ 非常高。然后它的进度是不稳定的：它在第 220 集左右首先达到了最大奖励，但它立即下降，然后上下反弹了几次，然后很快看起来终于稳定在最大奖励附近，在第 320 集左右，它的分数再次大幅下降。这被称为 **灾难性遗忘（catastrophic forgetting）**，它是几乎所有 RL 算法都面临的大问题之一：当代理探索环境时，它会更新其策略，但它在环境的一部分中学到的东西可能会破坏它之前在环境其他的一部分地方学到的东西。经验非常相关，学习环境不断变化——这对梯度下降来说并不理想！如果增加重放缓冲区的大小，该算法将较少受到此问题的影响。调整学习率也可能有所帮助。但事实是，强化学习很难：训练通常不稳定，您可能需要尝试许多超参数值和随机种子才能找到效果很好的组合。例如，如果您尝试将激活函数从“elu”更改为“relu”，性能会低很多。

注意：强化学习是出了名的困难，主要是因为训练不稳定以及对超参数值和随机种子的选择非常敏感。正如研究员 Andrej Karpathy 所说，“[Supervised learning] wants to work. [...] RL must be forced to work”。你需要时间、耐心、毅力，也许还需要一点运气。这是 RL 不像常规深度学习（例如，卷积网络）那样被广泛采用的主要原因。但是除了 AlphaGo 和 Atari 游戏之外，还有一些现实世界的应用程序：例如，谷歌使用 RL 来优化其数据中心成本，它被用于一些机器人应用程序、超参数调整和推荐系统。

您可能想知道为什么我们没有绘制损失图。事实证明，损失是模型性能的一个糟糕指标。损失可能会下降，但代理可能会表现更差（例如，当代理卡在环境的一个小区域并且 DQN 开始过度拟合该区域时，可能会发生这种情况）。相反，损失可能会上升，但代理可能会表现更好（例如，如果 DQN 低估了 Q-value 并且它开始正确地增加其预测，代理可能会表现更好，获得更多奖励，但损失可能会增加，因为 DQN 还设置了目标，目标也会更大）。因此，最好绘制奖励。

到目前为止，我们一直在使用的基本深度 Q-learning 算法对于学习玩 Atari 游戏来说太不稳定了。那么 DeepMind 是怎么做到的呢？好吧，他们调整了算法！

11. 深度 Q-Learning 变体（Deep Q-Learning Variants）

让我们来看看深度 Q-learning 算法的一些变体，它可以稳定和加速训练。

11.1 固定 Q-value 目标（Fixed Q-value Targets）

在基本的深度 Q-learning 算法中，模型既用于进行预测，也用于设置自己的目标。这可能导致类似于狗追自己尾巴的情况。这种反馈循环会使网络变得不稳定：它会发散、振荡、冻结等等。为了解决这个问题，在他们 2013 年的论文中，DeepMind 研究人员使用了两个 DQN 而不是一个：第一个是在线模型，它在每一步都学习并用于移动代理，另一个是目标模型仅用于定义目标。目标模型只是在线模型的克隆：

In [54]:
```
# extra code – creates the same DQN model as earlier

tf.random.set_seed(42)
```

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(32, activation="elu", input_shape=input_shape),
    tf.keras.layers.Dense(32, activation="elu"),
    tf.keras.layers.Dense(n_outputs)
])
```

In [55]:
```
target = tf.keras.models.clone_model(model)  # clone the model's architecture

target.set_weights(model.get_weights())  # copy the weights
```

然后，在 **training_step()** 函数中，我们只需要更改一行以在计算下一个状态的 Q-value 时使用目标模型而不是在线模型：

In [56]:
```
env.reset(seed=42)
np.random.seed(42)
tf.random.set_seed(42)
rewards = []
best_score = 0

batch_size = 32
discount_factor = 0.95
optimizer = tf.keras.optimizers.Nadam(learning_rate=1e-2)
loss_fn = tf.keras.losses.mean_squared_error

replay_buffer = deque(maxlen=2000)  # resets the replay buffer

def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones, truncateds = experiences
    next_Q_values = target.predict(next_states, verbose=0)  # <= CHANGED
    max_next_Q_values = next_Q_values.max(axis=1)
    runs = 1.0 - (dones | truncateds)  # episode is not done or truncated
    target_Q_values = rewards + runs * discount_factor * max_next_Q_values
    target_Q_values = target_Q_values.reshape(-1, 1)
    mask = tf.one_hot(actions, n_outputs)
    with tf.GradientTape() as tape:
        all_Q_values = model(states)
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
        loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))

    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

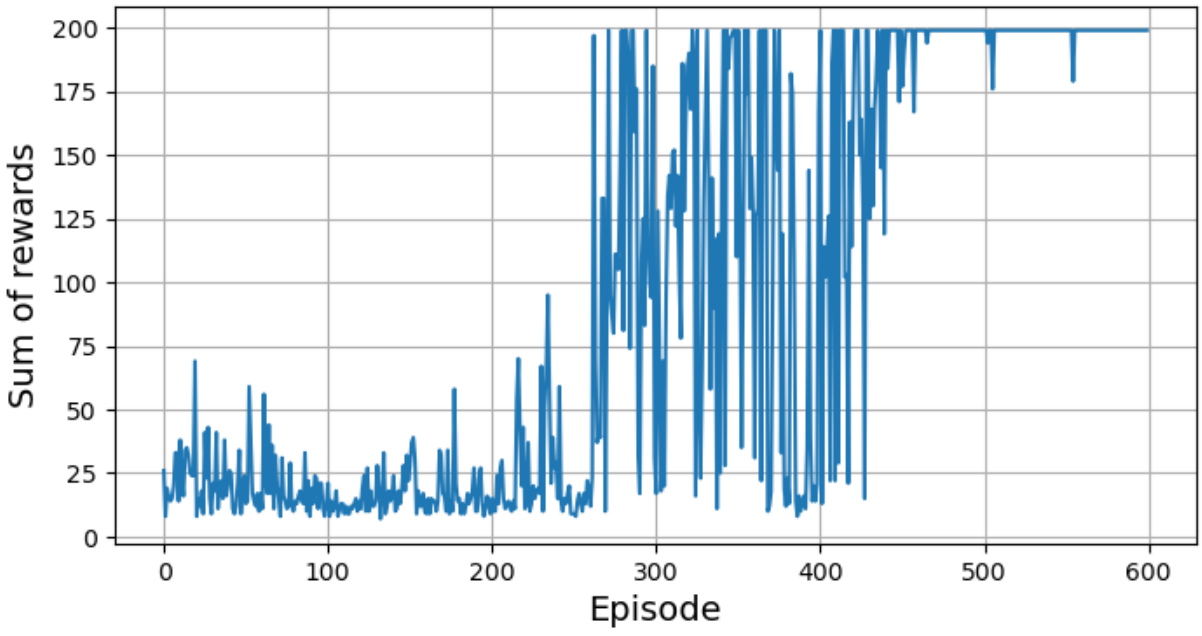
最后，在训练循环中，我们必须定期（例如，每 50 集）将在线模型的权重复制到目标模型：

In [62]:
```
for episode in range(600):
    obs, info = env.reset()
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
        obs, reward, done, info, truncated = play_one_step(env, obs, epsilon)
        if done or truncated:
            break
```

```
    # extra code – displays debug info, stores data for the next figure, and
    #              keeps track of the best model weights so far
    print(f"\rEpisode: {episode + 1}, Steps: {step + 1}, eps: {epsilon:.3f}",
          end="")
    rewards.append(step)
    if step >= best_score:
        best_weights = model.get_weights()
        best_score = step

    if episode > 50:
        training_step(batch_size)
        if episode % 50 == 0:                          # <= CHANGED
            target.set_weights(model.get_weights())  # <= CHANGED

    # Alternatively, you can do soft updates at each step:
    #if episode > 50:
        #training_step(batch_size)
        #target_weights = target.get_weights()
        #online_weights = model.get_weights()
        #for index, online_weight in enumerate(online_weights):
        #    target_weights[index] = (0.99 * target_weights[index]
        #                             + 0.01 * online_weight)
        #target.set_weights(target_weights)

model.set_weights(best_weights)  # extra code – restores the best model weights
```

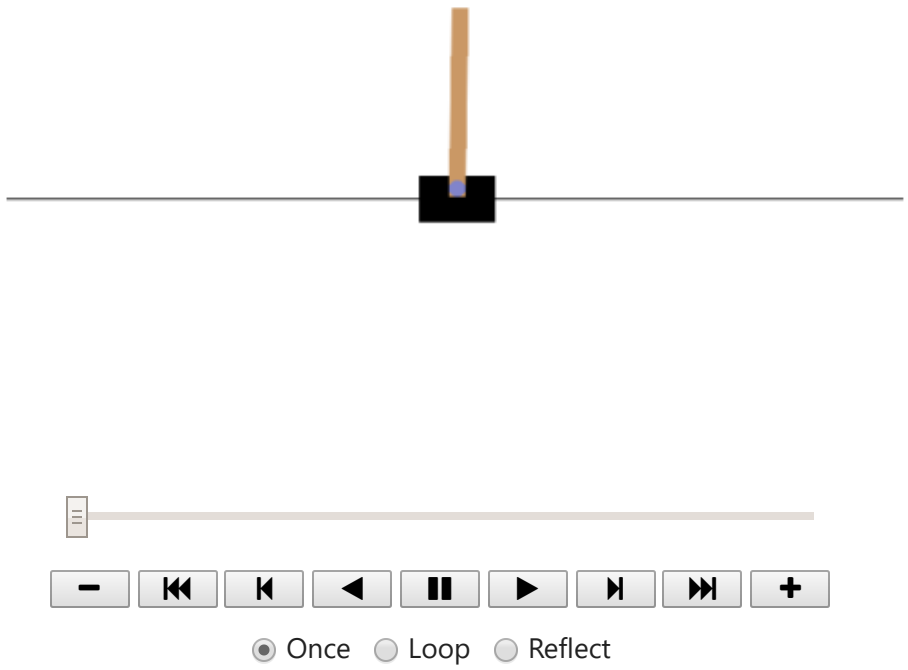
```
Episode: 600, Steps: 200, eps: 0.010
```

In [63]:
```
# extra code – this cell plots the learning curve
plt.figure(figsize=(8, 4))
plt.plot(rewards)
plt.xlabel("Episode", fontsize=14)
plt.ylabel("Sum of rewards", fontsize=14)
plt.grid(True)
plt.show()
```



In [64]:
```
# extra code – shows an animation of the trained DQN playing one episode
show_one_episode(epsilon_greedy_policy)
```

Out[64]:



由于目标模型的更新频率远低于在线模型，因此 Q-value 目标更稳定，我们之前讨论的反馈循环受到抑制，其影响也不那么严重。这种方法是 DeepMind 研究人员在其 2013 年论文中的主要贡献之一，它允许代理从原始像素学习玩 Atari 游戏。为了稳定训练，他们使用了 0.00025 的微小学习率，每 10000 步（而不是 50 步）更新一次目标模型，并且他们使用了 100 万次经验的非常大的回放缓冲区。他们非常缓慢地减少 **epsilon**，从 1 到 0.1，每 100 万步，他们让算法运行 5000 万步。此外，他们的 DQN 是一个深度卷积网络。

现在让我们来看看另一个 DQN 变体，它再次击败了 DQN 的技术水平。

11.2 Double DQN

在 2015 年的一篇论文中，DeepMind 研究人员调整了他们的 DQN 算法，提高了其性能并在一定程度上稳定了训练。他们称这种变体为 **double DQN**。该更新基于目标网络容易高估 Q-values 的观察结果。事实上，假设所有的动作都同样好：目标模型估计的 Q-values 应该是相同的，但由于它们是近似值，一些可能略大于其他，这纯属偶然。目标模型总是会选择最大的 Q-value，它会略大于平均 Q-value，很可能高估了真实的 Q-value（有点像在测量水池深度时计算最高随机波的高度)。为了解决这个问题，研究人员建议在为下一个状态选择最佳动

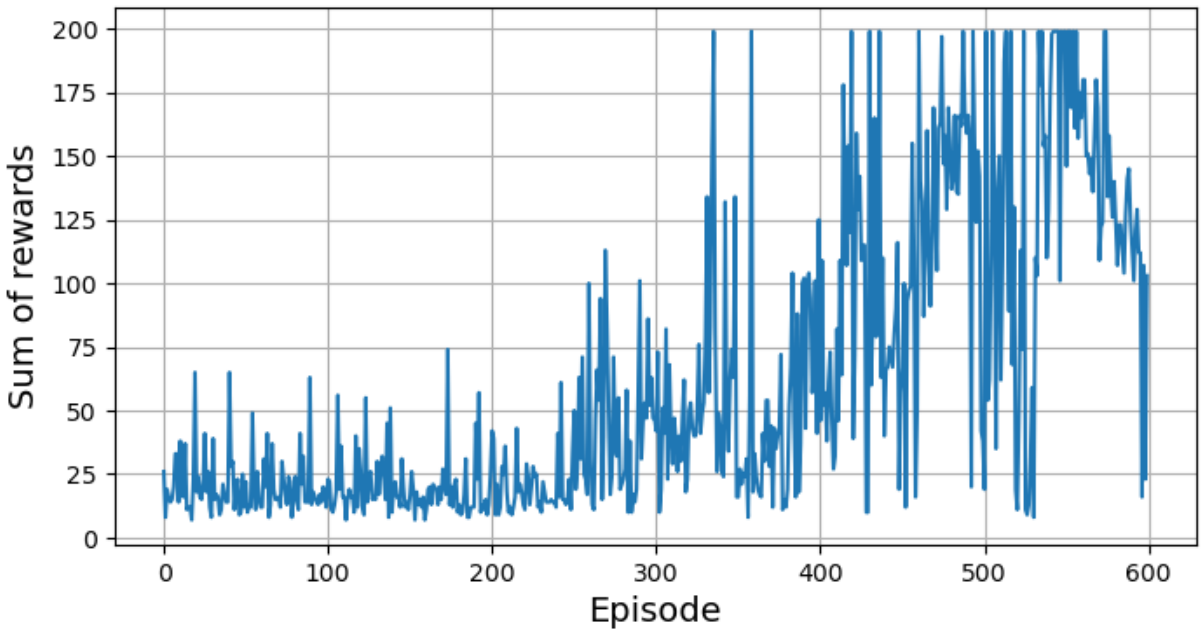
作时使用在线模型而不是目标模型，并且仅使用目标模型来估计这些最佳动作的 Q-value。这是更新后的 **training_step()** 函数：

In [57]:
```
tf.random.set_seed(42)

model = tf.keras.Sequential([
    tf.keras.layers.Dense(32, activation="elu", input_shape=input_shape),
    tf.keras.layers.Dense(32, activation="elu"),
    tf.keras.layers.Dense(n_outputs)
])

target = tf.keras.models.clone_model(model)  # clone the model's architecture
target.set_weights(model.get_weights())  # copy the weights

env.reset(seed=42)
np.random.seed(42)
tf.random.set_seed(42)
rewards = []
best_score = 0

batch_size = 32
discount_factor = 0.95
optimizer = tf.keras.optimizers.Nadam(learning_rate=1e-2)
loss_fn = tf.keras.losses.mean_squared_error

def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones, truncateds = experiences

    ##################### CHANGED SECTION ####################
    next_Q_values = model.predict(next_states, verbose=0)  # ≠ target.predict()
    best_next_actions = next_Q_values.argmax(axis=1)
    next_mask = tf.one_hot(best_next_actions, n_outputs).numpy()
    max_next_Q_values = (target.predict(next_states, verbose=0) * next_mask
                         ).sum(axis=1)
    ##########################################################

    runs = 1.0 - (dones | truncateds)  # episode is not done or truncated
    target_Q_values = rewards + runs * discount_factor * max_next_Q_values
    target_Q_values = target_Q_values.reshape(-1, 1)
    mask = tf.one_hot(actions, n_outputs)
    with tf.GradientTape() as tape:
        all_Q_values = model(states)
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
        loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))

    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))

replay_buffer = deque(maxlen=2000)

for episode in range(600):
    obs, info = env.reset()
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
```

```
        obs, reward, done, info, truncated = play_one_step(env, obs, epsilon)
        if done or truncated:
            break

    print(f"\rEpisode: {episode + 1}, Steps: {step + 1}, eps: {epsilon:.3f}",
          end="")
    rewards.append(step)
    if step >= best_score:
        best_weights = model.get_weights()
        best_score = step

    if episode > 50:
        training_step(batch_size)
        if episode % 50 == 0:
            target.set_weights(model.get_weights())

model.set_weights(best_weights)
```

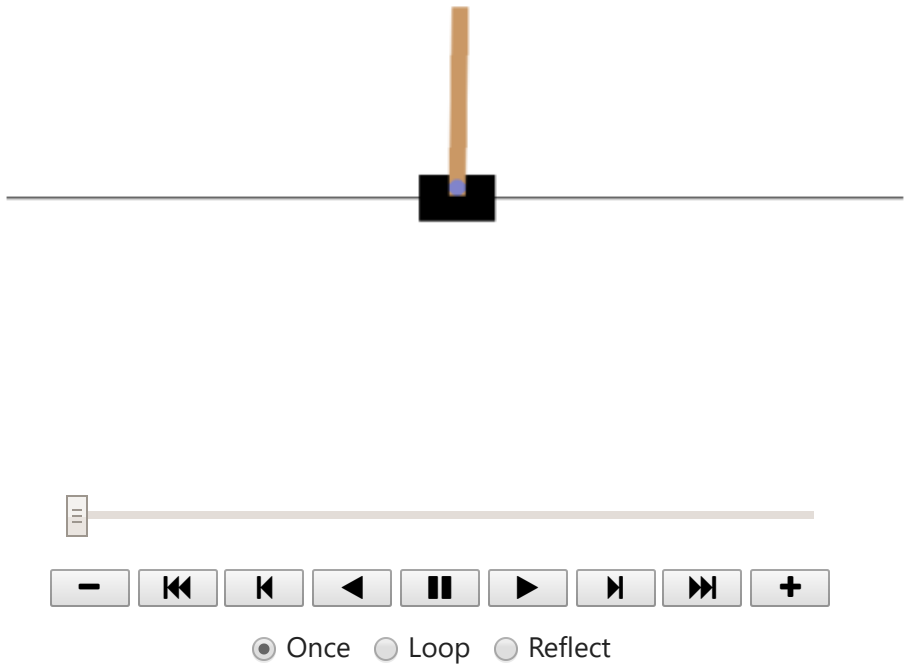
```
Episode: 600, Steps: 104, eps: 0.010
```

In [58]:
```
# extra code – this cell plots the learning curve
plt.figure(figsize=(8, 4))
plt.plot(rewards)
plt.xlabel("Episode", fontsize=14)
plt.ylabel("Sum of rewards", fontsize=14)
plt.grid(True)
plt.show()
```



In [59]:
```
# extra code – shows an animation of the trained DQN playing one episode
show_one_episode(epsilon_greedy_policy)
```

Out[59]:



仅仅几个月后，又提出了对 DQN 算法的另一项改进；我们接下来看看。

11.3 优先经验重放（Prioritized Experience Replay）

与其从重放缓冲区中 **均匀（uniformly）** 采样经验，不如更频繁地采样重要经验？这个想法被称为 **重要性抽样（importance sampling，IS）** 或 **优先经验重放（prioritized experience replay，PER）**，DeepMind 研究人员在 2015 年的一篇论文中引入了它（又一次！）。

更具体地说，如果经验可能导致快速学习进步，则它们被认为是“重要的”。但是我们如何估计呢？一种合理的方法是测量 TD 误差的大小 $\delta = r + \gamma \cdot V(s') - V(s)$。一个大的 TD 误差表明一个转换 (s, a, s') 非常令人惊讶，因此可能值得学习。当一个经验被记录在回放缓冲区中时，它的优先级被设置为一个非常大的值，以确保它至少被采样一次。然而，一旦它被采样（以及每次被采样），TD 误差 δ 就会被计算出来，并且这个经验的优先级被设置为 $p = |\delta|$（加上一个小常数以确保每个经验都具有非零采样概率）。对具有优先级的经验进行采样的概率 P 与 p^ζ 成正比，其中 ζ 是一个超参数，它控制我们希望重要性采样的贪婪程度：当 $\zeta = 0$ 时，我们只得到均匀采样，而当 $\zeta = 1$ 时，我们得到成熟的重要性抽样。在论文中，作者使用了 $\zeta = 0.6$，但最优值将取决于任务。

但是有一个问题：由于样本会偏向于重要的经验，我们必须在训练期间根据经验的重要性降低经验的权重来补偿这种偏差，否则模型只会过度拟合重要的经验。需要明确的是，我们希望更频繁地对重要经验进行采样，但这也意味着我们必须在训练期间赋予它们较低的权重。为此，我们将每个经验的训练权重定义为 $w = (nP)^{-\beta}$，其中 n 是回放缓冲区中的经验数量，β 是一个超参数，用于控制我们想要补偿重要性采样偏差的程度（0 表示完全没有，而 1 表示完全）。在论文中，作者在训练开始时使用 $\beta = 0.4$，并在训练结束时线性增加到 $\beta = 1$。同样，最优值将取决于任务，但如果你增加一个，你通常也想增加另一个。

现在让我们来看看DQN算法的最后一个重要变体。

11.4 Dueling DQN

DeepMind 研究人员在 2015 年的另一篇论文中介绍了 **Dueling DQN** 算法（DDQN，不要与 Double DQN 混淆，尽管这两种技术可以很容易地结合使用）。要理解它是如何工作的，我们必须首先注意到状态-动作对 (s, a) 的 Q-value 可以表示为 $Q(s, a) = V(s) + A(s, a)$，其中 $V(s)$ 是状态 s 的值，$A(s, a)$ 是在状态 s 中采取动作 a 与该状态下所有其他可能动作相比的 **优势（advantage）**。此外，状态的值等于该状态的最佳动作 a^* 的 Q-value（因为我们假设最优策略将选择最佳动作），因此 $V(s) = Q(s, a*)$，这意味着 $A(s, a^*) = 0$。在 Dueling DQN 中，模型估计状态值和每个可能动作的优势。由于最佳动作的优势应该为 0，因此模型会从所有预测优势中减去最大预测优势。这是一个简单的 DDQN 模型，使用函数式 API 实现：

In [60]:
```
tf.random.set_seed(42)  # extra code – ensures reproducibility on the CPU

input_states = tf.keras.layers.Input(shape=[4])
hidden1 = tf.keras.layers.Dense(32, activation="elu")(input_states)
hidden2 = tf.keras.layers.Dense(32, activation="elu")(hidden1)
state_values = tf.keras.layers.Dense(1)(hidden2)
raw_advantages = tf.keras.layers.Dense(n_outputs)(hidden2)
advantages = raw_advantages - tf.reduce_max(raw_advantages, axis=1,
                                            keepdims=True)
Q_values = state_values + advantages
model = tf.keras.Model(inputs=[input_states], outputs=[Q_values])
```

该算法的其余部分与前面的相同。事实上，你可以构建一个 Double Dueling DQN，并将其与优先经验重放相结合！更一般地说，许多强化学习技术可以结合使用，正如 DeepMind 在 2017 年的一篇论文中所展示的那样：该论文的作者将六种不同的技术结合到一个名为 Rainbow 的智能体中，其性能在很大程度上优于现有技术。

正如你所看到的，深度强化学习是一个快速增长的领域，还有更多需要发现！

12. 一些流行 RL 算法综述（Overview of Some Popular RL Algorithms）

在我们结束本章之前，让我们简要介绍一下其他一些流行的算法：

- **AlphaGo**：AlphaGo 使用基于深度神经网络的 **蒙特卡罗树搜索（Monte Carlo tree search，MCTS）** 变体在围棋比赛中击败人类冠军。MCTS 于 1949 年由 Nicholas Metropolis 和 Stanislaw Ulam 发明。 它在运行多次模拟后选择最佳着法，从当前位置开始反复探索搜索树，并在最有希望的分支上花费更多时间。当它到达一个之前没有访问过的节点时，它会随机播放直到游戏结束，并更新它对每个访问过的节点的估计（不包括随机移动），根据最终结果增加或减少每个估计。AlphaGo 基于相同的原理，但它使用策略网络来选择着法，而不是随机下棋。该策略网络使用策略梯度进行训练。原始算法涉及三个以上的神经网络，并且更加复杂，但在 AlphaGo Zero 论文中进行了简化，它使用单个神经网络来选择动作和评估游戏状态。 AlphaZero 论文推广了这种算法，使其不仅能够应对围棋游戏，而且能够应对国际象棋和将棋（日本象棋）。 最后，MuZero 论文继续改进该算法，即使智能体在不知道游戏规则的情况下开始，也优于之前的迭代！
- **Actor-critic algorithms**：Actor-critics 是一系列 RL 算法，将策略梯度与深度 Q-networks 相结合。一个 actor-critic 代理包含两个神经网络：策略网络 和 DQN。DQN 通过学习代理的经验进行正常训练。策略网络的学习方式与常规 PG 不同（并且快得多）：代理（演员）不是通过经历多个事件来估计每个动作的价值，然后对每个动作的未来折扣奖励求和，最后对其进行归一化，而是依赖关于 DQN（评论家）估计的动作值。这有点像运动员（代理）在教练（DQN）的帮助下学习。
- **Asynchronous advantage actor-critic (A3C)**：这是 DeepMind 研究人员在 2016 年引入的一个重要的 actor-critic 变体，其中多个代理并行学习，探索环境的不同副本。每隔一段时间，但异步（因此得名），每个代理将一些权重更新推送到主网络，然后从该网络中提取最新的权重。因此，每个代理都有助于改进主网络，并从其他代理所学到的知识中受益。此外，DQN 不是估计 Q-values ，而是估计每个动作的优势（因此名称中的第二个 A），从而稳定训练。
- **Advantage actor-critic (A2C)**：A2C 是 A3C 算法的一种变体，它消除了异步性。所有的模型更新都是同步的，所以梯度更新是在更大的批次上执行的，这允许模型更好地利用GPU的能力。
- **Soft actor-critic (SAC)**：SAC 是 Tuomas Haarnoja 和加州大学伯克利分校的其他研究人员于 2018 年提出的 actor-critic 变体。它不仅学习奖励，还学习最大化其行为的熵。换句话说，它试图尽可能地不可预测，同时仍然获得尽可能多的奖励。这会鼓励代理探索环境，从而加快训练速度，并降低在 DQN 产生不完美估计时重复执行相同操作的可能性。该算法展示了惊人的样本效率（与之前所有学习非常缓慢的算法相反）。
- **Proximal policy optimization (PPO)**：John Schulman 和其他 OpenAI 研究人员的这个算法基于 A2C，但它对损失函数进行了裁剪以避免过大的权重更新（这通常会导致训练不稳定）。PPO 是之前的 **信任域策略优化（trust region policy optimization，TRPO）** 算法的简化版，也是由 OpenAI 开发的。OpenAI 于 2019 年 4 月发布新闻，其基于 PPO 算法的名为 OpenAI Five 的人工智能在多人游戏 Dota 2 中击败了世界冠军。
- **Curiosity-based exploration**：RL 中反复出现的问题是奖励的稀疏性，这使得学习非常缓慢且效率低下。Deepak Pathak 和加州大学伯克利分校的其他研究人员提出了一种令人兴奋的方法来解决这个问题：为什么不忽略奖励，而只是让智能体对探索环境充满好奇呢？因此，奖励成为代理固有的，而不是来自环境。同样，激发孩子的好奇心比单纯奖励孩子取得好成绩更有可能带来好结果。这是如何运作的？智能体不断尝试预测其行为的结果，并寻找结果与其预测不符的情况。换句话说，它想要感到惊讶。如果结果是可

预测的（无聊的），它就会去别处。然而，如果结果不可预测，但代理注意到它无法控制它，一段时间后它也会感到无聊。出于好奇，作者成功地在许多视频游戏中训练了一个智能体：即使智能体不会因失败而受到惩罚，游戏也会重新开始，这很无聊，所以它学会了避免它。

- **Open-ended learning (OEL)**：OEL 的目标是训练代理能够不断学习新的和有趣的任务，这些任务通常是程序生成的。我们还没有到那一步，但在过去几年里已经取得了一些惊人的进步。例如，Uber AI 的一组研究人员在 2019 年发表的一篇论文介绍了 POET 算法，该算法生成多个带有凹凸不平的模拟 2D 环境，并在每个环境中训练一个代理：代理的目标是尽可能快地行走，同时避开障碍物。该算法从简单的环境开始，但随着时间的推移逐渐变得困难：这称为 **课程学习（curriculum learning）**。此外，虽然每个代理只在一个环境中接受训练，但它必须定期与所有环境中的其他智能体竞争。在每个环境中，获胜者都会被复制过来并取代之前的代理。通过这种方式，知识会定期跨环境传输，并选择最适应的代理。最后，代理比接受过单一任务训练的代理更能步行，而且他们可以应对更艰难的环境。当然，这个原则也可以应用到其他环境和任务中。如果您对 OEL 感兴趣，请务必查看增强型 POET 论文以及关于该主题的 DeepMind 2021 年论文。

注意：如果您想了解有关强化学习的更多信息，请查看 Phil Winder (O'Reilly) 的 Reinforcement Learning 一书。

本章涵盖了很多主题：策略梯度、马尔可夫链、马尔可夫决策过程、Q-learning、近似 Q-learning 和深度 Q-learning 及其主要变体（固定 Q-value 目标、Double DQN、Dueling DQN 和 优先经验重播），最后我们快速浏览了一些其他流行的算法。强化学习是一个巨大而令人兴奋的领域，每天都会涌现出新的想法和算法，所以我希望本章能激发您的好奇心：整个世界都可以探索！